



Centre *Informatique* pour les **L**ettres  
et les **S**ciences **H**umaines

## Apprendre C++ avec Qt : Annexe 7

### Tableaux multidimensionnels

1 - Vrais tableaux.....	2
Accès aux éléments .....	2
Initialisation .....	3
Déterminer le nombre d'éléments .....	3
Transmission à une fonction .....	4
Conclusion .....	5
2 - Un pointeur sur des tableaux.....	5
Création .....	6
Destruction .....	6
Conclusion .....	6
3 - Un tableau de pointeurs.....	7
Création et utilisation.....	7
Destruction .....	8
Transmission à une fonction .....	8
Conclusion .....	9
4 - Un pointeur sur des pointeurs .....	9
Création et utilisation.....	9
Destruction .....	10
Conclusion .....	10
5 - Bilan général.....	10

Bien que le langage C++ propose de meilleurs moyens pour structurer les données, il peut arriver que l'on rencontre encore des tableaux multidimensionnels "à la C". Une certaine familiarité avec ce genre d'objets<sup>1</sup> s'avère alors précieuse.

## 1 - Vrais tableaux

Il est très facile de créer un vrai tableau comportant deux (ou plus de deux) dimensions. La syntaxe utilisée est semblable à celle employée dans le cas des tableaux à une seule dimension, il convient simplement d'indiquer, pour chaque dimension, la taille que le tableau doit adopter.

```
1 char matrice[3][5]; //un tableau de trois lignes et cinq colonnes
```

### Accès aux éléments

L'accès aux éléments d'un tel tableau nécessite l'usage d'autant d'index qu'il y a de dimensions. Le fragment de code suivant annule tous les éléments du tableau créé ci-dessus :

```
2 int ligne, colonne;
3 for(ligne = 0 ; ligne < 3 ; ++ligne)
4     for(colonne = 0 ; colonne < 5 ; ++colonne)
5         matrice[ligne][colonne] = 0;
```

La notation employée pour accéder à un élément d'un tableau multidimensionnel est agréablement simple d'emploi, mais il n'est peut être pas inutile de s'attarder quelques instants sur le mécanisme qui sous-tend cette simplicité. Rappelons que la notation indexée n'est qu'un moyen de déréférencer un pointeur. Comment, dans ces conditions, l'expression

```
matrice[ligne][colonne]
```

peut-elle désigner une zone de mémoire destinée à stocker un char ?

On peut analyser cette expression en remarquant qu'elle n'est que le **déréférencement** d'une **première expression**. Pour que ce déréférencement permette d'accéder à une zone de mémoire destinée à stocker un char, il faut que l'évaluation de la première expression produise un **pointeur sur char**. Deux hypothèses sont donc envisageables :

- soit `matrice` est un tableau de pointeurs sur char, et `matrice[ligne]` en est simplement l'un des éléments,
- soit `matrice[ligne]` est un tableau de char (nous savons, en effet, que lorsque le nom d'un tableau figure dans une expression, celle-ci est évaluée en remplaçant ce nom par un pointeur sur le premier élément du tableau en question).

C'est cette seconde hypothèse qui est la bonne : la création de `matrice` s'est traduite par la réservation d'un bloc de mémoire destiné à stocker 15 caractères, et non des adresses (ce qui serait nécessaire si un tableau de pointeurs était impliqué dans cette affaire). L'expression `matrice[ligne]` désigne effectivement un tableau de cinq char, qui correspond à la ligne de `matrice` dont l'index est égal à `ligne`. Si l'on suppose que la représentation de `matrice` débute à l'adresse `n`, l'organisation des informations en mémoire peut être représentée ainsi :

adresse	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+12	n+13	n+14
noms	matrice														
	matrice[0] alias "la première ligne"					matrice[1] alias "la deuxième ligne"					matrice[2] alias "la troisième ligne"				
	matrice[0][0]	matrice[0][1]	matrice[0][2]	matrice[0][3]	matrice[0][4]	matrice[1][0]	matrice[1][1]	matrice[1][2]	matrice[1][3]	matrice[1][4]	matrice[2][0]	matrice[2][1]	matrice[2][2]	matrice[2][3]	matrice[2][4]

Représentation en mémoire d'un vrai tableau de char de trois lignes et cinq colonnes

La façon dont C++ s'y prend pour accéder aux éléments d'un vrai tableau impose donc que les différentes lignes qui composent celui-ci soient représentées dans une unique zone de mémoire, où elles apparaissent dans leur ordre naturel. Cette organisation ouvre la possibilité d'accéder de façon fiable à tous les éléments du tableau, via un simple pointeur sur le premier d'entre eux. En effet, après

<sup>1</sup> Ou, au moins, un bon texte de référence...

```
char matrice[3][5];
char * ptr = & matrice[0][0];
```

l'expression

```
matrice[ligne][colonne]
```

est rigoureusement équivalente à

```
ptr[(5 * ligne) + colonne]
```

comme on peut facilement s'en convaincre en examinant le tableau ci-dessus.

