Introduction à la programmation
Travaux pratiques: séance 8
INFO0201-1

B. Baert & F. Ludewig
Bruno.Baert@ulg.ac.be - F.Ludewig@ulg.ac.be

Programme de la séance

- Type constant (variables et pointeurs)
  $\rightarrow$ variables et pointeurs dont les valeurs ne peuvent être modifiées

Programme de la séance

- Type constant (variables et pointeurs)
  $\rightarrow$ variables et pointeurs dont les valeurs ne peuvent être modifiées
- Les pointeurs : suite
  - Opérations sur les pointeurs
  - Pointeurs de pointeurs

Programme de la séance

- Type constant (variables et pointeurs)
  $\rightarrow$ variables et pointeurs dont les valeurs ne peuvent être modifiées
- Les pointeurs : suite
  - Opérations sur les pointeurs
  - Pointeurs de pointeurs
- Les tableaux dynamiques à plusieurs dimensions
  - Tableaux de tableaux
  - Tableaux 2D continus

Rappels : notion de pointeur

- Pointeur = variable qui contient l'adresse d'une autre variable

Rappels : notion de pointeur

- Pointeur = variable qui contient l'adresse d'une autre variable
- Déclaration d'un pointeur : **type** *nom_du_pointeur ;

Rappels : notion de pointeur

- Pointeur = variable qui contient l'adresse d'une autre variable
- Déclaration d'un pointeur : **type** *nom_du_pointeur ;
- Allocation d'une variable : nom_pointeur = **new type** ;

Rappels : notion de pointeur

- Pointeur = variable qui contient l'adresse d'une autre variable
- Déclaration d'un pointeur : **type** *nom_du_pointeur ;
- Allocation d'une variable : nom_pointeur = **new type** ;
- Allocation d'un tableau : nom_pointeur = **new type**[taille] ;

Rappels : notion de pointeur

- Pointeur = variable qui contient l'adresse d'une autre variable
- Déclaration d'un pointeur : **type** *nom_du_pointeur ;
- Allocation d'une variable : nom_pointeur = **new type** ;
- Allocation d'un tableau : nom_pointeur = **new type**[taille] ;
- Libération d'une variable : **delete** nom_pointeur ;

Rappels : notion de pointeur

- Pointeur = variable qui contient l'adresse d'une autre variable
- Déclaration d'un pointeur : **type** *nom_du_pointeur ;
- Allocation d'une variable : nom_pointeur = **new type** ;
- Allocation d'un tableau : nom_pointeur = **new type**[taille] ;
- Libération d'une variable : **delete** nom_pointeur ;
- Libération d'un tableau : **delete[]** nom_pointeur ;

Rappels : notion de pointeur

- Pointeur = variable qui contient l'adresse d'une autre variable
- Déclaration d'un pointeur : **type** *nom_du_pointeur ;
- Allocation d'une variable : nom_pointeur = **new type** ;
- Allocation d'un tableau : nom_pointeur = **new type**[taille] ;
- Libération d'une variable : **delete** nom_pointeur ;
- Libération d'un tableau : **delete[]** nom_pointeur ;
- Relations variables-pointeurs :
  - int a ;
    a est une variable entière
    &a est une adresse mémoire (pointeur)
  - int *b ;
    b est un pointeur (adresse mémoire)
    *b est une variable entière

Type constant

Comme leur nom l'indique, les valeurs des *variables* peuvent être modifiées. On peut cependant vouloir déclarer des variables particulières, dont la valeur est fixée à la première affectation et ne peut plus être modifiée. Il existe pour cela le type spécifique **const**. Une variable qui est déclarée de type **const** doit être initialisée immédiatement (lors de la déclaration) et sa valeur ne pourra être modifée dans le programme, sous peine de générer une erreur dès la compilation.

Type constant

Comme leur nom l'indique, les valeurs des *variables* peuvent être modifiées. On peut cependant vouloir déclarer des variables particulières, dont la valeur est fixée à la première affectation et ne peut plus être modifiée. Il existe pour cela le type spécifique **const**. Une variable qui est déclarée de type **const** doit être initialisée immédiatement (lors de la déclaration) et sa valeur ne pourra être modifée dans le programme, sous peine de générer une erreur dès la compilation.

```
1  int main()
2  {
3    const double PI = 3.14159265358979323846264;
4    const int N = 1000;
5
6    N = 2000; // ERREUR !!!
7  }
```

Type constant : les pointeurs

Dans le cas des pointeurs, deux types de *constance* peuvent être envisagés :

