9. Tableaux dynamiques

1 Allocation dynamique de mémoire

1.1 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 8
`px` étant un pointeur sur un `double` : `sizeof(px)` vaut 8
`pc` étant un pointeur sur `char` : `sizeof(pc)` vaut 8

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 8
`sizeof(double*)` qui vaut 8
`sizeof(char*)` qui vaut 8

À noter que ces valeurs peuvent changer d'un ordinateur à un autre et d'un système d'exploitation à un autre. C'est pour cette raison qu'il faut toujours utiliser `sizeof`.

1.2 Allocation de `n` octets

L'allocation dynamique de mémoire permet à l'utilisateur de réserver un emplacement mémoire de `n` octets, par l'intermédiaire de l'adresse du premier de ces octets. La fonction :

```
malloc(n);
```
[1]

réserve `n` [2] octets et sa valeur est l'adresse du premier d'entre eux.
Il est très important de noter que `n` peut être une variable et pas seulement une constante, ce qui justifie l'adjectif « dynamique » .
Le nombre d'octets occupés par un `double` est `sizeof(double)`. Donc, si on veut réserver la place pour `n double` on écrit : `malloc(n*sizeof(double))`.

2 Tableau dynamique à un indice

2.1 Allocation d'un tableau dynamique à un indice

Réservons par exemple la place pour `n double` comme vu à la section précédente :

```
malloc(n*sizeof(double));
```

L'adresse du premier des octets réservés retournée par `malloc` n'a pas de type [3]. Une augmentation de 1 de cette adresse se traduit simplement par une augmentation de 1 du numéro d'octet. Si, par contre, on la convertit en adresse d'un type précis, cette augmentation est égale à la taille de ce type. Convertissons l'adresse `malloc(n*sizeof(double))` en adresse d'un `double` par :

```
(double*)malloc(n*sizeof(double));
```

et alors la valeur de `(double*)malloc(n)+1` sera celle de `(double*)malloc(n)` augmentée de 8 (taille d'un `double`). Plaçons enfin l'adresse fournie par `malloc` convertie en l'adresse d'un `double` dans un pointeur sur un `double` :

```
int n;
double* x;
...
n = 7;
```

1. La fonction `malloc` nécessite la directive `#include<stdlib.h>`
2. En principe `n` doit être de type `size_t` et non de type `int`. On n'en tient pas compte ici.
3. La fonction `malloc` est dite de type « générique » .

```
...
x = (double*)malloc(n*sizeof(double));
```

La variable `*x`, qui peut aussi être notée `x[0]`, est alors une variable de type `double` qui occupe les 8 premiers octets réservés par l'instruction `malloc`. Ce qui est intéressant c'est qu'à la suite de cette variable, il reste `n-1` emplacements mémoire disponibles, d'adresses consécutives, pouvant accueillir chacun une variable de type `double`. Puisque les adresses sont consécutives ces `n-1` variables sont tout simplement `*(x+1)`, `*(x+2)`, ..., `*(x+n-1)`, que, pour la commodité, on notera plutôt à l'aide de la notation `[]` : `x[1]`, `x[2]`, ..., `x[n-1]`.
Par ce procédé l'utilisateur a à sa disposition des variables indicées que l'on appellera tableaux dynamiques pour les distinguer des tableaux standards du C qui ne sont pas étudiés ni utilisés dans ce cours : en effet, excepté pour les cas très simples, l'utilisation des tableaux dynamiques est plus commode, logique et générale que celle des tableaux standards [4].
Sur le plan pratique il est important de retenir que les `n` éléments d'un tableau dynamique sont indicés de `0` à `n-1` et non de `1` à `n`.

En résumé, la forme condensée suivante de ce qui précède :

```
int n = 5;
double* x = (double*)malloc(n*sizeof(double));
```

crée un tableau dynamique de type `double` dont les éléments sont désignés par `x[0]`, `x[1]`, ..., `x[4]`.
La méthode se transpose de façon évidente à des tableaux d'entiers ou de caractères.

Pour libérer l'espace mémoire réservé quand on n'en a plus l'usage, on utilise simplement l'instruction :

```
free(x);
```

Si on ne le fait pas, cet espace mémoire reste inutilisable en pure perte jusqu'à la fin de l'exécution du programme ce qui ne peut que le ralentir et même, dans le cas d'allocations réitérées, conduire à une saturation complète de la mémoire.

2.2 Exemple d'utilisation de tableaux dynamiques à un indice

Les deux programmes suivants calculent le produit scalaire de deux vecteurs, le premier sans utiliser de tableau :

```
#include<iostream>
using namespace std;
int main() {
  double ux, uy, uz, vx, vy, vz, s;
  ...
  s = ux*vx + uy*vy + uz*vz;
  ...
  return 0;
}
```

le second en utilisant un tableau :