### Initialisation

L'initialisation d'un tel tableau peut se faire d'une façon analogue à celle employée pour les tableaux à une seule dimension :

```
char matrice[3][5] = { {0, 0, 0, 0, 0}, //première ligne de la matrice
                      {1, 1, 1, 1, 1}, //deuxième ligne de la matrice
                      {2, 2, 2, 2, 2} }; //troisième ligne de la matrice
```

Les accolades internes (qui délimitent les lignes) sont optionnelles lorsque toutes les valeurs initiales sont spécifiées. Leur présence améliore toutefois l'intelligibilité du code, et sera indispensable dans le cas d'une initialisation partielle telle que

```
char matrice[3][5] = { {0, 0}, //seules les deux premières
                      {1, 1}, //colonnes sont initialisées
                      {2, 2} };
```

Lorsque toutes les lignes sont (au moins partiellement) initialisées, le compilateur peut déterminer lui-même le **nombre de lignes**. Notre matrice peut donc aussi être définie ainsi :

```
char matrice[][5] = { {0, 0, 0, 0, 0},
                     {1, 1, 1, 1, 1},
                     {2, 2, 2, 2, 2} };
```

Le compilateur est en revanche incapable de déterminer lui-même le nombre de colonnes (ou la taille des autres dimensions, dans le cas d'une matrice à plus de deux dimensions).

Cette incapacité découle de la nécessité de garantir que le tableau peut non seulement être défini, mais également simplement déclaré (ce qui est parfois nécessaire). Dans le cas d'une déclaration, les valeurs d'initialisation ne figurent évidemment pas (aucune variable n'étant créée, le compilateur ne saurait qu'en faire). L'absence du nombre de lignes n'empêche pas le compilateur de savoir comment il doit évaluer une expression du type

```
matrice[ligne][colonne]
```

puisque celle-ci peut, comme nous l'avons vu, être réduite à une expression qui n'implique pas le nombre de lignes :

```
ptr[(ligne * nbColonnes) + colonnes]
```

Le nombre de colonnes (ou la taille de toutes les autres dimensions, dans le cas d'une matrice à plus de deux dimensions) est, en revanche, **indispensable** au compilateur pour calculer l'adresse d'un élément, et une déclaration telle que

```
char matrice[3][];
```

ne suffirait donc pas à permettre la compilation du fichier dans laquelle elle figurerait.

### Déterminer le nombre d'éléments

L'opérateur `sizeof()` peut être appliqué à un tableau multidimensionnel, mais les calculs nécessaires pour retrouver la structure du tableau se compliquent un peu lorsque le nombre de dimensions augmente. Si `matrice` est un tableau à deux dimensions, on peut écrire :

```
1 int tailleTotale = sizeof(matrice);
2 int tailleLigne = sizeof(matrice[0]);
3 int nbLignes = tailleTotale / tailleLigne;
4 int tailleElement = sizeof(matrice[0][0]);
5 int nbColonnes = tailleLigne / tailleElement;
```

Ce calcul repose sur le fait qu'un tableau à deux dimensions peut être vu comme un tableau à une dimension, dont chaque élément est lui-même un tableau à une dimension. Si l'on analyse `matrice` de cette façon, la **première ligne** s'appelle effectivement `matrice[0]`, et le nombre d'éléments (c'est à dire le nombre de lignes) s'obtient, comme d'habitude, en divisant la **taille du tableau** par celle de son **premier élément**. Le nombre de colonnes est obtenu, non moins logiquement, en divisant la taille d'une ligne par la **taille d'un élément**.

## Transmission à une fonction

Nous avons vu qu'une fonction devant accéder à un tableau unidimensionnel reçoit simplement l'adresse du premier élément de celui-ci :

```
1 char tabUniDim[10];
2 maFonction(tabUniDim); //équivalent à maFonction(& (tabUniDim[0]) );
```

Le même principe s'applique dans le cas des tableaux multidimensionnels :

```
1 char matrice[3][5];
2 maFonction(matrice); //équivalent à maFonction( & (matrice[0]) );
```

Remarquez que `matrice[0]` n'est pas à proprement parler le premier des éléments de `matrice`, puisque ceux-ci sont des `char` (parmi lesquels le titre de premier revient à `matrice[0][0]`), alors que `matrice[0]` est, comme nous l'avons vu, un tableau de cinq `char`.

