Programmation en C
Allocation dynamique

Régis Barbanchon

L1 Info-Math, Semestre 2

Les allocations automatique et statique (1/4)

Lors de la définition d'une variable à l'intérieur d'une fonction, on a rencontré jusqu'ici deux types d'allocation de mémoire :

l'allocation **automatique** et l'allocation **statique**.

Les allocations automatique et statique (1/4)

Lors de la définition d'une variable à l'intérieur d'une fonction, on a rencontré jusqu'ici deux types d'allocation de mémoire : l'allocation **automatique** et l'allocation **statique**.

- **l'allocation automatique :**
  - la variable est distincte pour chaque invocation de fonction ;
  - elle est créée automatiquement à l'entrée de sa portée ;
  - elle est détruite automatiquement à la sortie de sa portée ;
  - elle n'existe donc pas en dehors de l'exécution de la fonction ;
  - on ne peut donc pas en retourner l'adresse.

Les allocations automatique et statique (1/4)

Lors de la définition d'une variable à l'intérieur d'une fonction, on a rencontré jusqu'ici deux types d'allocation de mémoire :
l'allocation **automatique** et l'allocation **statique**.

- **l'allocation automatique :**
  - la variable est distincte pour chaque invocation de fonction ;
  - elle est créée automatiquement à l'entrée de sa portée ;
  - elle est détruite automatiquement à la sortie de sa portée ;
  - elle n'existe donc pas en dehors de l'exécution de la fonction ;
  - on ne peut donc pas en retourner l'adresse.
- **l'allocation statique :** (avec le mot-clé `static`)
  - une unique variable globale est créée ;
  - elle existe du début à la fin du programme ;
  - elle est réutilisée à chaque appel ;
  - elle pré-existe avant le premier appel de fonction ;
  - elle survit à la fin des exécutions de fonction ;
  - on peut donc en retourner l'adresse.

Les allocations automatique et statique (2/4)

Donnons nous une structure MyTime représentant l'heure, et écrivons une fonction retournant sa représentation-texte.

```
typedef struct MyTime { int hour, minute; } MyTime;
```

Le code suivant est illégal, car MyTime_ToAutoString() tente de retourner l'adresse de sa variable text qui est **automatique** :

```
char * MyTime_ToAutoString (MyTime my_time) {
  char text[140]; // automatic allocation when entering the function
  snprintf (text, 140, "%02d:%02d", my_time.hour, my_time.minute);
  return text;    // illegal return of automatic variable adress
}                 // automatic deallocation when leaving the function
```

```
int main (void){
  MyTime noon= (MyTime) { .hour= 12, .minute= 1 };
  char * noon_text= MyTime_ToAutoString (noon); // illegal
  printf ("%s\n", noon_text);
  return 0;
}
```

La compilation avec clang donne le diagnostic suivant :

```
warning: address of stack memory associated with local variable
     'text' returned [-Wreturn-stack-address]
 return text;
        ^~~~
```

Les allocations automatique et statique (3/4)

Le code suivant est légal, car `MyTime_ToStaticString()` retourne l'adresse de sa variable `text` qui est **statique** :

```
char * MyTime_ToStaticString (MyTime my_time) {
  static char text[140]; // allocated once at program start
  snprintf (text, 140, "%02d:%02d", my_time.hour, my_time.minute);
  return text;           // legal return of address
}                        // variable is never deallocated
```

L'usage d'une unique variable peut réserver des surprises :

```
int main (void){
  MyTime time1= (MyTime) { .hour= 23, .minute= 59 };
  MyTime time2= (MyTime) { .hour= 12, .minute=  1 };
  char * text1= MyTime_ToStaticString (time1); // legal
  char * text2= MyTime_ToStaticString (time2); // legal, overwrite same variable
  printf ("%p %s\n", (void *) text1, text1);   // yields unexpected output
  printf ("%p %s\n", (void *) text2, text2);
  return 0;
}
```

La sortie montre que le texte du 2ème appel à écrasé celui du 1er, et que `text1` et `text2` pointent sur la même variable `text` :

```
0x603080 12:01
0x603080 12:01
```

Les allocations automatique et statique (4/4)

Remarque : ne pas confondre le **retour illégal de l'adresse** d'une variable automatique et le **retour légal d'une copie** de sa valeur :

```
#define TWEET_SIZE 140
typedef struct Tweet { char text[TWEET_SIZE]; } Tweet;
```