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  int d = 3, i;
  double *u = (double*)malloc(d*sizeof(double)), *v = (double*)malloc(d*sizeof(double)), s;
  ...
  for(s = 0., i = 0; i < d; i++)
    s = s + u[i]*v[i];
  ...
  free(u); free(v);
  return 0;
}
```

4. Une comparaison des deux types de tableaux se trouve aux chapitres (hors programme) **Tableaux standards** et **Rapports entre tableaux standards, adresses et pointeurs**.

Le second programme paraît plus compliqué mais il est plus général : si on veut calculer dans un espace à plus de trois dimensions, il faut, dans le premier cas, créer de plus en plus de noms variables et écrire explicitement ces noms dans les opérations, ce qui peut devenir très lourd voire irréalisable alors qu'il suffit, dans le second cas, de changer la valeur de `d`.

2.3 Dépassement de la taille déclarée

Considérons le programme suivant :

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  int i, n = 4;
  double r, *x = (double*)malloc(n*sizeof(double)), *y = (double*)malloc(n*sizeof(double));
  r = 10.;
  for (i = 0; i < n; i++) {
    x[i] = i; y[i] = r+i;
  }
  for (i = 0; i < n; i++)
    cout << x[i] << " " << y[i] << endl;
  free(x); free(y);
  return 0;
}
```

dont le résultat est :

```
0 10
1 11
2 12
3 13
```

Si on déclare `x` et `y` comme de simples variables et non des pointeurs :

```
double r, x, y;
```

au lieu de :

```
double r, *x = (double*)malloc(n*sizeof(double)), *y = (double*)malloc(n*sizeof(double));
```

on a le diagnostic suivant à la compilation : *erreur : invalid types 'double[int]' for array subscript*
Mais si on excède la valeur déclarée de la dimension, par exemple :

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  int i, n = 4, p = 6;
  double r, *x = (double*)malloc(n*sizeof(double)), *y = (double*)malloc(n*sizeof(double));
  r = 10.;
  for (i = 0; i < p; i++) {
    x[i] = i; y[i] = r+i;  // cette boucle dépasse la dimension déclarée pour x (p>n)
  }
  for (i = 0; i < p; i++)
    cout << x[i] << " " << y[i] << endl; // idem
  free(x); free(y);
  return 0;
}
```

soit il y a un diagnostic à l'exécution tel que *Erreur de segmentation*, soit il n'y a aucun diagnostic et c'est très dangereux car un résultat est fourni mais il peut être complètement faux sans que l'utilisateur s'en aperçoive.
Il faut donc, à l'écriture des programmes, veiller de façon extrêmement vigilante à ne pas dépasser les dimensions déclarées pour les tableaux dynamiques.

2.4 Initialisation et affichage des tableaux

Les éléments des tableaux dynamiques ne sont pas initialisés à une valeur donnée lors de l'allocation de mémoire. Ils contiennent simplement la valeur correspondant à l'état, complètement indéterminé pour l'utilisateur, dans lequel se trouvaient les registres de mémoire avant l'allocation. Dans le cas où c'est nécessaire, l'utilisateur doit donc initialiser le tableau.

2.4.1 Exemples d'allocation, d'initialisation et d'affichage d'un tableau à un indice (vecteur)

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  int i, n = 5;
  double* x = (double*)malloc(n*sizeof(double));
  cout << "Initialisation d'un vecteur à 0 :" << endl;
  for (i = 0; i < n; i++)
    x[i] = 0.;
  for (i = 0; i < n; i++)
    cout << x[i] << " ";
  cout << endl;
  cout << "Initialisation d'un vecteur à une suite de carrés d'entiers :" << endl;
  for (i = 0; i < n; i++)
    x[i] = i*i;
  for (i = 0; i < n; i++)
    cout << x[i] << " ";
  cout << endl;
  free(x);
  return 0;
}
```

ce qui donne :

Initialisation d'un vecteur à 0 :
0 0 0 0 0
Initialisation d'un vecteur à une suite de carrés d'entiers :
0 1 4 9 16

2.4.2 Cas où les valeurs d'initialisation sont données explicitement

Quand les valeurs d'initialisation ne peuvent être exprimées par une formule programmable mais sont fournies explicitement (par exemple ce sont des valeurs expérimentales provenant d'une série de mesures) on a le choix entre trois méthodes :

1. Dans le cas où leur nombre se réduit à quelques unités, écrire un à un les éléments de tableau :

   ```
   x[0] = 1.32; x[1] = -0.37; x[2] = 6.78;
   ```

2. Dans le cas où leur nombre ne dépasse pas quelques dizaines, utiliser une fonction d'initialisation sur le modèle suivant :

   ```
   #include<iostream>
   #include<stdlib.h>
   #include<stdarg.h>
   using namespace std;
   void ini(double* x, int ni, ...) {
     int i;
     va_list ap;
     va_start(ap,ni);
     for (i = 0; i < ni; i++)
       x[i] = va_arg(ap,double);
   ```

```
  va_end(ap);
}
/*************************************************************************/
int main() {
  int i, nn = 3;
  double* t = (double*)malloc(nn*sizeof(double));
  ini(t, nn, 2.3, -4.1, 5.7);
  for (i = 0; i < nn; i++)
    cout << t[i] << " ";
  cout << endl;
  free(t);
  return 0;
}
```

La fonction nommée ici `ini` reçoit en argument :

le nom du tableau dynamique à initialiser
la dimension de ce tableau
la liste des valeurs d'initialisation

C'est une fonction à nombre variable d'arguments[5], dont le principe a été exposé au chapitre **Fonctions**. De telles fonctions sont déjà écrites et disponibles dans la bibliothèque des fonctions du Magistère, leur utilisation est décrite à l'annexe située en fin de ce chapitre.