Type constant : les pointeurs

Dans le cas des pointeurs, deux types de *constance* peuvent être envisagés :

- Un pointeur constant : le pointeur pointe toujours vers la même variable, on ne peut **changer l'adresse vers laquelle** il pointe ;

Type constant : les pointeurs

Dans le cas des pointeurs, deux types de *constance* peuvent être envisagés :

- Un pointeur constant : le pointeur pointe toujours vers la même variable, on ne peut **changer l'adresse vers laquelle** il pointe ;
- Un pointeur vers une constante : le pointeur est tel qu'il pointe **vers une variable dont la valeur est constante**. Dans ce cas, c'est la valeur de la variable vers laquelle le pointeur pointe qui ne peut être modifiée.

Type constant : les pointeurs

Dans le cas des pointeurs, deux types de *constance* peuvent être envisagés :

- Un pointeur constant : le pointeur pointe toujours vers la même variable, on ne peut **changer l'adresse vers laquelle** il pointe ;
- Un pointeur vers une constante : le pointeur est tel qu'il pointe **vers une variable dont la valeur est constante**. Dans ce cas, c'est la valeur de la variable vers laquelle le pointeur pointe qui ne peut être modifiée.

On déclare l'un ou l'autre de la manière suivante :

- pointeur constant : type* **const** nom_pointeur ;

Type constant : les pointeurs

Dans le cas des pointeurs, deux types de *constance* peuvent être envisagés :

- Un pointeur constant : le pointeur pointe toujours vers la même variable, on ne peut **changer l'adresse vers laquelle** il pointe ;
- Un pointeur vers une constante : le pointeur est tel qu'il pointe **vers une variable dont la valeur est constante**. Dans ce cas, c'est la valeur de la variable vers laquelle le pointeur pointe qui ne peut être modifiée.

On déclare l'un ou l'autre de la manière suivante :

- pointeur constant : type* **const** nom_pointeur ;
- pointeur vers valeur constante : **const** type * nom_pointeur ;

Type constant : les pointeurs

Dans le cas des pointeurs, deux types de *constance* peuvent être envisagés :

- Un pointeur constant : le pointeur pointe toujours vers la même variable, on ne peut **changer l'adresse vers laquelle** il pointe ;
- Un pointeur vers une constante : le pointeur est tel qu'il pointe **vers une variable dont la valeur est constante**. Dans ce cas, c'est la valeur de la variable vers laquelle le pointeur pointe qui ne peut être modifiée.

On déclare l'un ou l'autre de la manière suivante :

- pointeur constant : type* **const** nom_pointeur ;
- pointeur vers valeur constante : **const** type * nom_pointeur ;
- pointeur constant vers valeur constante :

const type* **const** nom_pointeur ;

Type constant : les pointeurs

```
1  int main()
2  {
3    int n = 42;
4    int k = 361;
5
6    int* p = &n; // pointeur non constant
7    *p = 45; // OK
8    p = &k; // OK
9
10   int* const cp = &a; // pointeur constant
11   *cp = 144; // OK
12   cp = &k; // ERREUR : modification du pointeur constant
13
14   cont int * pc = &a; // pointeur vers une constante
15   *pc = 618; // ERREUR : modification de la valeur constante pointée
16   pc = &k; // OK
17
18   const int const * cpc = &a;
19   *cpc = 12; // ERREUR : modification de la valeur constante pointée
20   cpc = &k; // ERREUR : modification du pointeur constant
21
22   return 0;
23 }
```

Equivalence tableau - pointeur

Un **tableau/pointeur** peut être vu comme un **pointeur/tableau** !

En effet, le nom d'un tableau n'est rien d'autre qu'un pointeur vers le premier élément du tableau.
De même, un pointeur vers une série de variables contigues dans la mémoire correspond à un tableau.
Ainsi, on peut écrire

```
1 int tab1[10]; // 'tab1' est un pointeur
2 int* tab2 = tab1; // 'tab2' est un "tableau"
3
4 tab2[5] = 0; // ceci fonctionne
5
6 *(tab1 + 1) = 12; // ceci aussi ==> tab[1] = 12;
```

Opérations sur les pointeurs

Même si toutes les opérations arithmétiques de base ne sont pas permises avec les pointeurs, les opérations d'incrémentation et décrémentation restent permises, avec une signification légèrement différente, et permettent de parcourir les éléments d'un tableau.

