Table des matières

Introduction

Une variable informatique n'est pas une simple variable mathématique. Derrière une déclaration `double a=3;` se cachent plusieurs choses :

— le nom de la variable qui est une chaîne de caractères utilisée dans le code,
— l'emplacement où elle est stockée dans la mémoire (et la taille qu'elle occupe),
— la valeur qui est inscrite dans la mémoire.

Derrière le nom de la variable (la *référence*) se cache toute une fiche d'identité qui est créée lors de la déclaration de la variable et qui contient en particulier l'emplacement de la mémoire (le *pointeur*) et la nature (le *type*) de la variable. En utilisant la référence ou le pointeur, on peut ainsi aller modifier la valeur inscrite dans la mémoire.

Le but de ce second chapitre est de présenter les différents moyens de manipuler une variable.

1 Références

1.1 Passage des arguments par référence dans les fonctions

Exercice 1 (De l'impossibilité, pour une fonction, de modifier son argument)**.** *Que va rendre le programme suivant ?*

```
#include <iostream>

void ajoute_un(int x) {x++;}

int main() {int x=2; ajoute_un(x);
        std::cout<<x<<std::endl;}
```

(nom du fichier : `Chapitre2/prog01.cpp`*)*

La raison pour laquelle dans la fonction `ajoute_un`, on ne peut modifier l'argument est que dans l'estimation de `ajoute_un(x)`, ce qui joue le rôle de `x` est une *copie*[1] du paramètre effectif `x`. On dit que dans `ajoute_un(x)`,

1. En particulier, pour des variables de grande taille, cela implique qu'une grande quantité de mémoire est utilisée pour une simple duplication...

l'argument `x` est passé par **valeur**, et que donc, il est en **lecture** seule et non en **écriture** (c'est à dire qu'on peut consulter sa valeur mais non la modifier). La solution est d'écrire une fonction `ajoute_un_ref` où l'argument est passé par **référence**.

```
#include <iostream>

void ajoute_un_ref(int &x) {x++;}

int main() {int x=2; ajoute_un_ref(x);
        std::cout<<x<<std::endl;
        //ajoute_un_ref(4);
        }
```

(nom du fichier : `Chapitre2/prog02.cpp`*)*

Dans `int &x`, `x` est un synonyme du paramètre effectif, qui est en **lecture-écriture** (c'est à dire qu'on peut le lire et le réécrire). *Attention*, cela interdit à présent de mettre une valeur à la place du nom de variable `x`, i.e. on ne peut pas décommenter la dernière ligne de l'exemple précédent ; l'avantage est de ne rien recopier dans la mémoire.

Notons aussi que l'on peut passer l'argument en référence tout en en bloquant la réécriture éventuelle (pour des raisons de sécurité et de clarté du code), avec le mot-clé `const` : l'intérêt d'une telle syntaxe réside alors dans le gain de temps dû au fait que le compilateur ne recopie pas l'argument à chaque appel de la fonction. Ainsi, le programme suivant va déclencher un message d'erreur du compilateur :

```
#include <iostream>

void ajoute_un_ref_const(const int &x) {x++;}

int main() {int x=2; ajoute_un_ref_const(x);
        std::cout<<x<<std::endl;}
```

(nom du fichier : `Chapitre2/prog03.cpp`*)*

Notons aussi que lorsque l'argument est passé en valeur, il peut être une variable, une constante ou une expression, alors que lorsqu'il est passé en référence, il ne peut être qu'une variable. Par exemple, l'appel à `ajoute_un_ref(3)` déclenchera un message d'erreur.

