

8. Tableaux dynamiques

La possibilité de combiner plusieurs variables du même type dans un tableau est souvent très utile. Pensez aux vecteurs et matrices, mais aussi à une liste des âges des membres d'un groupe, etc. Pour déclarer des tableaux on utilise des pointeurs et la fonction `malloc` (*memory allocation*, qui nécessite la directive `#include<stdlib.h>`).

Voici la déclaration d'un tableau à un indice de 4 doubles :

```
double* t; int n = 4; t = (double*)malloc(n*sizeof(double));
```

La fonction `malloc` réserve de la place (consécutive) dans la mémoire pour (ici) 4 doubles et retourne l'adresse de la première.

- ▶ On peut bien sûr remplacer partout `double` par un autre type de variable pour déclarer un tableau d'un autre type, par exemple :

```
int* ti = (int*)malloc(3*sizeof(int));
```
- ▶ La fonction `sizeof` donne la taille en octets d'un type de variable.
- ▶ La conversion de type (`double*`) devant le `malloc` est nécessaire, parce que `malloc` retourne une valeur du type `void*`, qui est une adresse sans information de quel type de variable y est stocké.

Après la déclaration on réfère aux 4 éléments de ce tableau comme suivant :

`t[0], t[1], t[2], t[3]`. **Remarque bien que ça commence à 0 et que `t[4]` n'existe pas !** Alternativement, `*t, *(t+1), *(t+2), *(t+3)` marche aussi.

- ▶ Pour ne pas encombrer la mémoire inutilement, il faut libérer l'espace réservée si l'on n'a plus besoin du tableau, avec la commande `free(t)`;

Exemple d'un programme qui déclare et initialise deux vecteurs et puis fait appel à une fonction pour en calculer le produit scalaire :

```
#include<iostream>
#include<stdlib.h>           // Necessary pour utiliser malloc
using namespace std;
double prodscal(double* a, double* b, int n) {
    int i; double s = 0.;    // N'oublie pas d'initialiser s a 0!
    for (i = 0; i < n; i++)
        s += a[i] * b[i];
    return s;
}
int main() {
    int i, n = 3;           // Facile a changer le programme pour une autre dimension
    double *a, *b;
    a = (double*)malloc(n*sizeof(double));
    b = (double*)malloc(n*sizeof(double));
    a[0] = 1.5; a[1] = 2.5; a[2] = 3.5;
    b[0] = 2.; b[1] = 3.; b[2] = 4.;
    cout << "Vecteur a : ";
    for (i = 0; i < n; i++)
        cout << a[i] << " ";
    cout << endl;
    cout << "Produit scalaire : " << prodscal(a,b,n) << endl; // 24.5
    free(a); free(b);
    return 0;
}
```

Tableaux à deux indices (matrices)

Pour créer un tableau à deux indices, il faut en fait créer un tableau de tableaux à un indice. Le tableau à deux indices consiste ainsi en

- ▶ un pointeur,
- ▶ qui pointe vers la première adresse d'un tableau de pointeurs,
- ▶ qui pointent chacun vers la première adresse de tableaux différents de variables (p.ex. doubles), qui sont les lignes du tableau à deux indices.

```
int i, p = 2, q = 3; // Matrice de 2 lignes et 3 colonnes
double** m;        // Pointeur sur un pointeur sur une double
// Reservation de la memoire pour le tableau des p pointeurs,
// dont la première adresse est mise dans m :
m = (double**)malloc(p * sizeof(double*));
// Reservation de la memoire pour les p tableaux des q doubles,
// dont les premières adresses sont mises dans les m[i] :
for(i = 0; i < p; i++)
    m[i] = (double*)malloc(q * sizeof(double));
// Initialisation des elements de la matrice :
m[0][0] = 1.1; m[0][1] = 1.2; m[0][2] = 1.3; // première ligne
m[1][0] = 2.1; m[1][1] = 2.2; m[1][2] = 2.3; // deuxième ligne
```

- ▶ Rappel important : le premier élément d'un tableau a le numéro 0. La ligne numéro p et la colonne numéro q n'existent donc pas!
→ **erreur de segmentation**
- ▶ On peut aussi créer des tableaux à deux indices dont les lignes n'ont pas toutes la même longueur.

