Rapport entre tableaux, adresses et pointeurs

Terminologie :

Pour un tableau, on convient d'employer le terme « dimension » pour désigner le nombre de valeurs que peut prendre un indice donné. On dit par exemple que :

```
double x[5][3]
```

est un tableau à deux indices, de dimension `5` selon le premier indice et `3` selon le second.

1 Préliminaire : l'opérateur `sizeof`

Cet opérateur fournit la dimension d'un objet en octets :

`i` étant un `int` : `sizeof(i)` vaut 4
`x` étant un `double` : `sizeof(x)` vaut 8
`c` étant un `char` : `sizeof(c)` vaut 1
`pi` étant un pointeur sur un `int` : `sizeof(pi)` vaut 4
`px` étant un pointeur sur un `double` : `sizeof(px)` vaut 4
`pc` étant un pointeur sur `char` : `sizeof(pc)` vaut 4

On peut aussi avoir directement la taille du type `int`, `double` ou `char` en écrivant :

`sizeof(int)` qui vaut 4
`sizeof(double)` qui vaut 8
`sizeof(char)` qui vaut 1
`sizeof(int *)` qui vaut 4
`sizeof(double *)` qui vaut 4
`sizeof(char *)` qui vaut 4

Pour un tableau `sizeof` donne le nombre d'octets occupés par l'ensemble des éléments du tableau :

`double t[5] ; sizeof(t) ;` donne 40

`double t[5][3] ; sizeof(t) ;` donne 120

Remarque

On a ainsi un moyen de connaître la dimension d'un tableau quelconque à un indice en écrivant :

```
sizeof(t)/sizeof(t[0]) ;
```

2 Signification d'un nom de tableau

Quand on déclare un tableau par :

```
double t[10];
```

on réserve 10×8 octets consécutifs pour dix variables de type `double` qui sont `t[0]`, `t[1]`, ..., `t[9]`[1].
Le nom `t` du tableau, employé seul, c'est à dire sans crochets, a une signification particulière : dans la plupart des cas il représente l'adresse du premier élément du tableau c'est à dire `&t[0]`. Ainsi, par exemple :

```
...
double t[10];
...
cout << (unsigned int)t << " " << (unsigned int)&t[0] << endl;
...
```

donne le résultat :

3217592848 3217592848

Cependant `t` n'a pas exactement les mêmes propriétés qu'une adresse (en l'occurence celle de la variable `t[0]`), ni non plus exactement celles d'un pointeur, ce qui sera mis en évidence dans la suite.
Mais on peut appliquer à `t` les opérateurs `*` et `[]`, comme à une adresse ou un pointeur. Et donc, si `t` représente l'adresse du premier élément du tableau `*t` est le premier élément du tableau. De même :

[1] On appelle ces variables « éléments » du tableau.

`t+1` représente l'adresse du second élément du tableau $\Longrightarrow$ `*(t+1)` est le second élément du tableau
...
`t+i` représente l'adresse du (`i+1`)$^{\text{ième}}$ élément du tableau $\Longrightarrow$ `*(t+i)` est le (`i+1`)$^{\text{ième}}$ élément du tableau
...
`t+9` représente l'adresse du dixième élément du tableau $\Longrightarrow$ `*(t+9)` est le dixième élément du tableau

Remarques

- On constate que l'indice `i` doit varier de `0` à `9` et non de `1` à `10`.
- Lorsqu'on écrit `*(t+i)`, rien n'empêche d'accéder à des variables situées avant ou après le tableau si `i` sort des limites. Comme on ne sait pas ce que sont ces variables le résultat ne peut être qu'aberrant. Il faut donc respecter soigneusement la dimension du tableau.

Si on utilise la notation `*(t+i)`, les crochets n'apparaissent que dans la déclaration. Pour des raisons de cohérence de notation, on utilise plutôt, en général, la notation avec des crochets. En effet on peut appliquer à l'adresse représentée par `t` les propriétés de l'opérateur `[]` vues précédemment :

la variable `*(t+i)` est équivalente à `t[i]`

et, autre façon de dire la même chose :

`t+i` représente l'adresse `&t[i]` du (`i+1`)$^{\text{ième}}$ élément et, en particulier, comme il se doit, `t` représente `&t[0]`.