Lors de la transmission d'un vrai tableau multidimensionnel à une fonction, la valeur effectivement transmise est l'adresse du premier sous-tableau, et non celle du premier élément.

Analysons l'exemple d'un tableau à deux dimensions devant être transmis à une fonction chargée de faire la somme de ses éléments. Si nous considérons ce tableau comme un tableau (unidimensionnel) de lignes, la fonction `calculeSomme()` va recevoir un pointeur sur une ligne et, si elle veut pouvoir utiliser ce pointeur comme un (faux) tableau pour accéder aux autres lignes, il lui faut connaître le nombre de lignes effectivement présentes dans ce tableau. Un fragment de code appelant la fonction `calculeSomme()` ressemblera donc à :

```
1 int matrice[][5] = { {0, 0, 0, 0, 0},
2                     {1, 1, 1, 1, 1},
3                     {2, 2, 2, 2, 2} };
4 int tailleTotale = sizeof(matrice);
5 int tailleLigne = sizeof(matrice[0]);
6 int nbLignes = tailleTotale / tailleLigne;
7 int somme = calculeSomme(matrice, nbLignes);
```

La seule véritable difficulté syntaxique concerne le premier paramètre de la fonction : la déclaration d'un pointeur sur un tableau exige l'utilisation de **parenthèses**.

```
1 int calculeSomme(int (*tab)[5], int nbLignes)
2 {
3     const int NB_COLONNES = sizeof(tab[0])/sizeof(tab[0][0]);
4     int ligne;
5     int colonne;
6     int resultat = 0;
7     for(ligne = 0 ; ligne < nbLignes ; ++ligne)
8         for (colonne = 0 ; colonne < NB_COLONNES; ++colonne)
9             resultat += tab[ligne][colonne];
10    return resultat;
11 }
```

Sans ces **parenthèses**, la déclaration serait celle d'un tableau de pointeurs, et non celle d'un pointeur sur un tableau : puisque `int tableau[10]` définit un "tableau d'int" susceptible de stocker 10 entiers, il est clair que `int * tab[5]` ne pourrait définir qu'un "tableau de pointeurs sur int" pouvant stocker 5 adresses.

Remarquez que, comme le paramètre `tab` est de type "pointeur sur un vrai tableau de cinq int", l'expression `tab[0]` désigne ce vrai tableau et peut donc utilement faire l'objet d'un appel à `sizeof()`. La ligne 3 recalcule une constante déjà connue, dans le seul but de lui attribuer un **nom** qui améliore la lisibilité et la maintenabilité du corps de la fonction.

Comme la position des parenthèses nécessaires à la déclaration d'un pointeur sur un tableau n'apparaît pas comme une évidence à tous les programmeurs, ceux-ci ont parfois recours à des notations alternatives telles que

```
int calculeSomme(int tab[][5]);
```

ou même

```
int calculeSomme(int tab[3][5]);
```

En dépit des apparences, ces deux notations n'expriment rien d'autre que le passage d'un pointeur sur un vrai tableau unidimensionnel, et le confort d'écriture n'est donc obtenu qu'au prix d'un gros mensonge sur la nature du paramètre. Un programmeur qui se laisserait abuser par ce mensonge et tenterait, par exemple, d'utiliser l'opérateur `sizeof()` pour recalculer le nombre de lignes s'exposerait à de graves déconvenues.

En plus de cette difficulté syntaxique, la transmission de vrais tableaux multidimensionnels pose un problème de généralité des fonctions qui les reçoivent. En effet, quelle que soit la notation adoptée, la déclaration de ce paramètre mentionne nécessairement le nombre de colonnes de la matrice (ainsi que la taille de toutes les autres dimensions, s'il s'agit d'un tableau à plus de deux dimensions). Il en résulte que deux tableaux à deux dimensions n'ayant pas le même nombre de colonnes ne peuvent pas être traités par la même fonction.

Comme la fonction reçoit un pointeur sur un sous-tableau, elle ne peut être utilisée que pour des tableaux composés de sous-tableaux de même type, et la taille des dimensions (ainsi, bien entendu, que leur nombre) fait partie du type d'un tableau.

