

Cours 11

Tableaux à deux dimensions (ou plus)
Fonctions d'algèbre linéaire

Vincent Lepetit

`vincent.lepetit@epfl.ch`

Les tableaux à deux dimensions

Les éléments des tableaux que nous avons vus jusqu'ici sont repérés par un seul indice:

```
int tab[3];
```

Ces tableaux sont des tableaux à une dimension.

| |
|--------|
| tab[0] |
| tab[1] |
| tab[2] |

On peut également définir des tableaux à deux dimensions, où chaque élément est repéré par 2 indices:

```
int tab[3][2];
```

| | |
|-----------|-----------|
| tab[0][0] | tab[0][1] |
| tab[1][0] | tab[1][1] |
| tab[2][0] | tab[2][1] |

Déclaration et Manipulation des éléments

Exemple de déclaration:

```
int x[2][3];
```

- le premier indice varie entre 0 et 1
- le deuxième indice varie entre 0 et 2.

Utilisation:

```
x[0][0] = 5;  
x[0][1] = 12;  
x[0][2] = 2;  
x[1][0] = 7;  
x[1][1] = 3;  
x[1][2] = 9;
```

```
x[0][0] = x[1][0] + x[0][1];
```

0 et 1

| | | | |
|----------|---|----|---|
| x | 0 | 1 | 2 |
| 0 | 5 | 12 | 2 |
| 1 | 7 | 3 | 9 |

Manipulations d'un tableau à deux dimensions

Soit le tableau x déclaré de la façon suivante:

```
const int NL = 10;  
const int NC = 20;  
int x[NL][NC];
```

| x | 0 | 1 | 2 | ... | NC-1 |
|----------|---|---|---|-----|------|
| 0 | ? | ? | ? | | ? |
| 1 | ? | ? | ? | | ? |
| ... | | | | | |
| NL-1 | ? | ? | ? | | ? |

Comment

- initialiser tous les éléments du tableau à 0 ?
- demander tous les éléments du tableau à l'utilisateur ?
- afficher le tableau ?

Initialiser les éléments à 0

Soit le tableau `x` déclaré de la façon suivante:

```
const int NL = 10;  
const int NC = 20;  
int x[NL][NC];
```

Pour initialiser tous les éléments du tableau `x`, il faut utiliser deux boucles `for` imbriquées:

```
for(int i = 0; i < NL; i++)  
    for(int j = 0; j < NC; j++)  
        x[i][j] = 0;
```

Remarque

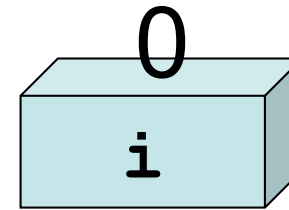
```
for(int i = 0; i < NL; i++)  
    for(int j = 0; j < NC; j++)  
        x[i][j] = 0;
```

s'interprète comme:

```
for(int i = 0; i < NL; i++)  
{  
    for(int j = 0; j < NC; j++)  
        x[i][j] = 0;  
}
```

Ces deux lignes constituent une seule instruction et les accolades ne sont pas nécessaires.

Pas-à-pas

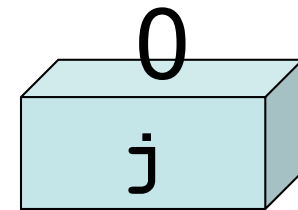
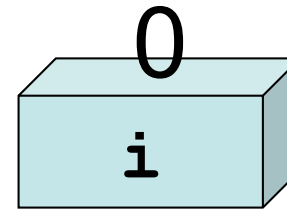


→ `for(int i = 0; i < NL; i++)`
 `for(int j = 0; j < NC; j++)`
 `x[i][j] = 0;`

| x | 0 | 1 | 2 | ... | NC-1 |
|----------|---|---|---|-----|------|
| 0 | ? | ? | ? | | ? |
| 1 | ? | ? | ? | | ? |
| ... | | | | | |
| NL-1 | ? | ? | ? | | ? |

Pas-à-pas

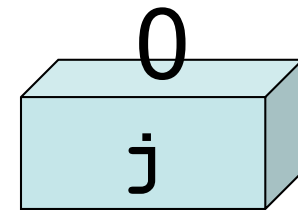
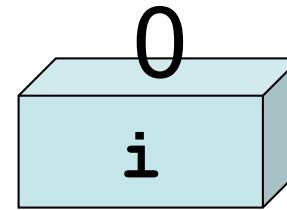
```
for(int i = 0; i < NL; i++)  
→ for(int j = 0; j < NC; j++)  
    x[i][j] = 0;
```



| x | 0 | 1 | 2 | ... | NC-1 |
|----------|---|---|---|-----|------|
| 0 | ? | ? | ? | | ? |
| 1 | ? | ? | ? | | ? |
| ... | | | | | |
| NL-1 | ? | ? | ? | | ? |

Pas-à-pas

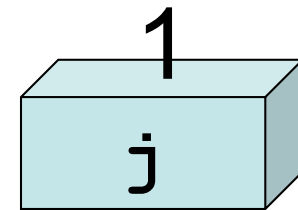
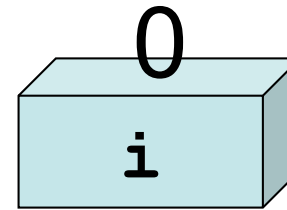
```
for(int i = 0; i < NL; i++)  
  for(int j = 0; j < NC; j++)  
    → x[i][j] = 0;
```



| x | 0 | 1 | 2 | ... | NC-1 |
|----------|----------|---|---|-----|------|
| 0 | 0 | ? | ? | | ? |
| 1 | ? | ? | ? | | ? |
| ... | | | | | |
| NL-1 | ? | ? | ? | | ? |

Pas-à-pas

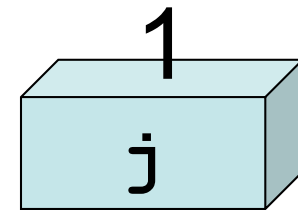
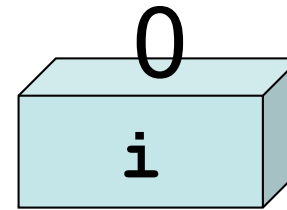
```
for(int i = 0; i < NL; i++)  
→ for(int j = 0; j < NC; j++)  
    x[i][j] = 0;
```



| x | 0 | 1 | 2 | ... | NC-1 |
|----------|---|---|---|-----|------|
| 0 | 0 | ? | ? | | ? |
| 1 | ? | ? | ? | | ? |
| ... | | | | | |
| NL-1 | ? | ? | ? | | ? |

Pas-à-pas

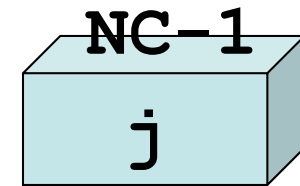
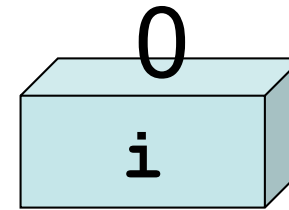
```
for(int i = 0; i < NL; i++)  
  for(int j = 0; j < NC; j++)  
    → x[i][j] = 0;
```



| x | 0 | 1 | 2 | ... | NC-1 |
|----------|---|---|---|-----|------|
| 0 | 0 | 0 | ? | | ? |
| 1 | ? | ? | ? | | ? |
| ... | | | | | |
| NL-1 | ? | ? | ? | | ? |

Pas-à-pas

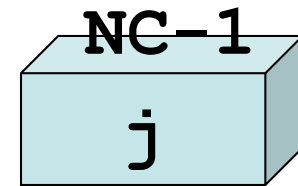
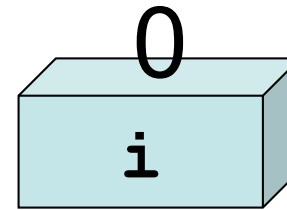
```
for(int i = 0; i < NL; i++)  
→ for(int j = 0; j < NC; j++)  
    x[i][j] = 0;
```



| x | 0 | 1 | 2 | ... | NC-1 |
|----------|---|---|---|-----|------|
| 0 | 0 | 0 | 0 | | ? |
| 1 | ? | ? | ? | | ? |
| ... | | | | | |
| NL-1 | ? | ? | ? | | ? |

Pas-à-pas

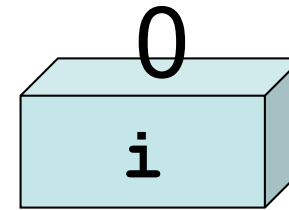

```
for(int i = 0; i < NL; i++)  
  for(int j = 0; j < NC; j++)  
    → x[i][j] = 0;
```



| x | 0 | 1 | 2 | ... | NC-1 |
|----------|---|---|---|-----|------|
| 0 | 0 | 0 | 0 | | 0 |
| 1 | ? | ? | ? | | ? |
| ... | | | | | |
| NL-1 | ? | ? | ? | | ? |

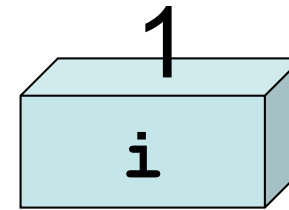
Pas-à-pas

```
for(int i = 0; i < NL; i++)  
  for(int j = 0; j < NC; j++)  
    x[i][j] = 0;
```



| x | 0 | 1 | 2 | ... | NC-1 |
|----------|---|---|---|-----|------|
| 0 | 0 | 0 | 0 | | 0 |
| 1 | ? | ? | ? | | ? |
| ... | | | | | |
| NL-1 | ? | ? | ? | | ? |

Pas-à-pas

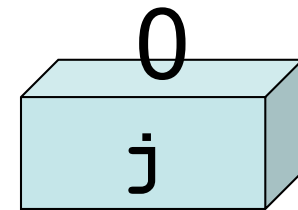
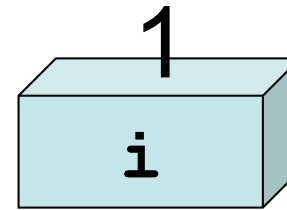


→ `for(int i = 0; i < NL; i++)`
 `for(int j = 0; j < NC; j++)`
 `x[i][j] = 0;`

| x | 0 | 1 | 2 | ... | NC-1 |
|----------|---|---|---|-----|------|
| 0 | 0 | 0 | 0 | | 0 |
| 1 | ? | ? | ? | | ? |
| ... | | | | | |
| NL-1 | ? | ? | ? | | ? |

Pas-à-pas

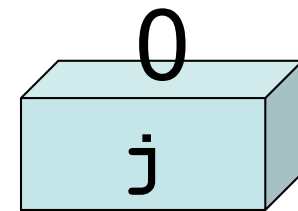
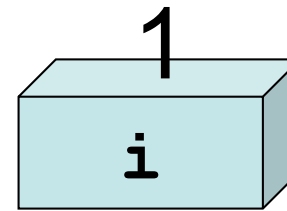
```
for(int i = 0; i < NL; i++)  
→ for(int j = 0; j < NC; j++)  
    x[i][j] = 0;
```



| x | 0 | 1 | 2 | ... | NC-1 |
|----------|---|---|---|-----|------|
| 0 | 0 | 0 | 0 | | 0 |
| 1 | ? | ? | ? | | ? |
| ... | | | | | |
| NL-1 | ? | ? | ? | | ? |

Pas-à-pas

```
for(int i = 0; i < NL; i++)  
  for(int j = 0; j < NC; j++)  
    → x[i][j] = 0;
```



| x | 0 | 1 | 2 | ... | NC-1 |
|----------|---|---|---|-----|------|
| 0 | 0 | 0 | 0 | | 0 |
| 1 | 0 | ? | ? | | ? |
| ... | | | | | |
| NL-1 | ? | ? | ? | | ? |

Affichage: Raisonner sur un exemple

Considérons le cas particulier où `NL` vaut 3, `NC` vaut 2:

```
int x[3][2];
```

Si l'on fait:

```
for(int i = 0; i < 3; i++)  
    for(int j = 0; j < 2; j++)  
        cout << x[i][j] << endl;
```

le tableau sera affiché sur une seule longue colonne:

```
8  
3  
9  
7  
12  
5
```

| | | |
|----------|----------|----------|
| x | 0 | 1 |
| 0 | 8 | 3 |
| 1 | 9 | 7 |
| 2 | 12 | 5 |

Raisonner sur un exemple

Si on veut afficher le tableau `x[3][2]` avec 3 lignes et 2 colonnes:

```
8 3
9 7
12 5
```

on pourrait l'afficher sans utiliser de boucles en écrivant:

```
cout << x[0][0] << " " << x[0][1] << endl;
cout << x[1][0] << " " << x[1][1] << endl;
cout << x[2][0] << " " << x[2][1] << endl;
```

Afficher une ligne avec une boucle

L'affichage d'une ligne

```
cout << x[0][0] << " " << x[0][1] << endl;
```

peut se décomposer en:

```
cout << x[0][0] << " ";  
cout << x[0][1] << " ";  
cout << endl;
```

qui peut s'écrire avec une boucle `for`:

```
for(int j = 0; j < 2; j++)  
    cout << x[0][j] << " ";  
cout << endl;
```

L'instruction

```
cout << endl;
```

n'est pas contenue dans la boucle.

Elle est exécutée après.

Afficher plusieurs lignes

Pour afficher toutes les lignes, il suffit d'inclure le code qui affiche une ligne:

```
for(int j = 0; j < 2; j++)  
    cout << x[0][j] << " ";  
cout << endl;
```

dans une seconde boucle `for`.

Le 0 correspondait au numéro de la ligne à afficher, il faut donc le remplacer par la variable compteur de cette seconde boucle `for`, ici `i`:

```
for(int i = 0; i < 3; i++)  
{  
    for(int j = 0; j < 2; j++)  
        cout << x[i][j] << " ";  
    cout << endl;  
}
```

Remarque

Cette fois, les accolades sont obligatoires.

Il y a deux instructions dans la boucle `for(int i... :`

```
for(int i = 0; i < 3; i++)
```

```
{
```

```
    for(int j = 0; j < 2; j++)
```

```
        cout << x[i][j] << " ";
```

```
    cout << endl;
```

```
}
```

} Une instruction

} Une deuxième instruction

Généraliser

Le cas particulier est valable pour $NL = 3$ et $NC = 2$:

```
for(int i = 0; i < 3; i++)
{
    for(int j = 0; j < 2; j++)
        cout << x[i][j] << " ";
    cout << endl;
}
```

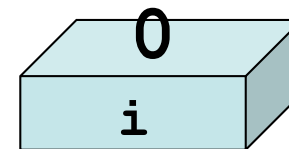
Le cas général est facile à écrire maintenant:

```
for(int i = 0; i < NL; i++)
{
    for(int j = 0; j < NC; j++)
        cout << x[i][j] << " ";
    cout << endl;
}
```

Pas-à-pas

```
→ for(int i = 0; i < 3; i++)  
  {  
    for(int j = 0; j < 2; j++)  
      cout << x[i][j] << " ";  
    cout << endl;  
  }
```

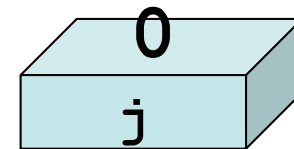
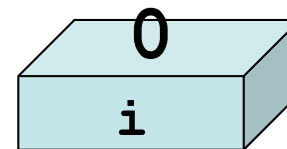
| | | |
|----------|----------|----------|
| x | 0 | 1 |
| 0 | 8 | 3 |
| 1 | 9 | 7 |
| 2 | 12 | 5 |



Pas-à-pas

```
for(int i = 0; i < 3; i++)  
{  
→ for(int j = 0; j < 2; j++)  
    cout << x[i][j] << " ";  
    cout << endl;  
}
```

| x | 0 | 1 |
|----------|----------|----------|
| 0 | 8 | 3 |
| 1 | 9 | 7 |
| 2 | 12 | 5 |

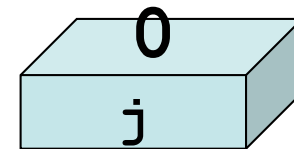
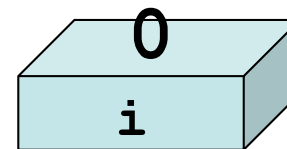


Pas-à-pas

```
for(int i = 0; i < 3; i++)  
{  
    for(int j = 0; j < 2; j++)  
→ cout << x[i][j] << " ";  
    cout << endl;  
}
```

| | | |
|----------|----------|----------|
| x | 0 | 1 |
| 0 | 8 | 3 |
| 1 | 9 | 7 |
| 2 | 12 | 5 |

8 ◆

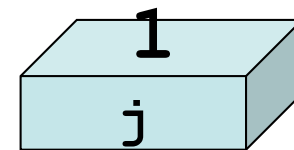
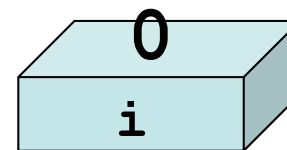


Pas-à-pas

```
for(int i = 0; i < 3; i++)  
{  
→ for(int j = 0; j < 2; j++)  
    cout << x[i][j] << " ";  
    cout << endl;  
}
```

| x | 0 | 1 |
|----------|----------|----------|
| 0 | 8 | 3 |
| 1 | 9 | 7 |
| 2 | 12 | 5 |

8 ◆

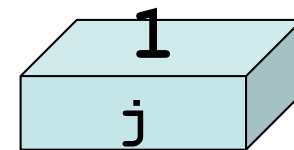
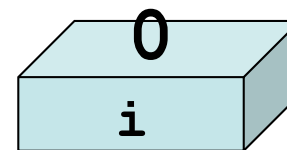


Pas-à-pas

```
for(int i = 0; i < 3; i++)  
{  
→ for(int j = 0; j < 2; j++)  
    cout << x[i][j] << " ";  
    cout << endl;  
}
```

| x | 0 | 1 |
|----------|----------|----------|
| 0 | 8 | 3 |
| 1 | 9 | 7 |
| 2 | 12 | 5 |

8 ◆

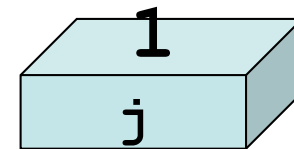
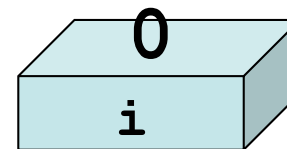


Pas-à-pas

```
for(int i = 0; i < 3; i++)  
{  
    for(int j = 0; j < 2; j++)  
→ cout << x[i][j] << " ";  
    cout << endl;  
}
```

| | | |
|----------|----------|----------|
| x | 0 | 1 |
| 0 | 8 | 3 |
| 1 | 9 | 7 |
| 2 | 12 | 5 |

```
8 3 ◆
```