On comprend ainsi qu'un élément de tableau `t[i]` est un objet composite : c'est le résultat de l'application de l'opérateur `[]` à l'adresse représentée par `t` et non une simple notation d'un seul bloc.

Remarque

La donnée de `t` et `t[0]` détermine complètement les adresses de tous les éléments du tableau : l'adresse de l'élément `i` est donnée par `t+i` pour `i` allant de `0` à `sizeof(t)/sizeof(t[0])-1`

3 Un nom de tableau ne s'identifie exactement ni à une adresse ni à un pointeur

3.1 Différence avec une adresse

Première différence

Considérons l'exemple suivant :

```
...
double t[7];
...
cout << (unsigned int)t << " " << (unsigned int)&t << " " << (unsigned int)&t[0] << endl;
...
```

qui donne le résultat :

3217592848 3217592848 3217592848

Il est normal, d'après ce qui a été dit précédemment, que `t` et `&t[0]` donnent le même résultat. Par contre, si `t` était une simple adresse, l'expression `&t` représenterait `&(&t[0])` c'est à dire l'adresse d'une adresse, ce qui n'a pas de sens, et serait rejetée par le compilateur. Or on constate que `&t` a bien une valeur, qui est d'ailleurs tout simplement `&t[0]`.

Seconde différence

Les instructions :

```
...
double t[7];
...
cout << sizeof(t) << " " << sizeof(&t) << " " << sizeof(&t[0]) << endl;
...
```

donnent le résultat :

56 4 4

`sizeof(t)` donne donc le nombre total d'octets occupés par le tableau (7×8=56 octets) alors que `sizeof(&t)` et `sizeof(&t[0])` donnent le nombre d'octets occupés par l'adresse `&t[0]` (4 octets). Ici encore, donc, le nom de tableau `t` se distingue d'une adresse.

3.2 Différence avec un pointeur

Le nom de tableau `t` ne peut pas non plus être assimilé à un pointeur auquel on aurait attribué la valeur `&t[0]`. En effet l'adresse que représente le nom du tableau est imposée par le compilateur au moment de la déclaration `double t[7]` et ne peut pas être changée par la suite. Supposons que l'on ait :

```
double t[7] ; double *p ;
```

`t` est donc un tableau et `p` un pointeur.
On peut écrire éventuellement :

```
p=t ;
```

mais non :

```
t=p ;
```

Un nom de tableau ne pourrait donc au mieux qu'être un pointeur constant mais ce n'est pas exactement le cas non plus. En effet les instructions :

```
...
double t[7];
const double *p=t;
cout << (unsigned int)t << " " << (unsigned int)p << " " << (unsigned int)&t[0] << endl;
cout << sizeof(t) << " " << sizeof(p) << endl;
cout << (unsigned int)&t << " " << (unsigned int)&p << endl;
...
```

donnent le résultat :

3220290728 3220290728 3220290728
56 4
3220290728 3220290788

La première ligne du résultat indique que `t` et `p` ont tous les deux la valeur `&t[0]` ce qui est normal.
La seconde ligne indique que `t` n'est pas un pointeur constant sur un `double` puisque dans ce cas `sizeof(t)` devrait valoir 4 et il vaut 56.
La troisième ligne indique que `t` ne s'identifie pas non plus à un pointeur constant sur un objet de 56 octets car dans ce cas `&t` désignerait l'adresse de ce pointeur et non `&t[0]` qui en serait le contenu.