Remarque

Dans une telle fonction à nombre variable d'arguments il est très important de respecter strictement le type des arguments. Si les arguments sont du type double et on veut passer la valeur 2 à la fonction, il faut l'écrire comme `2.` avec un point. Dans ce cas il n'y a pas de conversion automatique de type, comme pour les fonctions avec un nombre fixe d'arguments, et si l'on ne respecte pas les types le résultat sera complètement aberrant.

3. Dans les autres cas écrire préalablement les valeurs d'initialisation dans un fichier de données puis les faire lire et placer dans les éléments de tableau par le programme.

2.5 Tableau dynamique à un indice en argument d'une fonction

Pour transmettre un tableau dynamique à une fonction il suffit de transmettre :
- l'adresse du premier élément du tableau dans un pointeur argument muet de la fonction
- le nombre d'éléments du tableau.

Prenons l'exemple d'une fonction calculant la norme d'un vecteur à trois composantes. Cette fonction et son appel par le `main` peuvent s'écrire :

```
#include<iostream>
#include<stdlib.h>
#include<math.h>
using namespace std;
//-------------------------------------
double norme(double* x, int p) {
  int i; double s;
  for (s = 0., i = 0; i < p; i++)
    s = s + x[i]*x[i];
  return sqrt(s);
}
//-------------------------------------
int main() {
  int i, n1 = 3, n2 = 5;
  double* v1 = (double*)malloc(n1*sizeof(double));
  for (i = 0; i < n1; i++)
    v1[i] = i;
```

5. Rappel : il n'est pas demandé aux étudiants de maîtriser les fonctions de ce type, mais la connaissance de leur existence et de leur utilisation peut être utile en pratique, comme ici par exemple.

```
  double* v2 = (double*)malloc(n2*sizeof(double));
  for (i = 0; i < n2; i++)
    v2[i] = i+3;
  cout << norme(v1,n1) << endl;
  cout << norme(v2,n2) << endl;
  free(v1); free(v2);
  return 0;
}
```

Lors de l'appel `norme(v1,n1)` l'adresse contenue dans le pointeur `v1`, qui est celle de l'élément `v1[0]`, est mise dans le pointeur `x`, comme si on avait écrit `x = v1`[6]. Le pointeur `x` contenant la même adresse que le pointeur `v1`, il donne accès, dans la fonction, par l'intermédiaire de l'opérateur `[]`, aux mêmes emplacements de mémoire que le pointeur `v1` : `x[0]` et `v1[0]` correspondent au même emplacement de mémoire et contiennent donc la même valeur, de même pour `x[1]` et `v1[1]`, etc. Durant l'exécution de la fonction, `x` et `v1` sont deux noms différents pour le même tableau. Dans la transmission à la fonction il n'y a pas duplication des valeurs des éléments de `v1` dans `x` mais simplement passage de l'adresse du premier élément de `v1`.
Ceci a pour conséquence que toute modification des éléments de `x` dans la fonction entraîne automatiquement celle des éléments de `v1` : s'il y a modification de `x`, le tableau `v1` ne contient plus, après l'appel de la fonction, les mêmes valeurs que celles qu'il contenait avant, l'argument effectif est modifié. Ce mécanisme peut entraîner des effets indésirables si l'on n'y prend pas garde, mais peut inversement être utilisé de façon avantageuse pour transmettre des valeurs dans le sens fonction appelée vers fonction appelante et non plus seulement dans le sens fonction appelante vers fonction appelée. Par exemple, la fonction suivante reçoit en entrée un vecteur quelconque dans un tableau et retourne le vecteur unitaire parallèle et de même sens dans le même tableau :

```
//vecteur_unitaire_un_vecteur.cpp
#include<iostream>
#include<stdlib.h>
#include<math.h>
using namespace std;
int f(double* v, int n) {
  double c, q;
  int i;
  for (c = 0., i = 0; i < n; i++)
    c = c + v[i]*v[i];
  q = sqrt(c);
  if (q == 0.)
    return 0;
  for (i = 0; i < n; i++)
    v[i] = v[i]/q;
  return 1;
}
int main() {
  int i, n = 2;
  double* x = (double*)malloc(n*sizeof(double));
  x[0] = 1.; x[1] = 2.;
  for (i = 0; i < n; i++)
    cout << x[i] << " ";
  cout << endl;
  f(x,n);
  for (i = 0; i < n; i++)
    cout << x[i] << " ";
  cout << endl;
  free(x);
  return 0;
}
```

Si on veut que le vecteur d'entrée ne soit pas modifié il faut utiliser deux pointeurs :

6. Cela se passe comme pour des variables ordinaires : la valeur contenue dans la variable argument effectif est placée dans la variable argument muet.

```
//vecteur_unitaire_deux_vecteurs.cpp
#include<iostream>
#include<stdlib.h>
#include<math.h>
using namespace std;
int f(const double* v, double* u, int n) // on met const pour être sûr que
{                                        // v ne sera pas modifié
  double c, q;
  int i;
  for(c = 0., i = 0; i < n; i++)
    c = c + v[i]*v[i];
  q = sqrt(c);
  if(q == 0.) {
    for(i = 0; i < n; i++)
      u[i] = 0;
    return 0;
  }
  for(i = 0; i < n; i++)
    u[i] = v[i]/q;
  return 1;
}
int main() {
  int i, n = 2;
  double* x = (double*)malloc(n*sizeof(double));
  double* xu = (double*)malloc(n*sizeof(double));
  x[0] = 1.; x[1] = 2.;
  f(x,xu,n);
  for(i = 0; i < n; i++)
    cout << x[i] << " ";
  cout << endl;
  for(i = 0; i < n; i++)
    cout << xu[i] << " ";
  cout << endl;
  free(x); free(xu);
  return 0;
}
```

3 N'employer un tableau que lorsque c'est réellement nécessaire

On ne doit employer un tableau que s'il faut conserver des valeurs pour les réutiliser en un autre point du programme. Sinon on mobilise des mémoires pour rien et, de plus, on subit les contraintes liées à la déclaration.