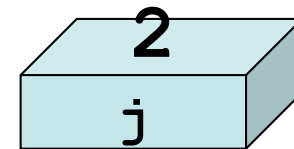
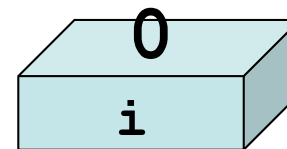


Pas-à-pas

```
for(int i = 0; i < 3; i++)  
{  
→ for(int j = 0; j < 2; j++)  
    cout << x[i][j] << " ";  
    cout << endl;  
}
```

| | | |
|----------|----------|----------|
| x | 0 | 1 |
| 0 | 8 | 3 |
| 1 | 9 | 7 |
| 2 | 12 | 5 |

```
8 3 ◆
```

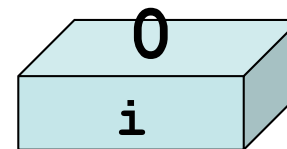


Pas-à-pas

```
for(int i = 0; i < 3; i++)  
{  
    for(int j = 0; j < 2; j++)  
        cout << x[i][j] << " ";  
    → cout << endl;  
}
```

| x | 0 | 1 |
|----------|----------|----------|
| 0 | 8 | 3 |
| 1 | 9 | 7 |
| 2 | 12 | 5 |

```
8 3  
◆
```

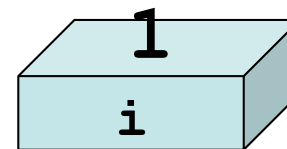


Pas-à-pas

```
→ for(int i = 0; i < 3; i++)  
  {  
    for(int j = 0; j < 2; j++)  
      cout << x[i][j] << " ";  
    cout << endl;  
  }
```

| | | |
|----------|----------|----------|
| x | 0 | 1 |
| 0 | 8 | 3 |
| 1 | 9 | 7 |
| 2 | 12 | 5 |

```
8 3  
◆
```

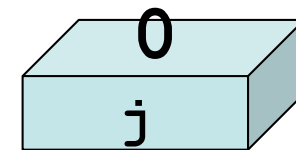
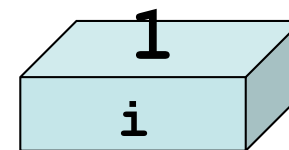


Pas-à-pas

```
for(int i = 0; i < 3; i++)  
{  
→ for(int j = 0; j < 2; j++)  
    cout << x[i][j] << " ";  
    cout << endl;  
}
```

| x | 0 | 1 |
|----------|----------|----------|
| 0 | 8 | 3 |
| 1 | 9 | 7 |
| 2 | 12 | 5 |

```
8 3  
◆
```

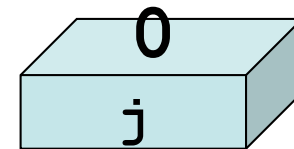
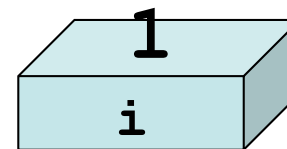


Pas-à-pas

```
for(int i = 0; i < 3; i++)  
{  
    for(int j = 0; j < 2; j++)  
→ cout << x[i][j] << " ";  
    cout << endl;  
}
```

| x | 0 | 1 |
|----------|----------|----------|
| 0 | 8 | 3 |
| 1 | 9 | 7 |
| 2 | 12 | 5 |

```
8 3  
9 ◆
```



Pas-à-pas

```
for(int i = 0; i < 3; i++)  
{  
    for(int j = 0; j < 2; j++)  
        cout << x[i][j] << " ";  
    cout << endl;  
}
```

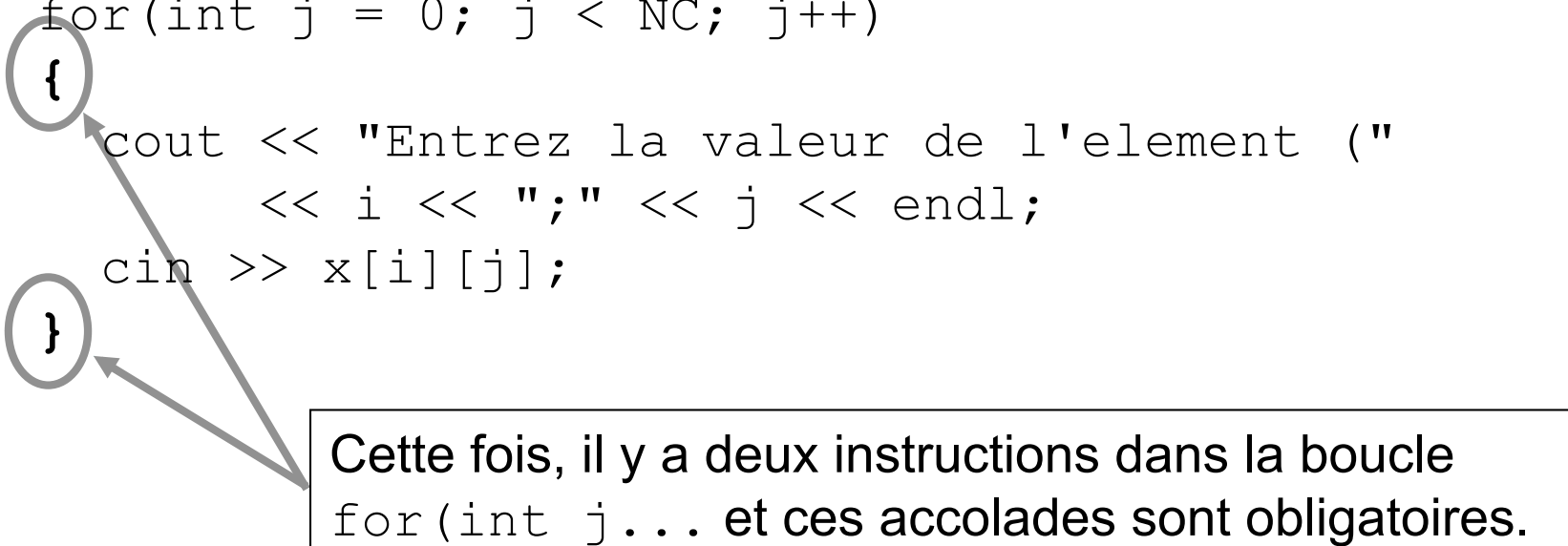


| | | |
|----------|----------|----------|
| x | 0 | 1 |
| 0 | 8 | 3 |
| 1 | 9 | 7 |
| 2 | 12 | 5 |

```
8 3  
9 7  
12 5  
◆
```

Lecture

```
for(int i = 0; i < NL; i++)  
    for(int j = 0; j < NC; j++)  
    {  
        cout << "Entrez la valeur de l'element ("  
            << i << ";" << j << endl;  
        cin >> x[i][j];  
    }
```



Cette fois, il y a deux instructions dans la boucle
for(int j... et ces accolades sont obligatoires.

Exécution:

```
Entrez la valeur de l'element (0,0):  
8  
Entrez la valeur de l'element (0,1):  
9  
Entrez la valeur de l'element (0,2):  
12  
Entrez la valeur de l'element (1,0):  
3  
...
```

Tableaux à 2 dimensions et fonctions

Une fonction peut aussi considérer des tableaux à plusieurs dimensions:

```
void mise_a_zero_2_dims(int x[3][2])
{
    for(int i = 0; i < 3; i++)
        for(int j = 0; j < 2; j++)
            x[i][j] = 0;
}
```

Cette fonction peut alors être utilisée de la façon suivante:

```
int t1[3][2], t2[3][2];
...
mise_a_zero_2_dims(t1);
mise_a_zero_2_dims(t2);
...
```

ICI:

- t1 ET t2 DOIVENT AVOIR LA MEME TAILLE QUE t;
- ON NE PEUT PAS UTILISER DE POINTEURS.

Exercice

Écrire la fonction d'en-tête:

```
void triangle(char T[N][N])
```

qui initialise le tableau de caractères passé en paramètre à:

```
+++++++  
.+.....+  
..+.....+  
...+.....+  
....+.....+  
.....+..+  
.....+.+  
.....++  
.....+
```

Écriture de fonctions d'algèbre linéaire

Fonctions d'algèbre linéaire

Nous allons écrire plusieurs fonctions permettant la manipulation de vecteurs et de matrices, comme l'addition ou la multiplication de matrices.

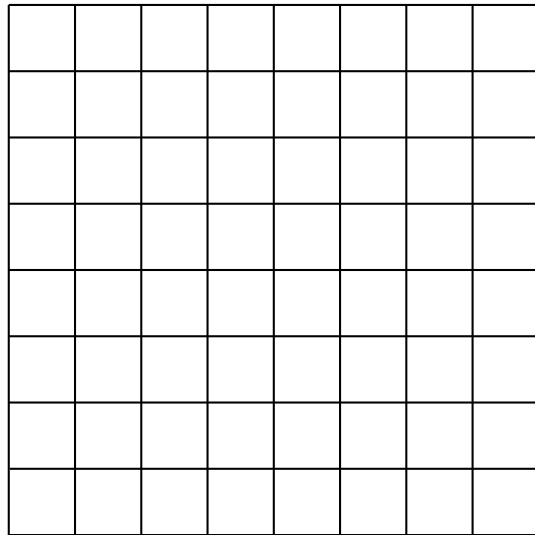
Il faut d'abord décider d'une façon de représenter une matrice ou un vecteur dans le programme C.

Première représentation possible

Pour représenter les matrices, on peut utiliser des tableaux à 2 dimensions.

```
const int dim_max = 50;
```

```
float M[dim_max][dim_max];
```



M pour `dim_max = 8`

Première représentation possible

Pour représenter les matrices, on peut utiliser des tableaux à 2 dimensions.

```
const int dim_max = 50;
```

```
float M[dim_max][dim_max];
```

Par exemple, la fonction permettant d'additionner deux matrices aurait l'en-tête suivant:

```
void add_mat(float A[dim_max][dim_max],  
            float B[dim_max][dim_max],  
            float Res[dim_max][dim_max]);
```

Problème: nous ne voulons pas nous limiter aux matrices de taille `dim_max`, mais aussi pouvoir considérer également des matrices plus petites.

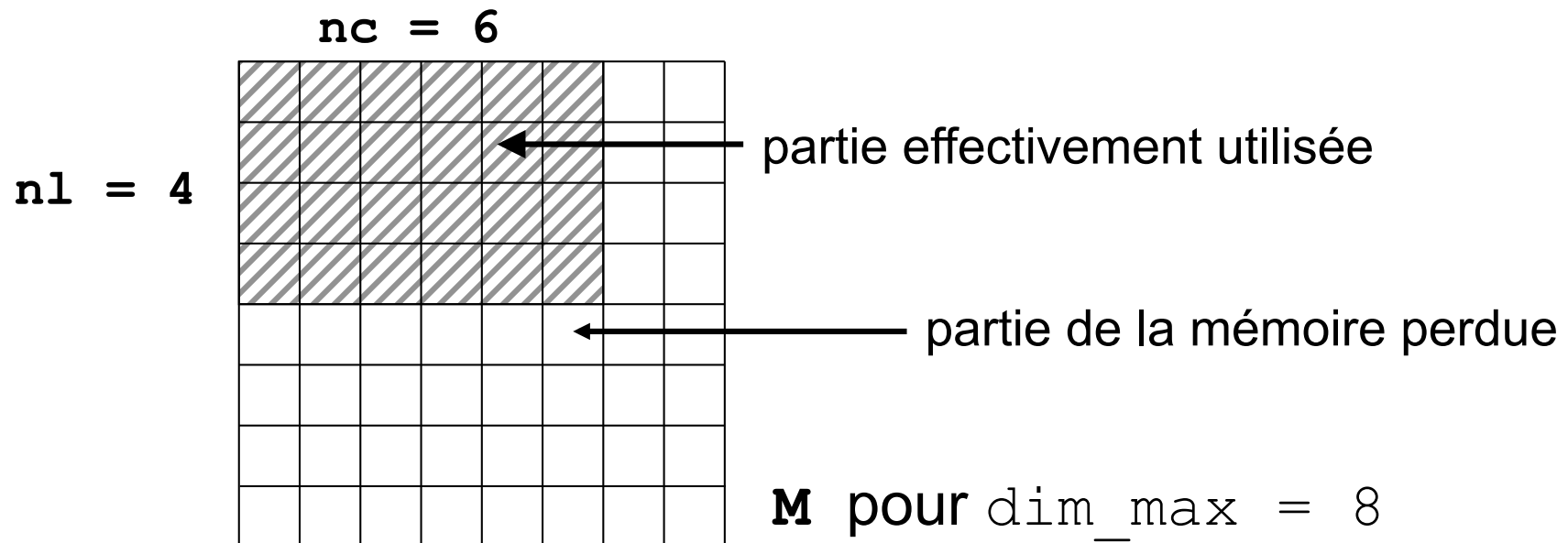
Considérer des matrices plus petites

Il faut ajouter aux paramètres de `add_mat` le nombre de lignes et le nombre de colonnes des matrices:

```
void add_mat(float A[dim_max][dim_max],  
            float B[dim_max][dim_max],  
            float Res[dim_max][dim_max],  
            int nl, int nc)
```

(`nl` : nombre de lignes; `nc` : nombre de colonnes)

Donc seule une partie des tableaux serait utilisée pour stocker les matrices:



Type Matrice

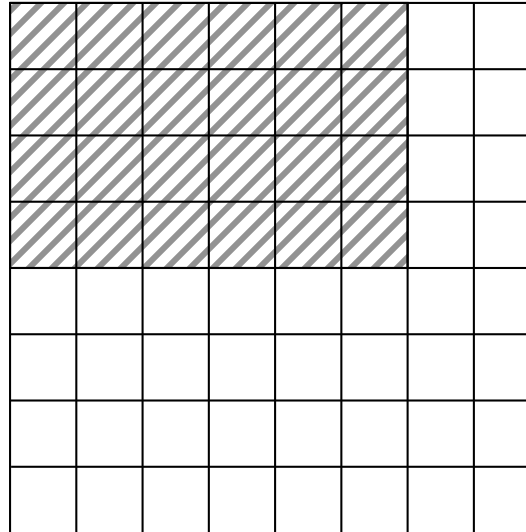
Il est possible d'utiliser les tableaux 2D du C pour définir des matrices, mais cette solution a des inconvénients.

Nous allons créer notre propre type de tableau 2D permettant de définir des matrices de différentes tailles.

Nous allons définir une structure `Matrice` qui contiendra:

- un pointeur sur une zone mémoire allouée dynamiquement, contenant les éléments de la matrice;
- le nombre de lignes et le nombre de colonnes de la matrice.

Allocation dynamique



Pour éviter d'utiliser inutilement de la mémoire, on peut utiliser une allocation dynamique pour allouer les tableaux.

(Allocation dynamique → permet de définir la taille de la matrice au moment de l'*exécution* du programme)

L'allocation dynamique se fait en utilisant `new`.

Comment allouer dynamiquement un tableau à 2 dimensions ?

`new` ne permet d'allouer qu'un emplacement contigu de la mémoire.

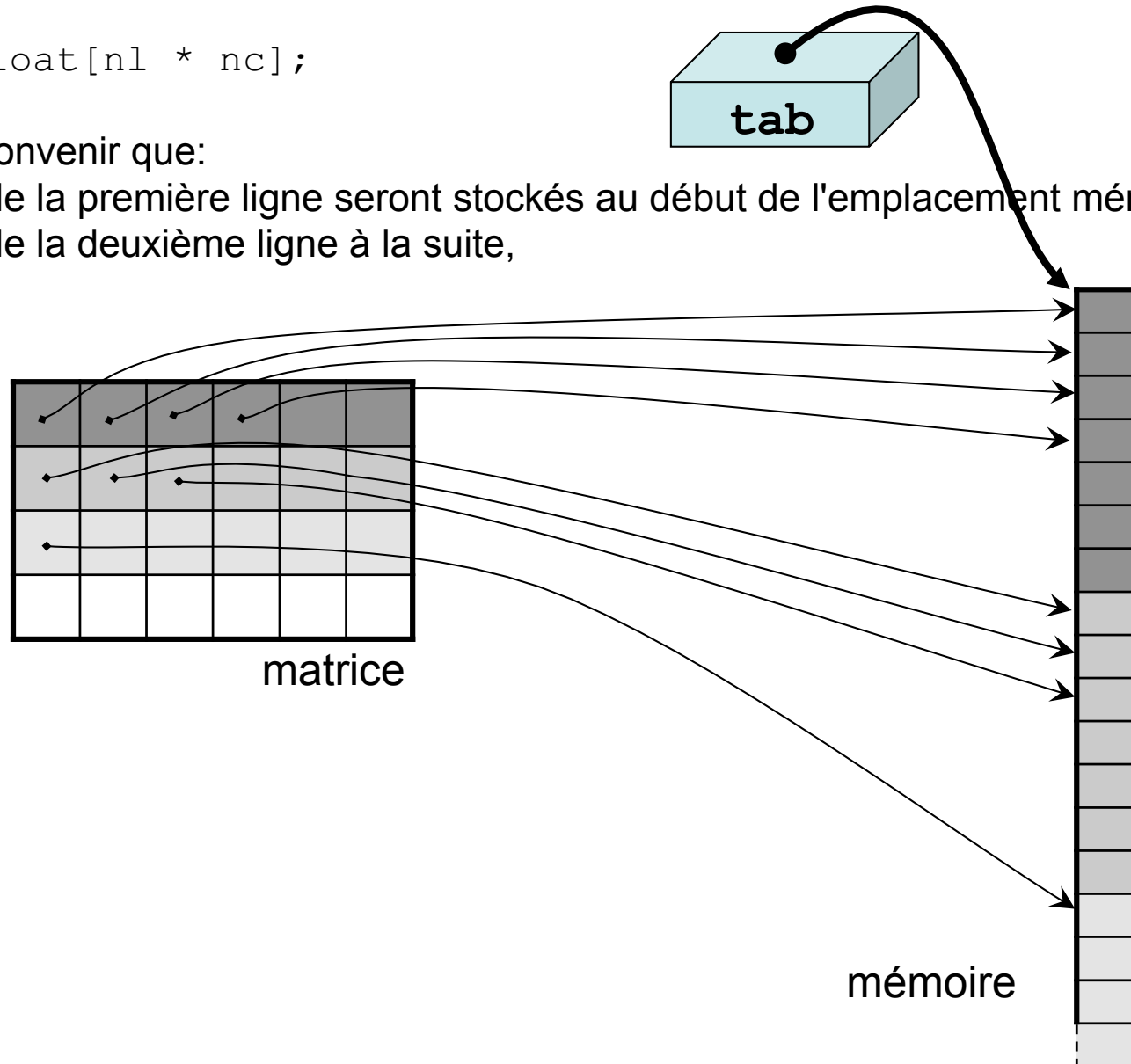
Allocation dynamique de matrices

Allouer une zone contenant tous les éléments du tableau:

```
tab = new float[nl * nc];
```

On peut alors convenir que:

- les éléments de la première ligne seront stockés au début de l'emplacement mémoire,
- les éléments de la deuxième ligne à la suite,
- etc...