Par exemple, la fonction suivante retourne un tweet par copie :

```
Tweet MyTime_ToTweet (MyTime my_time) {
  Tweet tweet;  // automatic allocation when entering the function
  snprintf (tweet.text, TWEET_SIZE, "%02d:%02d", my_time.hour, my_time.minute);
  return tweet; // legal copy of value into caller variable
}               // automatic deallocation when leaving the function
```

```
int main (void){
  MyTime time1= (MyTime) { .hour= 23, .minute= 59 };
  MyTime time2= (MyTime) { .hour= 12, .minute=  1 };
  Tweet tweet1= MyTime_ToTweet (time1); // tweet copied into tweet1
  Tweet tweet2= MyTime_ToTweet (time2); // tweet copied into tweet2
  printf ("%p %s\n", (void *) tweet1.text, tweet1.text);
  printf ("%p %s\n", (void *) tweet2.text, tweet2.text);
  return 0;
}
```

Une sortie possible (les adresses changent selon les exécutions) :

```
0x7fffc769b128 23:59
0x7fffc769b098 12:01
```

Allocation dynamique : `malloc()` et `free()`

L'allocation dynamique consiste à allouer (et désallouer) explicitement des blocs de bytes contigüs en mémoire.

Les deux fonctions suivantes sont déclarées dans `<stdlib.h>` :

```
void * malloc (size_t nb_bytes);

void free (void * block);
```

Allocation dynamique : `malloc()` et `free()`

L'allocation dynamique consiste à allouer (et désallouer) explicitement des blocs de bytes contigüs en mémoire.

Les deux fonctions suivantes sont déclarées dans `<stdlib.h>` :

```
void * malloc (size_t nb_bytes);

void free (void * block);
```

- `void * block= malloc(n)` alloue dynamiquement `n` bytes :
  - l'adresse du bloc est retournée dans `block` en cas de succès.
  - l'alignement du bloc convient pour tout type de donnée.
  - `NULL` est retourné dans `block` en cas d'échec.

Allocation dynamique : malloc() et free()

L'allocation dynamique consiste à allouer (et désallouer) explicitement des blocs de bytes contigüs en mémoire.

Les deux fonctions suivantes sont déclarées dans <stdlib.h> :

```
void * malloc (size_t nb_bytes);

void free (void * block);
```

- `void * block= malloc(n)` alloue dynamiquement `n` bytes :
  - l'adresse du bloc est retournée dans `block` en cas de succès.
  - l'alignement du bloc convient pour tout type de donnée.
  - `NULL` est retourné dans `block` en cas d'échec.

- `free(block)` désalloue un bloc alloué dynamiquement :
  - `free(block)` est illégal si `block` n'est pas issu de `malloc()`.
  - `free(block)` est illégal si `block` a déjà été désalloué.
  - `free(block)` est légal (et ne fait rien) si `block` est `NULL`.

Allocation dynamique : pour une variable simple (1/2)

Pour réserver un bloc dédié à une variable d'un certain type, on utilise la primitive `sizeof` qui donne sa taille en byte :

```
int main (void) {
  int * devil_number= malloc (sizeof (int));
  assert (devil_number != NULL);
  * devil_number= 666;
  printf ("%d\n", * devil_number);
  free (devil_number);
  return 0;
}
```

```
666
```

```
int main (void) {
  double * devil_number= malloc (sizeof (double));
  assert (devil_number != NULL);
  * devil_number= 6.6;
  printf ("%f\n", * devil_number);
  free (devil_number);
  return 0;
}
```

```
6.6
```

Rappelons qu'un pointeur `void *` se convertit en pointeur typé sans cast explicite. On ne caste donc pas le retour de `malloc()`.

Allocation dynamique : pour une variable simple (2/2)

En préférant `sizeof expr` plutôt que `sizeof (TypeName)`,
on peut éviter la répétition de la mention du type `TypeName`,
et ainsi éviter des erreurs de maintenance lorsque le type change :

```
int main (void) {
  int * devil_number= malloc (sizeof * devil_number);
  assert (devil_number != NULL);
  * devil_number= 666;
  printf ("%d\n", * devil_number);
  free (devil_number);
  return 0;
}
```

```
int main (void) {
  double * devil_number= malloc (sizeof * devil_number);
  assert (devil_number != NULL);
  * devil_number= 6.6;
  printf ("%f\n", * devil_number);
  free (devil_number);
  return 0;
}
```

Remarque : la primitive `sizeof expr` n'évalue pas `expr`, et donc `sizeof * devil_number` ne déréférence pas `devil_number`.

Allocation dynamique : pour une variable tableau

Pour allouer un tableau il suffit d'allouer sa taille en byte,
c-à-d, le nombre de cases multiplié par la taille d'une cellule.