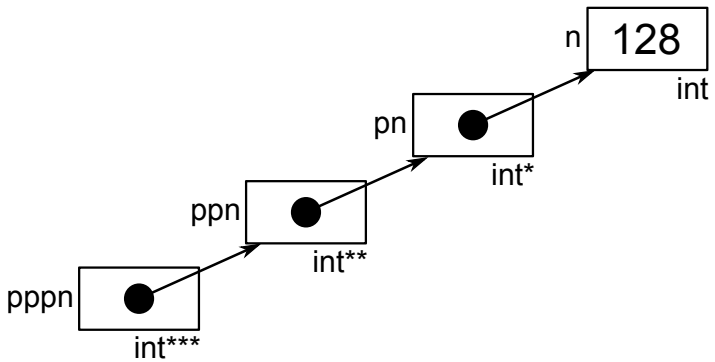
En effet, lorsque ces opérations sont appliquées à des pointeurs, l'ajout de **une unité** signifie "avancer dans la mémoire d'une quantité égale à la taille de l'objet vers lequel pointe le pointeur".

```
1  int* t = new int[100]; // 4325756
2
3  ++t; // 4325756 + 4 = 4325760
4  *t = 0; // t[1] = 0
5
6  t = t + 3; // 4325760 + 3*4 = 4325772
7  *t = 0; // t[1+3] = t[4] = 0
8
9  --t; // 4325772 - 4 = 4325768
10 *t = 0; // t[4-1] = t[3] = 0
```

Pointeurs de pointeurs

Un pointeur est un type de variable. On peut donc imaginer de créer **un pointeur qui pointe vers une variable qui est elle-même un pointeur** vers une autre variable.
On déclare ainsi un **pointeur de pointeur**. On peut même déclarer des pointeurs de pointeurs de pointeurs, et ainsi de suite.

```
1  int n = 128; // variable
2  int* pn = &n; // pointeur
3  int** ppn = &pn; // pointeur vers un pointeur
4  int*** pppn = &ppn; // pointeur de pointeur de pointeur
5  *pppn // --> = ppn
6  **ppn = 32; // n = 32;
7  ***pppn = 64; // --> n = 64;
```

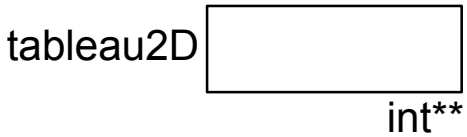


Tableaux de tableaux

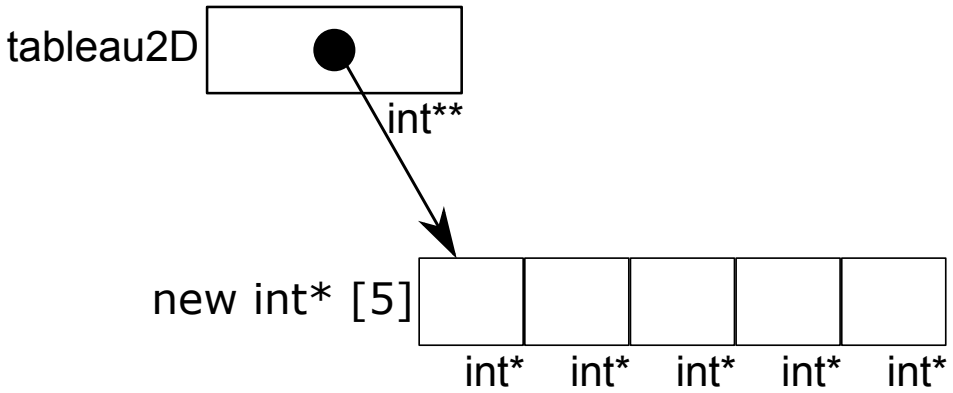
Pour allouer un tableau à 2 dimensions, il suffit de considérer un premier tableau à 1D qui représente par exemple le nombre de lignes du tableau. Dans chaque élément, on alloue un nouveau tableau qui contiendra toutes les cases de la ligne correspondante. On obtient ainsi un tableau de tableaux qui équivaut à un tableau 2D.

```
1  const int NB_LIGNES = 5;
2  const int NB_COLONNES = 4;
3  // Allocation
4  int **tableau2D = new int* [NB_LIGNES];
5  for(int i=0; i < NB_LIGNES; i++)
6  {
7    tableau2D[i] = new int[NB_COLONNES];
8  }
9  // Initialisation
10 for(int i=0; i < NB_LIGNES; i++)
11   for(int j=0; j < NB_COLONNES; j++)
12     tableau2D[i][j] = i*j;
```