Exemple

On veut calculer les n premiers termes de la suite : $u_n = au_{n-1} + bu_{n-2}$, connaissant u_0 et u_1.

1) Avec un tableau :

```
/* Calcul de la suite u(n)=a*u(n-1)+b*u(n-2) en utilisant inutilement un tableau */
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  int i, nn = 50;
  double a = 5.6, b = -3.4, *u = (double*)malloc(nn*sizeof(double));
  u[0] = -4.2; u[1] = 7.1;
  for(i = 2; i < nn; i++)
    u[i] = a*u[i-1] + b*u[i-2];
  for(i = 0; i < nn; i++)
    cout << "u(" << i << ")=" << u[i] << endl;
```

```
  free(u);
  return 0;
}
```

Cela ne sert à rien de stocker les valeurs de u_n dans un tableau puisqu'elles sont affichées au fur et à mesure à l'écran et qu'on ne s'en sert pas ensuite dans le programme.

2) Sans tableau :

```
/* Calcul de la suite u(n)=a*u(n-1)+b*u(n-2) sans utiliser de tableau */
#include<iostream>
using namespace std;
int main() {
  int i, nn = 50;
  double a = 5.6, b = -3.4, u, u0 = -4.2, u1 = 7.1;
  cout << "u(0)=" << u0 << endl;
  cout << "u(1)=" << u1 << endl;
  for(i = 2; i < nn; i++) {
    u = a*u1 + b*u0;
    cout << "u(" << i << ")=" << u << endl;
    u0 = u1; u1 = u;
  }
  return 0;
}
```

La seconde méthode est plus simple.
Mais si on a besoin des u_n pour une tâche qui ne peut être effectuée au vol (par exemple les faire afficher par valeur croissante) il faut utiliser la première méthode.
On dit que dans le premier cas on a un accès direct aux u_n alors que dans le second on a seulement un accès séquentiel.

Autre exemple

On veut calculer la moyenne et l'écart-type d'un ensemble de valeurs entrées au clavier par l'utilisateur :

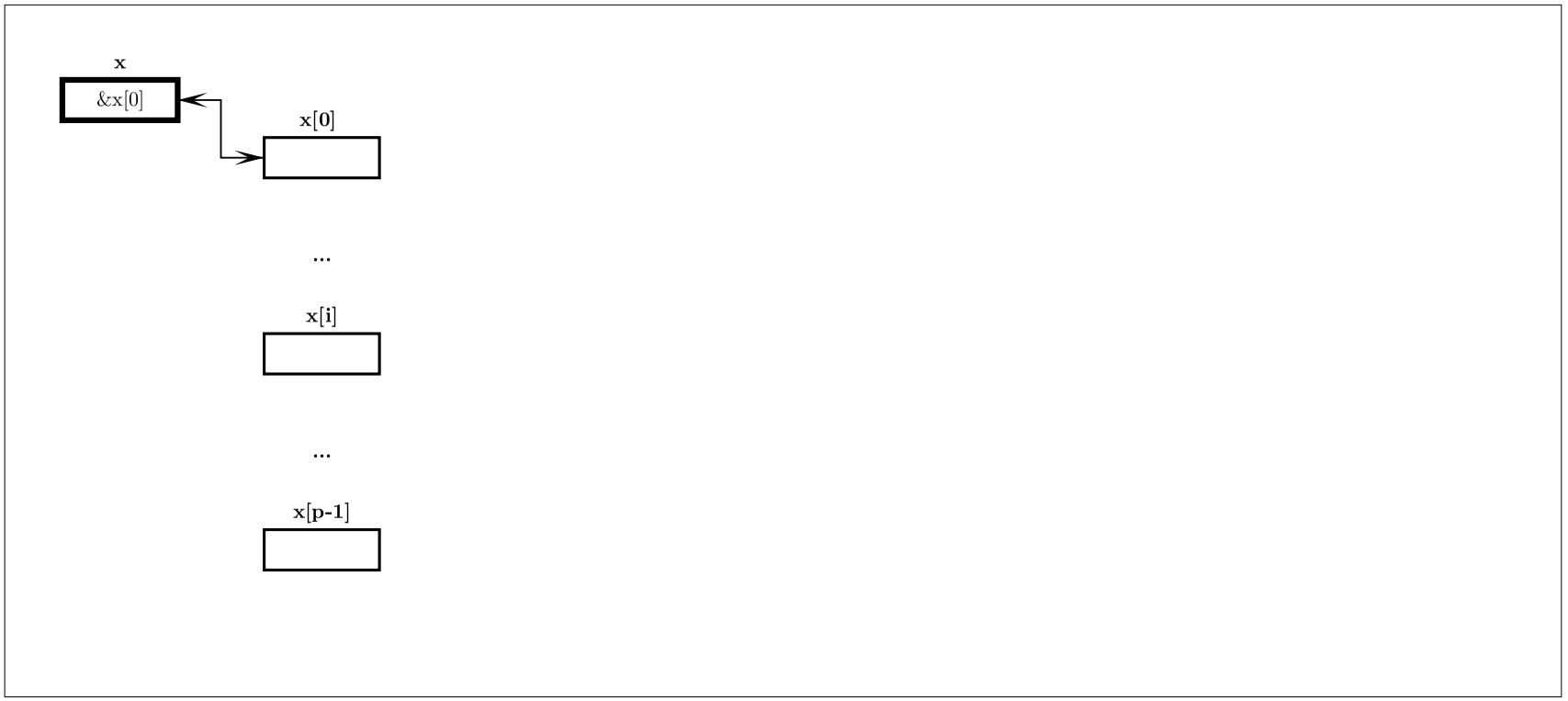
```
#include<iostream>
#include <math.h>
using namespace std;
int main() {
  int i, n;
  double s, s2, x, moy;
  cout << "Nombre de valeurs ? "; cin >> n;
  for(s = 0, s2 = 0, i = 1; i <= n; i++) {
    cin >> x;
    s += x; s2 += x*x;
  }
  moy = s/n;
  cout << "moyenne=" << moy << "  écart-type=" << sqrt(s2/n-moy*moy) << endl;
  return 0;
}
```