Ici, `reals` pointe sur un tableau de 5 flottants,
alloué dynamiquement puis désalloué :

```
int main (void) {
  int length= 5;
  double * reals= malloc (length * sizeof * reals);
  assert (reals != NULL);
  for (int k= 0; k < length; k++) reals [k]= k*k + 0.5;
  for (int k= 0; k < length; k++) printf (" %f ", reals[k]);
  printf ("\n");
  free (reals);
  return 0;
}
```

```
 0.5   1.5   4.5   9.5   16.5
```

Remarque : On a utilisé l'expression `length * sizeof * reals`.
On aurait aussi pu écrire `length * sizeof reals[0]`.
Les deux sont équivalentes à `length * sizeof (double)`.

Allocation dynamique et fonctions (1/2) : sur une chaîne

L'allocation dynamique permet d'allouer un bloc dans une fonction est de désallouer ce bloc dans une autre fonction :

```
char * MyTime_ToDynamicString (MyTime my_time) {
  int length= snprintf (NULL, 0, "%02d:%02d", my_time.hour, my_time.minute);
  char * text= malloc (1 + length * sizeof * text); // one extra char for '\0'
  assert (text != NULL);
  snprintf (text, 1+length, "%02d:%02d", my_time.hour, my_time.minute);
  return text;
}
```

Remarque : la forme `snprintf(NULL, 0, format,...)` permet de calculer la longueur de la chaîne formatée.

```
int main (void) {
  MyTime time1= (MyTime) { .hour= 12, .minute=  1 };
  MyTime time2= (MyTime) { .hour= 23, .minute= 59 };
  char * text1= MyTime_ToDynamicString (time1);
  char * text2= MyTime_ToDynamicString (time2);
  printf ("%p %s\n", (void *) text1, text1);
  printf ("%p %s\n", (void *) text2, text2);
  free (text1); free (text2);
  return 0;
}
```

```
0x802420 12:01
0x802440 23:59
```

Allocation dynamique et fonctions (2/2) : sur une structure

Écrivons un initialiseur, puis un constructeur et un destructeur pour la structure MyTime allouée dynamiquement :

```
void MyTime_Init (MyTime * my_time, int hour, int minute) { // initializer
  my_time->hour= hour;
  my_time->minute= minute;
}
```

```
MyTime * MyTime_Create (int hour, int minute) { // constructor
  MyTime * my_time= malloc (sizeof * my_time);
  assert (my_time != NULL);
  MyTime_Init (my_time, hour, minute);
  return my_time;
}
```

```
void MyTime_Destroy (MyTime * my_time) { // destructor
  free (my_time);
}
```

Exemple d'utilisation :

```
int main (void) {
  MyTime * my_time= MyTime_Create (12, 1);
  printf ("%d:%d\n", my_time->hour, my_time->minute);
  MyTime_Init (my_time, 23, 59);
  printf ("%d:%d\n", my_time->hour, my_time->minute);
  MyTime_Destroy (my_time);
  return 0;
}
```

Tableau encapsulé dans une structure (1/7)

Écrivons une structure `ArrayOfDouble`
représentant un tableau de flottants qui connaît sa longueur.

```
typedef struct ArrayOfDouble {
  double * cells;
  int length;
} ArrayOfDouble;
```

Les cases sont allouées dynamiquement via le champ `cells`.

Voici l'initialiseur de `ArrayOfDouble` :

```
void ArrayOfDouble_Init (ArrayOfDouble * array, int length) { // initializer
  array->cells= malloc (length * sizeof * array->cells);
  assert (array->cells != NULL);
  array->length= length;
}
```

Voici le finaliseur de `ArrayOfDouble` :

```
void ArrayOfDouble_Clean (ArrayOfDouble * array) { // finalizer
  free (array->cells);
}
```

Tableau encapsulé dans une structure (2/7)

Noter que si son champ `cells` est alloué **dynamiquement**,
la structure `ArrayOfDouble` qui l'encapsule
peut elle-même être allouée **automatiquement**.

Par exemple :

```
int main (void)
{
  ArrayOfDouble array; // automatic allocation
  ArrayOfDouble_Init (& array, 5); // initializer
  for (int k= 0; k < array.length; k++) array.cells[k]= k*k+ 0.5;
  for (int k= 0; k < array.length; k++) printf (" %f ", array.cells[k]);
  printf ("\n");
  ArrayOfDouble_Clean (& array); // finalizer
  return 0;
}
```

La sortie produite est :

```
0.5  1.5  4.5  9.5  16.5
```

Tableau encapsulé dans une structure (3/7)

On peut également allouer `ArrayOfDouble` dynamiquement.
Voici alors le constructeur et le destructeur nécessaires :

```
ArrayOfDouble * ArrayOfDouble_Create (int length) { // constructor
  ArrayOfDouble * array= malloc (sizeof * array);
  assert (array != NULL);
  ArrayOfDouble_Init (array, length);
  return array;
}
```

```
void ArrayOfDouble_Destroy (ArrayOfDouble * array) { // destructor
  ArrayOfDouble_Clean (array); // we must free array->cells first
  free (array);                // we must free array last
}
```

Exemple d'utilisation analogue au précédent :

```
int main (void) {
  ArrayOfDouble * array= ArrayOfDouble_Create (5); // constructor
  for (int k= 0; k < array->length; k++) array->cells[k]= k*k+ 0.5;
  for (int k= 0; k < array->length; k++) printf (" %f ", array->cells[k]);
  printf ("\n");
  ArrayOfDouble_Destroy (array); // destructor
  return 0;
}
```

Tableau encapsulé dans une structure (4/7)

Voici deux fonctions accédant aux cases directement avec `[]`.