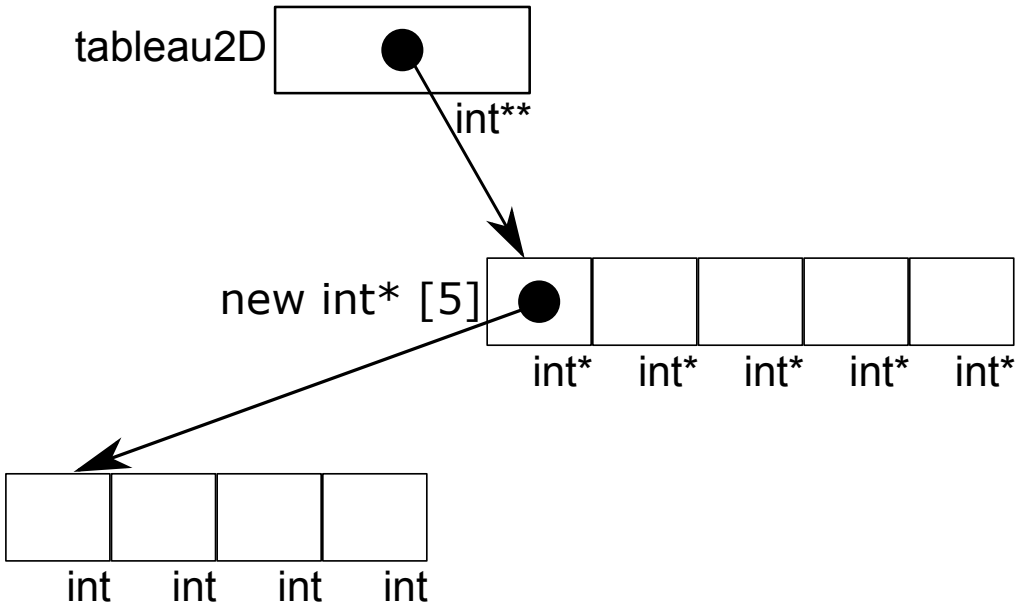
Tableaux de tableaux



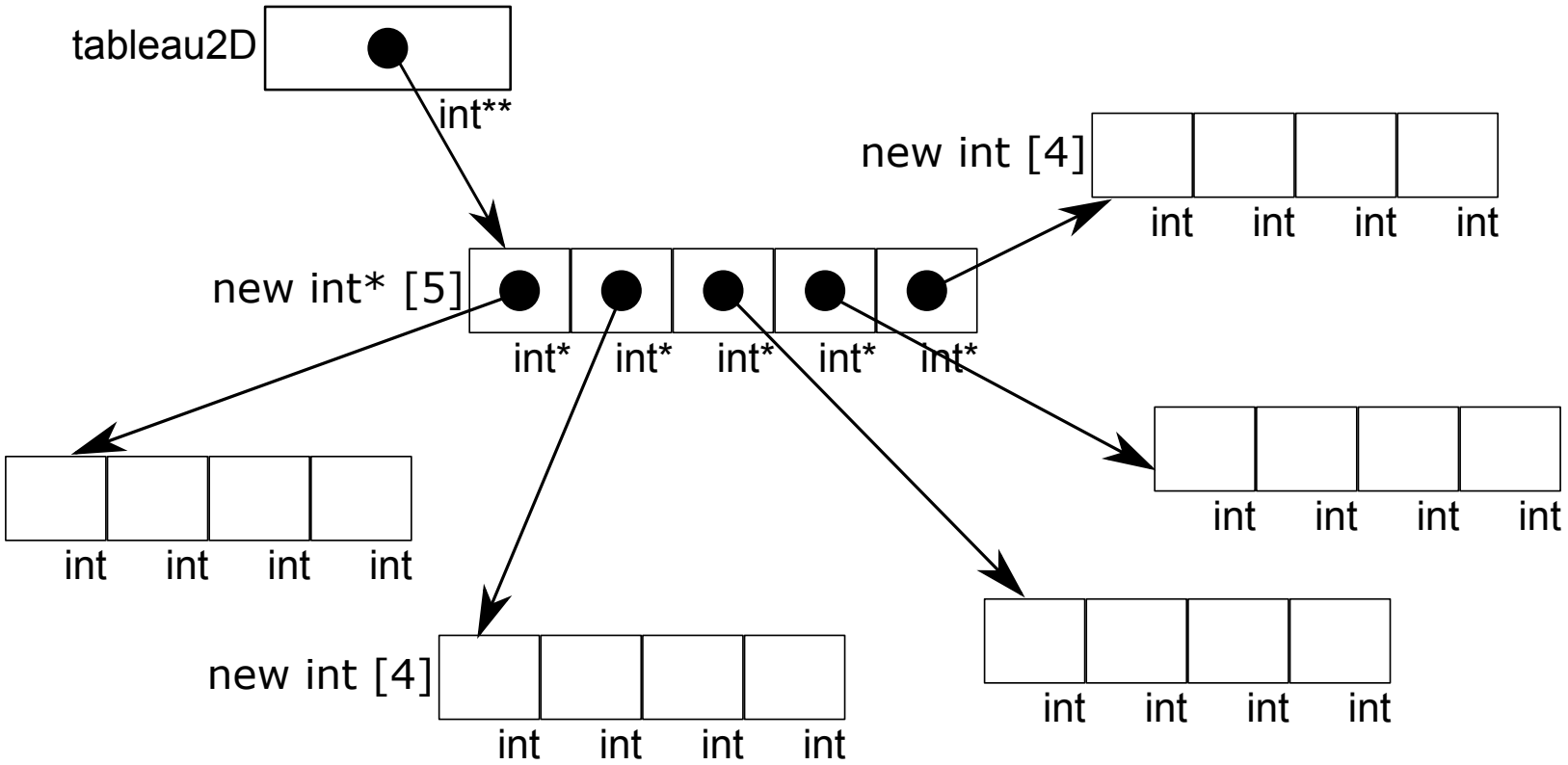
Tableaux de tableaux



Tableaux de tableaux



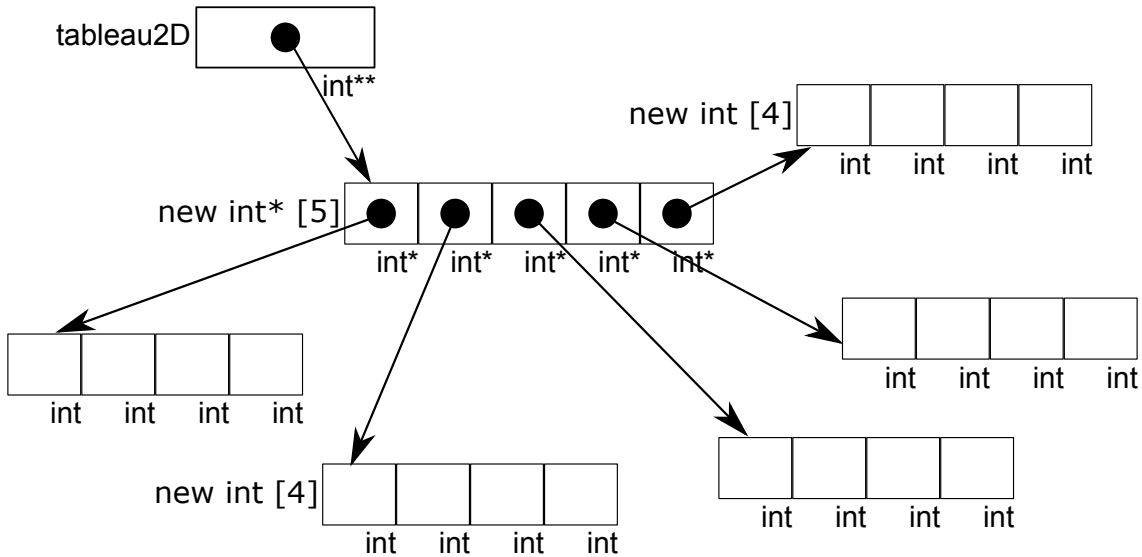
Tableaux de tableaux



Tableaux de tableaux

Pour libérer un tableau de tableaux, il est nécessaire de libérer chacun des tableaux contenus dans le premier tableau