Structure Matrice

Une matrice peut donc être représentée par un tableau de `float`:

```
float * tab;
```

Pour faire des opérations sur cette matrice, il faut encore connaître sa taille (nombre de lignes et de colonnes).

→ Créons donc une structure réunissant un pointeur, un nombre de lignes et un nombre de colonnes:

```
struct Matrice  
{  
    float * tab;  
    int nl, nc;  
};
```

Une variable de type `Matrice` contient à la fois le pointeur vers les éléments de la matrice, et la taille de la matrice.

Allocation d'une structure Matrice

```
struct Matrice  
{  
    float * tab;  
    int nl, nc;  
};
```

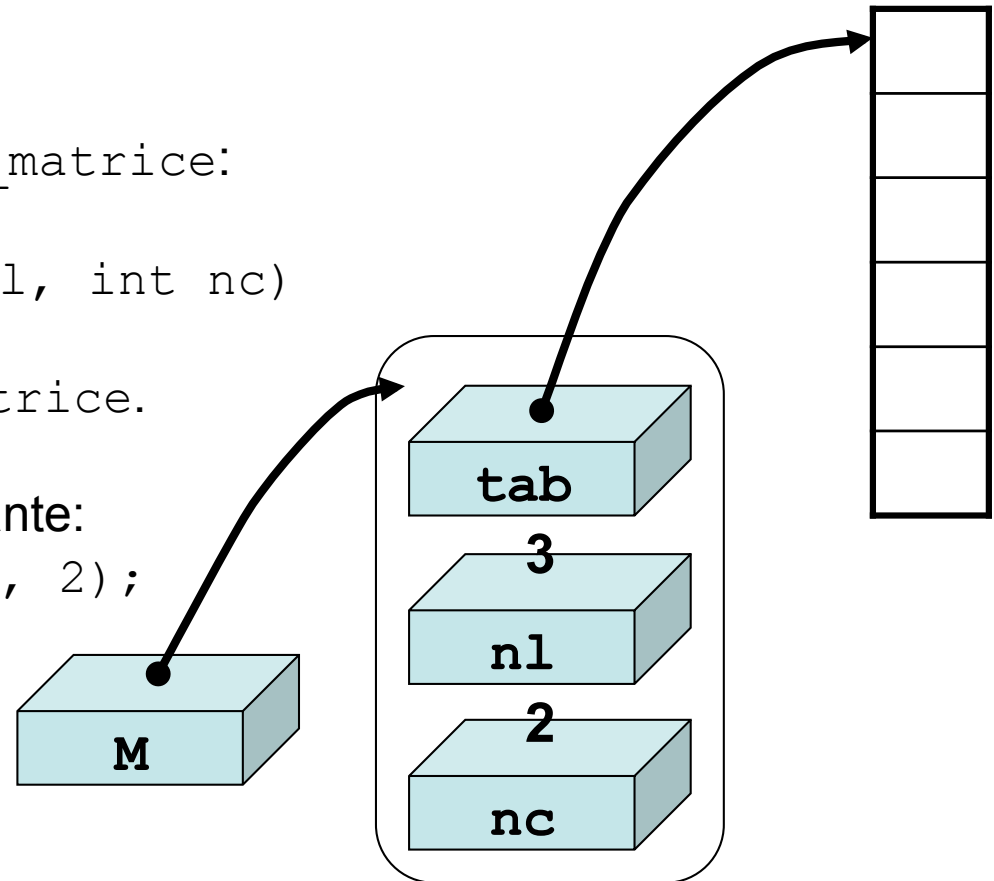
Nous allons écrire une fonction `alloue_matrice`:

```
Matrice * alloue_matrice(int nl, int nc)
```

pour allouer facilement une structure `Matrice`.

Fonctions qui s'utilisent de la façon suivante:

```
Matrice * M = alloue_matrice(3, 2);
```



Allocation d'une structure `Matrice`

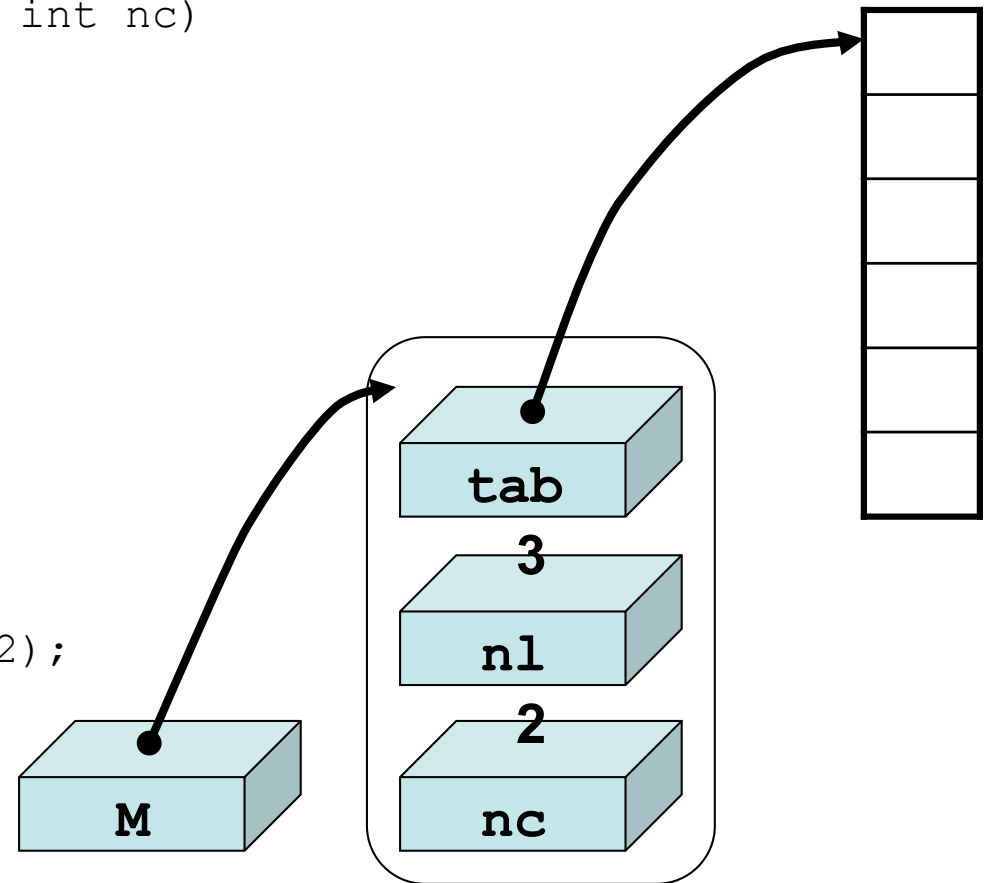
```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice * res = new Matrice;

    res->tab = new float[nl * nc];
    res->nl = nl;
    res->nc = nc;

    return res;
}
```

Fonction qui s'utilise de la façon suivante:

```
Matrice * M = alloue_matrice(3, 2);
```




Pas-à-pas

```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice * res = new Matrice;

    res->tab = new float[nl * nc];
    res->nl = nl;
    res->nc = nc;

    return res;
}
```

Fonction qui s'utilise de la façon suivante:

 `Matrice * M = alloue_matrice(3, 2);`

Pas-à-pas

```
→ Matrice * alloue_matrice(int nl, int nc)
{
    Matrice * res = new Matrice;

    res->tab = new float[nl * nc];
    res->nl = nl;
    res->nc = nc;

    return res;
}
```

Fonction qui s'utilise de la façon suivante:

```
Matrice * M = alloue_matrice(3, 2);
```

Pas-à-pas

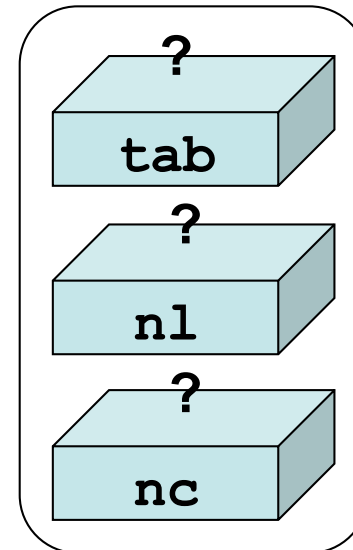
```
Matrice * alloue_matrice(int nl, int nc)
{
    3      2
    → Matrice * res = new Matrice;

    res->tab = new float[nl * nc];
    res->nl = nl;
    res->nc = nc;

    return res;
}
```

Fonction qui s'utilise de la façon suivante:

```
Matrice * M = alloue_matrice(3, 2);
```

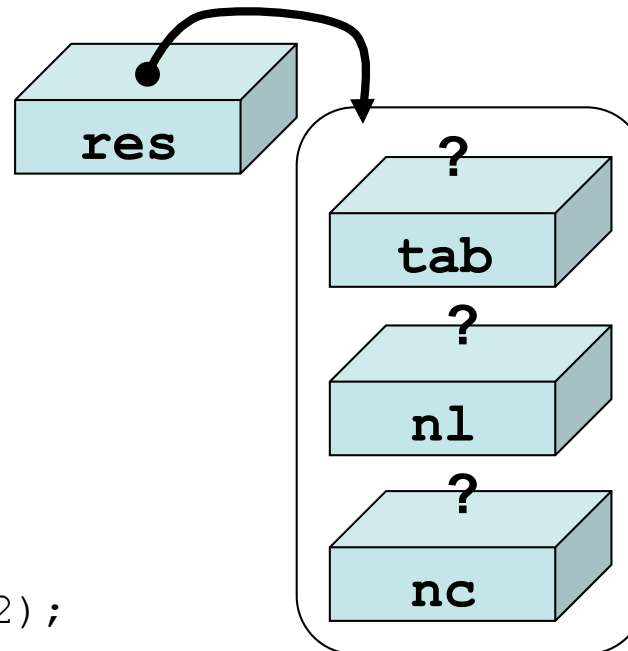


Pas-à-pas

```
Matrice * alloue_matrice(int nl, int nc)
{
  → Matrice * res = new Matrice;

  res->tab = new float[nl * nc];
  res->nl = nl;
  res->nc = nc;

  return res;
}
```



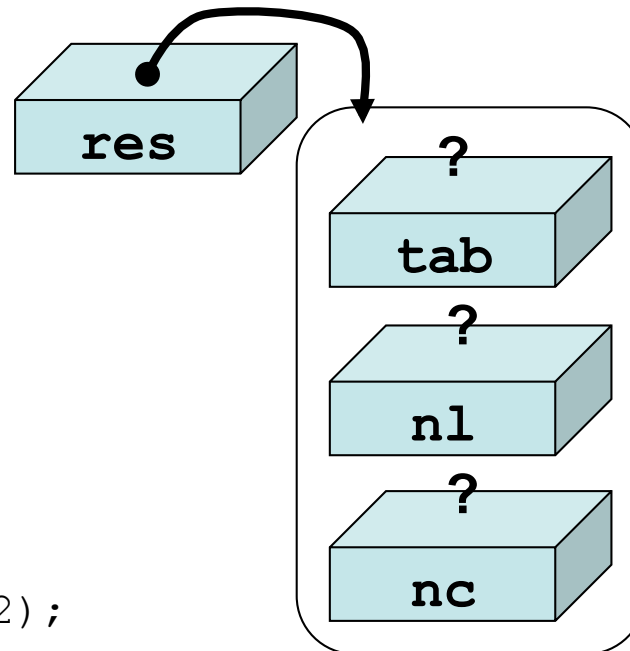
Fonction qui s'utilise de la façon suivante:

```
Matrice * M = alloue_matrice(3, 2);
```

Pas-à-pas

```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice * res = new Matrice;
    → res->tab = new float[nl * nc];
    res->nl = nl;
    res->nc = nc;

    return res;
}
```



Fonction qui s'utilise de la façon suivante:

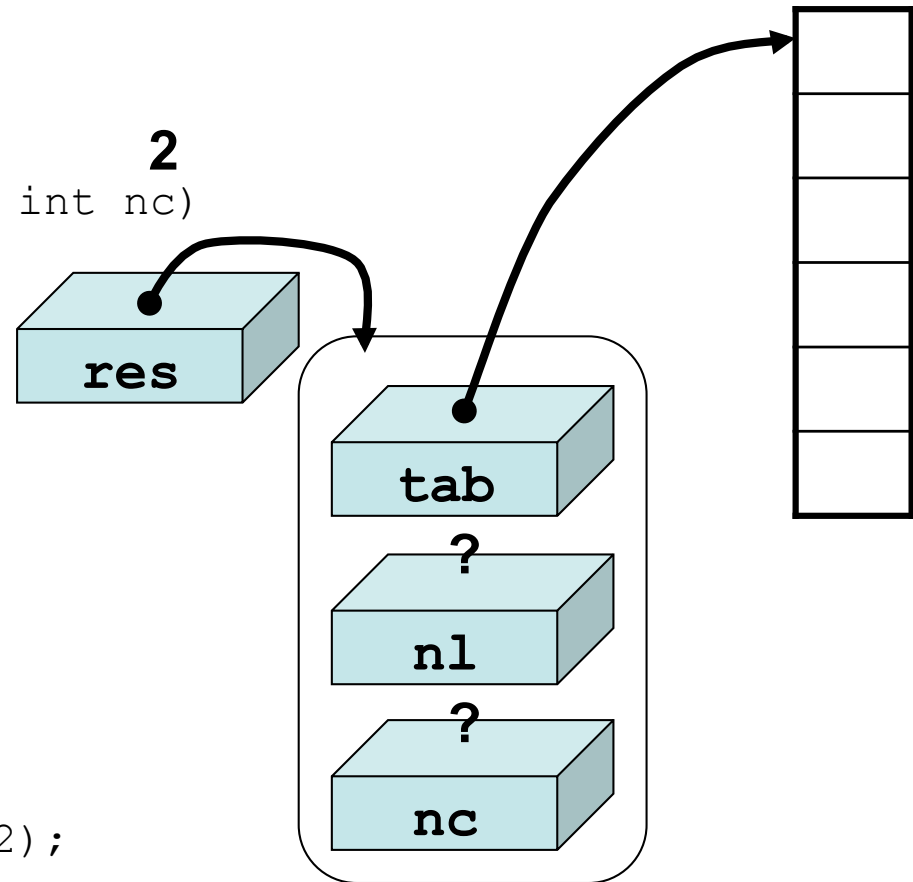
```
Matrice * M = alloue_matrice(3, 2);
```

Pas-à-pas

```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice * res = new Matrice;
    → res->tab = new float[nl * nc];
    res->nl = nl;
    res->nc = nc;

    return res;
}
```

Fonction qui s'utilise de la façon suivante:
Matrice * M = alloue_matrice(3, 2);

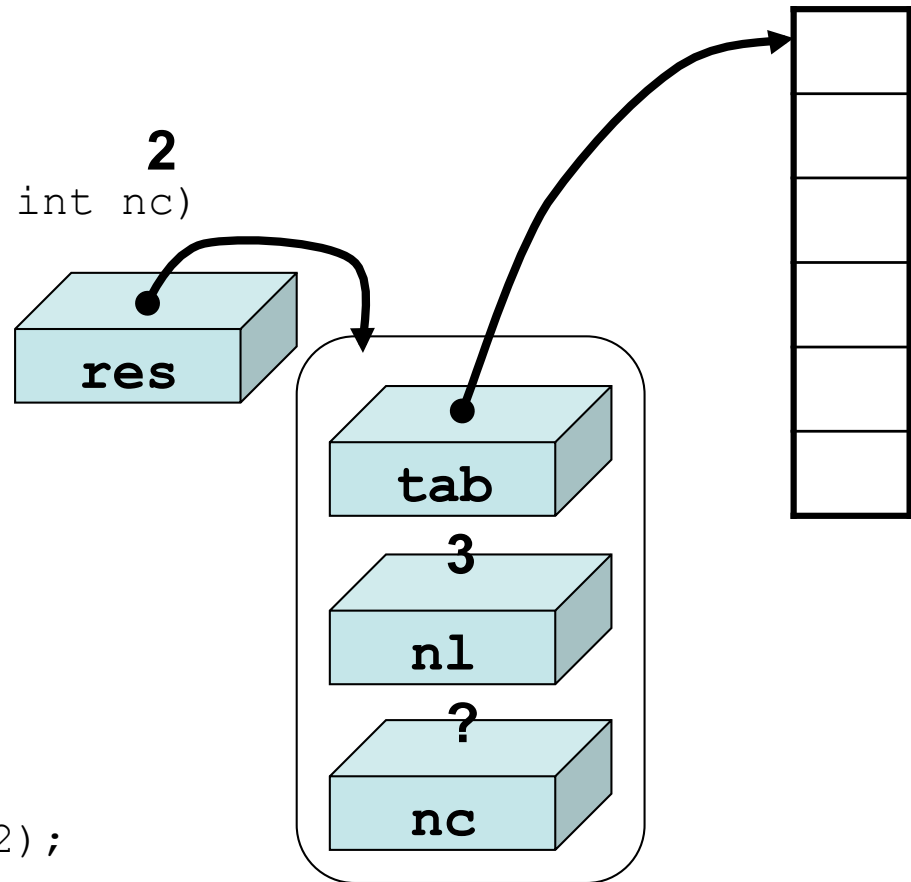


Pas-à-pas

```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice * res = new Matrice;

    res->tab = new float[nl * nc];
    → res->nl = nl;
    res->nc = nc;

    return res;
}
```



Fonction qui s'utilise de la façon suivante:

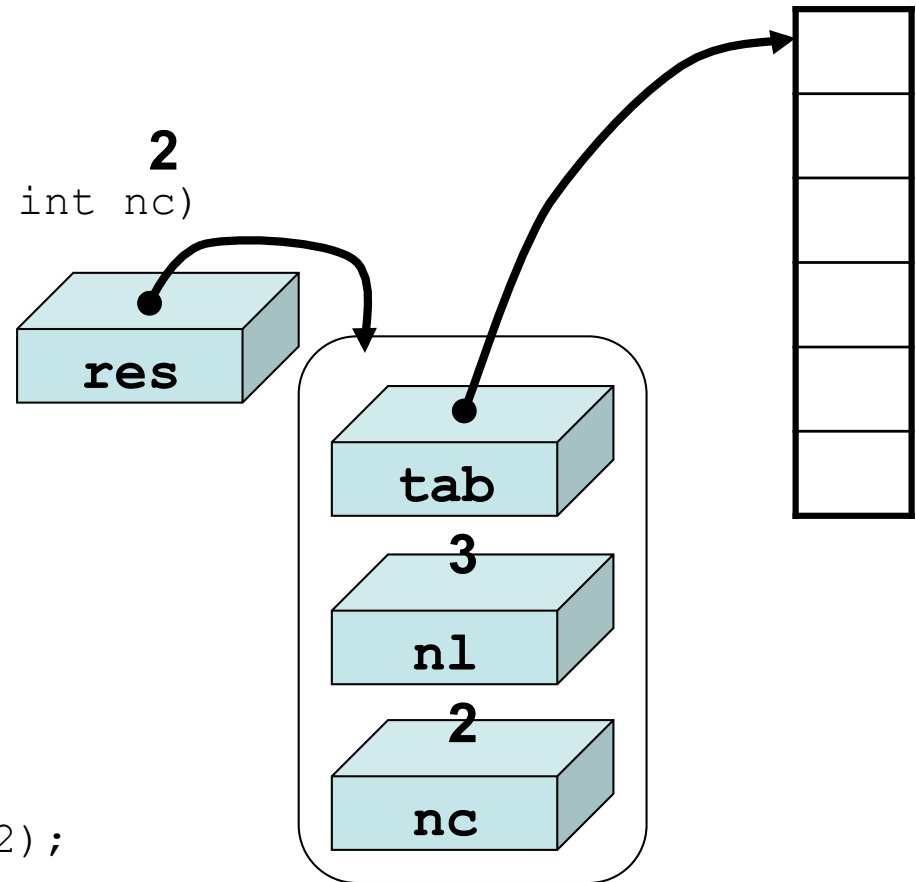
```
Matrice * M = alloue_matrice(3, 2);
```

Pas-à-pas

```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice * res = new Matrice;

    res->tab = new float[nl * nc];
    res->nl = nl;
    → res->nc = nc;

    return res;
}
```



Fonction qui s'utilise de la façon suivante:

```
Matrice * M = alloue_matrice(3, 2);
```

Pas-à-pas

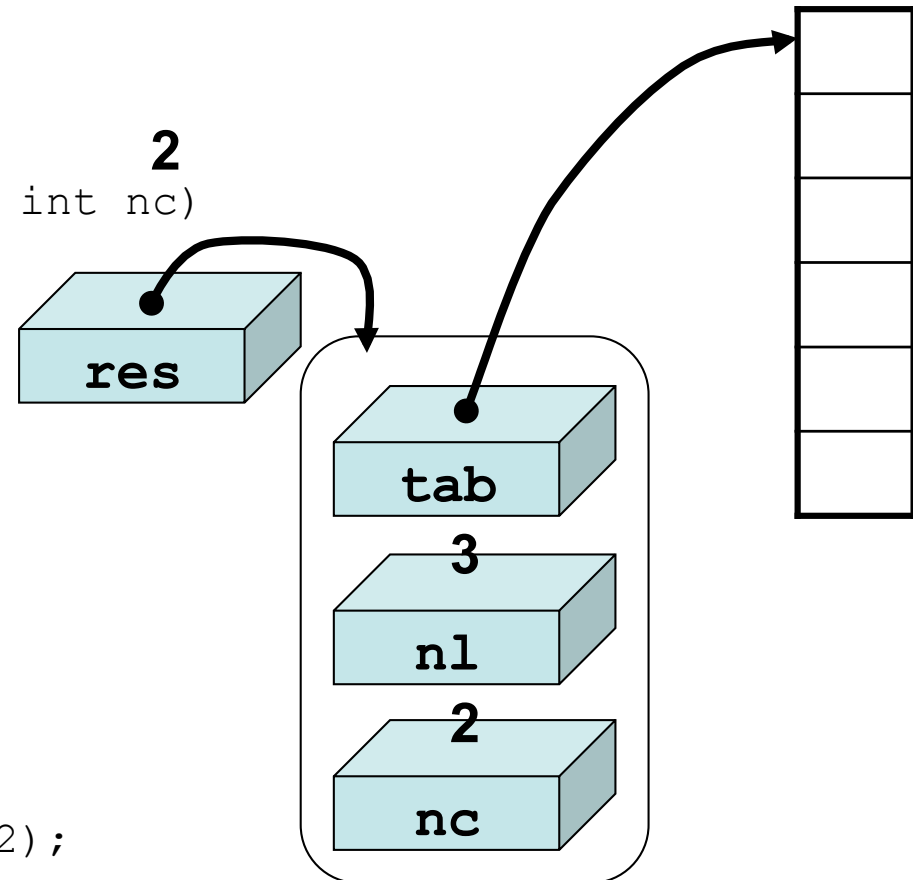
```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice * res = new Matrice;

    res->tab = new float[nl * nc];
    res->nl = nl;
    res->nc = nc;

    → return res;
}
```

Fonction qui s'utilise de la façon suivante:

```
Matrice * M = alloue_matrice(3, 2);
```



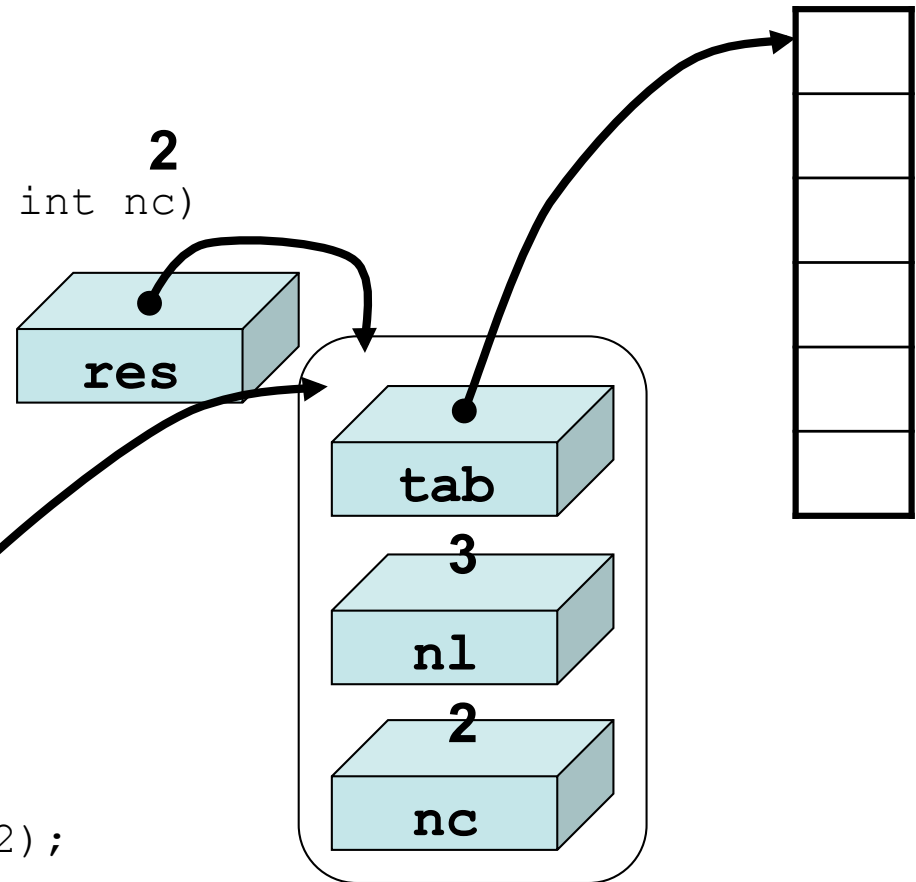
Pas-à-pas

```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice * res = new Matrice;

    res->tab = new float[nl * nc];
    res->nl = nl;
    res->nc = nc;

    → return res;
}
```

Fonction qui s'utilise de la façon suivante:
Matrice * M = alloue_matrice(3, 2);



Pas-à-pas

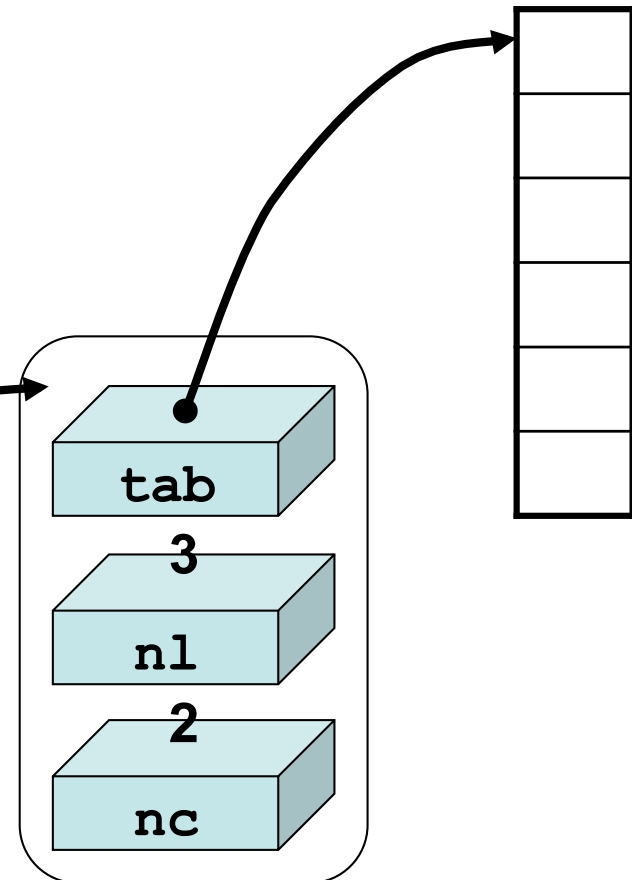
```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice * res = new Matrice;

    res->tab = new float[nl * nc];
    res->nl = nl;
    res->nc = nc;

    return res;
}
```

Fonction qui s'utilise de la façon suivante:

→ `Matrice * M = alloue_matrice(3, 2);`



Pas-à-pas

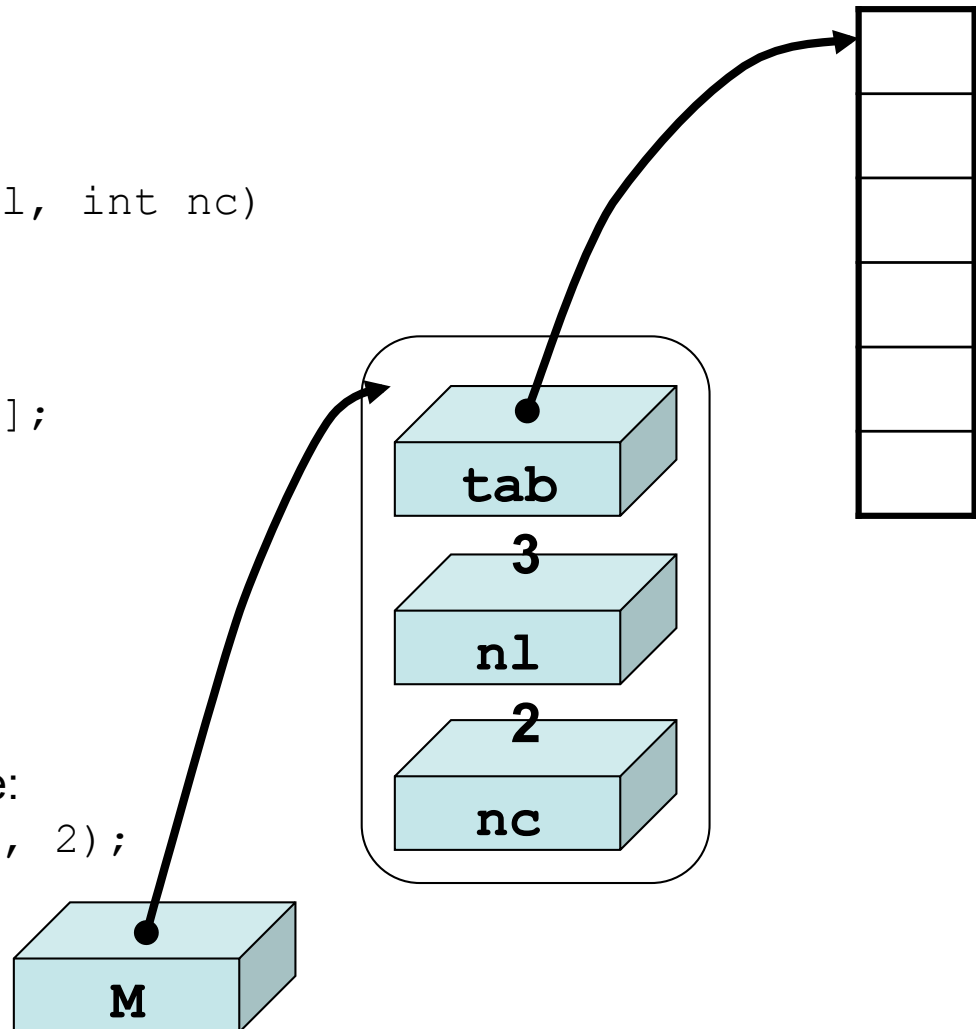
```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice * res = new Matrice;

    res->tab = new float[nl * nc];
    res->nl = nl;
    res->nc = nc;

    return res;
}
```

Fonction qui s'utilise de la façon suivante:

→ `Matrice * M = alloue_matrice(3, 2);`



Pièges de la fonction `alloue_matrice`:

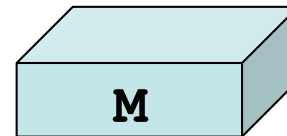
Piège 1

IL NE FAUT PAS FAIRE:

```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice res;

    res.tab = new float[nl * nc];
    res.nl = nl;
    res.nc = nc;

    return &res; // !!! On retourne un pointeur
                // sur une variable locale !!!
}
```



Ici, `res` est une structure `Matrice`, non plus un pointeur sur une zone mémoire allouée dynamiquement.

Mais `res` est une variable locale, qui disparaît quand on sort de la fonction.

La fonction retourne ici un pointeur sur une zone mémoire qui risque fort d'être remplie par d'autres valeurs lors de la suite du déroulement du programme.

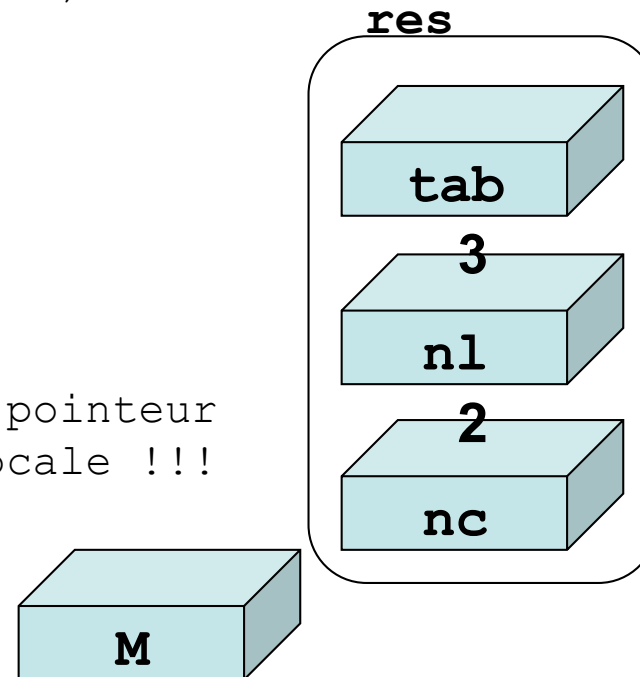
Pas-à-pas

IL NE FAUT PAS FAIRE:

```
Matrice * alloue_matrice(int nl, int nc)
{
  → Matrice res;

  res.tab = new float[nl * nc];
  res.nl = nl;
  res.nc = nc;

  return &res; // !!! On retourne un pointeur
               // sur une variable locale !!!
}
```

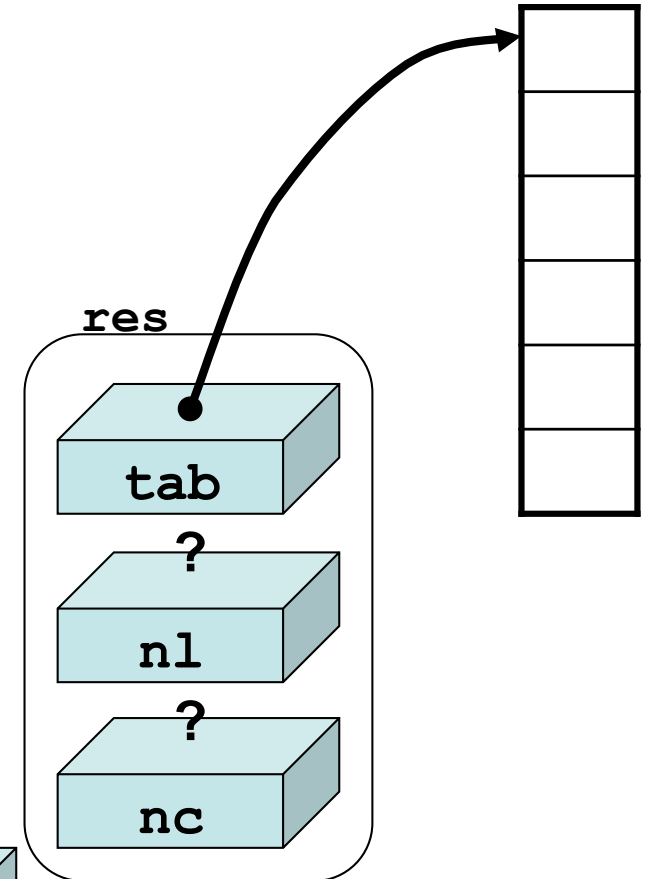
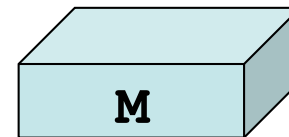