Pour libérer la mémoire si l'on n'a plus besoin du tableau à deux indices :

```
for(i = 0; i < p; i++) // d'abord liberer les p tableaux des q doubles
    free(m[i]);
free(m); // puis liberer le tableau des p pointeurs
```

Pour initialiser un tableau on a 4 méthodes :

1. S'il y a une certaine logique dans les valeurs on peut utiliser des boucles :

```
for(i = 0; i < p; i++)
    for(j = 0; j < q; j++)
        m[i][j] = i+1 + 0.1*(j+1);
```

2. Si le tableau est petit on peut le faire à la main (`m[0][0] = 1.1`; etc.)

3. Pour un tableau de taille moyenne il existe des fonctions dans la bibliothèque du magistère (`#include<bibli_fonctions.h>`), voir l'annexe du chapitre 9 du poly de cours :

```
ini_D_2(m, p, q, 1.1, 1.2, 1.3, 2.1, 2.2, 2.3);
```

Dans cette fonction il n'y a pas de conversion de type automatique : mettre 2 au lieu de 2. pour initialiser un tableau de doubles donne un résultat aberrant !

4. Pour un grand tableau avec des valeurs sans logique, taper les valeurs dans un fichier de données et faire lire ce fichier et mettre les valeurs dans le tableau par le programme.

Tout se généralise pour des tableaux à plus d'indices (hors programme). Par exemple, on pourrait utiliser une variable `double*** Gamma`; pour les symboles de Christoffel à trois indices de la relativité générale.

Exemple d'un programme qui calcule le produit de deux matrices :

```
#include<iostream>
#include<iomanip>
#include<fstream>
#include<stdlib.h>
using namespace std;
double** produit(double** a, double** b, int m, int n, int p) {
    int i, j, k;
    double** c = (double**)malloc(m * sizeof(double*));
    for(i = 0; i < m; i++) {
        c[i] = (double*)malloc(p * sizeof(double));
        for(k = 0; k < p; k++) {
            c[i][k] = 0.;
            for(j = 0; j < n; j++)
                c[i][k] += a[i][j] * b[j][k];
        }
    }
    return c;
}
void affichage(double** a, int m, int n) { // Remarque bien que le pointeur a
    int i, j; // ne contient pas d'information sur les dimensions de la matrice
    for(i = 0; i < m; i++) { // et qu'on a donc besoin des arguments m et n.
        for(j = 0; j < n; j++)
            cout << setw(7) << a[i][j];
        cout << endl;
    }
    cout << endl;
}
```

```

void liberer(double** a, int m) {
    int i;
    for(i = 0; i < m; i++) free(a[i]);
    free(a);
}

int main() {
    fstream fich;
    int i, j, m = 3, n = 2, p = 6;
    double **a, **b, **c;
    a = (double**)malloc(m * sizeof(double*));
    for (i = 0; i < m; i++) {
        a[i] = (double*)malloc(n * sizeof(double));
        for(j = 0; j < n; j++) // les elements de a sont initialises
            a[i][j] = i+1 + 0.1*(j+1); // avec des boucles (formule explicite)
    }
    fich.open("matrice.dat", ios::in);
    b = (double**)malloc(n * sizeof(double*));
    for(i = 0; i < n; i++) {
        b[i] = (double*)malloc(p * sizeof(double));
        for(j = 0; j < p; j++) // les elements de b sont initialises
            fich >> b[i][j]; // a partir d'un fichier
    }
    fich.close();
    affichage(a,m,n); affichage(b,n,p);
    c = produit(a,b,m,n,p);
    affichage(c,m,p);
    liberer(a,m); liberer(b,n); liberer(c,m);
    return 0; }

```

Version plus complète du programme qui trace l'allure d'une fonction avec des # (non pas tournée de 90°, valeurs négatives, axes, etc.) :

```
#include<iostream>
#include<math.h>
#include<stdlib.h>
using namespace std;
int xmin = -30, xmax = 30, ymin = -8, ymax = 8; // variables globales
double f(double x) { // fonction a tracer
    return 8.*sin(x*M_PI/10.);
}
void colonne(double y, char* col) { // Remplit une ligne de la matrice
    int i; // pour indiquer ou mettre les #
    if (y > ymax) y = ymax + 1; // (= colonne de la matrice transposee affichee)
    for (i = ymin; i < y; i++) // Les ordonnees de ymin a ymax correspondent
        col[i-ymin] = '#'; // aux indices 0 a ymax-ymin de la matrice
}
void affichage(char** m) {
    int i, j;
    for (j = ymax-ymin; j >= 0; j--) { // Remarque bien l'echange de i et j
        for (i = 0; i <= xmax-xmin; i++) // pour afficher la transposee de m
            if (i == -xmin && j == -ymin) cout << "+";
            else if (i == -xmin) cout << "|";
            else if (j == -ymin) cout << "-";
            else cout << m[i][j];
        cout << endl;
    }
}
```