On voit qu'il est inutile de stocker les valeurs dans un tableau.

4 Tableaux dynamiques à plus d'un indice

4.1 Allocation d'un tableau dynamique à plus d'un indice

Les deux figures suivantes illustrent les explications de cette section :

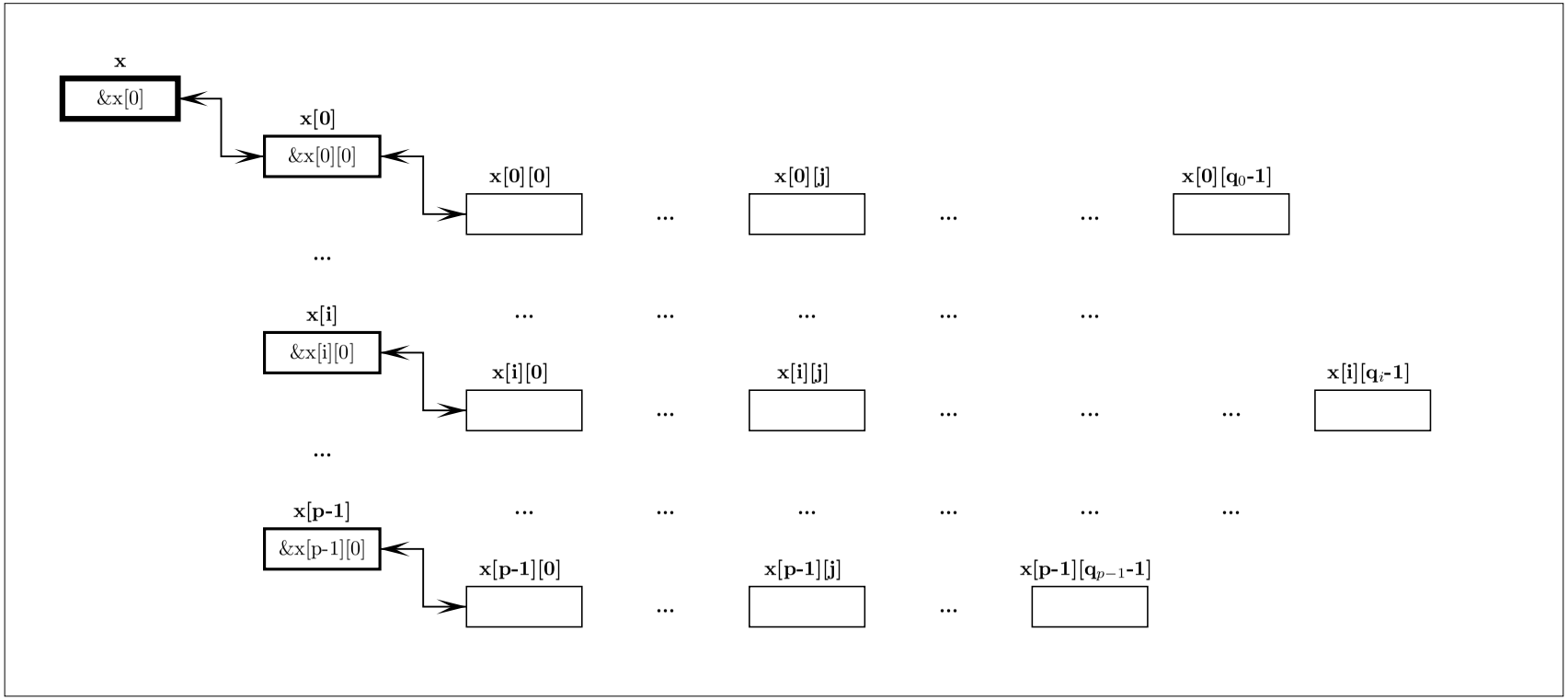


Les instructions ci-contre produisent l'allocation de mémoire schématisée ci-dessus.

```
double **x ;
int p ;
p=... ;
x=(double **)malloc(p*sizeof(double *)) ;
```

A l'issue de cette première étape on a créé `p` pointeurs de `double` : `x[0] ... x[p-1]`, dans lesquels on n'a pour l'instant mis aucune adresse.