### Conclusion

Outre la rigidité liée au fait que toutes les dimensions d'un vrai tableau doivent être fixées lors de l'écriture du programme, les vrais tableaux multidimensionnels présentent donc aussi l'inconvénient de ne pas se prêter à la mise au point de fonctions facilement réutilisables. Pour s'affranchir de ces limitations, il est bien entendu possible de recourir à des pointeurs qui, tout en étant d'un usage bien plus souple, permettent d'obtenir des effets analogues. Il existe plusieurs façons de simuler la présence d'un tableau multidimensionnel à l'aide de pointeurs car, s'agissant de tableaux de tableaux, tous les panachages de vrais et de faux sont envisageables. Si l'on s'en tient aux tableaux à deux dimensions, trois options s'offrent à nous :

- un faux tableau de vrais tableaux (c'est à dire un pointeur sur des tableaux),
- un vrai tableau de faux tableaux (c'est à dire un tableau de pointeurs) et
- un faux tableau de faux tableaux (c'est à dire un pointeur sur des pointeurs).

Une fois le principe compris, la généralisation de la démarche au cas des faux tableaux ayant plus de deux dimensions ne devrait pas vous poser trop de problèmes.

## 2 - Un pointeur sur des tableaux

La première façon d'envisager la création d'un faux tableau multidimensionnel est de stocker les adresses de vrais tableaux dans un faux tableau : la structure ainsi réalisée correspond directement à celle dont dispose une fonction à laquelle on passe un vrai tableau multidimensionnel. Si nous utilisons un pointeur nommé `ptrSurTab` pour créer un faux tableau de trois lignes et cinq colonnes, l'organisation des données en mémoire peut être représentée ainsi :

Nom	<code>ptrSurTab</code>
Type	pointeur sur "tableau de cinq char"
Contenu	l'adresse n

adresse	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11	n+12	n+13	n+14
	<code>ptrSurTab[0]</code>					<code>ptrSurTab [1]</code>					<code>ptrSurTab [2]</code>				
noms	<code>ptrSurTab[0][0]</code>	<code>ptrSurTab[0][1]</code>	<code>ptrSurTab[0][2]</code>	<code>ptrSurTab[0][3]</code>	<code>ptrSurTab[0][4]</code>	<code>ptrSurTab[1][0]</code>	<code>ptrSurTab[1][1]</code>	<code>ptrSurTab[1][2]</code>	<code>ptrSurTab[1][3]</code>	<code>ptrSurTab[1][4]</code>	<code>ptrSurTab[2][0]</code>	<code>ptrSurTab[2][1]</code>	<code>ptrSurTab[2][2]</code>	<code>ptrSurTab[2][3]</code>	<code>ptrSurTab[2][4]</code>

La similarité avec le cas d'un vrai tableau est évidente, puisque la zone de mémoire où sont stockés les éléments du tableau est organisée exactement de la même façon. La seule nuance est que nous n'accédons plus à cette zone en utilisant son nom (elle n'en a pas) mais en déréférençant `ptrSurTab`, qui contient son adresse.

Il s'agit là d'une nuance assez subtile puisque, de toute façon, le compilateur s'empresse de remplacer le nom d'un tableau par une adresse dès qu'il s'agit d'accéder aux éléments...

## Création

Par rapport au cas d'un vrai tableau, l'introduction d'un pointeur rendu valide grâce à un appel à `new[]` offre toutefois un avantage important : le nombre de lignes du tableau n'a plus à être connu au moment de la compilation, mais peut n'être déterminé qu'au moment de l'exécution. S'il existe une fonction `demandeNombreLignes()` chargée d'instaurer un dialogue permettant à l'utilisateur d'indiquer le nombre de lignes souhaitées, nous écrirons donc :

```
1 char (*ptrSurTab)[5]; //un pointeur sur "tableau de cinq char"
2 int nbLignes = demandeNombreLignes();
3 ptrSurTab = new char[nbLignes][5];
```

Il est important de bien comprendre que la **taille du sous-tableau** dont notre pointeur va contenir l'adresse est partie intégrante du type du pointeur. Un pointeur sur vrais tableaux ne permet donc pas de réaliser des faux tableaux dont le nombre de colonnes (ou la taille des autres dimensions, le cas échéant) serait déterminé lors de l'exécution.

Il s'agit d'un faux tableau composé de vrais tableaux. Seul le nombre de vrais tableaux est donc "dynamique", la taille de ces vrais tableaux doit être connue lors de la compilation.