Exercice 2. *Écrire une fonction* `echangeur` *de prototype*

```
void echangeur(int &a, int &b)
```

dont l'effet est de permuter les valeurs des deux variables entières `a` *et* `b`*. La tester.*

Exercice 3 (De l'impossibilité de modifier une constante)**.** *Donner la sortie du programme suivant. Pourrait-on décommenter l'avant-dernière ligne ?*

```
#include <iostream>
using namespace std;

void f(int i) {i++;}

void g(int &i) {i++;}

int main() {int i=4;
        f(i);   cout<<i<<endl;
        f(0);   g(i);
        cout<<i<<endl;
        //g(0);
        }
```

(nom du fichier : `Chapitre2/prog04.cpp`*)*

1.2 Références à une variable et fonctions rendant une référence

Le passage des arguments par référence dans les fonctions est un cas particulier d'une approche plus générale : celle des *références* à des variables. Une référence à une variable est un synonyme (un *autre nom*) pour cette variable, déclarée comme suit :

```
int x;

int &r=x;
```

Ici, `r` est une référence à `x`. À partir de cette commande, une seule valeur et un seul emplacement mémoire de `int` se cache derrière les variables `x` et `r`, jusqu'à ce qu'on change la référence `r` vers une autre variable que `x`.

Exercice 4. *Que va rendre le programme suivant ?*

```
#include <iostream>
using namespace std;

int main() {int x=2; int &r=x;
        cout<<x<<" "<<r<<endl;
        x=5;
        cout<<x<<" "<<r<<endl;
        r=-1;
        cout<<x<<" "<<r<<endl;}
```

(nom du fichier : `Chapitre2/prog05.cpp`*)*

Les références peuvent être utilisées avantageusement dans des fonctions rendant une référence, comme dans l'exercice suivant.

Exercice 5 (Référence vers une variable comme un alias de cette variable)**.** *Que vont rendre les programmes suivants ?*

```
#include <iostream>
using namespace std;

int& max( int& m, int& n ) {return ( m > n ? m : n );}

int main() {int m = 2;
        int n = 3;
        max(m,n) = 1;
        cout << "m = " << m << ", n = " << n << endl;}
```

(nom du fichier : `Chapitre2/prog06.cpp`*)*

```
#include <iostream>
using namespace std;

int& max(int &m, int &n, int &p) {
 if ((m>=n)&&(m>=p)) return m;
 else if ((n>=m)&&(n>=p)) return n;
 else if ((p>=m)&&(p>=n)) return p;}

int main() {int a=1,b=2,c=3;
        cout<<a<<" "<<b<<" "<<c<<endl;
        max(a,b,c)=0;
        cout<<a<<" "<<b<<" "<<c<<endl;}
```

(nom du fichier : `Chapitre2/prog07.cpp`*)*

En effet, le signe d'affectation `=` prend une référence à gauche et une valeur à droite si bien que l'endroit où stocker une valeur peut lui-même être déterminé par une fonction qui renvoie une référence !

2 Tableaux statiques (non dynamiques)

2.1 Définition

Un tableau est une suite d'éléments de même type accessibles par indexation. Pour définir un tableau appelé `mon_tableau` unidimentionnel de 5 éléments de type `mon_type`, on utilise la syntaxe suivante :

```
int N=5;

mon_type mon_tableau[N];
```

Les éléments du tableau sont alors numérotés de 0 à `N`-1, et on accède au ième avec la commande `mon_tableau[i]` (lecture et écriture). Les tableaux peuvent aussi se définir avec des accolades au moment de leur déclaration (et uniquement au moment de leur déclaration !) :

```
double T[5]={1.1, 2, 6, 7, 3};
```

En général, l'usage de tableaux statiques n'est pas nécessairement une bonne idée. En effet, rares sont les applications où l'on sache à l'avance la taille des tableaux *avant* compilation. C'est une situation fréquente quand on programme pour soi ; vis-à-vis d'un client dont on ne sait pas ce qu'il fera du programme, au contraire, seul un programme *compilé* peut être livré et il est hors de question qu'il aille modifier des paramètres du code. De plus, au cours d'une même exécution, il est souvent souhaitable que la taille des tableaux puisse varier. Comme contre-exemple à cette règle générale, citons néanmoins la situation suivante : dans un programme de géométrie dans l'espace, on sait à l'avance que tous les tableaux seront de taille 3 et l'usage de tableaux statiques est acceptable !.