Première étape de l'allocation de mémoire pour un tableau-pointeur à deux indices



Les instructions ci-contre complètent celles de la première étape pour obtenir l'allocation de mémoire d'un tableau-pointeur à deux indices schématisée ci-dessus.
On peut choisir d'avoir ou non des lignes toutes de la même longueur. Pour avoir des lignes toutes de longueur q il suffit faire $q_0 = \cdots = q_i = \cdots = q_{p-1} = q$, on obtient alors un tableau-pointeur rectangulaire.

```
int q0, ... qi, ... qp-1 ;
q0=... ; ... qi=... ; ... qp-1=... ;
x[0]=(double *)malloc(q0*sizeof(double)) ;
...
x[i]=(double *)malloc(qi*sizeof(double)) ;
...
x[p-1]=(double *)malloc(qp-1*sizeof(double)) ;
```

A l'issue de cette seconde étape on a mis des adresses de `double` dans chacun des pointeurs de `double` : `x[0] ... x[p-1]` et on a créé `q0 + ... + qi + ... + qp-1` variables ordinaires de type `double` auxquelles on n'a pour l'instant attribué aucune valeur.

Seconde étape de l'allocation de mémoire pour un tableau-pointeur à deux indices

Considérons la déclaration :

```
double** x;
```

`x` est un pointeur de pointeur de `double` susceptible de recevoir l'adresse d'un pointeur de `double`
`*x` ou `x[0]` est un pointeur de `double` susceptible de recevoir l'adresse d'un `double`.

Ajoutons maintenant à la déclaration précédente :

```
int p;
...
p = ...;
x = (double**)malloc(p * sizeof(double*));
```

cette dernière instruction met dans `x` une adresse de pointeur de `double` à partir de laquelle `p` emplacements de pointeurs de `double` sont réservés.
Ces `p` pointeurs de `double` sont alors désignés par `*x`, `*(x+1)`, ..., `*(x+p-1)` ou `x[0]`, `x[1]`, ..., `x[p-1]` [7]. Chacun de ces `p` pointeurs peut servir à créer un tableau dynamique à un indice en ajoutant par exemple aux instructions précédentes :

```
int q0, q1, ...;
...
q0 = ...; q1 = ...; ...
x[0] = (double*)malloc(q0 * sizeof(double));
x[1] = (double*)malloc(q1 * sizeof(double));
...
```

On peut choisir, comme cas particulier : `q0=q1=q2=...=q` et les instructions précédentes s'écrivent :

```
int q;
...
q = ...;
for(i = 0; i < p; i++)
  x[i] = (double*)malloc(q * sizeof(double));
...
```

`x[i][j]` (avec `i=0 ... p-1` et `j=0 ... q-1`) est alors une variable de type `double` dont l'adresse est `x[i]+j`, l'adresse du pointeur de `double` `x[i]` étant `x+i`.
On a ainsi créé un tableau dynamique à deux indices qui est une pure généralisation du tableau dynamique à un indice et il s'utilise de la même façon.
Pour libérer l'espace mémoire réservé, il faut procéder dans l'ordre inverse comparé aux `malloc` : d'abord faire des `free(x[i])` et puis un `free(x)` :

```
for(i = 0; i < p; i++)
  free(x[i]);
free(x);
```

4.1.1 Exemples d'allocation, d'initialisation et d'affichage d'un tableau à deux indices (matrice)

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  int i, j, n = 5;
  double** x = (double**)malloc(n * sizeof(double*));
  for(i = 0; i < n; i++)
    x[i] = (double*)malloc(n * sizeof(double));
  cout << "Initialisation d'une matrice carrée à 0 :" << endl;
  for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
      x[i][j] = 0.;
  for(i = 0; i < n; i++) {
```

7. A partir d'ici on choisit de n'utiliser que la notation « crochets » .

```
      for(j = 0; j < n; j++)
        cout << x[i][j] << " ";
      cout << endl;
    }
    cout << "Initialisation d'une matrice carrée à l'unité :" << endl;
    for(i = 0; i < n; i++)
      for(j = 0; j < n; j++)
        if(i == j)
          x[i][j] = 1.;
        else
          x[i][j] = 0.;
    for(i = 0; i < n; i++) {
      for(j = 0; j < n; j++)
        cout << x[i][j] << " ";
      cout << endl;
    }
    for(i = 0; i < n; i++)
      free(x[i]);
    free(x);
    return 0;
  }
```

ce qui donne :

Initialisation d'une matrice carrée à 0 :
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
Initialisation d'une matrice carrée à l'unité :
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1

L'allocation, l'initialisation et l'affichage pourront être rendus simples et concis par l'utilisation de fonctions.

4.1.2 Cas où les valeurs d'initialisation sont données explicitement

Mêmes méthodes que pour un seul indice. Voir fonctions à l'annexe située à la fin de ce chapitre.

4.2 Tableau dynamique à plus d'un indice en argument d'une fonction

Remarque préliminaire

A l'allocation du tableau dynamique à deux indices `x` faite dans la sous-section précédente ajoutons les instructions :

```
double** y;
y = x;
```

`y` contient alors la même adresse de pointeur de `double` que `x`, `x[i]` et `y[i]` désignent le même emplacement de pointeur de `double` donc le même pointeur de `double`, `x[i][j]` et `y[i][j]` désignent le même emplacement de `double` donc le même `double`, `x` et `y` sont donc deux noms différents pour le même tableau dynamique à deux indices, toute modification des éléments de l'un est mécaniquement appliquée à ceux de l'autre.
On le vérifie sur l'exemple suivant :