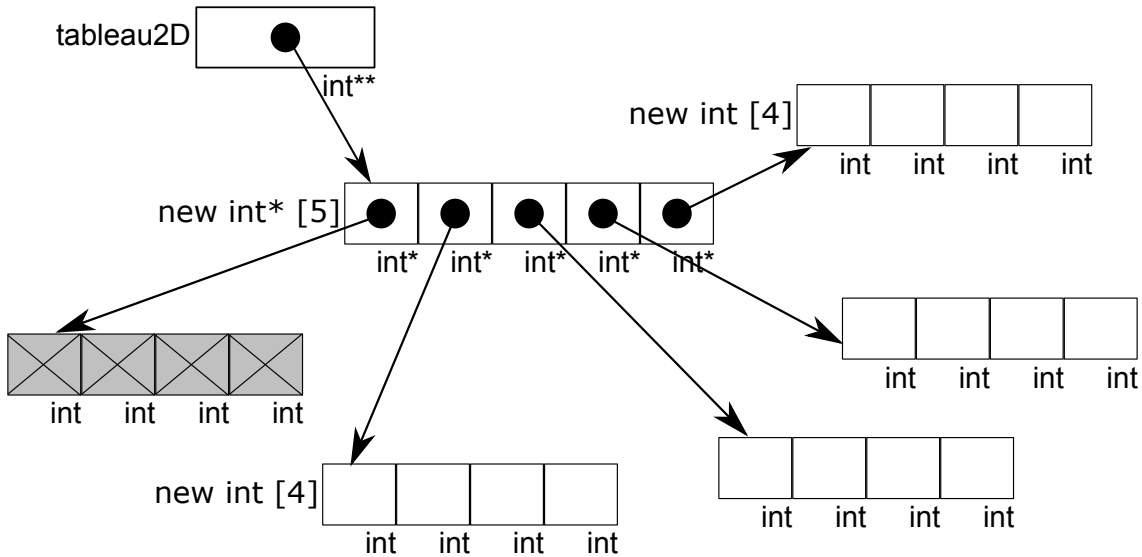
```
1  // Libération de la mémoire
2  for(int i=0; i < NB_LIGNES; i++)
3  {
4    // Chacun des sous-tableaux
5    delete[] tableau2D[i];
6  }
7  // Le tableau conteneur
8  delete[] tableau2D;
```



Tableaux de tableaux

Pour libérer un tableau de tableaux, il est nécessaire de libérer chacun des tableaux contenus dans le premier tableau

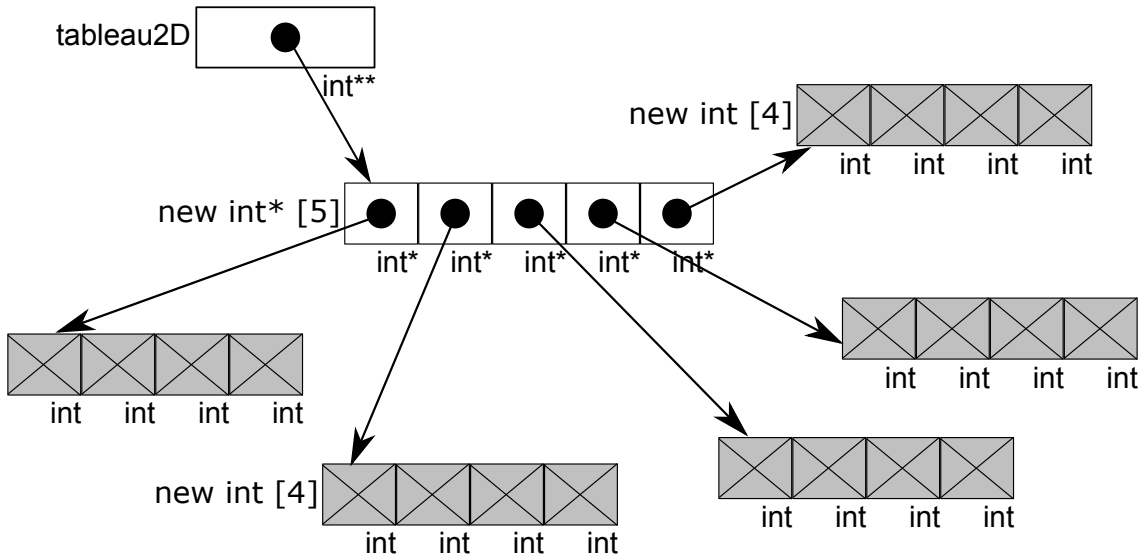
```
1  // Libération de la mémoire
2  for(int i=0; i < NB_LIGNES; i++)
3  {
4    // Chacun des sous-tableaux
5    delete[] tableau2D[i];
6  }
7  // Le tableau conteneur
8  delete[] tableau2D;
```



Tableaux de tableaux

Pour libérer un tableau de tableaux, il est nécessaire de libérer chacun des tableaux contenus dans le premier tableau