Une 1ère en écriture, `FillWith()`, qui remplit le tableau :

```
void ArrayOfDouble_FillWith (ArrayOfDouble * array, double value) {
  for (int k= 0; k < array->length; k++) {
    array->cells[k]= value;
  }
}
```

Une 2nde en lecture, `Print()`, qui affiche le tableau sur un flux :

```
void ArrayOfDouble_Print (ArrayOfDouble const * array, FILE * file) {
  fprintf (file, "[ ");
  for (int k= 0; k < array->length; k++) {
    fprintf (file, "%f ", array->cells[k]);
  }
  fprintf (file, "]\n");
}
```

Exemple d'utilisation :

```
int main (void) {
  ArrayOfDouble * array= ArrayOfDouble_Create (5);
  ArrayOfDouble_FillWith (array, 0.0);
  ArrayOfDouble_Print (array, stdout);
  ArrayOfDouble_Destroy (array);
  return 0;
}
```

Tableau encapsulé dans une structure (5/7)

Forçons nous à accéder aux cases via une fonction d'adressage. `CellAt()` retourne l'adresse d'une case à un index donné.

```
double * ArrayOfDouble_CellAt (ArrayOfDouble const * array, int index) {
  assert (ArrayOfDouble_IndexIsValid (array, index));
  return array->cells + index;   // or:   return & array->cells[index];
}
```

Elle permet de s'assurer via l'assertion sur le prédicat suivant que l'index est valide, c-à-d, dans l'intervalle `[0..length-1]` :

```
bool ArrayOfDouble_IndexIsValid (ArrayOfDouble const * array, int index) {
  return 0 <= index && index < array->length;
}
```

On peut alors éventuellement écrire un **getter** et un **setter** :

```
double ArrayOfDouble_ValueAt (ArrayOfDouble const * array, int index) {
  double * cell= Array_CellAt (array, index);
  return * cell;
}
```

```
void ArrayOfDouble_SetValueAt (ArrayOfDouble * array, int index, double value) {
  double * cell= Array_CellAt (array, index);
  * cell= value;
}
```

Tableau encapsulé dans une structure (6/7)

La fonction de remplissage `FillWith()` initiale. . .

```
void ArrayOfDouble_FillWith (ArrayOfDouble * array, double value)
{
  for (int k= 0; k < array->length; k++)
     array->cells [k]= value;
}
```

devient, en utilisant la fonction d'adressage `CellAt()` :

```
void ArrayOfDouble_FillWith (ArrayOfDouble * array, double value)
{
  for (int k= 0; k < array->length; k++) {
    double * cell= ArrayOfDouble_CellAt (array, k);
    * cell= value;
  }
}
```

ou encore, en utilisant le setter `SetValueAt()` :

```
void ArrayOfDouble_FillWith (ArrayOfDouble * array, double value)
{
  for (int k= 0; k < array->length; k++)
    ArrayOfDouble_SetValueAt (array, k, value);
}
```

Tableau encapsulé dans une structure (7/7)

Enfin, la fonction d'affichage `Print()` initiale…

```
void Array_Print (ArrayOfDouble const * array, FILE * file) {
  fprintf (file, "[ ");
  for (int k= 0; k < array->length; k++)
    fprintf (file, "%f ", array->cells [k]);
  fprintf (file, "]\n");
}
```

devient, en utilisant la fonction d'adressage `CellAt()` :

```
void ArrayOfDouble_Print (ArrayOfDouble const * array, FILE * file) {
  fprintf (file, "[ ");
  for (int k= 0; k < array->length; k++) {
    double * cell= ArrayOfDouble_CellAt (array, k);
    fprintf (file, "%f ", * cell);
  }
  fprintf (file, "]\n");
}
```

ou encore, en utilisant le getter `ValueAt()` :

```
void ArrayOfDouble_Print (ArrayOfDouble const * array, FILE * file) {
  fprintf (file, "[ ");
  for (int k= 0; k < array->length; k++)
    fprintf (file, "%f ", ArrayOfDouble_ValueAt (array, k));
  fprintf (file, "]\n");
}
```