Pas-à-pas

IL NE FAUT PAS FAIRE:

```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice res;
    → res.tab = new float[nl * nc];
    res.nl = nl;
    res.nc = nc;

    return &res; // !!! On retourne un pointeur
                // sur une variable locale !!!
}
```

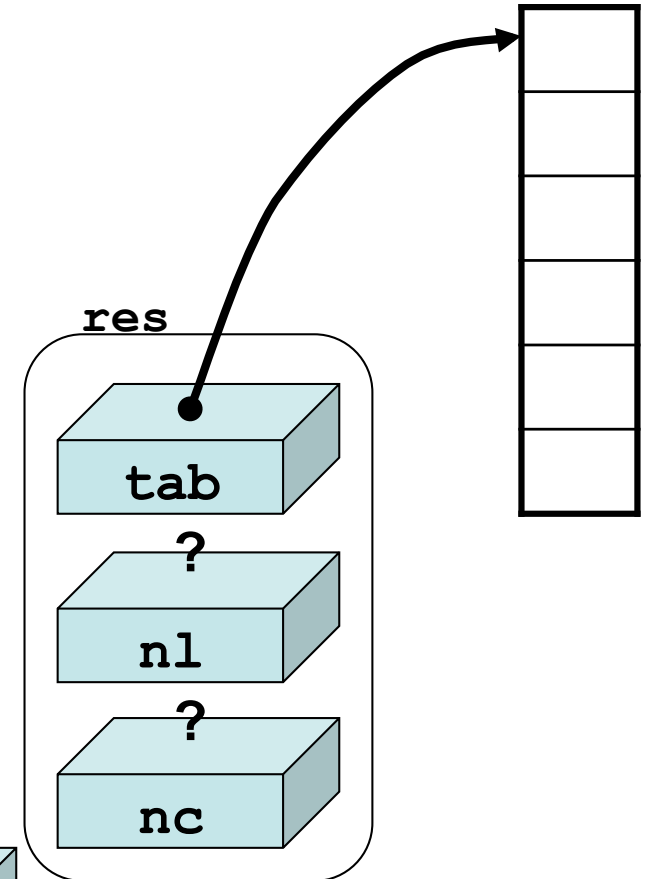
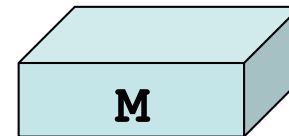


Pas-à-pas

IL NE FAUT PAS FAIRE:

```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice res;
    → res.tab = new float[nl * nc];
    res.nl = nl;
    res.nc = nc;

    return &res; // !!! On retourne un pointeur
                // sur une variable locale !!!
}
```

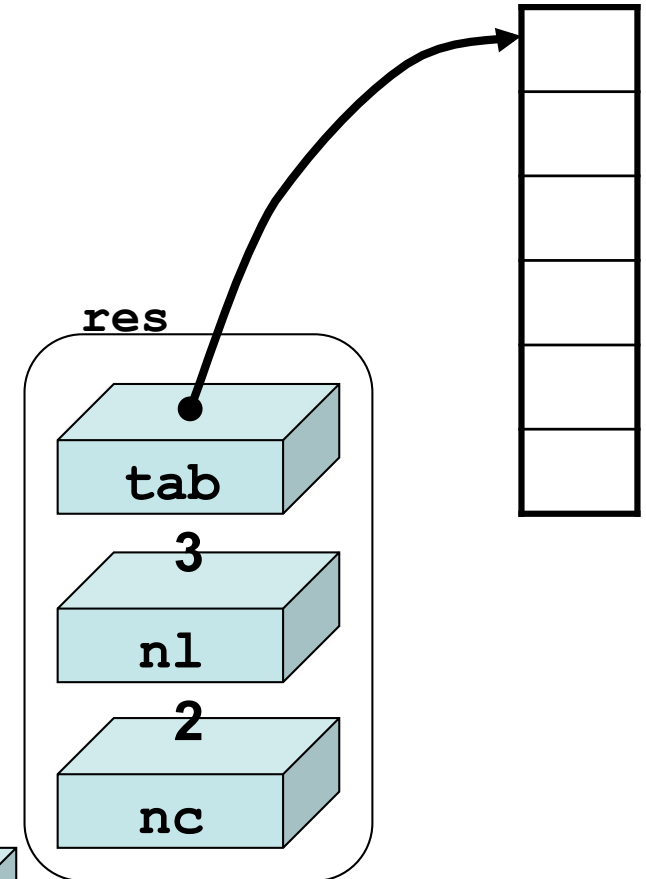
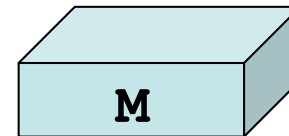


Pas-à-pas

IL NE FAUT PAS FAIRE:

```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice res;

    res.tab = new float[nl * nc];
    res.nl = nl;
    res.nc = nc;
    → return &res; // !!! On retourne un pointeur
                // sur une variable locale !!!
}
```



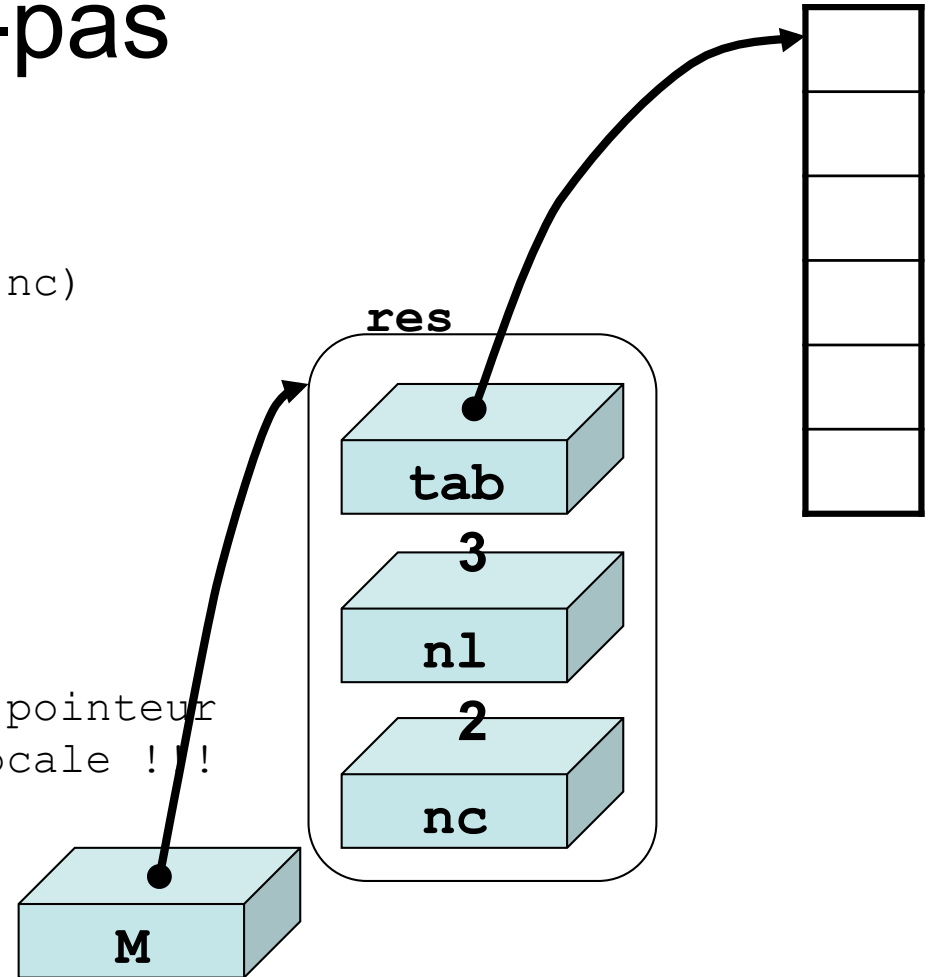
Pas-à-pas

IL NE FAUT PAS FAIRE:

```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice res;

    res.tab = new float[nl * nc];
    res.nl = nl;
    res.nc = nc;

    → return &res; // !!! On retourne un pointeur
                // sur une variable locale !!!
}
```



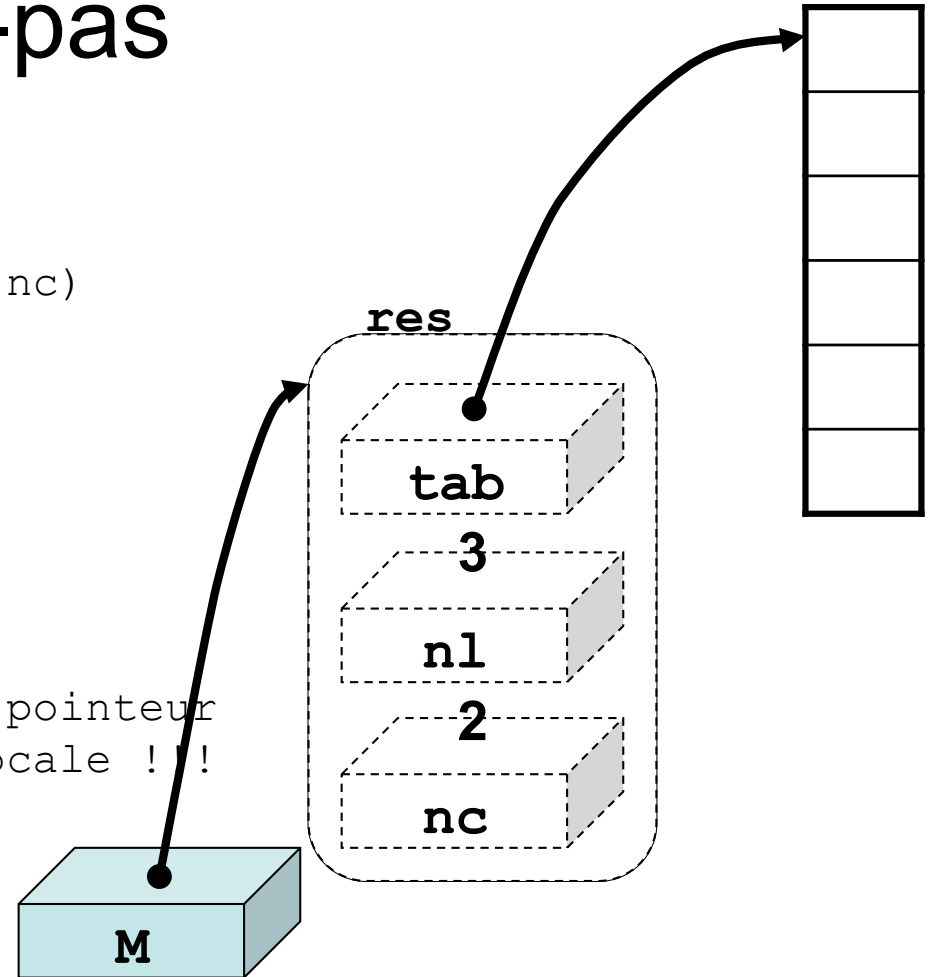
Pas-à-pas

IL NE FAUT PAS FAIRE:

```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice res;

    res.tab = new float[nl * nc];
    res.nl = nl;
    res.nc = nc;

    return &res; // !!! On retourne un pointeur
                // sur une variable locale !!!
}
```



Pièges de la fonction `alloue_matrice`:

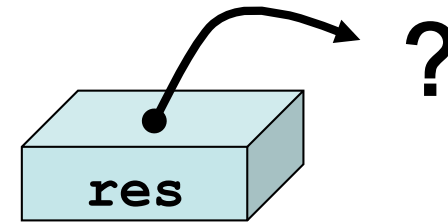
Piège 2

IL NE FAUT PAS FAIRE:

```
Matrice * alloue_matrice(int nl, int nc)
{
    Matrice * res; // !!! res pointe n'importe ou !!!

    res->tab = new float[nl * nc];
    res->nl = nl;
    res->nc = nc;

    return res;
}
```



Ici, `res` est de nouveau un pointeur, mais `new` n'est pas appelée pour allouer de la mémoire.

`res` contient une valeur inconnue, et pointe quelque part en mémoire...

En fonction de ce que contient effectivement `res`, les instructions `res->tab = ...`, `res->nl = ...` peuvent provoquer deux choses:

- `res` pointe sur une zone mémoire n'appartenant pas au programme, et ces instructions vont provoquer un `Segmentation fault`;
- sinon `res` pointe sur une zone mémoire appartenant au programme, et ces instructions vont écraser des données du programme.

Fonction `libere_matrice`

Il faut encore écrire une fonction qui permet de libérer la mémoire occupée par une structure `Matrice`, une fois que celle-ci n'est plus utilisée:

Cette fonction a l'en-tête suivant:

```
void libere_matrice(Matrice * M)
```

Il faut commencer par libérer la mémoire allouée pour les éléments.
Le champ `tab` de `M` pointe sur le début de cette mémoire:

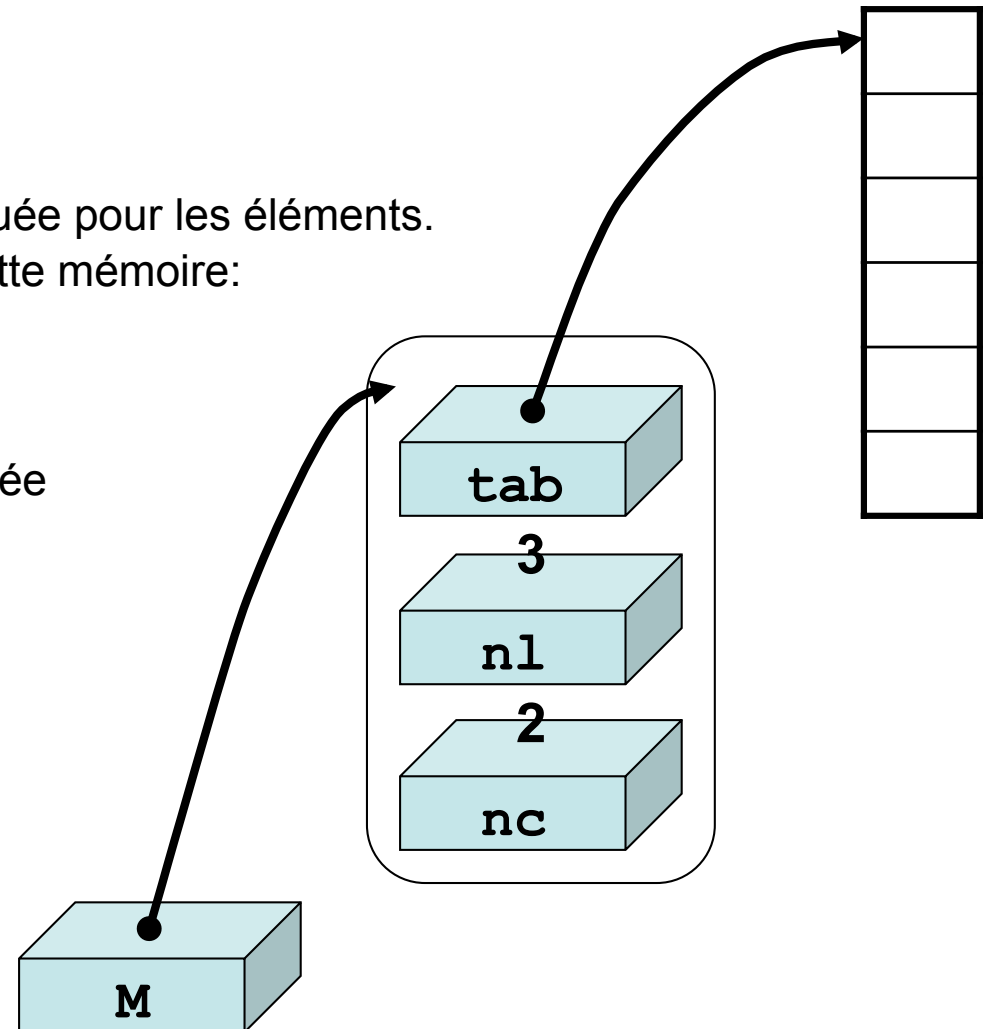
```
delete [] M->tab;
```

On peut maintenant libérer la mémoire occupée par la structure elle-même:

```
delete M;
```

La fonction complète:

```
void libere_matrice(Matrice * M)
{
    delete [] M->tab;
    delete M;
}
```



Pas-à-pas

```
Matrice * alloue_matrice(int nl, int nc)
{
  ...
}
```

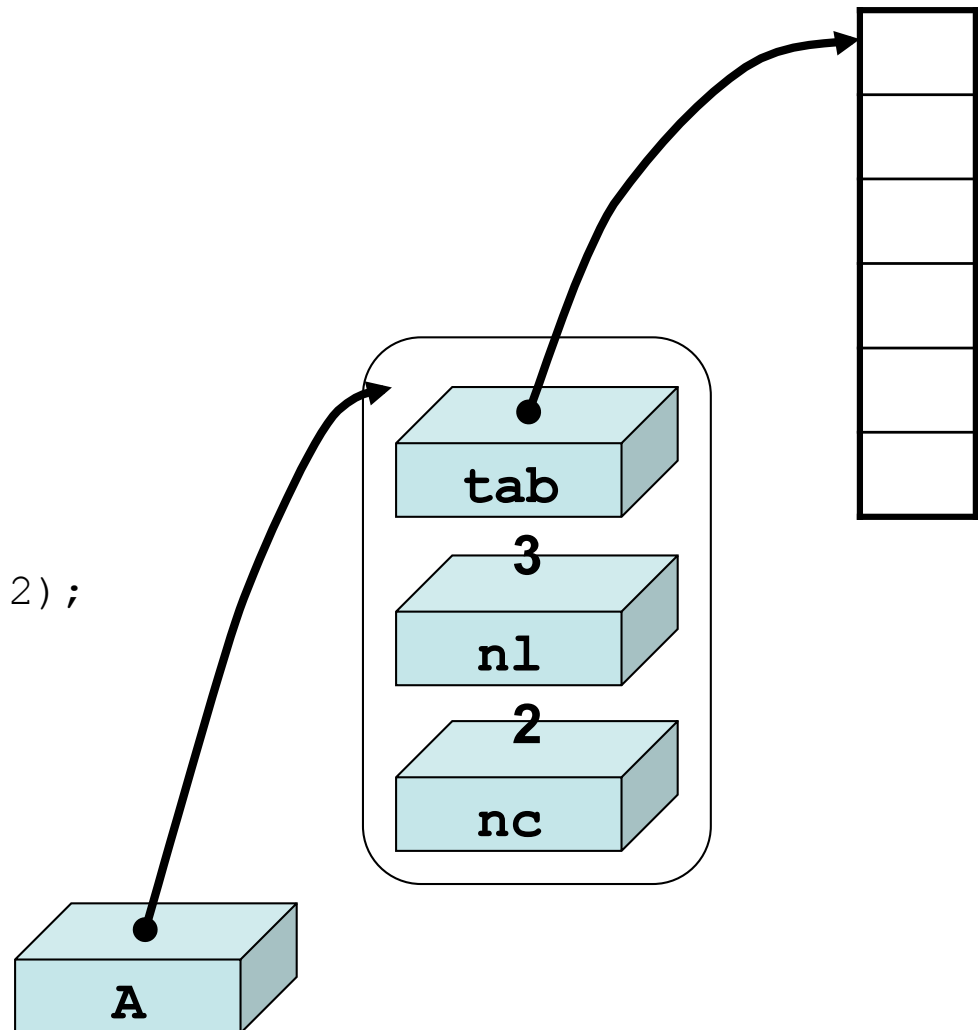
```
void libere_matrice(Matrice * M)
{
  delete [] M->tab;
  delete M;
}
```