2.2 Un raccourci utile : l'instruction `typedef`

Cette construction, dont la syntaxe est

`typedef` *déclaration*

permet de donner un nom à un type dans le but d'alléger par la suite les expressions où il figure. Pour déclarer un nom de type, on fait suivre le mot réservé **`typedef`** d'une expression identique à une déclaration de variable, dans laquelle le rôle du nom de la variable est joué par le nom du type qu'on veut définir. Exemple :

```
typedef double mes_matrices[10]
```

définit `mes_matrices` comme le nom du type de matrices de 10 éléments de type **`double`**. Ensuite, on déclare une telle matrice `M` par la syntaxe

```
mes_matrices M;
```

2.3 Les tableaux multidimentionnels

Tout se passe comme pour les tableaux unidimentionnels. Il suffit de savoir que les tableaux p-dimentionnels de taille $n_1 \times n_2 \times \cdots \times n_p$ sont définis par la suite des n_1 tableaux $p-1$-dimentionnels de taille $n_2 \times \cdots \times n_p$ qui les constituent, chacun d'entre eux étant défini par la suite des n_2 tableaux $p-2$-dimentionnels de taille $n_3 \times \cdots \times n_p$ qui le constituent, etc...

Exercice 6 (Utilisation de typedef, les matrices sont définies par lignes en C++)**.** *Donner la sortie du programme suivant.*

```
#include <iostream>
using namespace std;

int main() {typedef double mes_matrices[3][2];
        mes_matrices m = { {1,2}, {2,3}, {3,4}} ;
        mes_matrices n = { 1, 2, 2, 3, 3, 4 } ;
        for(int i=0; i<3; i++) {
                for(int j=0;j<2;j++) cout<<(m[i][j]==n[i][j])<<endl;}}
```

(nom du fichier : `Chapitre2/prog08.cpp`*)*

2.4 Passage des tableaux en arguments des fonctions

On peut, comme dans l'exemple suivant, écrire des fonctions définies pour des tableaux de taille fixe

```
#include <iostream>
using namespace std;

double f(double t[4]){return t[0];}

int main() {typedef double mes_matrices[2][4];
        mes_matrices m = { {1,2,3,4}, {4,5,6,7} };
        mes_matrices n = { 1, 2, 3, 4, 5, 6, 7 } ;
        for(int i=0; i<2; i++) {
                cout<<f(m[i])<<" "<<f(n[i])<<endl;}}
```

(nom du fichier : `Chapitre2/prog09.cpp`*)*

L'idée couramment utilisée (cf exercice suivant), pour pouvoir traiter le cas de tableaux de longueurs diverses, est de passer la taille du tableau en argument et d'utiliser les déclarations

```
mon_type T[]
```

(qui signifie "`T` est un tableau d'un nombre quelconque d'éléments de type `mon_type`").

Il est **important** d'avoir toujours en tête que `C++` n'est pas capable de retrouver la taille d'un tableau à partir de sa seule adresse : il faut donc toujours faire attention à conserver l'information sur la taille du tableau !

Exercice 7. *Que va rendre le programme suivant ?*

```
#include <iostream>
using namespace std;

void affiche_tableau(double T[], int N){
        for(int i=0; i<N; i++) cout<<T[i]<<" ";
        cout<<endl;}

int main() {double T[20] = { 1, 2, 3, 4, 5, 6, 7 } ;
        affiche_tableau(T, 20);}
```

(nom du fichier : `Chapitre2/prog10.cpp`*)*

2.5 Exercices

Exercice 8 (Tableaux 1)**.** *Écrire un programme qui demande à l'utilisateur de saisir 10 entiers, que l'on stocke dans un tableau ainsi qu'un entier V. Le programme doit rechercher si V se trouve dans le tableau et doit supprimer la dernière occurence de V en décalant d'une case vers la gauche les éléments suivants et en rajoutant un 0 à la fin du tableau. Le programme doit ensuite afficher le tableau final.*

Exercice 9 (Tableaux 2)**.** *Écrire un programme qui demande à l'utilisateur un entier $n \geq 2$, de taper n entiers, qui seront stockés dans un tableau. Le programme doit ensuite afficher soit "le tableau est croissant (au sens large)", soit "le tableau est décroissant (au sens large)", soit "le tableau est constant", soit "le tableau est quelconque".*