Arithm. de pointeur pour un tableau à 1 dimension

Rappelons que les crochets de tableau sont du sucre syntaxique cachant une arithmétique de pointeur d'adressage de case :

- `array[k]` $\Longleftrightarrow$ `* (array + k)`
- `& array[k]` $\Longleftrightarrow$ `array + k`

Remarque : la longueur du tableau n'intervient pas.
Voilà pourquoi on n'est pas obligé d'écrire...

```
#define LENGTH 10
void FillArrayWithZero (double array[LENGTH]) {
  for (int k= 0; k < LENGTH; k++)  array[k]= 0.0;
}
```

avec `LENGTH` connu à la compilation, mais que l'on peut écrire...

```
void FillArrayWithZero (double array[], int length) {
  for (int k= 0; k < length; k++)  array[k]= 0.0;
}
```

avec `length` inconnu à la compilation, puisque in fine on a :

```
void FillArrayWithZero (double * array, int length) {
  for (int k= 0; k < length; k++)  *(array + k)= 0.0;
}
```

Arithm. de pointeur pour un tableau à 2 dimensions (1/4)

La situation est différente avec les tableaux à 2 dimensions :

```
#define NB_ROWS 3
#define NB_COLS 5
void FillMatrixWithZero (double matrix [NB_ROWS][NB_COLS]) {
  for (int row= 0; row < NB_ROWS; row++)
    for (int col= 0; col < NB_COLS; col++)
       matrix [row][col]= 0.0;
}
```

Arithm. de pointeur pour un tableau à 2 dimensions (1/4)

La situation est différente avec les tableaux à 2 dimensions :

```
#define NB_ROWS 3
#define NB_COLS 5
void FillMatrixWithZero (double matrix [NB_ROWS][NB_COLS]) {
  for (int row= 0; row < NB_ROWS; row++)
    for (int col= 0; col < NB_COLS; col++)
       matrix [row][col]= 0.0;
}
```

Lorsque la case `matrix[row=2][col=3]` est adressée. . .

	[0]	[1]	[2]	[3]	[4]
[0]					
[1]					
[2]				0.0	

Arithm. de pointeur pour un tableau à 2 dimensions (1/4)

La situation est différente avec les tableaux à 2 dimensions :

```
#define NB_ROWS 3
#define NB_COLS 5
void FillMatrixWithZero (double matrix [NB_ROWS][NB_COLS]) {
  for (int row= 0; row < NB_ROWS; row++)
    for (int col= 0; col < NB_COLS; col++)
      matrix [row][col]= 0.0;
}
```

Lorsque la case `matrix[row=2][col=3]` est adressée...

	[0]	[1]	[2]	[3]	[4]
[0]					
[1]					
[2]				0.0	

c'est en fait la case d'index plat `13= (2 * 5) + 3` qui est adressée si l'on imagine le tableau comme étant plat, comme ci-dessous...

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[1][0]	[1][1]	[1][2]	[1][3]	[0][4]	[2][0]	[2][1]	[2][2]	[2][3]	[2][4]
													0.0	
[0*5]	[1]	[2]	[3]	[4]	[1*5]	[6]	[7]	[8]	[9]	[2*5]	[11]	[12]	[2*5+3]	[14]

Arithm. de pointeur pour un tableau à 2 dimensions (1/4)

La situation est différente avec les tableaux à 2 dimensions :

```
#define NB_ROWS 3
#define NB_COLS 5
void FillMatrixWithZero (double matrix [NB_ROWS][NB_COLS]) {
  for (int row= 0; row < NB_ROWS; row++)
    for (int col= 0; col < NB_COLS; col++)
      matrix [row][col]= 0.0;
}
```

Lorsque la case `matrix[row=2][col=3]` est adressée. . .

	[0]	[1]	[2]	[3]	[4]
[0]					
[1]					
[2]				0.0	

c'est en fait la case d'index plat `13= (2 * 5) + 3` qui est adressée si l'on imagine le tableau comme étant plat, comme ci-dessous. . .

[0] [0]	[0] [1]	[0] [2]	[0] [3]	[0] [4]	[1] [0]	[1] [1]	[1] [2]	[1] [3]	[0] [4]	[2] [0]	[2] [1]	[2] [2]	[2] [3]	[2] [4]
													0.0	
[0*5]	[1]	[2]	[3]	[4]	[1*5]	[6]	[7]	[8]	[9]	[2*5]	[11]	[12]	[2*5+3]	[14]

c'est-à-dire, la case d'index plat `(row * NB_COLS) + col`.