La syntaxe permettant d'utiliser `new[]` pour réserver une zone de mémoire qui sera manipulée par l'intermédiaire d'un "pointeur sur vrais tableaux" ressemble d'assez près à la définition d'un vrai tableau : deux couples de crochets entourent simplement les tailles des deux dimensions. Cette ressemblance est toutefois assez superficielle, car, lorsqu'on écrit

```
char vraiTableau[3][5];
```

le type de la variable est "(vrai) tableau de char comportant trois lignes et cinq colonnes", alors que, dans le cas de

```
new char[3][5];
```

on demande à `new[]` une zone de mémoire permettant de stocker **trois** objets, et il se trouve que ces objets sont de type "(vrai) tableau de cinq char"

D'un certain point de vue, il aurait peut-être été préférable que le langage C++ distingue plus clairement le **nombre d'objets créés** du **type de ces objets**. Une écriture telle que

```
new (char[5]) [3]; //de la place pour 3 "tableaux de 5 char"
```

serait, certes, plus régulière par rapport au cas unidimensionnel

```
new char [3]; //de la place pour 3 "char"
```

mais elle présenterait le défaut rédhibitoire d'inverser l'ordre ligne/colonne par rapport à la définition d'un vrai tableau, ce qui serait un vrai cauchemar.

## Destruction

Etant donné que la création de notre faux tableau s'est appuyée sur un appel à `new[]`, il est bien entendu nécessaire, lorsque le tableau cesse d'être utile, de libérer la mémoire qui lui a été ainsi réservée. Comme nous venons de le voir, `new[]` n'a en fait créé qu'un faux tableau unidimensionnel, dont l'apparence multidimensionnelle provient du fait que ses éléments sont des (vrais) tableaux. Du point de vue de la gestion de la mémoire il s'agit donc d'un simple tableau unidimensionnel, et sa destruction nécessite simplement l'application de l'opérateur `delete[]` à un pointeur contenant l'adresse de la zone à libérer. Le faux tableau que nous avons créé ci-dessus peut donc être détruit ainsi :

```
delete[] ptrSurTab;
```

## Conclusion

Si l'approche "pointeur sur tableaux" présente l'inconvénient de ne rendre dynamique que le nombre de lignes du faux tableau créé, elle présente en revanche l'avantage évident de donner naissance à des objets pouvant être traités par les mêmes fonctions que les vrais tableaux dotés du même nombre de colonnes.

Un autre inconvénient du recours à cette approche est que, du fait de sa ressemblance avec celle permettant la création d'un vrai tableau, la syntaxe `new[3][5]` est aussi facile à utiliser que prompt à faire naître des contresens dans les esprits innocents.

### 3 - Un tableau de pointeurs

Lorsqu'on dispose de plusieurs faux tableaux du même type, il est possible de stocker leurs adresses dans un (vrai) tableau. Imaginons, par exemple que nous disposions de trois pointeurs sur char nommés p0, p1 et p2. Il est facile d'utiliser new[ ] pour faire pointer chacun d'entre eux sur une zone de mémoire capable d'accueillir cinq valeurs de type char :

```
1 const int NB_COLONNES = 5;
2 char * p0 = new char [NB_COLONNES]; //allocation d'une première ligne
3 char * p1 = new char [NB_COLONNES]; //allocation d'une deuxième ligne
4 char * p2 = new char [NB_COLONNES]; //allocation d'une troisième ligne
```

Pour regrouper les adresses contenues dans ces pointeurs, il faut un **tableau de trois pointeurs sur char**. En supposant que les allocations précédentes ont réussi, nous écrirons donc :

```
5 char * tabDePtr[3] = {p0, p1, p2};
```

La beauté de la chose est qu'un élément de ce tableau (tabDePtr[1], par exemple) est un pointeur sur char contenant l'adresse (p1, en l'occurrence) d'une zone réservée en vue du stockage de cinq char. Si ce **pointeur** est **déréférencé** à l'aide de l'opérateur [ ], l'expression ainsi créée désigne l'une de ces cinq données de type char, et nous pouvons écrire :

```
tabDePtr[1][3] = 12; //équivalent à p1[3] = 12;
```

#### Création et utilisation

La création de ce faux tableau (ou d'une matrice comportant beaucoup plus de trois lignes) peut être obtenue plus directement, puisque les pointeurs p1, p2 et p3 ne servent qu'à rendre bien visibles les différentes étapes du processus de création et peuvent donc être supprimés :

```
1 const int NB_LIGNES = 3;
2 const int NB_COLONNES = 5;
3 char * tabDePtr[NB_COLONNES]; //un tableau de pointeurs sur char
4 int ligne;
5 for (ligne = 0 ; ligne < NB_LIGNES ; ++ligne)
6 {
7     tabDePtr[ligne] = new char[NB_COLONNES];
8     if (tabDePtr[ligne] == NULL)
9         //il faut ici prendre les mesures nécessaires
10 }
```

