Comment passer d'une matrice à un vecteur et réciproquement

```
/*
```
En ce qui concerne la 'bonne' déclaration d'une matrice (telle qu'utilisée ici), le lecteur se reportera à l'autre document nommé « Comment 'bien' déclarer et utiliser en C un tableau de dimension 2 ». Dans ce document, on ne rappellera que rapidement comment il faut faire et le lecteur verra que, selon les cas, la meilleure façon restera de travailler avec seulement un vecteur (voir à la fin).
```
*/


#include <stdio.h>            // pour printf, puts
#include <stdlib.h>           // pour malloc, free


/*
```
ICI, le programme principal `main` renvoie des codes de bon/mauvais fonctionnement ; il s'agit des valeurs `EXIT_SUCCESS` et `EXIT_FAILURE` définies dans `stdlib.h` .
```
*/

int main (void)
{


/*
```
On introduit une première instruction `typedef` afin de pouvoir changer plus facilement de type de données traitées par ce programme. Par exemple, si l'on veut 'convertir' ce programme pour qu'il fonctionne avec des entiers (`short int` par exemple), il suffit de modifier qu'une seule ligne en remplaçant `typedef double TYPE` par `typedef short int TYPE` .
Le lecteur remarquera que, pour que le programme puisse être correct sans autre modification, il a fallu compliquer un tout petit peu les instructions de visualisation des données.
De même, les deux autres instructions `typedef` servent à introduire (c'est de la meilleure pratique que d'introduire ainsi des nouveaux tpyes) les deux autres types manipulés par ce programme, à savoir, le type `VECTOR` et le type `MATRIX` construits à partir de `TYPE` précédemment défini.
```
*/

   typedef double TYPE ;
// typedef float TYPE ;
// typedef long int TYPE ;
// typedef short int TYPE ;
// typedef unsigned char TYPE ;
// ...
```

```
   typedef TYPE * VECTOR ;
   typedef TYPE * * MATRIX ;


/*
```
Ici, on se définit `dim` comme étant la longueur du vecteur ; cette longueur devant correspondre exactement (c'est mieux !) au nombre de cases de la matrice correspondante, on définit donc la valeur de `dim` à partir de celles de `dim1` et `dim2` , valeurs du nombre de lignes et de colonnes de la matrice.
De plus, à titre pédagogique, pour un vecteur de 8 cases (notre exemple), on se définit quatre possibilités de voir un tel vecteur soit comme une matrice 2 x 4 , soit comme une matrice 4 x 2 ; Noter aussi les possibilités d'avoir des matrices 1 x 8 ou 8 x 1 (ce qui n'est pas équivalent à un vecteur …).
```
*/

// int const dim1 = 8 , dim2 = 1 ;
   int const dim1 = 4 , dim2 = 2 ;
// int const dim1 = 2 , dim2 = 4 ;
// int const dim1 = 1 , dim2 = 8 ;

   int const dim = dim1 * dim2 ;


   int i, j ;                  // les indices des boucles


/*
```
On déclare :

- d'une part, la matrice de départ `m_in` et le vecteur qui lui sera équivalent `v_out` utilisés dans la première partie de notre exemple (on commence toujours par le plus facile …)
- d'autre part, le vecteur de départ `v_in` et la matrice que l'on voudra lui rendre équivalente `m_out` utilisés dans la seconde partie de notre exemple (il y a plus de travail à faire …).

```
*/

   VECTOR v_out , v_in ;
   MATRIX m_in , m_out ;


   puts ( " Premiere partie : Matrice ==> Vecteur \n" ) ;


/*
```
Dans cette première partie, on part d'une matrice construite avec des cases contiguës `m_in` .
Le but poursuivi est de créer un alias de la matrice (c'est-à-dire ayant la même adresse et, bien sûr, sans réservation de mémoire supplémentaire) qui soit vu comme un (simple) vecteur `v_out` .
```
*/
```

```
/*
```
Dans un premier temps, on va donc créer cette matrice et la remplir par des valeurs reconnaissables. Ceci est fait en allouant d'abord, pour l'objet de type `MATRIX` , un vecteur destiné à recevoir les 'pointeurs' des lignes, puis en allouant la mémoire nécessaire aux éléments de la matrice pour le premier pointeur de ligne i.e. `m_in[0]` soit encore `*m_in` ; il ne reste plus qu'à indiquer que les autres pointeurs de lignes correspondent au début des autres lignes de la matrice : c'est le rôle de la boucle `for` . Remarquer que l'on se sert dans la boucle `for` d'une des propriétés fondamentales de l'arithmétique des pointeurs. (Pour plus de détails, voir le document déjà signalé au début).
```
*/

   m_in = (MATRIX) malloc( dim1 * sizeof(VECTOR) ) ;
   if ( NULL == m_in ) return(EXIT_FAILURE) ;

   *m_in = (VECTOR) malloc( dim * sizeof(TYPE) ) ;
   if ( NULL == *m_in ) return(EXIT_FAILURE) ;

   for ( i = 1 ; i < dim1 ; ++i ) m_in[i] = m_in[i-1] + dim2 ;


   for ( i = 0 ; i < dim1 ; ++i)
   {
      for ( j = 0 ; j < dim2 ; ++j ) m_in[i][j] = (TYPE)(10*i+j) ;
   }

   for ( i = 0 ; i < dim1 ; ++i)
   {
      for ( j = 0 ; j < dim2 ; ++j )
      printf( " m_in[%d][%d] = %g \n" , i, j, (double)m_in[i][j] ) ;
   }
   puts(" ") ;