4 Passage d'un tableau à un indice en argument d'une fonction

Reprenons l'exemple, vu au chapitre **Fonctions**, qui illustre la façon la plus générale de transmettre un tableau en argument d'une fonction :

```
#include<iostream>
#include<cmath>
using namespace std;
double norme(double x[],int p)
{
  int i; double s;
  s=0.;
  for(i=0;i<p;i++) s=s+x[i]*x[i];
  return sqrt(s);
}
int main(void)
{
  const int n1=3,n2=8;
  double v1[n1]={1.,2.,3.},v2[n2]={4.,5.,6.,7.,8.,9.,10.,11.};
  cout << norme(v1,n1) << endl;
  cout << norme(v2,n2) << endl;
  return 0;
}
```

Dans le programme principal `v1` et `v2` sont des tableaux. Dans l'appel `norme(v1,n1)`, `v1` représente l'adresse `&v1[0]`. C'est donc seulement l'adresse du premier élément du tableau `v1` que l'on passe à la fonction dans `x`, comme si on écrivait : `x=&v1[0]`. `x` ne peut pas être un tableau parce si c'était le cas `x` représenterait l'adresse `&x[0]` et ce serait comme si on écrivait `&x[0]=&t[0]`, mais on sait qu'il est impossible d'attribuer une valeur à une adresse. `x` est en réalité un pointeur et donc, en dépit des apparences, il n'y a pas de tableau dans la fonction mais seulement un pointeur `x`. Grâce à l'opérateur `[]` ce pointeur s'utilise exactement comme un tableau et on ne s'aperçoit pas qu'il ne s'agit pas d'un tableau. On pourrait écrire l'en-tête de la fonction `norme` :

```
double norme(double *x,int p)
```

ce serait strictement équivalent.
On comprend aussi pourquoi une fonction d'en tête `... f(...,double x[],int n,...)` peut être appelée avec un tableau à un indice de n'importe quelle dimension. En effet, pour pouvoir accéder aux éléments dans la fonction, il suffit de lui passer l'adresse du premier, les autres suivant par incrémentation de l'adresse. Le nombre d'éléments (`n` ici) ne sert qu'à indiquer jusqu'où on peut incrémenter l'adresse.
Le pointeur `x` contient l'adresse du premier élément du tableau `v1`, donc, durant l'exécution de la fonction, `*(x+i)`, que l'on peut noter indifféremment `x[i]`, et `v1[i]` sont associés au même emplacement de mémoire. `x[i]` n'est qu'une autre façon d'accéder à l'emplacement de mémoire réservé à `v1[i]` par la déclaration `double v1[n1]` dans le programme principal. On voit qu'il s'agit d'une transmission par adresse des éléments du tableau `v1` et que tout modification de `x[i]` dans la fonction est immédiatement répercutée sur `v1[i]`. Donc pour un tableau en argument la transmission se fait dans les deux sens : de la fonction appelante vers la fonction appelée et de la fonction appelée vers la fonction appelante, contrairement au cas de la transmission par valeur où la transmission n'a lieu que de la fonction appelante vers la fonction appelée. Ceci peut-être utilisé par exemple pour renvoyer, en plus de la norme du vecteur, le vecteur unitaire porté par `v1` :

```
#include<iostream>
#include<cmath>
using namespace std;
double norme(double x[],int p)
{
  int i; double s,nor;
  s=0.;
  for(i=0;i<p;i++) s=s+x[i]*x[i];
  nor=sqrt(s);
  if(nor==0){cout << "Le vecteur est nul, il ne définit pas un vecteur unitaire" << endl; return no
  for(i=0;i<p;i++) x[i]=x[i]/nor;
  return nor;
}
int main(void)
{
  const int n1=3,n2=8;
  int i;
  double v1[n1]={1.,2.,3.},v2[n2]={4.,5.,6.,7.,8.,9.,10.,11.};
  cout << "Composantes du vecteur initial : "; for(i=0;i<n1;i++) cout << v1[i] << " "; cout << endl
  cout << "Norme=" << norme(v1,n1) << endl;
  cout << "Composantes du vecteur unitaire : "; for(i=0;i<n1;i++) cout << v1[i] << " "; cout << end
  cout << endl;
  cout << "Composantes du vecteur initial : "; for(i=0;i<n2;i++) cout << v2[i] << " "; cout << endl
  cout << "Norme=" << norme(v2,n2) << endl;
  cout << "Composantes du vecteur unitaire : "; for(i=0;i<n2;i++) cout << v2[i] << " "; cout << end
  return 0;
}
```