On peut ensuite accéder aux éléments exactement comme s'il s'agissait d'un vrai tableau :

```
11 int colonne;
12 for (ligne = 0 ; ligne < 3 ; ++ligne)
13     for (colonne = 0 ; colonne < 5 ; ++colonne)
14         tabDePtr[ligne][colonne] = 0;
```

Bien que ce fragment de code ressemble étrangement à celui proposé page 2, l'organisation de l'information en la mémoire est ici tout à fait différente. Elle peut être représentée ainsi :

adresse	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11
nom		tabDePtr[0]				tabDePtr[1]				tabDePtr[2]		
contenu		l'adresse A0				l'adresse A1				l'adresse A2		

*tabDePtr est un tableau de trois adresses (chacune occupe quatre octets)*

adresse	A0	A0+1	A0+2	A0+3	A0+4
nom	tabDePtr[0][0]	tabDePtr[0][1]	tabDePtr[0][2]	tabDePtr[0][3]	tabDePtr[0][4]

*L'adresse A0 est celle d'une zone réservée au stockage de cinq char*

adresse	A1	A1+1	A1+2	A1+3	A1+4
nom	tabDePtr[1][0]	tabDePtr[1][1]	tabDePtr[1][2]	tabDePtr[1][3]	tabDePtr[1][4]

*L'adresse A1 est celle d'une autre zone réservée au stockage de cinq char*

adresse	A2	A2+1	A2+2	A2+3	A2+4
nom	tabDePtr[2][0]	tabDePtr[2][1]	tabDePtr[2][2]	tabDePtr[2][3]	tabDePtr[2][4]

*L'adresse A2 est celle d'une troisième zone réservée au stockage de cinq char*



Par rapport à un vrai tableau (cf. page 2), cette organisation présente plusieurs particularités :

- Les zones de mémoire correspondant aux différentes lignes de la matrice correspondent chacune à un bloc différent et ne sont donc pas nécessairement adjacentes. Ce détail peut s'avérer crucial car, s'il permet parfois une utilisation plus efficace de la mémoire, il s'oppose en revanche catégoriquement à une réinterprétation de `tabDePtr` qui considérerait qu'il s'agit en fait d'un tableau unidimensionnel de 15 caractères.
- Si le nombre de lignes est fixé définitivement lors de la compilation, leur taille peut être décidée (et peut même assez facilement être modifiée) au cours de l'exécution du programme.
- L'ordre des lignes dans le tableau peut très facilement être modifié : pour permuter deux lignes, il suffit d'échanger la valeur de deux pointeurs (si les lignes sont longues, le fait de ne pas avoir à échanger toutes ces valeurs peut faire gagner un temps considérable).
- Rien n'oblige toutes les lignes à comporter le même nombre de colonnes. Bien que cette idée puisse sembler a priori étrange (en particulier parce qu'elle complique le parcours intégral de la matrice, élément par élément), elle est souvent mise en œuvre, notamment lorsque l'accès aux éléments est dédaigné au profit de l'accès aux lignes, comme dans l'exemple suivant :

```
1 char * message[] = { "Essai", "Texte un peu plus long", "Bref" };
2 int ligne;
3 for (ligne=0 ; ligne < sizeof(message)/sizeof(message[0]) ; ++ligne)
4     affiche(message[ligne]);
```

Remarquez que, dans cet exemple, les pointeurs éléments du tableau `message` ne sont pas rendus valides par allocation dynamique, mais par simple initialisation à l'aide d'une chaîne littérale. Il s'agit néanmoins bel et bien d'un (vrai) tableau de pointeurs désignant des (faux) tableaux de `char` (les chaînes littérales) qui ont des longueurs différentes.

- Si certaines des lignes prévues s'avèrent inutiles, il n'est pas nécessaire de leur allouer de la mémoire. Le "gaspillage" de mémoire est alors minime, quel que soit le nombre de colonnes.

Lorsque cette technique est employée, il est préférable de rendre explicitement `NULL` les pointeurs inutilisés, de façon à favoriser la détection des erreurs de programmation.