...

```
Matrice * A = alloue_matrice(3, 2);
```

...

```
→ libere_matrice(A);
```



Pas-à-pas

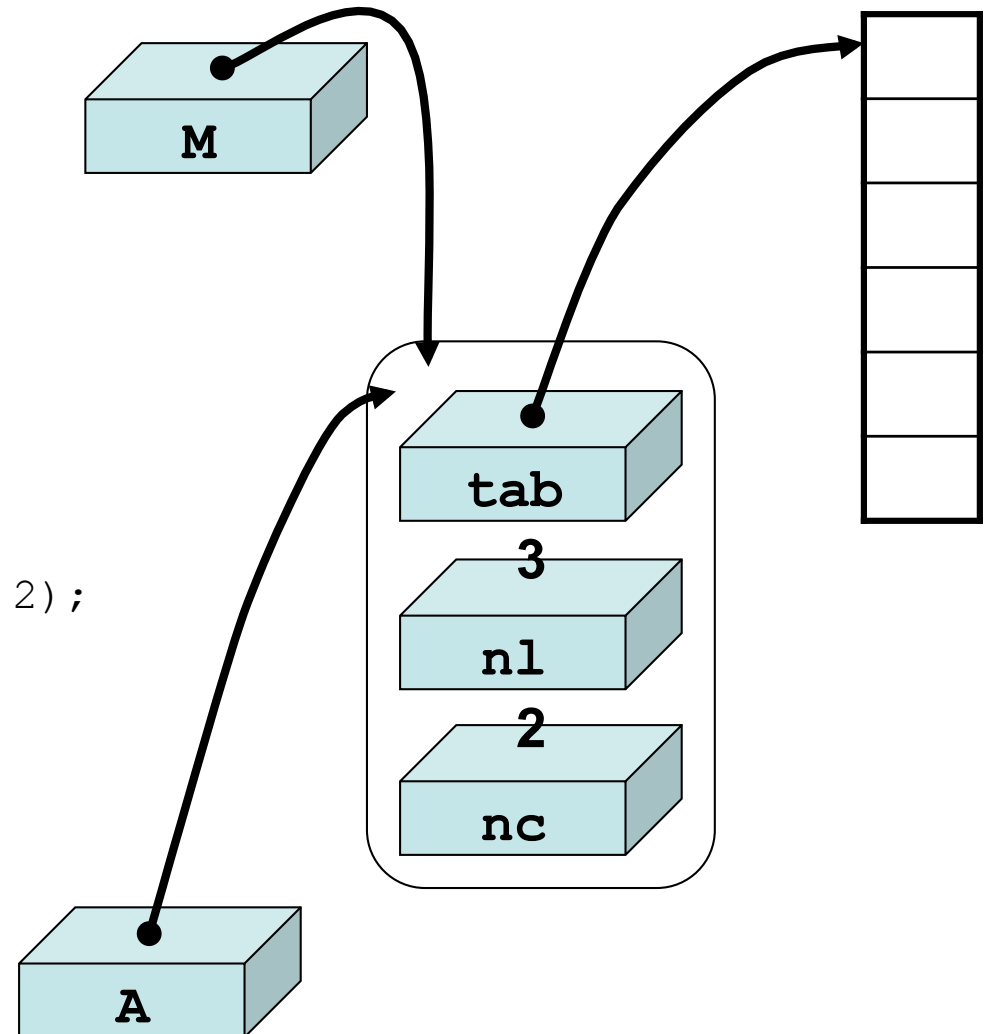
```
Matrice * alloue_matrice(int nl, int nc)
{
  ...
}
```

```
→ void libere_matrice(Matrice * M)
{
  delete [] M->tab;
  delete M;
}
```

...

```
Matrice * A = alloue_matrice(3, 2);
```

```
...
libere_matrice(A);
```



Pas-à-pas

```
Matrice * alloue_matrice(int nl, int nc)
```

```
{
```

```
...
```

```
}
```

```
void libere_matrice(Matrice * M)
```

```
{
```

```
→ delete [] M->tab;
```

```
delete M;
```

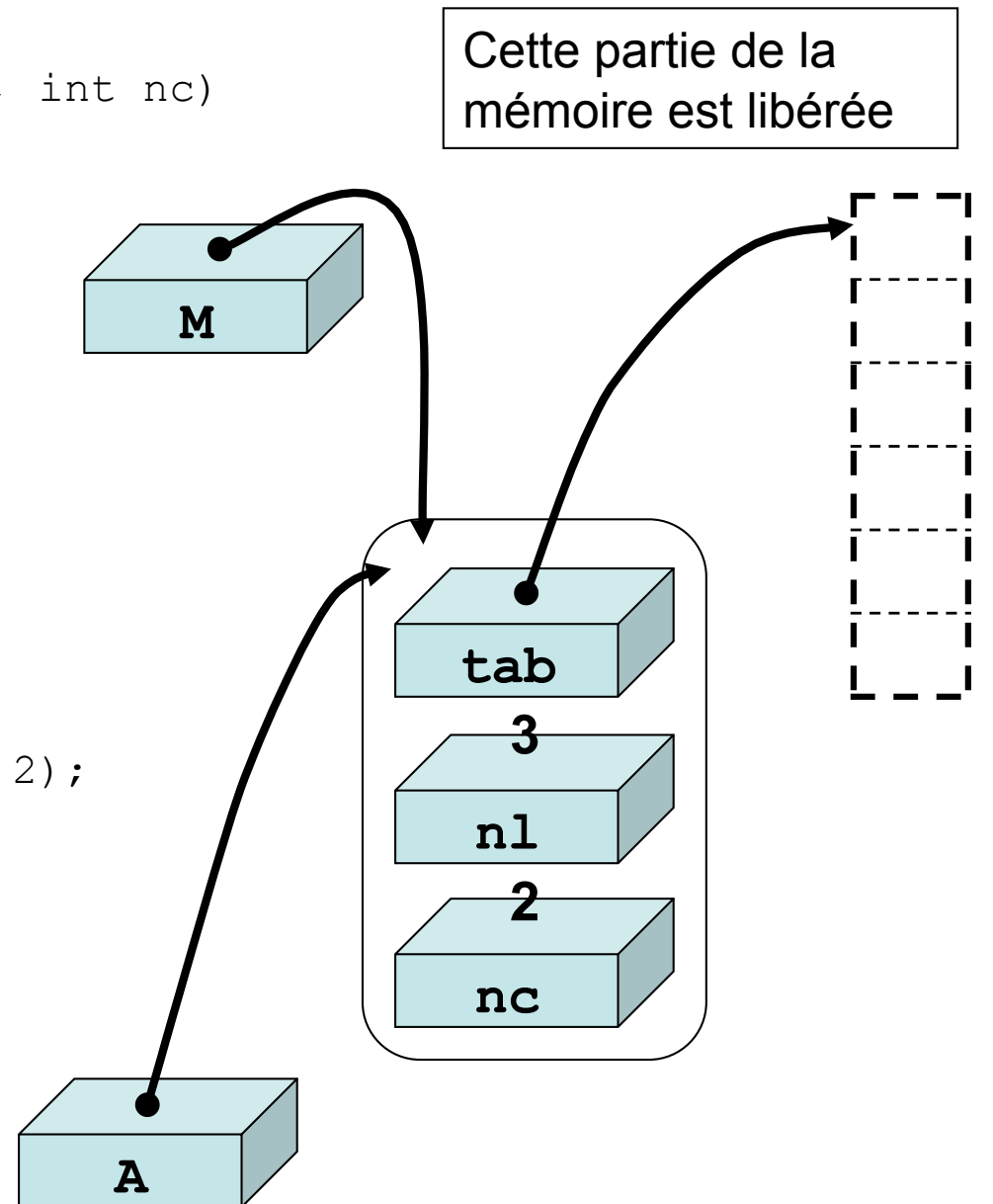
```
}
```

```
...
```

```
Matrice * A = alloue_matrice(3, 2);
```

```
...
```

```
libere_matrice(A);
```



Pas-à-pas

```
Matrice * alloue_matrice(int nl, int nc)
{
  ...
}
```

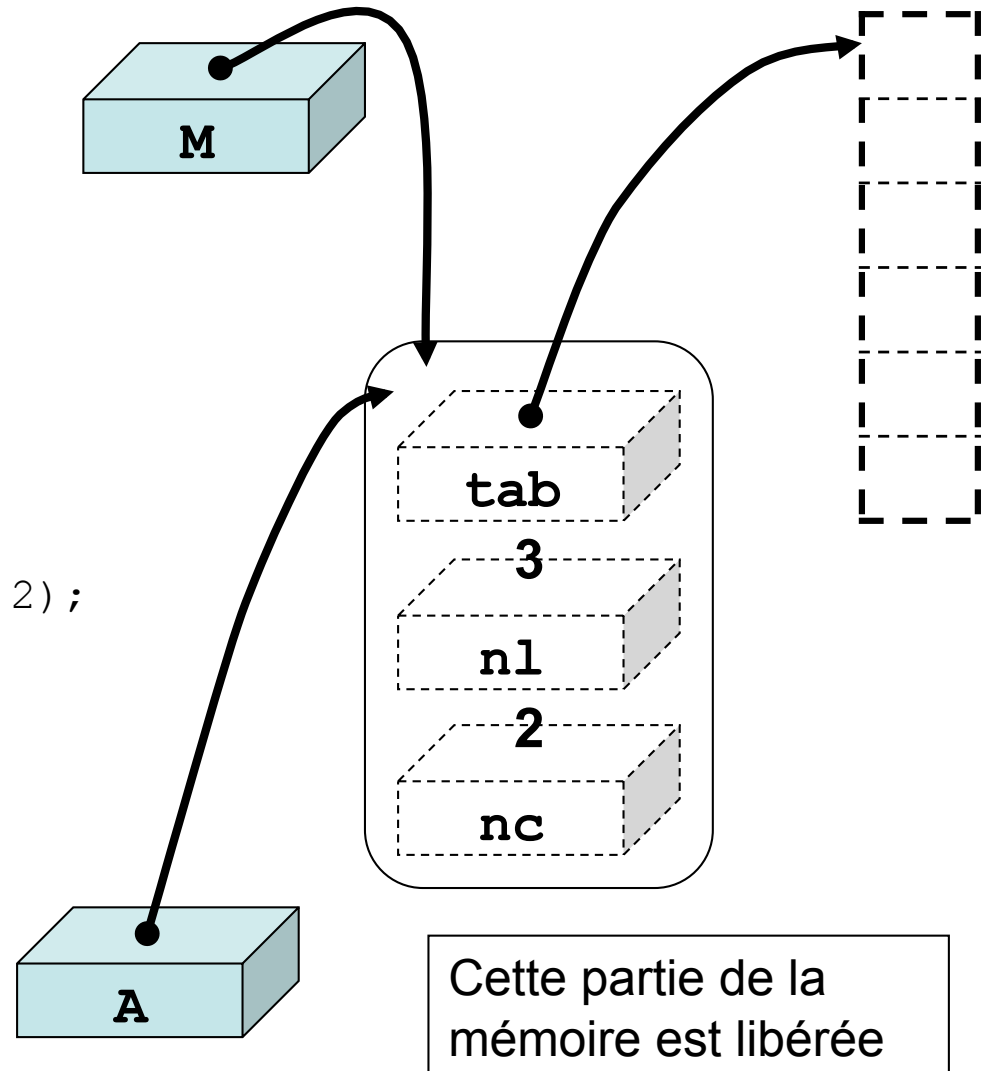
```
void libere_matrice(Matrice * M)
{
  delete [] M->tab;
  → delete M;
}
```

...

```
Matrice * A = alloue_matrice(3, 2);
```

...

```
libere_matrice(A);
```



Pas-à-pas

```
Matrice * alloue_matrice(int nl, int nc)
{
  ...
}
```

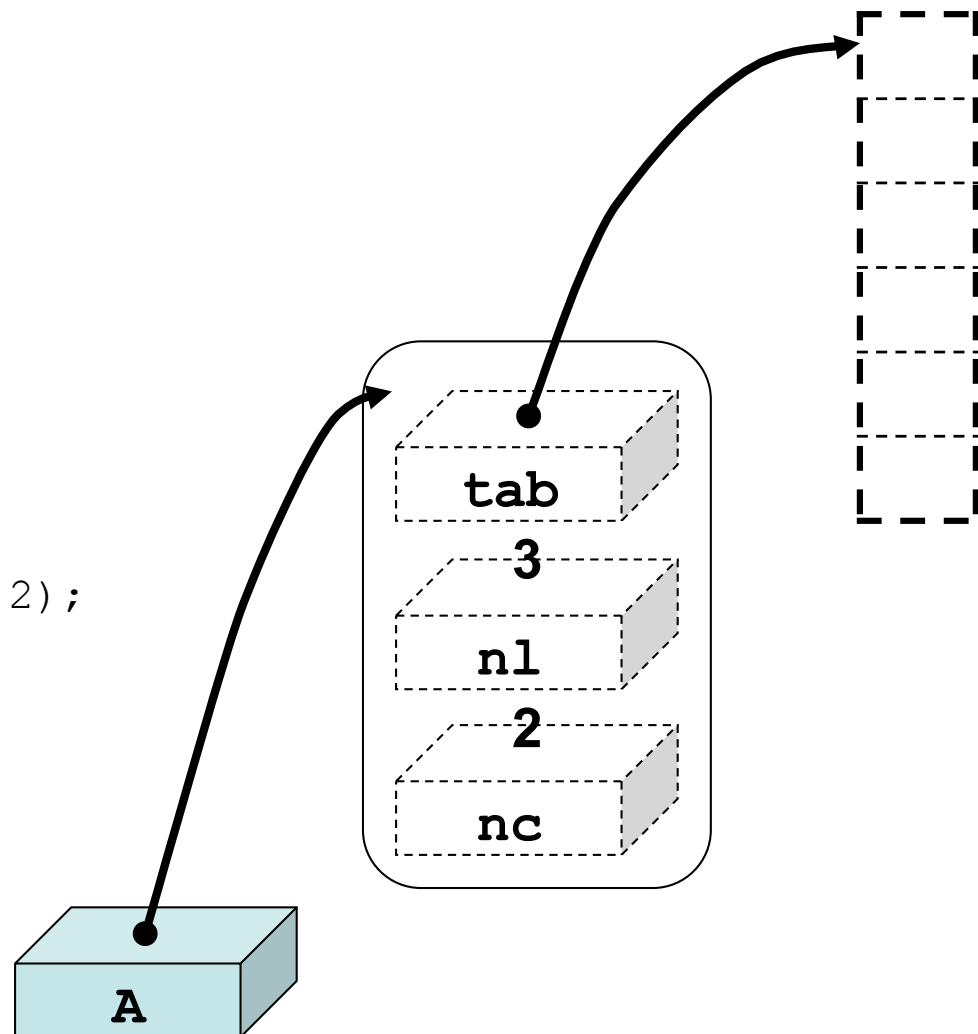
```
void libere_matrice(Matrice * M)
{
  delete [] M->tab;
  delete M;
}
```

...

```
Matrice * A = alloue_matrice(3, 2);
```

...

```
libere_matrice(A);
```



Remarque

Le C++ systématise l'utilisation de fonctions similaires aux fonctions `alloue_matrice` et `libere_matrice`.

On a déjà vu que les classes C++ ressemblent aux structures, mais réunissent non seulement des champs (comme `n1` et `nc` pour notre structure `Matrice`) mais aussi des fonctions.

Parmi ces fonctions, les classes C++ ont des fonctions spéciales:

- des fonctions appelées *constructeurs*, qui sont utilisées pour l'allocation et sont assez similaires à `alloue_matrice`.
- de même, il existe des fonctions appelées *destructeurs*, pour la libération de la mémoire, assez similaires à `libere_matrice`.

Accéder aux éléments

On ne peut plus utiliser les notations des tableaux à deux dimensions:

```
e1 = M[i][j];
```

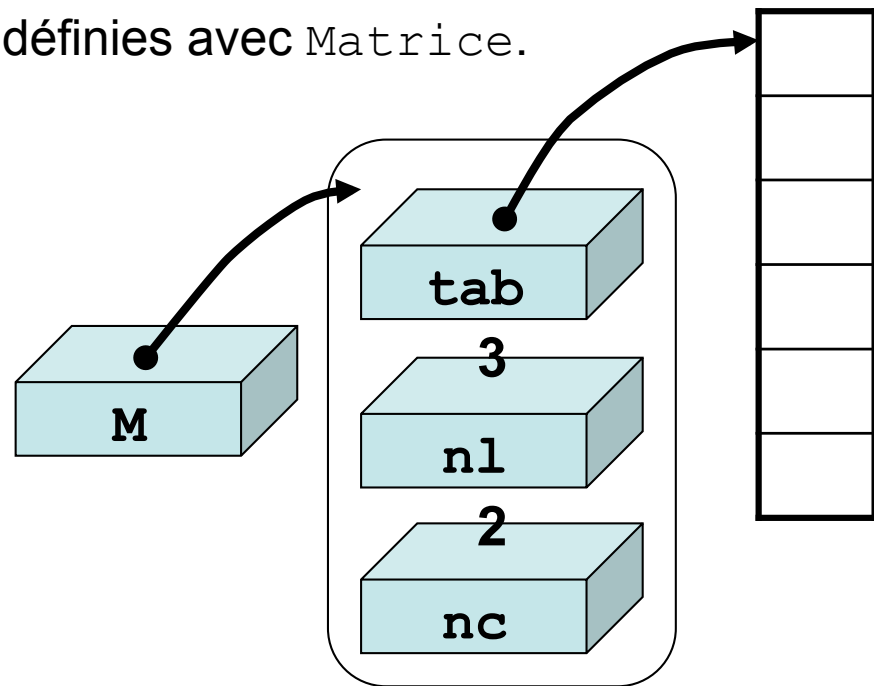
ou

```
M[i][j] = 0;
```

pour accéder aux éléments des matrices définies avec `Matrice`.