Bien entendu il est indispensable que le type du pointeur `x` soit le même que celui du tableau `v1` sinon les incrémentations d'adresses `x+i` et `t+i` ne se feraient plus du même nombre d'octets et ce serait le chaos. De toutes façons le compilateur détecte une éventuelle incohérence des types.
L'argument muet, nommé `x` ici, est toujours un pointeur et jamais un tableau, même lorsqu'on écrit l'en-tête, comme on a le droit de le faire :

`... f(...,double x[n],...)`[2]

c'est équivalent à l'écrire :

`... f(...,double *x,...)`

Dans les deux cas c'est uniquement l'adresse du premier élément du tableau en argument effectif qui est transmise à la fonction et, dans le premier cas, `n` est ignoré.
On peut vérifier que l'argument muet est toujours un pointeur dans l'exemple suivant :

```
#include<iostream>
using namespace std;
const int n=7;
void f(double s[n])
{
  cout << sizeof(s) << endl;
  double y=22; s=&y; cout << "*s=" << *s << endl;
}
int main()
{
  double t[n];
  cout << sizeof(t) << endl;
  //double z=33; t=&z; cout << "*t=" << *t << endl;  // <-- pas possible
  f(t);
  return 0;
}
```

dont le résultat est :

56
4
**s=22*

Dans la fonction `f`, `s` est un pointeur, car sa taille est 4 et non 56 et il peut se trouver à gauche d'un signe =, contrairement au tableau `t` du `main`.

Remarque

On a vu que la transmission des tableaux en argument d'une fonction se faisant par adresse et non par valeur, toute modification de l'argument muet dans la fonction se répercute sur l'argument effectif, contrairement au cas des variables simples. Si on veut être sûr que le tableau passé en argument ne soit pas modifié par la fonction, on peut écrire :

`... f(...,const double s[],int n,...)` au lieu de `... f(...,double s[],int n,...)`

5 Cas des tableaux à plus d'un indice

Il est tentant de vouloir généraliser la méthode pour passer un tableau à un indice en argument d'une fonction, qui vient d'être exposée, au cas de deux indices ou plus. On souhaiterait pouvoir écrire, dans le cas de deux indices par exemple, l'en-tête d'une fonction `f` :

`... f(...,double s[][],...)`

ou :

`... f(...,double **s,...)`

et son appel :

```
double t[p][q];
...
... f(...,t,...);
```

mais cela ne fonctionne pas.
Il faut d'abord noter que la notation `s[][]` n'est pas définie[3], elle ne l'est que pour une seule paire de crochets. Elle ne peut donc pas être équivalente à `**s`, comme dans le cas à un seul indice.

[2] `n` étant une constante.
[3] Même en argument d'une fonction.

Mais le plus important est que, même avec l'en-tête `... f(...,double **s,...)`, le passage du tableau à deux indices ne se fait pas, parceque `s` ne peut recevoir que l'adresse d'un pointeur sur un `double`, c'est à dire d'un objet dont la taille est de 4 octets alors que `t` représente l'adresse d'un objet de `p`×`q`×8 octets.

La fin de ce chapitre est destinée à expliciter cette difficulté et à examiner trois façons de la contourner que l'on rencontre souvent dans la littérature écrite en C. Aucune de ces trois façons n'est complètement satisfaisante, il vaudra mieux, dans de nombreux cas, utiliser les « tableaux-pointeurs[4] » à la place des tableaux vus jusqu'ici. Le principe et l'utilisation de ces tableaux-pointeurs seront vus au chapitre **Allocation dynamique de mémoire**.
En conclusion :

les tableaux ne sont ni des adresses ni des pointeurs
ils sont simples à employer uniquement pour des usages très élémentaires, on leur préférera les pointeurs pour des usages plus élaborés.