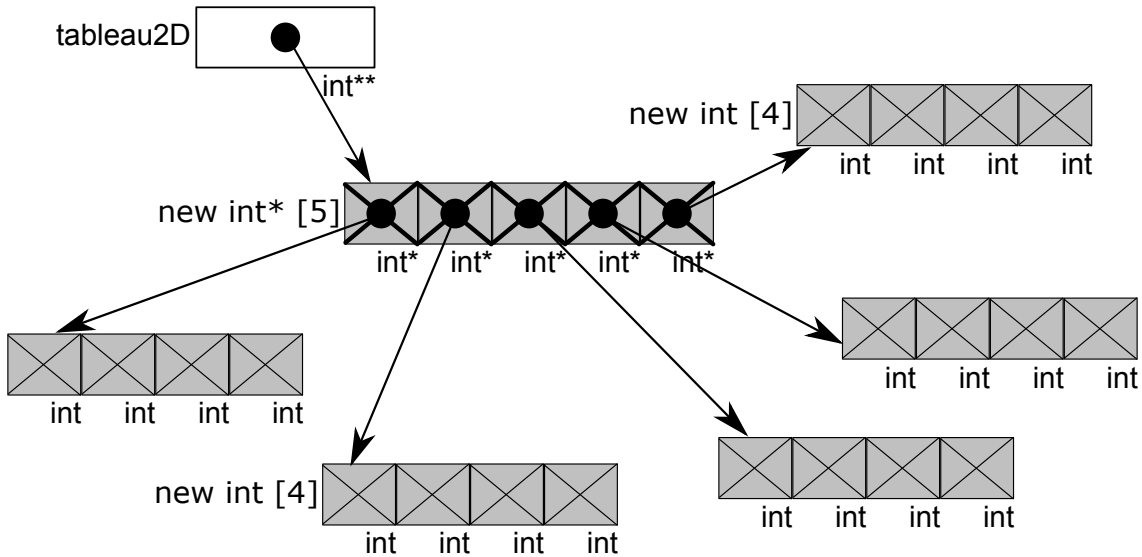
```
1  // Libération de la mémoire
2  for(int i=0; i < NB_LIGNES; i++)
3  {
4    // Chacun des sous-tableaux
5    delete[] tableau2D[i];
6  }
7  // Le tableau conteneur
8  delete[] tableau2D;
```



Tableaux de tableaux

Pour libérer un tableau de tableaux, il est nécessaire de libérer chacun des tableaux contenus dans le premier tableau

```
1  // Libération de la mémoire
2  for(int i=0; i < NB_LIGNES; i++)
3  {
4    // Chacun des sous-tableaux
5    delete[] tableau2D[i];
6  }
7  // Le tableau conteneur
8  delete[] tableau2D;
```



Tableaux dynamiques 2D efficaces

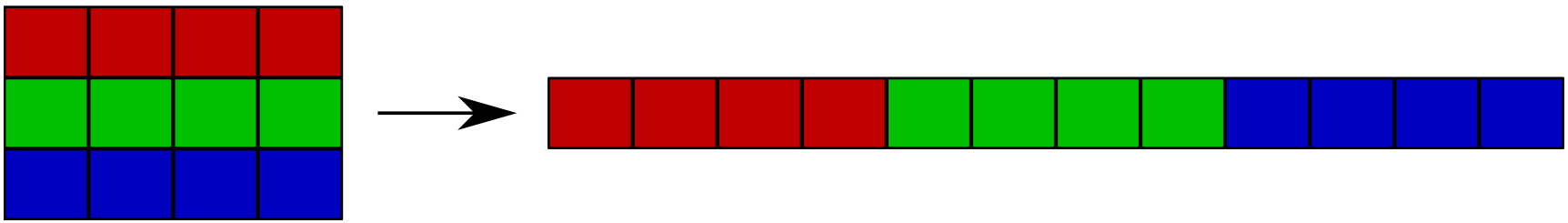
Un inconvénient du tableau de tableaux réside dans le fait que les sous-tableaux ne sont pas nécessairement **placés de manière contigue dans la mémoire**.

Tableaux dynamiques 2D efficaces

Un inconvénient du tableau de tableaux réside dans le fait que les sous-tableaux ne sont pas nécessairement **placés de manière contigue dans la mémoire**.
Si un tel tableau est utilisé pour stocker une matrice et réaliser des calculs, les accès à des endroits différents de la mémoire impliqueront une **perte de performance**.