Nous allons faire à la place:

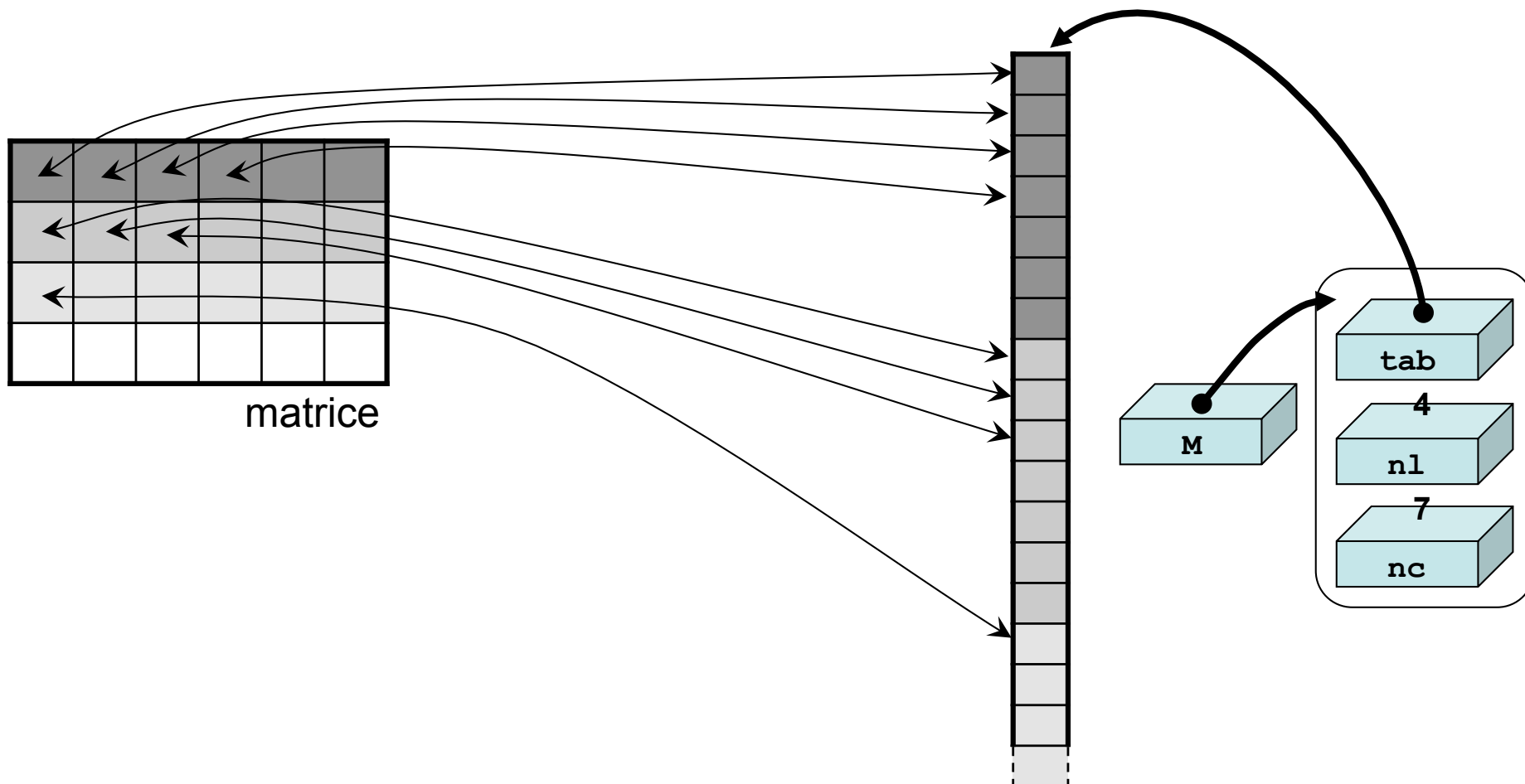
```
e1 = M->tab[dépend de i et j];
```



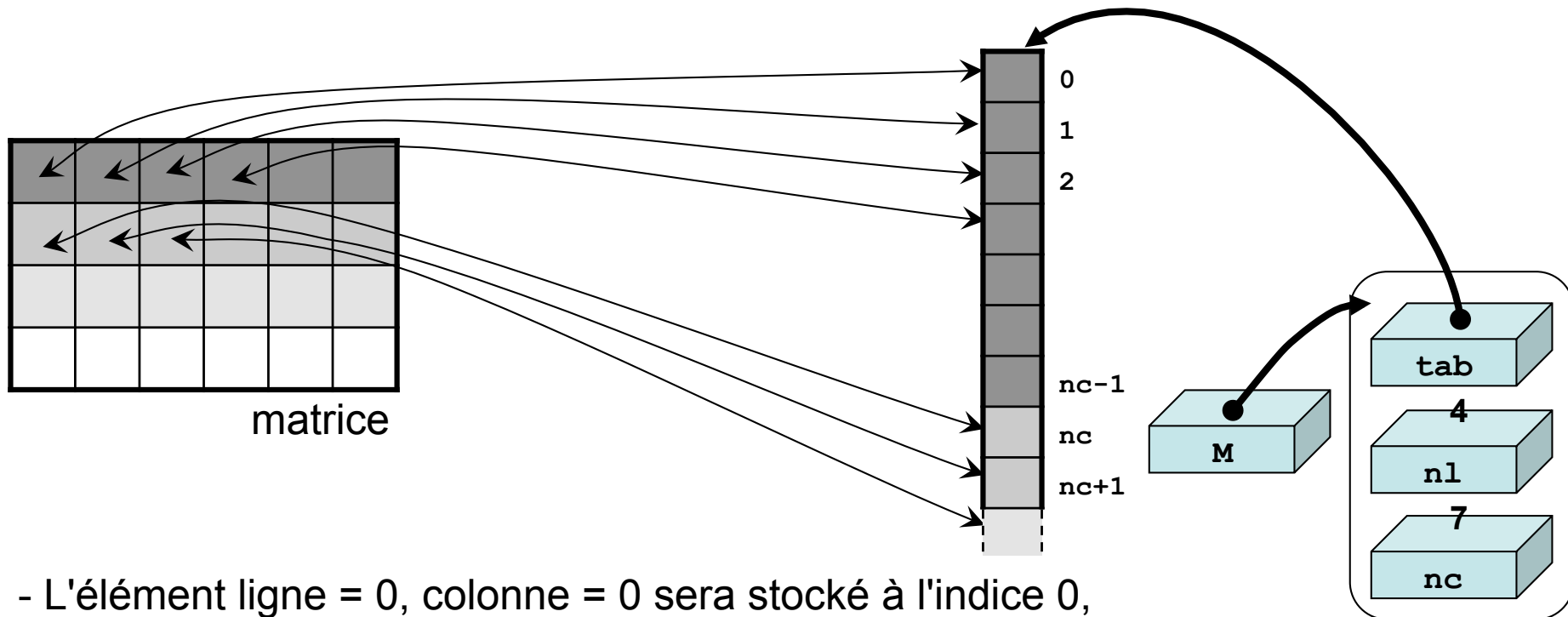
Accéder aux éléments

On peut alors convenir que:

- les éléments de la première ligne seront stockés au début de l'emplacement mémoire,
- les éléments de la deuxième ligne à la suite,
- etc...



Accéder aux éléments



- L'élément ligne = 0, colonne = 0 sera stocké à l'indice 0,
 - l'élément ligne = 0, colonne = 1 sera stocké à l'indice 1,
 - ...
 - l'élément ligne = 1, colonne = 0 sera stocké à l'indice nc ,
 - ...
 - l'élément ligne = 2, colonne = 0 sera stocké à l'indice $2nc$,
 - ...
- l'élément i, j sera stocké à l'indice $i * nc + j$

On peut donc accéder à l'élément i, j de la façon suivante:

```
e1 = M->tab[i * M->nc + j];
```

Accéder aux éléments avec des fonctions

La fonction suivante retourne la valeur de l'élément i, j de la matrice dynamique M :

```
float get(Matrice * M, int i, int j)
{
    return M->tab[i * M->nc + j];
}
```

La fonction suivante met la valeur e à la place de l'élément i, j :

```
void set(Matrice * M, int i, int j, float e)
{
    M->tab[i * M->nc + j] = e;
}
```

Fonction affiche

La fonction `affiche` sera donc codée de la façon suivante:

```
void affiche(Matrice * M)
{
    for(int i = 0; i < M->nl; i++)
    {
        for(int j = 0; j < M->nc; j++)
            cout << get(M, i, j) << " ";
        cout << endl;
    }
}
```

Cette fonction pourra être utilisée de la façon suivante:

```
Matrice * A = alloue_matrice(3, 2);
```

```
// Initialiser A
```

```
affiche(A) ;
```

Pas-à-pas


```
float get(Matrice * M, int i, int j)
{
    return M->tab[i * M->nc + j];
}
```

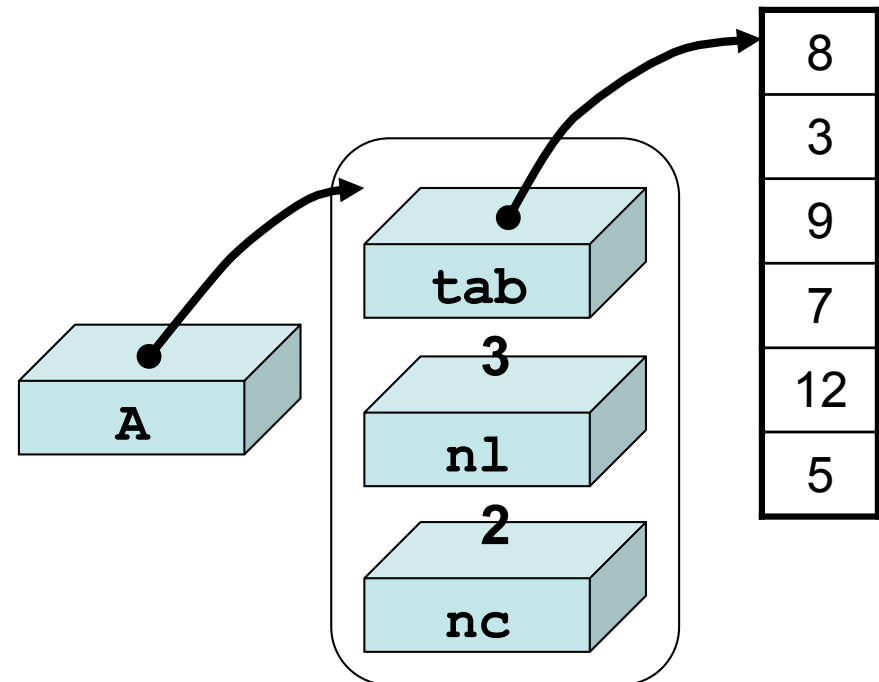
```
void affiche(Matrice * M)
{
    for(int i = 0; i < M->nl; i++)
    {
        for(int j = 0; j < M->nc; j++)
            cout << get(M, i, j) << " ";
        cout << endl;
    }
}
```

...

```
Matrice * A = alloue_matrice(3, 2);
```

```
// Initialiser A
```

 `affiche(A);`



Pas-à-pas

```
float get(Matrice * M, int i, int j)
{
    return M->tab[i * M->nc + j];
}
```

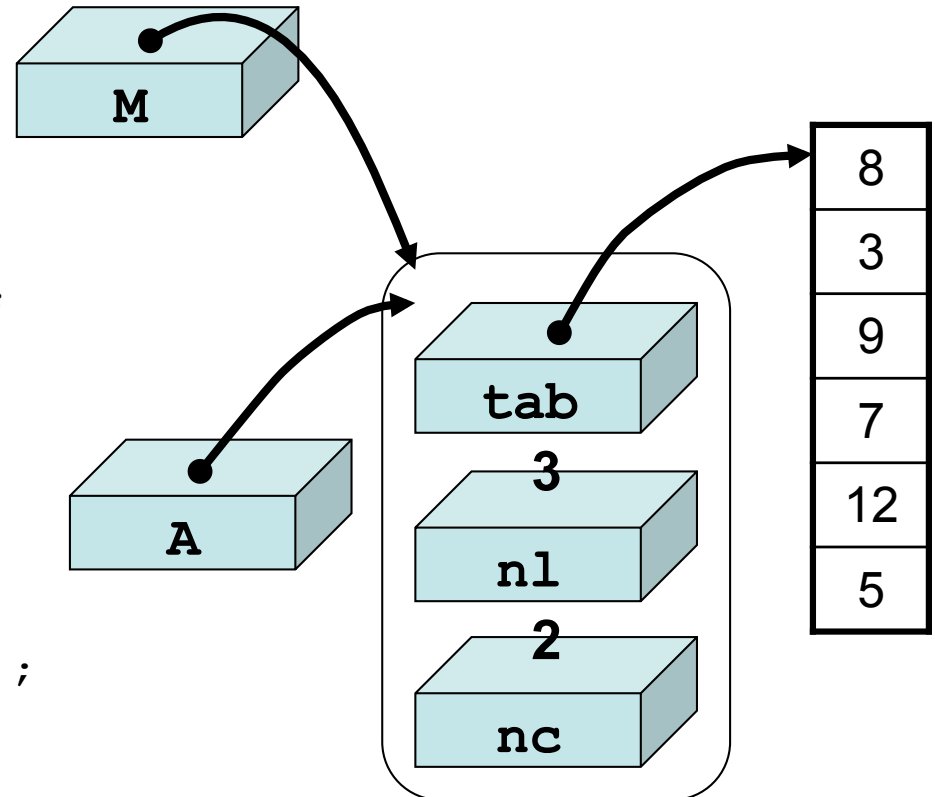
```
→ void affiche(Matrice * M)
{
    for(int i = 0; i < M->nl; i++)
    {
        for(int j = 0; j < M->nc; j++)
            cout << get(M, i, j) << " ";
        cout << endl;
    }
}
```

...

```
Matrice * A = alloue_matrice(3, 2);
```

```
// Initialiser A
```

```
affiche(A);
```



Pas-à-pas

```
float get(Matrice * M, int i, int j)
{
    return M->tab[i * M->nc + j];
}
```

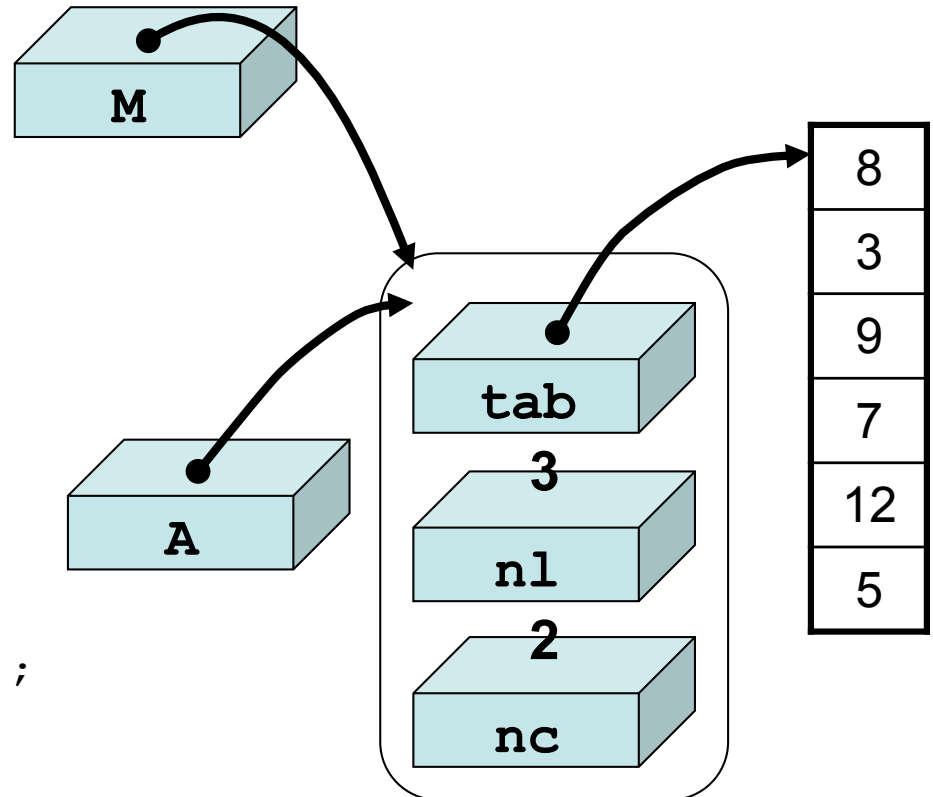
```
void affiche(Matrice * M)
{
    for(int i = 0; i < M->nl; i++)
    {
        for(int j = 0; j < M->nc; j++)
        → cout << get(M, i, j) << " ";
        cout << endl;
    }
}
```

...