Toute la fin de ce chapitre est hors programme

5.1 Signification d'un nom de tableau à plus d'un indice

Raisonnons sur l'exemple d'un tableau à deux indices. Les résultats sont généralisables à un nombre quelconque d'indices. Un tableau à deux indices est un tableau de tableaux. Le tableau déclaré par :

```
double t[P][Q] ;
```

est un tableau `t[P]` dont chacun des `P` éléments est lui-même un tableau de `Q` `double`.
Un élément quelconque est noté `t[i][j]` (on peut le noter aussi `(t[i])[j]` ce qui ne sert à rien sauf à souligner le fait que c'est un tableau de tableaux).
D'après les règles concernant les noms de tableaux à un indice vues précédemment :

`t` représente `&t[0]`
`t[0]` représente `&t[0][0]`

or on a vu que, pour un tableau `x` à un indice, `&x` est égale à `&x[0]`. Donc en appliquant cette propriété au tableau `t[0]` :

`&t[0]` est égale à `&t[0][0]` et donc `t` représente aussi `&t[0][0]`.

donc, au total, `t`, `&t`, `t[0]`, `&t[0]` représentent la même adresse `&t[0][0]`, comme on peut le vérifier sur l'exemple suivant :

```
#include<iostream>
using namespace std;
#define P 3
#define Q 7
int main()
{
  double t[P][Q];
  cout << (unsigned int)t << " " << (unsigned int)&t << " " << (unsigned int)t[0] << " " << (unsign
  << " " << (unsigned int)&t[0][0] << endl;
  return 0;
}
```

dont le résultat est :

3215152928 3215152928 3215152928 3215152928 3215152928

Par contre :

`t` est du type tableau de `P` tableaux de `Q` `double`
`t[0]` est du type tableau de `Q` `double`
`t[0][0]` est du type `double`

en particulier :

[4] Un tableau-pointeur n'est pas le même objet qu'un tableau de pointeurs.

`sizeof(t)` = `P`×`Q`×8 (octets)
`sizeof(t[0])` = `Q`×8
`sizeof(t[0][0])` = 8

ce que l'on vérifie en ajoutant l'instruction :

```
cout << sizeof(t) << " " << sizeof(t[0]) << " " << sizeof(t[0][0]) << endl;
```

dans l'exemple précédent, qui donne :

168 56 8

Donc `t`, `t[0]` et `t[0][0]` ont la même adresse mais ont des tailles différentes, ce qui fait que :

`&t+1` représente une adresse située `P`×`Q`×8 octets plus loin que `&t`
`&t[0]+1` (ou `t+1`) représente une adresse située `Q`×8 octets plus loin que `&t[0]` (ou `t`)
`&t[0][0]+1` (ou `t[0]+1`) représente une adresse située 8 octets plus loin que `t[0][0]` (ou `t[0]`)

On remarque que :

```
Q=sizeof(t[0])/sizeof(t[0][0])
P=sizeof(t)/sizeof(t[0])
```

donc la donnée de `t`, `t[0]` et `t[0][0]` détermine complètement les adresses de tous les éléments du tableau : l'adresse de l'élément `i,j` est donnée par `&t[0][0]+i*Q+j`[5] pour `i` allant de `0` à `P-1` et `j` allant de `0` à `Q-1`.