```
#include<iostream>
#include<stdlib.h>
#include<iomanip>
using namespace std;
int main() {
  double **x, **y;
  int i, j, p, q;
  p = 3; q = 4;
  // On utilise le pointeur de pointeur x pour déclarer un tableau dynamique à deux indices
  x = (double**)malloc(p * sizeof(double*));
  for(i = 0; i < p; i++)
    x[i] = (double*)malloc(q * sizeof(double));
  //-----------------------------------------------------------------------------
  // On donne des valeurs quelconques aux éléments du tableau dynamique x
  for(i = 0; i < p; i++)
    for(j = 0; j < q; j++)
      x[i][j] = i*q + j;
  // On fait afficher les valeurs de x
  cout << endl;
  for(i = 0; i < p; i++) {
    for(j = 0; j < q; j++)
      cout << setw(3) << x[i][j];
    cout << endl;
  }
  // On met dans y l'adresse qui se trouve dans x
  y = x; // On pourrait aussi bien mettre cette instruction ci-dessus, juste après
         // x = (double**)malloc(p * sizeof(double*));
  // On fait afficher les valeurs de y
  cout << endl;
  for(i = 0; i < p; i++) {
    for(j = 0; j < q; j++)
      cout << setw(3) << y[i][j];
    cout << endl;
  }
  //-----------------------------------------------------------------------------
  // On donne des valeurs quelconques aux éléments du tableau dynamique y
  for(i = 0; i < p; i++)
    for(j = 0; j < q; j++)
      y[i][j] = i + j*p;
  // On fait afficher les valeurs de y
  cout << endl;
  for(i = 0; i < p; i++) {
    for(j = 0; j < q; j++)
      cout << setw(3) << y[i][j];
    cout << endl;
  }
  // Bien noter qu'ici on n'écrit pas x = y; qui serait possible mais inutile
  // On fait afficher les valeurs de x
  cout << endl;
  for(i = 0; i < p; i++) {
    for(j = 0; j < q; j++)
      cout << setw(3) << x[i][j];
    cout << endl;
  }
  //-----------------------------------------------------------------------------
  // Libérer la mémoire réservée; on pourrait de façon équivalente
  // utiliser free(y[i]) et free(y)
```

```
    for(i = 0; i < p; i++)
      free(x[i]);
    free(x);
    return 0;
  }
```

Considérons maintenant l'exemple de l'appel de la fonction `f` suivante par le `main` :

```
... f(..., double** y, int p, int q, ...) {
  ...
}
//-------------------------------------------------------
int main() {
  double** x;
  int i, p, q;
  ...
  p = ...; q = ...;
  x = (double**)malloc(p*sizeof(double*));
  ...
  for(i = 0; i < p; i++)
    x[i] = (double*)malloc(q*sizeof(double));
  ...
  ... f(..., x, p, q, ...) ...;
  ...
}
```

L'appel de la fonction `f` avec l'argument effectif `x` revient à faire `y = x;`
Le temps de l'exécution de `f` :

- `y` contient la même adresse que `x`
- `y` donne accès aux mêmes emplacements mémoire que `x`, emplacements que l'on peut atteindre par la notation « crochets » `y[i][j]`
- travailler sur les éléments `y[i][j]` est strictement équivalent à travailler sur les éléments `x[i][j]`.

5 Allocation automatisée des tableaux dynamiques

L'emploi des tableaux dynamiques tel que présenté jusqu'ici est assez lourd. On peut le simplifier en utilisant une propriété à priori non évidente de l'allocation dynamique : si elle est faite dans une fonction elle demeure après la fin de l'exécution de cette fonction. Ceci est illustré par l'exemple suivant :

```
#include<iostream>
#include<stdlib.h>
using namespace std;
double* allo() {
  int i, n = 5;
  double* z = (double*)malloc(n*sizeof(double));
  for(i = 0; i < 5; i++)
    z[i] = i;
  for(i = 0; i < 5; i++)
    cout << z[i] << endl;
  cout << endl;
  return z;
}
int main() {
  int i;
  double* x;
  x = allo();
  for(i = 0; i < 5; i++)
    cout << x[i] << endl;
  free(x);
```

```
  return 0;
}
```

qui donne le résultat :

```
0
1
2
3
4

0
1
2
3
4
```

On vérifie que l'emplacement mémoire réservé dans la fonction `allo` le reste après la fin de l'exécution de cette fonction[8]. En contre-partie, puisque chaque appel de la fonction alloue un nouvel emplacement mémoire, il ne faut pas oublier de désallouer dès que c'est possible par l'instruction `free`, sans quoi la place inutilement réservée peut croître jusqu'à saturer la mémoire.
On peut alors automatiser l'allocation d'un tableau dynamique en utilisant cette propriété comme dans l'exemple suivant :

```
#include<iostream>
#include<stdlib.h>
using namespace std;
double* allo(int n) {
  double* z = (double*)malloc(n * sizeof(double));
  return z;
}
int main() {
  int n;
  n = ...;
  double* x = allo(n);
  ...
  return 0;
}
```

et l'allocation d'un tableau dynamique se fait donc simplement par l'instruction : [9]