### Destruction

Lorsqu'un faux tableau multidimensionnel constitué d'un tableau de pointeurs cesse d'être utile, il faut restituer au système les zones qui ont été attribuées à chacune de ses lignes. Si `tabDePtr` est un tableau dont les lignes ont été créées par allocation dynamique, il pourra être détruit de la façon suivante :

```
1 int ligne;
2 for (ligne = 0 ; ligne < sizeof(tabDePtr)/sizeof(tabDePtr[0]) ; ++ligne)
3     delete[] tabDePtr[ligne];
```

Bien entendu, si les lignes n'ont pas été créées par allocation dynamique mais sont, par exemple, des adresses de chaînes littérales, aucun `delete[]` n'est nécessaire (ou possible).

### Transmission à une fonction

Une fonction à qui l'on doit transmettre un tableau de pointeurs reçoit un pointeur sur le premier élément du tableau, c'est à dire un "**pointeur sur pointeur**". A titre d'exemple, voici comment pourrait être définie une fonction donnant une valeur nulle à chacun des éléments de `tabDePtr`, ou de tout autre vrai tableau de faux tableaux de caractères qui lui serait transmis :

```
1 void miseAZeroMatrice(char ** matrice, int nbLignes, int nbColonnes)
2 {
3     int ligne;
4     int colonne;
5     for (ligne = 0 ; ligne < nbLignes ; ++ligne)
6         for (colonne = 0 ; colonne < nbColonnes ; ++colonne)
7             matrice[ligne][colonne] = 0;
8 }
```

L'obligation dans laquelle nous nous trouvons de transmettre explicitement le nombre de lignes et le nombre de colonnes a ses bons côtés : la même fonction peut traiter tous les tableaux de pointeurs sur `char`, quels que soient leurs nombres de lignes et de colonnes !



## Conclusion

Les tableaux de pointeurs offrent de nombreux avantages, dont le plus évident est la possibilité de ne fixer la taille des lignes qu'au cours de l'exécution. La permutation de lignes complètes par simple échange des valeurs des pointeurs correspondants et la possibilité d'avoir des lignes de longueurs différentes sont aussi des caractéristiques qui s'avèrent parfois précieuses.

Au titre des inconvénients, il faut citer une certaine lourdeur de mise en œuvre, puisque la création et la destruction du tableau doivent être effectuées ligne par ligne, et exigent donc en général chacune une boucle. Un autre inconvénient est que le nombre de lignes doit, pour sa part, être fixé définitivement lors de l'écriture du programme.

## 4 - Un pointeur sur des pointeurs

La création d'un (vrai) tableau de pointeurs permet, nous l'avons vu, de simuler la présence d'un tableau multidimensionnel. La même logique peut être appliquée au cas d'un faux tableau de pointeurs : au lieu de stocker les adresses des lignes dans un vrai tableau, il suffit d'allouer dynamiquement une zone de mémoire permettant de stocker le nombre d'adresses nécessaires. Dans le cas d'une matrice de char comportant trois lignes et cinq colonnes, l'organisation en mémoire correspondant à cette approche peut être représentée ainsi :

nom	ptrSurPtr
contenu	l'adresse n

*ptrSurPtr contient l'adresse d'une zone de mémoire contenant une adresse*

adresse	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8	n+9	n+10	n+11
nom		ptrSurPtr[0]				ptrSurPtr[1]				ptrSurPtr[2]		
contenu		l'adresse A0				l'adresse A1				l'adresse A2		

*L'adresse n est celle d'une zone réservée au stockage de trois adresses (chacune occupe quatre octets)*

adresse	A0	A0+1	A0+2	A0+3	A0+4
nom	ptrSurPtr[0][0]	ptrSurPtr[0][1]	ptrSurPtr[0][2]	ptrSurPtr[0][3]	ptrSurPtr[0][4]

*L'adresse A0 est celle d'une zone réservée au stockage de cinq char*

adresse	A1	A1+1	A1+2	A1+3	A1+4
nom	ptrSurPtr[1][0]	ptrSurPtr[1][1]	ptrSurPtr[1][2]	ptrSurPtr[1][3]	ptrSurPtr[1][4]

*L'adresse A1 est celle d'une autre zone réservée au stockage de cinq char*

adresse	A2	A2+1	A2+2	A2+3	A2+4
nom	ptrSurPtr[2][0]	ptrSurPtr[2][1]	ptrSurPtr[2][2]	ptrSurPtr[2][3]	ptrSurPtr[2][4]

*L'adresse A2 est celle d'une troisième zone réservée au stockage de cinq char*

Si on la compare à celle représentée page 7, cette organisation ne présente qu'une seule originalité : la zone de mémoire qui contient les adresses des lignes n'est plus une variable à laquelle on accède grâce à son nom, mais une zone réservée à l'aide de `new[]`, à laquelle on accède en déréférençant `ptrSurPtr`, qui contient son adresse. L'organisation du stockage des éléments est, en revanche, tout à fait identique.