5.2 Passage d'un tableau à plus d'un indice en argument d'une fonction

Comme cela a déjà été mentionné ci-dessus, il n'est pas possible d'écrire, pour un tableau à deux indices :

```
... f(...,double **s,...)
```

et :

```
double t[P][Q];
...
... f(...,t,...);
```

Le compilateur diagnostique une incompatibilité de types.
Dans le cas à un indice le pointeur sur un `double` `s` reçoit `t` c'est à dire `&t[0]`, donc bien l'adresse d'un `double`. Ensuite, dans la fonction, on accède à tous les éléments de `t` à l'aide de la notation crochets appliquée à `s` et tout se passe bien.
Dans le cas à deux indices, `t` est un tableau de tableaux : dans l'exemple ci-dessus un tableau de `P` tableaux de `Q` `double`. Par ailleurs, `s` est un pointeur sur un pointeur de `double` : il ne peut recevoir que l'adresse d'un pointeur sur un `double`, c'est à dire d'un objet dont la taille est de 4 octets. Or `t` représente l'adresse `&t[0]` qui est celle d'un objet (`t[0]`) de `Q`×8 octets, d'où l'incompatibilité de type. Trois méthodes classiques pour y remédier sont exposées dans les paragraphes suivants.

5.2.1 Par un pointeur simple en argument muet

C'est la méthode la plus élémentaire. Un pointeur sur un `double` en argument muet reçoit l'adresse du premier élément du tableau. L'en-tête est le même que dans le cas d'un tableau à un indice :

```
... f(...,double s[],int p,int q,...) /* ou, équivalent : ... f(...,double *s,int p,int q,...) */
```

et l'appel se fait par :

```
double t[P][Q];
...
f(...,&t[0][0],P,Q,...);
```

L'adresse du premier élément du tableau `&t[0][0]` est celle d'un `double`, elle a donc la bonne taille pour pouvoir être mise dans `s`[6]. Durant l'exécution de la fonction, `*(s+i*Q+j)`, pour lequel on utilise plutôt la notation `s[i*Q+j]`, n'est qu'une autre façon de désigner l'élément `t[i][j]`. L'inconvénient est que `s` n'est utilisable qu'avec une seule paire de crochets, puisque ce n'est pas un pointeur sur un pointeur, et on ne retrouve pas la notation normale pour un tableau à deux indices avec deux paires de crochets. Le tableau à deux indices `t` est « déplié » dans un « tableau » à un indice. Cette méthode est rustique mais elle a l'avantage de la simplicité et elle est facilement généralisable à un

[5] Ou `t[0]+i*Q+j`.
[6] Ce qui ne serait pas le cas de `&t[0]`, `&t` ou `t`.

nombre quelconque d'indices. C'est celle qui a été présentée au chapitre **Fonctions**, section **Passage des tableaux en argument**, sous-section **Cas des tableaux à plus d'un indice**.

Remarque

Si on veut interdire la modification on peut, comme dans le cas à un seul indice, écrire l'en-tête :

```
... f(...,const double s[],int p,int q,...)
```

5.2.2 Par un pointeur de taille adaptée

On peut aussi mettre en argument muet un pointeur susceptible de recevoir l'adresse d'un objet de la taille de `&t[0]`. Dans le cas de l'exemple il faut un pointeur sur un tableau de `Q double`, déclaré par `double (*s)[Q]`[7]. En argument d'une fonction cette déclaration peut également s'écrire `double[][Q]`. On a donc, en tête :

```
... f(...,double s[][Q],...) /* ou ... f(...,double (*s)[Q],...), ou ... f(...,double s[P][Q],...) */
```

et appel :

```
double t[P][Q];
...
... f(...,t,...);
```

On remarque qu'il n'est pas nécessaire de faire figurer la dimension du premier indice, `P` dans ce cas, qui de toutes façons n'est pas utilisée : on peut mettre soit rien, soit n'importe quelle valeur, cela ne change rien[8]. Ceci a déjà été remarqué à propos de la transmission des tableaux à un indice, pour lesquels il est inutile de faire figurer l'unique dimension dans l'argument muet[9].
`s` contient l'adresse `&t[0]` donc `s+i` est l'adresse `&t[i]` et `*(s+i)` contient l'adresse `&t[i][0]`. Donc `*(*(s+i)+j)` est égal à `t[i][j]` et comme il est aussi égal par définition à `s[i][j]` on a, comme souhaité, `s[i][j]=t[i][j]` dans la fonction.
On voit tout de suite l'inconvénient majeur de cette méthode : la fonction ne peut s'appliquer qu'à des tableaux dont la dimension selon le second indice est `Q`. Dans certains cas particuliers ce n'est pas gênant[10], mais en général c'est insuffisant : une fonction qui ne peut multiplier que des matrices `P`×`Q` par des matrices `Q`×`Q`, `Q` étant fixé, est d'un intérêt limité. Cette méthode est celle qui est le plus couramment indiquée dans les livres d'informatique.