Arithm. de pointeur pour un tableau à 2 dimensions (2/4)

Pour adresser une case, `double * cell= & matrix[row][col]`, on a donc de façon sous-jacente l'arithmétique de pointeur :

```
cell= (double *) matrix + (row * NB_COLS) + col.
```

Arithm. de pointeur pour un tableau à 2 dimensions (2/4)

Pour adresser une case, `double * cell= & matrix[row][col]`, on a donc de façon sous-jacente l'arithmétique de pointeur :

```
cell= (double *) matrix + (row * NB_COLS) + col.
```

Le nombre de colonnes `NB_COLS` (c-à-d, la taille d'une ligne) intervient dans le calcul, mais pas le nombre de lignes `NB_ROWS`.

Arithm. de pointeur pour un tableau à 2 dimensions (2/4)

Pour adresser une case, `double * cell= & matrix[row][col]`,
on a donc de façon sous-jacente l'arithmétique de pointeur :

`cell= (double *) matrix + (row * NB_COLS) + col.`

Le nombre de colonnes `NB_COLS` (c-à-d, la taille d'une ligne) intervient dans le calcul, mais pas le nombre de lignes `NB_ROWS`.

On peut donc éliminer `NB_ROWS` dans `FillMatrixWithZero()` :

```
void FillMatrixWithZero (double matrix [][NB_COLS], int nb_rows) {
  for (int row= 0; row < nb_rows; row++)
    for (int col= 0; col < NB_COLS; col++)
       matrix [row][col]= 0.0;
}
```

Arithm. de pointeur pour un tableau à 2 dimensions (2/4)

Pour adresser une case, `double * cell= & matrix[row][col]`, on a donc de façon sous-jacente l'arithmétique de pointeur :

`cell= (double *) matrix + (row * NB_COLS) + col.`

Le nombre de colonnes `NB_COLS` (c-à-d, la taille d'une ligne) intervient dans le calcul, mais pas le nombre de lignes `NB_ROWS`.

On peut donc éliminer `NB_ROWS` dans `FillMatrixWithZero()` :

```
void FillMatrixWithZero (double matrix [][NB_COLS], int nb_rows) {
  for (int row= 0; row < nb_rows; row++)
    for (int col= 0; col < NB_COLS; col++)
       matrix [row][col]= 0.0;
}
```

En revanche, on ne peut pas éliminer `NB_COLS` comme suit :

```
// impossible, does not compile !
void FillMatrixWithZero (double matrix [][], int nb_rows, int nb_cols) {
  for (int row= 0; row < nb_rows; row++)
    for (int col= 0; col < nb_cols; col++)
       matrix [row][col]= 0.0;
}
```

Arithm. de pointeur pour un tableau à 2 dimensions (3/4)

En fait, le compilateur voit le paramètre formel `matrix` comme un pointeur sur la première ligne de la matrice.

Introduisons l'alias `MatrixRow` pour le type *ligne de 5 flottants* :

```
#define NB_COLS 5
typedef double MatrixRow [NB_COLS];
```

On peut alors réexprimer `FillMatrixWithZero()` comme...

```
void FillMatrixWithZero (MatrixRow * matrix, int nb_rows) {
  for (int row= 0; row < nb_rows; row++)
    for (int col= 0; col < NB_COLS; col++)
       matrix[row][col]= 0.0;
}
```

Ou encore, en se passant de `typedef`, obtenir l'infâme...

```
void FillMatrixWithZero (double (* matrix) [NB_COLS], int nb_rows) {
  for (int row= 0; row < nb_rows; row++)
    for (int col= 0; col < NB_COLS; col++)
      matrix[row][col]= 0.0;
}
```

que l'on voit parfois dans les diagnostics d'erreur de compilation.

Arithm. de pointeur pour un tableau à 2 dimensions (4/4)

Curiosité peu connue du C99, on peut spécifier des variables dans les crochets d'un paramètre tableau d'une fonction, à condition qu'elles en soient elles-mêmes des paramètres :

```
// legal in C99
void FillMatrixWithZero (int nb_rows, int nb_cols, double matrix [nb_rows][nb_cols]) {
  for (int row= 0; row < nb_rows; row++)
    for (int col= 0; col < nb_cols; col++)
       matrix [row][col]= 0.0;
}
```

De façon curieuse, elles doivent aussi obligatoirement se trouver devant le paramètre tableau, et jamais après.

Le code suivant ne compile donc pas :

```
// illegal in C99, does not compile
void FillMatrixWithZero (double matrix [nb_rows][nb_cols], int nb_rows, int nb_cols) {
  for (int row= 0; row < nb_rows; row++)
    for (int col= 0; col < nb_cols; col++)
       matrix [row][col]= 0.0;
}
```

3 modèles pour l'alloc dynamique de tableau 2D

Voici 3 façons d'allouer dynamiquement un tableau 2D
à `nb_rows` lignes et `nb_cols` colonnes :

3 modèles pour l'alloc dynamique de tableau 2D

Voici 3 façons d'allouer dynamiquement un tableau 2D
à `nb_rows` lignes et `nb_cols` colonnes :

- **le modèle plat :** on alloue un seul tableau unidimensionnel de longueur `nb_rows * nb_cols`, et on passe par une fonction d'adressage effectuant l'arithmétique de pointeur nécéssaire pour convertir les indices `row, col` en une adresse de case.