```
/*
```
On va maintenant dire que le vecteur `v_out` est équivalent à la matrice `m_in` . Comme les cases mémoires sont déjà allouées, il suffit de donner comme valeur à ce pointeur la valeur qui lui convient ! Nous voyons, à la fois pour des raisons de correspondance de type (chaque entité de part et d'autre de l'affectation doivent être du même type, ici `TYPE *` ) et pour des raisons de taille d'allocation (c'est bien à `*m_in` que l'on a alloué la taille nécessaire au stockage des données), que la seule possibilité est celle qui est indiquée. Remarquons toutefois que si la matrice avait été allouée de manière statique, l'indirection n'étant pas possible, on aurait été obligé d'écrire l'affectation autrement : `v_out = &m_in[0][0] ;`
```
*/

   v_out = *m_in ;   // tout est là

   puts(" On fait les verifications : ") ;

   printf(" pour m_in : %p \n", m_in) ;
   printf(" pour *m_in : %p \n", *m_in) ;
   printf(" pour v_out : %p \n", v_out) ;

   puts (" OK pour les adresses \n") ;
```
```

```
   for ( i = 0 ; i < dim ; ++i )
      printf( " v_out[%d] = %g \n" , i, (double)v_out[i] ) ;
   puts ( " OK aussi pour les valeurs \n" ) ;


/*
```
Même si ce n'est pas strictement nécessaire dans le programme principal, on n'oublie pas de désallouer proprement ce qui a été alloué dynamiquement.
```
*/

   free( *m_in ) ;
   free( m_in ) ;

   puts( " FIN DE LA PREMIERE PARTIE \n\n" ) ;


/*
```

Et maintenant, on va essayer de faire l'inverse !!

```
À savoir, en partant d'un vecteur v_in (dont les cases seront donc, par construction, contiguës), on veut créer un alias (c'est-à-dire ayant la même adresse mais, comme on le verra par la suite et contrairement au premier cas, au prix d'une réservation de mémoire supplémentaire) qui soit vu comme une matrice m_out , au choix 8x1, 4x2, 2x4 ...
*/

   puts ( " Seconde partie : Vecteur ==> Matrice \n" ) ;


/*
On alloue d'abord les cases du vecteur puis on les remplit avec des valeurs reconnaissables.
*/

   v_in = malloc( dim * sizeof(TYPE) ) ;
   if ( NULL == v_in ) return(EXIT_FAILURE) ;

   for ( i = 0 ; i < dim ; ++i ) v_in[i] = (TYPE)(i+100) ;

   for ( i = 0 ; i < dim ; ++i )
        printf( " v_in[%d] = %g \n" , i, (double)v_in[i] ) ;
   puts (" ") ;


/*
Comme on veut travailler au final avec une matrice, on n'a pas d'autre choix que de créer (ce qui implique l'utilisation de mémoire supplémentaire) comme dans le premier cas, un vecteur destiné à recevoir les 'pointeurs' des lignes de la (future) matrice.
*/

   m_out = (MATRIX) malloc( dim1 * sizeof(VECTOR) ) ;
   if ( NULL == m_out ) return(EXIT_FAILURE) ;
```

```
/*
```
Il ne reste plus qu'à donner la bonne valeur (l'adresse du vecteur `v_in`) au premier 'pointeur' de lignes de la matrice ; remarquons, comme précédemment que c'est bien `*m_out` qui est bien du même type que `v_in` , à savoir `TYPE *`).
```
*/

   *m_out = v_in ;


/*
```
Sans faire l'allocation de `m_out` , on aurait pu penser qu'il était *suffisant* de faire :
```
   m_out = (MATRIX)v_in ;    *m_out = v_in ;
```
ce qui implique, en ce qui concerne la valeur des adresses, que non seulement celle de `*m_out` est la même que celle de `v_in` (normal) mais aussi celle de `m_out` . Malheureusement, du fait de l'absence d'allocation pour les 'pointeurs' des lignes, ceux-ci seront donc stockés dans les premières cases de `m_out` , cet objet ayant la même adresse que `*m_out` , écrasant par là les valeurs des premières cases du vecteur `v_in` (correspondantes à la première ligne de la matrice). Faire l'expérience pour voir !!