5.2.3 Par un tableau de pointeurs

Au lieu de transmettre uniquement l'adresse du premier élément du tableau `&t[0][0]` dans un pointeur, on peut transmettre l'ensemble des adresses `&t[i][0]`[11] pour `i=0..P-1` dans un tableau de `P` pointeurs sur un `double`. L'en-tête s'écrit :

```
... f(...,double **s,int p,int q,...) */
```

et l'appel :

```
double t[P][Q], *tp[P];
for(i=0; i<P; i++) tp[i]=t[i]; /* équivalent a tp[i]=&t[i][0] */
...
f(...,tp,P,Q,...);
...
```

`tp` est un tableau de `P` pointeurs sur un `double`. Le pointeur sur un `double` `tp[0]` contient `&t[0][0]`.
`s` est un pointeur sur un pointeur sur un `double`. Il reçoit `&tp[0]` qui est bien l'adresse d'un pointeur sur un `double`. `s[0]` est identique à `&tp[0]`. `s[0][0]` est le contenu de ce qui se trouve à l'adresse contenue dans `s[0]` donc dans `tp[0]`, c'est à dire `t[0][0]`. `s[0][1]` est le contenu de ce qui se trouve à l'adresse `&t[0][0]+1` donc c'est `t[0][1]`, de même `s[0][2]` est `t[0][2]` et ainsi de suite jusqu'à `t[0][Q-1]`.
Comme les pointeurs `tp[0]`, `tp[1]` et `tp[2]` sont des éléments consécutifs d'un même tableau ils ont des adresses

[7] Ce n'est pas la même chose que `double *s[Q]` qui est un tableau de `Q` pointeurs sur un `double`, les crochets étant prioritaires sur l'étoile (cf Kernighan et Ritchie, p. 110).
[8] Pour un tableau à un nombre quelconque d'indices, seule la dimension du premier indice peut ne pas figurer.
[9] De façon générale, en argument muet d'une fonction et seulement dans ce cas, la déclaration `double t[N][P][Q]...` (nombre quelconque d'indices) est équivalente à `double t[][P][Q]...` et `t` est alors un pointeur sur un tableau de `P`×`Q`×... `double`.
[10] S'il ne s'agit par exemple que de matrices de rotation dans l'espace à trois dimensions.
[11] C'est à dire les adresses de chaque début de ligne de la matrice représentée par `t[i][j]`.

consécutives. Donc `s[1]` est identique à `&tp[1]`, `s[2]` est identique à `&tp[2]`, etc.. En répétant le raisonnement fait pour `s[0]` on voit que `s[1][j]` et `t[1][j]` sont identiques pour `j=0, ..., Q-1` et ainsi de suite jusqu'à `s[P-1][j]` et `t[P-1][j]`.
L'avantage de cette troisième méthode est que, dans la fonction `f`, `s` s'emploie avec la notation naturelle `s[i][j]`.
Son inconvénient est qu'il faut passer par l'intermédiaire d'un tableau de pointeurs (`tp` ici) qui doit être déclaré et initialisé pour chaque tableau passé en argument.
Elle est employée dans certaines bibliothèques de programmes.

5.3 Conclusion

On voit que, pour les tableaux à plus d'un indice, aucune des trois façons qui viennent d'être exposées n'est complètement satisfaisante, c'est pourquoi on préférera l'utilisation des tableaux-pointeurs, dans sa version automatisée, présentée au chapitre **Allocation dynamique de mémoire**.