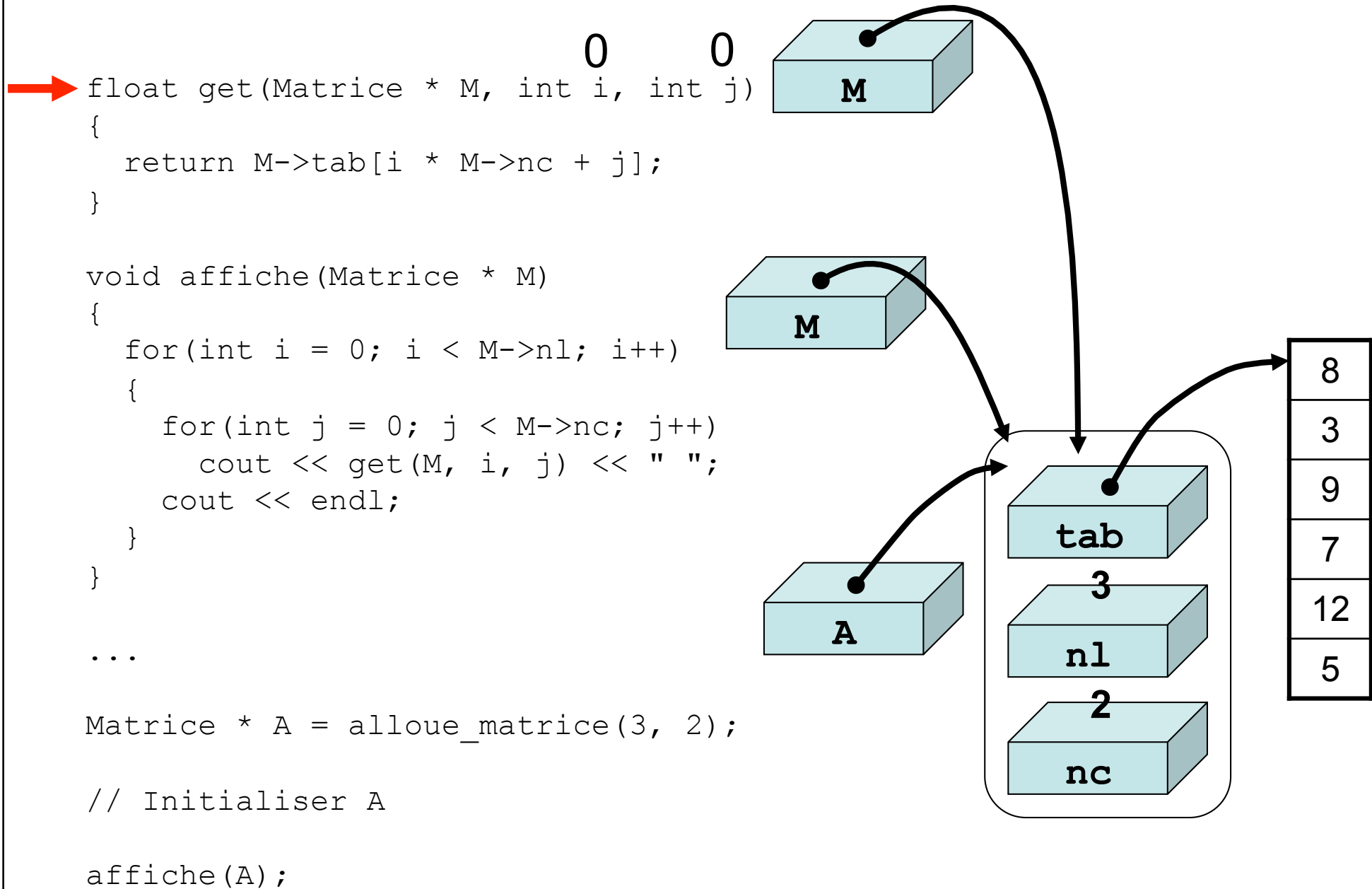
```
Matrice * A = alloue_matrice(3, 2);
```

```
// Initialiser A
```

```
affiche(A);
```



Pas-à-pas



Pas-à-pas

```
float get(Matrice * M, int i, int j)
{
  return M->tab[i * M->nc + j];
}
```

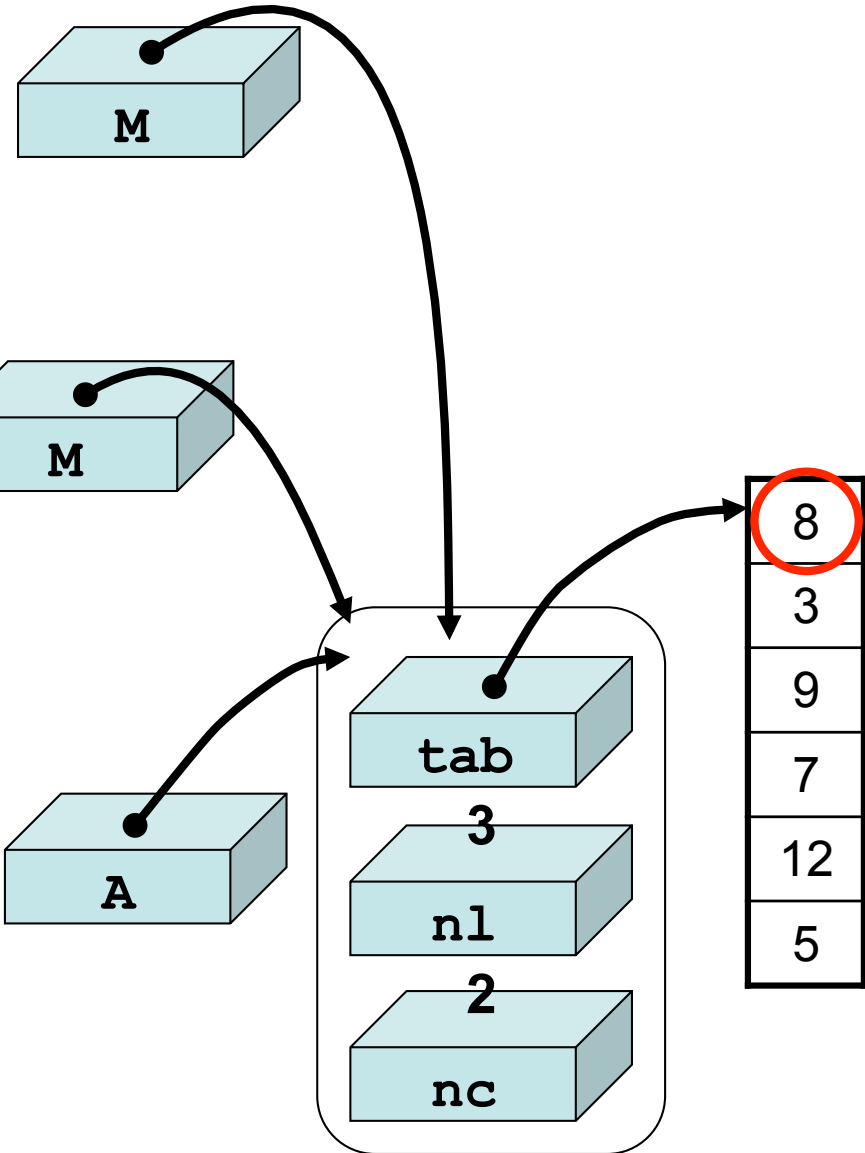
```
void affiche(Matrice * M)
{
  for(int i = 0; i < M->nl; i++)
  {
    for(int j = 0; j < M->nc; j++)
      cout << get(M, i, j) << " ";
    cout << endl;
  }
}
```

...

```
Matrice * A = alloue_matrice(3, 2);
```

```
// Initialiser A
```

```
affiche(A);
```



Pas-à-pas

```
float get(Matrice * M, int i, int j)
{
    return M->tab[i * M->nc + j];
}
```

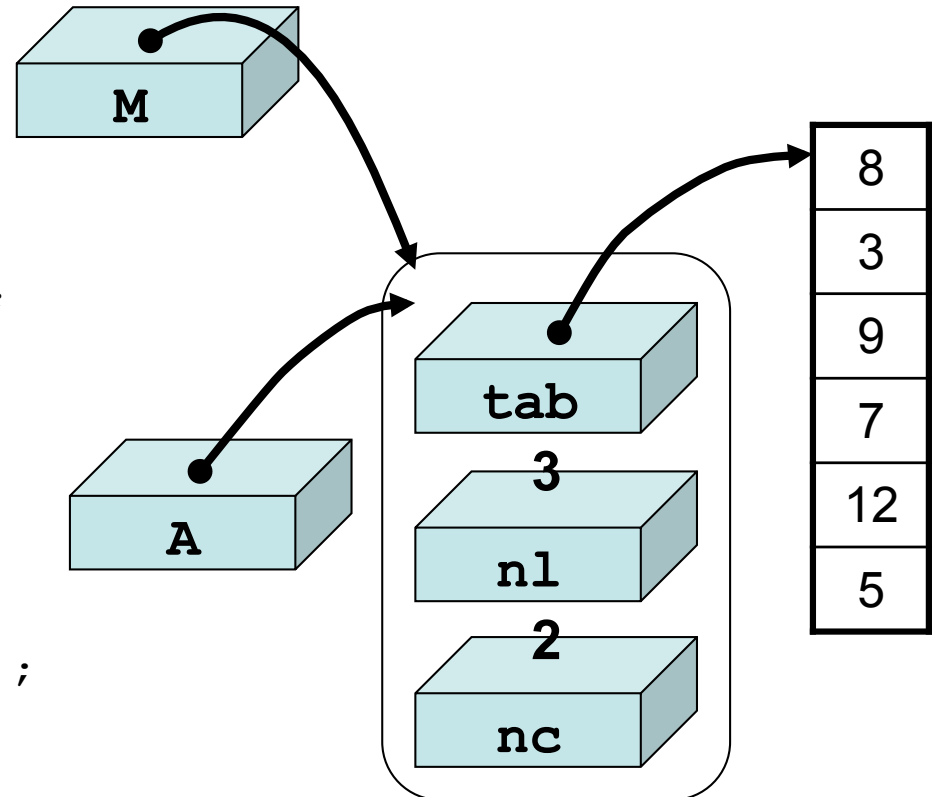
```
void affiche(Matrice * M)
{
    for(int i = 0; i < M->nl; i++)
    {
        for(int j = 0; j < M->nc; j++)
        → cout << get(M, i, j) << " ";
        cout << endl;
    }
}
```

...

```
Matrice * A = alloue_matrice(3, 2);
```

```
// Initialiser A
```

```
affiche(A);
```



Pas-à-pas

```
float get(Matrice * M, int i, int j)
{
    return M->tab[i * M->nc + j];
}
```

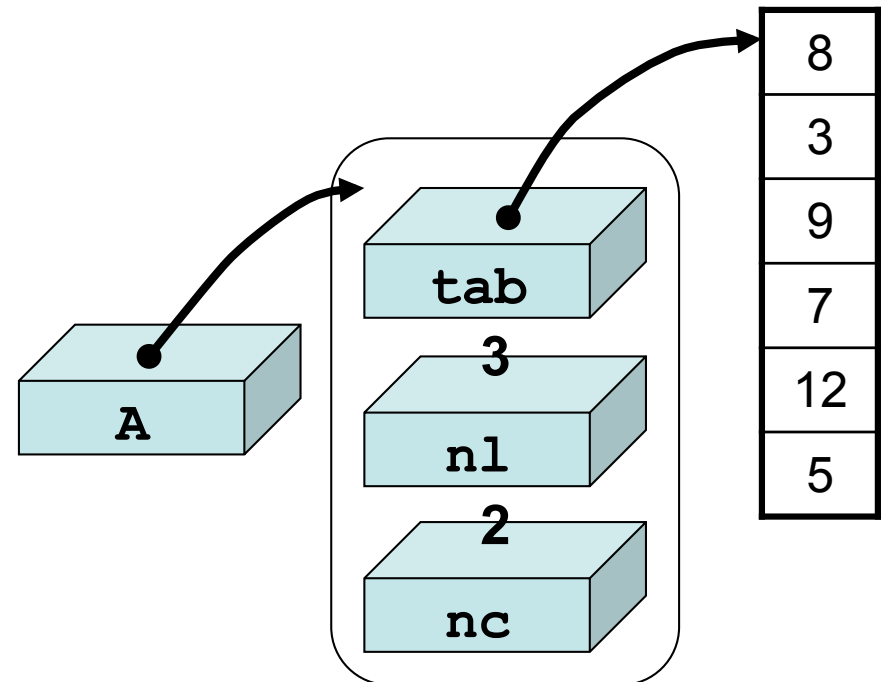
```
void affiche(Matrice * M)
{
    for(int i = 0; i < M->nl; i++)
    {
        for(int j = 0; j < M->nc; j++)
            cout << get(M, i, j) << " ";
        cout << endl;
    }
}
```

...

```
Matrice * A = alloue_matrice(3, 2);
```

```
// Initialiser A
```

```
affiche(A);
```



Fonctions pour accéder aux éléments

Les fonctions `get` et `set` peuvent vérifier que la ligne et la colonne de l'élément auquel on veut accéder sont bien valides:

```
float get(Matrice * M, int i, int j)
{
    if (i < 0)
    {
        cout << "Erreur a l appel de get(Matrice.. i < 0: i = "
              << i << endl;
        return 0.;
    }
    if (i >= M->nl)
    {
        cout << "Erreur a l appel de get(Matrice.. i >= nl: i = "
              << i << endl;
        return 0.;
    }

    ... // Idem for j

    return M->tab[i * M->nc + j];
}
```

Comment définir l'en-tête d'une fonction ?

La fonction `add` qui additionne deux matrices aura l'en-tête suivant:

```
void add(Matrice * A, Matrice * B, Matrice * Res)
```

A la sortie de la fonction `add`, `Res` contiendra la somme de `A` et `B`.

Quelles fonctions écrire ?

Fonctions relatives aux vecteurs:

`affiche`;

`set_0`, qui initialise les éléments d'un vecteur à 0;

`init`, qui permet d'initialiser un vecteur avec des valeurs passées en paramètre

`mult`, qui multiplie un vecteur par un scalaire;

`add`, qui additionne deux vecteurs;

`produit_scalaire`, qui calcule le produit scalaire de deux vecteurs;

...

Fonctions relatives aux matrices:

`affiche`,

`set_0`, qui initialise une matrice à la matrice nulle;

`set_Id`, qui initialise une matrice à la matrice Identité;

`init`, qui permet d'initialiser un vecteur avec des valeurs passées en paramètre

`trace`, qui calcule la trace d'une matrice;

`transpose`, qui transpose une matrice;

`mult`, qui multiplie une matrice par un scalaire;

`add`, qui additionne deux matrices;

`mult`, qui calcule le produit d'une matrice et d'un vecteur;

`mult`, qui calcule le produit de deux matrices;

`inverse`, `determinant`, `valeurs_propres`, ...

...

Exemple d'utilisation de nos fonctions

```
Matrice * M1 = alloue_matrice(2, 2);  
Matrice * M2 = alloue_matrice(2, 2);  
Matrice * R  = alloue_matrice(2, 2);
```

```
init(M1, 1, 2,  
      3, 4);
```

```
init(M2, -2, 1,  
      5, 3);
```

```
cout << "La matrice M1 vaut:" << endl;  
affiche(M1);
```

```
cout << "La trace de M1 vaut " << trace(M1) << endl;
```

```
mult(M1, M2, R);
```

```
cout << "Le produit de M1 et M2 vaut:" << endl;  
affiche(R);
```

M1 est initialisée à $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

M2 est initialisée à $\begin{pmatrix} -2 & 1 \\ 5 & 3 \end{pmatrix}$

Écrivez les fonctions:

- `set_0` qui initialise une matrice à la matrice nulle;
- `set_Id` qui initialise une matrice à la matrice Identité. La fonction `set_Id` retournera:
 - *false* si la matrice passée en paramètre n'est pas carrée, et
 - *true* si l'opération s'est bien passée;
- `init` permettant d'initialiser une matrice.

Il faut écrire une fonction différente pour chaque taille de matrice que vous voudrez initialiser:

```
void init(Matrice * M,  
          float c1, float c2, float c3, float c4)
```

permettra d'initialiser une matrice 2x2;

```
void init(Matrice * M,  
          float c1, float c2, float c3,  
          float c4, float c5, float c6)
```

permettra d'initialiser une matrice 2x3 ou 3x2.

Ces fonctions devront donc tester la valeur de `n1` et `nc` pour gérer les cas, et mettre les paramètres `c1, c2...` au bon endroit.

- `add` qui additionne deux matrices.

Structure Vecteur

De la même façon, on peut définir une structure `Vecteur`:

```
struct Vecteur
{
    float * tab;
    int n;
};
```

On n'a bien sûr ici besoin que d'une valeur (`n`) pour définir la taille du vecteur.

```
Vecteur * alloue_vecteur(int n)
{
    Vecteur * res = new Vecteur;

    res->tab = new float[n];
    res->n = n;

    return res;
}
```

Structure Vecteur

```
struct Vecteur
{
    float * tab;
    int n;
};
```

```
float get(Vecteur * V, int i)
{
    return V->tab[i];
}
```

```
void set(Vecteur * V, int i, float e)
{
    V->tab[i] = e;
}
```

Remarque:

On peut avoir DEUX fonctions nommées `set` (idem pour `get`), l'une pour les matrices l'autre pour les vecteurs. Le compilateur sait faire la différence grâce au type des paramètres.

Fonction affiche pour Vecteur

```
void affiche(Vecteur * V)
{
    for(int i = 0; i < V->n; i++)
        cout << get(V, i) << " ";
    cout << endl;
}
```

```
Matrice * A = alloue_matrice(2, 3);
Vecteur * V = alloue_vecteur(5);
(...)
affiche(A); // affiche pour les matrices
affiche(V); // affiche pour les vecteurs
```

Exercices

Écrivez les fonctions `trace` et `produit_scalaire`:

- `float trace(Matrice * A)`

La trace d'une matrice (nécessairement carrée) est la somme des éléments sur la diagonale.

- `float produit_scalaire(Vecteur * V1, Vecteur * V2)`

Exercices

- Écrivez la fonction `mult` qui multiplie une matrice et un vecteur:

```
void mult(Matrice * A, Vecteur * V, Vecteur * Res)
```

`Res` contiendra le produit de la matrice `A` et du vecteur `V`;

- Écrivez la fonction `mult` qui multiplie deux matrices entre elles:

```
void mult(Matrice * A, Matrice * B, Matrice * Res)
```

`Res` contiendra le produit des matrices `A` et `B`.

Exercices

De la même façon, écrivez les fonctions:

`set_0`, qui initialise les éléments d'un vecteur à 0;

`mult` qui multiplie un vecteur par un scalaire;

`add` qui additionne deux vecteurs;

`transpose`, qui calcule la transposée d'une matrice;

`mult` qui calcule le produit d'une matrice par un scalaire;

`puissance_mat`, qui élève une matrice carrée à la puissance n ;

`determinant`, qui calcule le déterminant d'une matrice;

...

Des fonctions moins mathématiques:

`decale_a_gauche`, qui décale d'une colonne vers la gauche les éléments d'une matrice.

La colonne de gauche sera déplacée sur la colonne de droite;

`decale_a_droite`, qui décale d'une colonne vers la droite les éléments d'une matrice.

La colonne de droite sera déplacée sur la colonne de gauche;