3 modèles pour l'alloc dynamique de tableau 2D

Voici 3 façons d'allouer dynamiquement un tableau 2D
à `nb_rows` lignes et `nb_cols` colonnes :

- **le modèle plat :** on alloue un seul tableau unidimensionnel de longueur `nb_rows * nb_cols`, et on passe par une fonction d'adressage effectuant l'arithmétique de pointeur nécéssaire pour convertir les indices `row, col` en une adresse de case.

- **le modèle en strates :** on alloue un tableau par ligne, c-à-d, `nb_rows` tableaux de longueur `nb_cols`, ainsi qu'un tableau auxiliaire de longueur `nb_rows` pour accéder à ces lignes.

3 modèles pour l'alloc dynamique de tableau 2D

Voici 3 façons d'allouer dynamiquement un tableau 2D
à `nb_rows` lignes et `nb_cols` colonnes :

- **le modèle plat :** on alloue un seul tableau unidimensionnel de longueur `nb_rows * nb_cols`, et on passe par une fonction d'adressage effectuant l'arithmétique de pointeur nécéssaire pour convertir les indices `row, col` en une adresse de case.

- **le modèle en strates :** on alloue un tableau par ligne, c-à-d, `nb_rows` tableaux de longueur `nb_cols`, ainsi qu'un tableau auxiliaire de longueur `nb_rows` pour accéder à ces lignes.

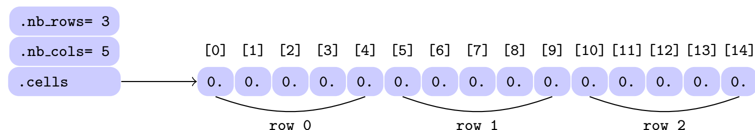
- **le modèle hybride des deux autres :** on alloue 2 tableaux, un 1er de longueur `nb_rows * nb_cols` pour les cases, et un 2nd de longueur `nb_rows` pour accéder aux lignes dans le 1er.

Le modèle plat les tableaux 2D (1/2)

On va coder le modèle plat avec la structure `FlatGrid` :

```
typedef struct FlatGrid {
  double * cells;
  int nb_rows, nb_cols;
} FlatGrid;
```

Son modèle peut être schématisé comme suit :



Dans une `FlatGrid grid`, la case d'indices `(row, col)` correspond à une case `grid.cells [flat_index]`, où l'index plat `flat_index= row * grid.nb_cols + col`.

Le modèle plat pour les tableaux 2D (2/2)

L'initialiseur est simple, avec un seul appel à `malloc()` :

```
void FlatGrid_Init (FlatGrid * grid, int nb_rows, int nb_cols) {
  int nb_cells= nb_rows * nb_cols;
  grid->cells= malloc (nb_cells * sizeof * grid->cells);
  grid->nb_rows= nb_rows;
  grid->nb_cols= nb_cols;
}
```

Le finaliseur est simple, avec un seul appel à `free()` :

```
void FlatGrid_Clean (FlatGrid * grid) {
  free (grid->cells);
}
```

L'adressage des cases est plus compliquée (arithm. de pointeur) :

```
int FlatGrid_Index (FlatGrid const * grid, int row, int col) {
  return grid->nb_cols * row + col;
}
```

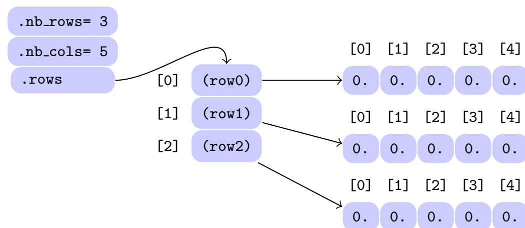
```
double * FlatGrid_CellAt (FlatGrid const * grid, int row, int col) {
  assert (0 <= row && row < grid->nb_rows);
  assert (0 <= col && col < grid->nb_cols);
  int index= FlatGrid_Index (grid, row, col);
  return grid->cells + index;
}
```

Le modèle en strates pour les tableaux 2D (1/2)

On va coder le modèle en strates avec la structure `LayeredGrid` :

```
typedef struct LayeredGrid {
  double ** rows;
  int nb_rows, nb_cols;
} LayeredGrid;
```

Son modèle peut être schématisé comme suit :



Dans une `LayeredGrid grid`, la case d'indices `(row, col)` correspond à la case `grid.rows [row][col]`.