Tableaux dynamiques 2D efficaces

Un inconvénient du tableau de tableaux réside dans le fait que les sous-tableaux ne sont pas nécessairement **placés de manière contigue dans la mémoire**.
Si un tel tableau est utilisé pour stocker une matrice et réaliser des calculs, les accès à des endroits différents de la mémoire impliqueront une **perte de performance**.
Pour remédier à ce problème, on préfèrera allouer un grand tableau 1D qui aura la même longueur que le nombre total de cellules du tableau 2D correspondant.

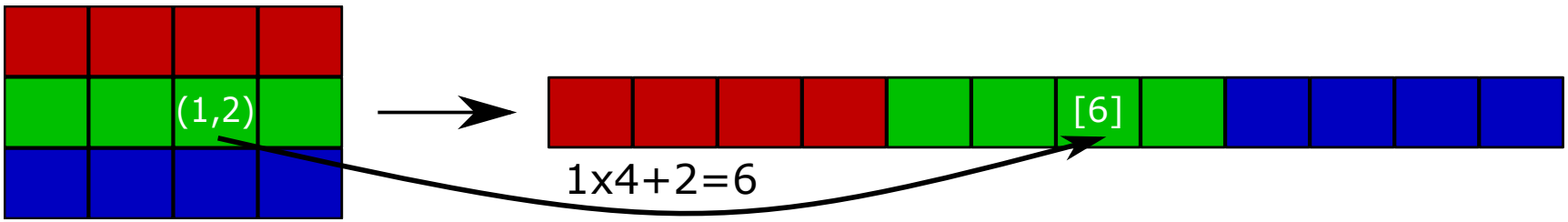


Tableaux dynamiques 2D efficaces : accès aux éléments

Pour accéder aux éléments d'un tel tableau 2D remis en ligne, il faut convertir un couple (i, j) représentant l'élément de la ligne i et de la colonne j en un indice x du tableau 1D.
La relation entre les deux est la suivante :

$$\text{indice}_{\text{tableau1D}} = x = i * \text{NB\_COLONNES} + j$$

où NB_COLONNES est le nombre de colonnes du tableau 2D.



ATTENTION : L'indice du premier élément, de ligne ou de colonne, possède toujours la valeur zéro.

Exercices

1. Transposition d'une matrice carrée :
   - Ecrire un programme qui demande à l'utilisateur de saisir une matrice carrée M de taille variable et la stocker dans un tableau 2D dynamique ;
   - Afficher la matrice transposée M^T dans la console.
2. Produit tensoriel de deux vecteurs :
   Le produit tensoriel de deux tenseurs d'ordre 1 (vecteurs 1D) $u(m \times 1)$ et $v(1 \times n)$ correspond à la matrice $A(m \times n)$, résultat du produit matriciel de u et v^{T} :

   $$A = u \otimes v = uv^{\mathsf{T}}$$

   Ecrire un programme qui
   - demande à l'utilisateur de saisir deux vecteurs de tailles variables (la taille est demandée à l'utilisateur lors de l'exécution) ;
   - calcule le produit tensoriel de ces deux vecteurs dans un tableau dynamique continu dans la mémoire (2e méthode) ;
   - affiche le résultat à la fois à l'écran et dans un fichier nommé "produit_tensoriel.txt" ;

2. Interpolation de Lagrange :
Les polynômes de Lagrange permettent d'interpoler une série de points par un polynôme qui passe exactement par ces points. Si l'on possède $n+1$ points (x_i, y_i), le polynôme de degré le plus petit pour lequel $f(x_i) = y_i$ est de degré n et est égal à

$$P_n(x) = \sum_{j=0}^{n} y_j \left(\prod_{i=0, i\neq j}^{n} \frac{x - x_i}{x_j - x_i} \right)$$

que l'on peut également décomposer comme suit :

$$P_n(x) = \sum_{j=0}^{n} y_j l_j(x), \text{ où } l_j = \prod_{i=0, i\neq j}^{n} \frac{x - x_i}{x_j - x_i}$$

Ecrire un programme qui

- charge des points à partir du fichier "data_interp.dat", disponible à l'adresse `http://spin02.phys.ulg.ac.be/`
- interpole les points par la méthode de Lagrange dans l'intervalle $[-5 : 5]$ avec un pas de 0.01 ;
- ecrit ces points dans un fichier nommé "interpolation.txt" ;

Tracer le résultat avec un programme externe (Matlab, GnuPlot, Excel,...)
Conseil : structurer le programme avec des fonctions pour ne pas répéter inutilement certaines opérations.