Et comme dans le cas précédent, il faut indiquer que les autres pointeurs de lignes correspondent au début des différentes lignes de la matrice. Remarquer que l'on se sert dans la boucle `for` d'une des propriétés fondamentales de l'arithmétique des pointeurs.
```
*/

   for ( i = 1 ; i < dim1 ; ++i ) m_out[i] = m_out[i-1] + dim2 ;


   puts( " On fait les verifications : " ) ; //  quelques vérifications !

   printf( " pour v_in     : %p \n", v_in     ) ;
   printf( " pour m_out    : %p \n", m_out    ) ;
   printf( " pour *m_out   : %p \n", *m_out   ) ;
   printf( " pour m_out[0] : %p \n", m_out[0] ) ;     // idem !

   puts (" OK pour les adresses \n") ;


   for ( i = 1 ; i < dim1 ; ++i )
      printf( " pour m_out[%d] : %p \n", i, m_out[i] ) ;

   if ( dim1 > 0 )
      puts (" OK aussi pour les adresses attendues \n") ;


   for ( i = 0 ; i < dim1 ; ++i)
   {
      for ( j = 0 ; j < dim2 ; ++j )
      printf( " m_out[%d][%d] = %g \n" , i, j, (double)m_out[i][j] ) ;
   }

   puts (" OK aussi pour les valeurs \n") ;
```

```
/*
```
Même si ce n'est pas strictement nécessaire dans le programme principal, on n'oublie pas de désallouer proprement ce qui a été alloué dynamiquement.
```
*/

   free( v_in ) ;
   free( m_out ) ;

   puts( " FIN DE LA SECONDE PARTIE \n\n" ) ;


   return(EXIT_SUCCESS) ;
}
```

Remarques finales :

Au total, nous voyons que l'utilisation d'une matrice (tableau à 2 dimensions) consomme toujours plus de mémoire qu'un seul vecteur de taille (mathématique) identique.
Aussi et même si les cases de la matrice sont contiguës (je n'imagine pas d'ailleurs que l'on puisse, pour des matrices carrées, rectangulaires et même triangulaires ou à bande, faire autrement !), l'usage d'un tableau à 2 dimensions requiert toujours 2 indirections (c'est l'opérateur `*` ou sa version cachée en `[..][..]`) ce qui consomme aussi un peu plus de temps de calcul dans les accès mémoire.
Ceci dit, un des gros avantages de l'utilisation des matrices en tant que telles, reste la proximité entre l'objet mathématique (et les utilisations que l'on en fait) et sa version informatique. Si cela était demandé (en exercice, par exemple) de faire une procédure de résolution d'un système linéaire par la méthode du (plus grand) pivot en n'utilisant que des objets (informatiques) de type vecteur, le code écrit serait bien moins facile à lire et à construire que sa version naturelle en utilisant des matrices.

Remerciements à Alain NoulleZ pour sa participation à la résolution de ce problème.

Et voici, les résultats du programme (bien sûr les sorties obtenues pour la valeur des pointeurs dépendent du système utilisé).

```
Premiere partie : Matrice ==> Vecteur

 m_in[0][0] = 0
 m_in[0][1] = 1
 m_in[1][0] = 10
 m_in[1][1] = 11
 m_in[2][0] = 20
 m_in[2][1] = 21
 m_in[3][0] = 30
 m_in[3][1] = 31

 On fait les verifications :
 pour m_in  : 002F07A8
 pour *m_in : 002F07F0
 pour v_out : 002F07F0
 OK pour les adresses
```

```
v_out[0] = 0
v_out[1] = 1
v_out[2] = 10
v_out[3] = 11
v_out[4] = 20
v_out[5] = 21
v_out[6] = 30
v_out[7] = 31
OK aussi pour les valeurs

FIN DE LA PREMIERE PARTIE


Seconde partie : Vecteur ==> Matrice

v_in[0] = 100
v_in[1] = 101
v_in[2] = 102
v_in[3] = 103
v_in[4] = 104
v_in[5] = 105
v_in[6] = 106
v_in[7] = 107

On fait les verifications :
pour v_in     : 002F07A8
pour m_out    : 002F0820
pour *m_out   : 002F07A8
pour m_out[0] : 002F07A8
OK pour les adresses

pour m_out[1] : 002F07B8
pour m_out[2] : 002F07C8
pour m_out[3] : 002F07D8
OK aussi pour les adresses attendues

m_out[0][0] = 100
m_out[0][1] = 101
m_out[1][0] = 102
m_out[1][1] = 103
m_out[2][0] = 104
m_out[2][1] = 105
m_out[3][0] = 106
m_out[3][1] = 107
OK aussi pour les valeurs

FIN DE LA SECONDE PARTIE
```

Page blanche