Le modèle en strates pour les tableaux 2D (2/2)

L'initialiseur est compliqué avec `nb_rows+1` appels à `malloc()` :

```
void LayeredGrid_Init (LayeredGrid * grid, int nb_rows, int nb_cols) {
  grid->rows= malloc (nb_rows * sizeof * grid->rows);
  assert (grid->rows != NULL);
  for (int k= 0; k < nb_rows; k++) {
    grid->rows [k]= malloc (nb_cols * sizeof * grid->rows [k]);
    assert (grid->rows [k] != NULL);
  }
  grid->nb_rows= nb_rows;
  grid->nb_cols= nb_cols;
}
```

Le finaliseur est compliqué avec `nb_rows+1` appels à `free()` :

```
void LayeredGrid_Clean (LayeredGrid * grid) {
  for (int k= 0; k < grid->nb_rows; k++)
    free (grid->rows [k]);
  free (grid->rows); // must be done last
}
```

La fonction d'adressage est simple (utilisation des crochets) :

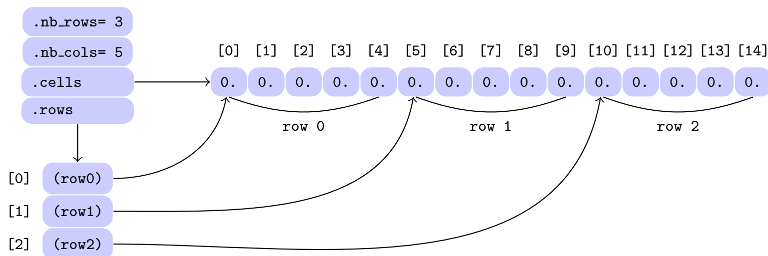
```
double * LayeredGrid_CellAt (LayeredGrid * grid, int row, int col) {
  assert (0 <= row && row < grid->nb_rows);
  assert (0 <= col && col < grid->nb_cols);
  return & grid->rows[row][col];
}
```

Le modèle hybride pour les tableaux 2D (1/2)

On va coder le modèle hybride avec la structure `HybridGrid` :

```
typedef struct HybriddGrid {
  double * cells;
  double ** rows;
  int nb_rows, nb_cols;
} HybridGrid;
```

Son modèle peut être schématisé comme suit :



Dans une `HybridGrid grid`, la case d'indices `(row, col)` correspond à la case `grid.rows [row][col]`.

Le modèle hybride pour les tableaux 2D (2/2)

L'initialiseur est compliqué, avec deux appels à `malloc()`, mais aussi `nb_rows` fléchages à brancher :

```
void HybridGrid_Init (HybridGrid * grid, int nb_rows, int nb_cols) {
  int nb_cells= nb_rows * nb_cols;
  grid->cells= malloc (nb_cells * sizeof * grid->cells);
  grid->rows = malloc (nb_rows  * sizeof * grid->rows );
  assert (grid->cells != NULL  &&  grid->rows  != NULL);
  for (int k= 0; k < nb_rows; k++)
    grid->rows [k]= grid->cells + k * nb_cols;
  grid->nb_rows= nb_rows;
  grid->nb_cols= nb_cols;
}
```

Le finaliseur est simple, avec seulement deux appels à `free()` :

```
void HybridGrid_Clean (LayeredGrid * grid) {
  free (grid->cells);
  free (grid->rows);
}
```

La fonction d'adressage est simple (utilisation des crochets) :

```
double * HybridGrid_CellAt (LayeredGrid const * grid, int row, int col) {
  assert (0 <= row && row < grid->nb_rows);
  assert (0 <= col && col < grid->nb_cols);
  return & grid->rows [row][col];
}
```

Utilisation des 3 modèles de tableau 2D

Les 3 modèles s'utilisent de façon identique :

```
int main (void) {
  FlatGrid grid;
  FlatGrid_Init (& grid, 3, 5);
  double * cell= FlatGrid_CellAt (& grid, 2, 3)
  * cell= 0.0;
  FlatGrid_Clean (& grid);
  return 0;
}
```

```
int main (void) {
  LayeredGrid grid;
  LayeredGrid_Init (& grid, 3, 5);
  double * cell= LayeredGrid_CellAt (& grid, 2, 3);
  * cell= 0.0;
  LayeredGrid_Clean (& grid);
  return 0;
}
```

Enfin, on peut déréférencer le retour de la fonction d'adressage sans passer par une variable intermédiaire `cell` :

```
int main (void) {
  HybridGrid grid;
  HybridGrid_Init (& grid, 3, 5);
  * HybridGrid_CellAt (& grid, 2, 3)= 0.0;
  HybridGrid_Clean (& grid);
  return 0;
}
```