```
double* x = allo(n);
```

Dans `allo` on peut ajouter un test vérifiant que l'allocation s'est faite correctement :

```
double* allo(int n) {
  double* z = (double*)malloc(n * sizeof(double));
  if(z == NULL) {
    cerr << "Mémoire insuffisante" << endl;
    exit -1;
  }
  return z;
}
```

`cerr` est la sortie standard d'erreur, qui permet de séparer les diagnostics d'erreur à l'exécution des sorties normales produites par `cout`.
De même on écrit une fonction pour désallouer :

```
void desallo(double* z) {
  free(z);
}
```

8. Cet exemple n'est pas à lui seul une preuve : l'emplacement mémoire pourrait « par hasard » ne pas avoir été ré-utilisé.

9. Une version alternative de la fonction `allo`, renvoyant l'adresse en argument et non par la valeur de la fonction, serait : `void allo(double** z, int n) {*z = (double*)malloc(n * sizeof(double));}` Remarque bien les `*` supplémentaires ! Dans ce cas l'allocation d'un tableau dynamique se ferait par : `double* x; allo(&x, n);`

et dans le programme appelant il suffit d'écrire :

```
desallo(x)
```

pour libérer l'espace mémoire.

Remarque

Dans ce cas la fonction `desallo` est inutile, il suffit en effet d'écrire directement `free(x)`. Mais, à partir d'un tableau dynamique à deux indices, il devient intéressant d'en disposer.

Des fonctions bâties sur le principe de `allo` et `desallo` pour l'allocation et la désallocation de tableaux dynamiques de type `char`, `int` ou `double` à un, deux ou trois indices et de noms respectifs :

`C_1`, `C_2`, `C_3`, `I_1`, `I_2`, `I_3`, `D_1`, `D_2`, `D_3`
`f_C_1`, `f_C_2`, `f_C_3`, `f_I_1`, `f_I_2`, `f_I_3`, `f_D_1`, `f_D_2`, `f_D_3`

ainsi que des fonctions destinées à l'initialisation des tableaux dynamiques, de noms respectifs :

`ini_C_1`, `ini_C_2`, `ini_C_3`, `ini_I_1`, `ini_I_2`, `ini_I_3`, `ini_D_1`, `ini_D_2`, `ini_D_3`

sont déjà écrites et mises à la disposition des utilisateurs dans la bibliothèque des fonctions du Magistère. Leur mode d'emploi est décrit à l'annexe située à la fin de ce chapitre.

Annexe : fonctions simplifiant l'usage des tableaux dynamiques

On rappelle qu'on appelle dimension du $n^{ième}$ indice d'un tableau dynamique le nombre de valeurs prises par cet indice.
Pour simplifier l'allocation, l'initialisation et la désallocation explicite des tableaux dynamiques, des fonctions ont été écrites et sont disponibles dans la bibliothèque des fonctions du Magistère. Leurs noms sont les suivants :

	Allocation			Initialisation			Désallocation		
Nombre d'indices	Type du tableau			Type du tableau			Type du tableau		
	`char`	`int`	`double`	`char`	`int`	`double`	`char`	`int`	`double`
1	C_1	I_1	D_1	ini_C_1	ini_I_1	ini_D_1	f_C_1	f_I_1	f_D_1
2	C_2	I_2	D_2	ini_C_2	ini_I_2	ini_D_2	f_C_2	f_I_2	f_D_2
3	C_3	I_3	D_3	ini_C_3	ini_I_3	ini_D_3	f_C_3	f_I_3	f_D_3

Leur mode d'emploi est exposé dans la suite de cette annexe.

Allocation des tableaux dynamiques

Les encadrés suivants décrivent l'allocation, l'initialisation explicite éventuelle et la désallocation de tableaux dynamiques de type `double` pour un, deux ou trois indices.
Pour les types `int` ou `char`, il suffit de remplacer `double` par `int` ou `char` et `D` par `I` ou `C`.

Un indice

```
#include<bibli_fonctions.h>
...
int p;
p = ...;
double* x = D_1(p);
ini_D_1(x, p, 4.5, -6.4, ...); // initialisation explicite éventuelle
...
f_D_1(x,p);
...
```

Deux indices

```
#include<bibli_fonctions.h>
...
int p, q;
p = ...; q = ...;
double** x = D_2(p,q);
ini_D_2(x, p, q, 4.5, -6.4, ...); // initialisation explicite éventuelle
...
f_D_2(x,p,q);
...
```

Pour l'initialisation les valeurs de la séquence `4.5,-6.4,...` sont attribuées aux éléments de tableau dans l'ordre `x[0][0]`, `x[0][1]`, `x[0][2]`, ..., `x[1][0]`, `x[1][1]`, `x[1][2]`, etc., c'est à dire en faisant varier le second, et donc dernier, indice en premier.

Trois indices

```
#include<bibli_fonctions.h>
...
int p, q, r;
p = ...; q = ...; r = ...;
double*** x = D_3(p,q,r);
ini_D_3(x, p, q, r, 4.5, -6.4, ...); // initialisation explicite éventuelle
...
f_D_3(x,p,q,r);
...
```

Pour l'initialisation les valeurs de la séquence `4.5,-6.4,...` sont attribuées aux éléments de tableau selon la même règle que pour les tableaux à deux indices, c'est à dire en faisant varier le dernier indice en premier et l'avant dernier en second.