## Création et utilisation

La création d'un faux tableau destiné à stocker des adresses ne pose aucun problème particulier. Il faut simplement se souvenir qu'un faux tableau est un pointeur sur le type de ses éléments, ce qui signifie ici un pointeur sur un pointeur.

```

1  const int NB_LIGNES = 3;
2  const int NB_COLONNES = 5;
3  char ** ptrSurPtr = new char * [NB_LIGNES]; //un faux tableau de pointeurs
4  if (ptrSurPtr != NULL)
5      { //on peut créer les lignes
6          int ligne;
7          for(ligne = 0 ; ligne < NB_LIGNES ; ++ligne)
8              {
9                  ptrSurPtr[ligne] = new char[NB_COLONNES];
10                 if (ptrSurPtr[ligne] == NULL)
11                     //il faut appeler au secours !
12             }
13     }
```

Le type de la variable `ptrSurPtr` correspond exactement à celui du paramètre reçu par une fonction à laquelle on transmet un (vrai) tableau de pointeurs. Une même fonction est donc en mesure de traiter indifféremment des faux tableaux multidimensionnels, que ceux-ci soient des vrais ou des faux tableaux de pointeurs. Les fonctions de ce genre étant également indifférentes aux nombres de lignes et de colonnes du faux tableau à traiter, il devient enfin possible d'écrire du code ayant une certaine généralité d'usage !

### Destruction

Un faux tableau multidimensionnel construit à partir d'un pointeur sur pointeurs peut être détruit d'une façon analogue à celle employée dans le cas d'un (vrai) tableau de pointeurs : les lignes doivent tout d'abord être détruites les unes après les autres. Une fois la destruction des lignes terminées, une étape supplémentaire est toutefois nécessaire, puisqu'il faut également libérer la mémoire réservée au stockage des adresses des lignes. Le tableau élaboré ci-dessus pourrait donc être détruit à l'aide des instructions suivantes :

```
14 for (ligne = 0 ; ligne < NB_LIGNES ; ++ligne)
15     delete[] ptrSurPtr[ligne];
16 delete[] ptrSurPtr;
```

### Conclusion

L'utilisation d'un faux tableau multidimensionnel basé sur un pointeur sur pointeurs n'est pas significativement plus complexe que celle d'un faux tableau analogue basé sur un (vrai) tableau de pointeurs. Aux avantages de cette dernière méthode, le pointeur sur pointeurs ajoute celui de rendre dynamique le nombre de colonnes (et les tailles des autres dimensions, dans le cas d'un tableau à plus de deux dimensions)

## 5 - Bilan général

Quel que soit le genre de tableau auquel on a affaire, l'accès aux éléments est toujours obtenu de la même façon : il suffit de faire suivre le nom de la variable (pointeur ou tableau) par autant de couples de crochets qu'il y a de dimensions, chacun de ces couples encadrant la valeur de l'index de l'élément visé sur la dimension concernée.

Une fois qu'ils ont été créés, les différents tableaux possibles s'utilisent suffisamment indifféremment pour que certains programmeurs aient tendance à oublier qu'il s'agit tout de même de structures de données profondément différentes. Cet oubli est fréquemment la cause de difficultés (notamment lors de la déclaration des paramètres des fonctions) et est parfois à l'origine de graves erreurs de programmation.

Les avantages et les inconvénients des différentes approches envisageables dans le cas d'une matrice bidimensionnelle peuvent être résumés par le tableau suivant :

	Vrai tableau	Pointeur sur tableaux	Tableau de pointeurs	Pointeur sur pointeurs
Création	Très facile	Moins facile	Moins facile	Moins facile
Initialisation	Très facile	Impossible	Impossible	Impossible
Libération de la mémoire	Automatique	Facile	Moins facile	Moins facile
Nombre de lignes "dynamique"	non	oui	non	oui
Nombre de colonnes "dynamique"	non	non	oui	oui
Acceptable par une même fonction quel que soit le nombre de colonnes	non	non	oui	oui
Réinterprétable comme un tableau unidimensionnel	oui	oui	non	non
Lignes de tailles inégales	Impossibles	Impossibles	Possibles	Possibles
Permutation "instantanée" de deux lignes	non	non	oui	oui

Ces caractéristiques générales se retrouvent dans le cas des tableaux à plus de deux dimensions, même si le nombre d'options possibles rend difficilement envisageable une description systématique, ne serait-ce que du cas des matrices tri-dimensionnelles.