Exercice 10 (Tableaux dynamiques et références)**.** *Écrire une fonction* `f` *de prototype*

```
bool f(int t[], int n, int &v)
```

(`t` *est un tableau d'entiers de taille* `n`*).* `f` *doit renvoyer par un booléen* `b` *indiquant s'il existe une valeur comprise (au sens large) entre 1 et 4 dans les* `n` *premières cases du tableau* `t`*. Si* `f` *renvoie* `true`*,* `v` *est égal à la valeur de la première case du tableau comprise entre 1 et 4. Tester cette fonction.*

3 Pointeurs

3.1 Adresse d'une variable

Comme nous l'avons vu en introduction, une variable a trois attributs fondamentaux : son type, son nom (référence) et son adresse dans la mémoire. Les deux premiers nous sont maintenant bien connus, le troisième est celui qui mène à la notion de pointeur. L'adresse d'une variable `x` est elle-même un entier (le numéro du bit où commence l'objet), auquel on accède par la commande `&x`, qui indique l'endroit, dans la mémoire, où est stockée la variable `x` (l'espace de stockage alloué à `x` dépend ensuite bien entendu de son type : un booléen nécessite moins d'espace qu'un réel). **Attention**, le symbole `&` (esperluette) fait penser aux références, vues au paragraphe précédent, mais il ne s'agit pas de la même chose (en pratique, il n'y a jamais d'ambiguité).

3.2 Déclaration d'un pointeur

Un *pointeur* `p` vers une variable de type `mon_type` est l'adresse d'une variable de type `mon_type`. Un tel objet **se déclare** ainsi :

```
mon_type *p;
```

(on peut aussi écrire `mon_type* p;`) ou bien en désignant directement la variable dont il est l'adresse, ainsi :

```
mon_type x;

mon_type *p=&x;
```

La variable pointée par un pointeur `p`, appelé son *contenu* est accessible par la commande `*p`. Ainsi, pour toute variable `x`, on a

`*&x==x`

et pour tout pointeur `p`, on a

`&*p==p`

Exercice 11 (Pointeurs)**.** *Quels seront les effets des exécutions des programmes suivants ?*

```
#include <iostream>
using namespace std;

int main(){
        double *a;//Definit a comme un pointeur vers un double
        double c; //Definit l'emplacement memoire pour un double, appele c
        c=5.0; //Attribue au double c la valeur 5.0
        a=&c; //a devient l'adresse de c
        cout<<*a<<endl;}
```

(nom du fichier : `Chapitre2/prog11.cpp`*)*

```
#include <iostream>
using namespace std;

int main(){double *a;
        double *b;
        double c=1.0,d=2.0,f,g;
        double *e;
        double *h;
        a=&c;
        b=&d;
        f=*a+*b;
        e=&f;
        g=(*a+*b+*e)/2.0;
        h=&g;
        cout<<*h*2.0<<endl;}
```

(nom du fichier : `Chapitre2/prog12.cpp`*)*

```
#include <iostream>
using namespace std;

int main() {int x=1,y=2;
        int *p=&x,*q;
        q=&y;
        int z=*q;
        int *r=&z;
        cout<<*r+x<<endl;}
```

(nom du fichier : `Chapitre2/prog13.cpp`*)*

Comme le montre l'exercice suivant, on peut créer des pointeurs vers des pointeurs, etc...

Exercice 12 (Pointeurs vers des pointeurs)**.** *Quels sera l'effet de l'exécution du programme suivant ?*

```
#include <iostream>
using namespace std;

int main(){double x=2;
        double* px=&x;
        double** ppx=&px;
        double*** pppx=&ppx;
        ***pppx=5;
        cout<<x<<endl;}
```

(nom du fichier : `Chapitre2/prog14.cpp`*)*

Exercice 13 (Une fonction ne peut modifier son argument que s'il est passé en référence ou par adresse)**.** *Donner la sortie du programme suivant.*

```
#include <iostream>
using namespace std;

void f(int i) {i++;}

void g(int *i) {(*i)++;}

void h(int &i) {i++;}

int main() {int i=2;
        f(i);
        cout<<i<<endl;
        g(&i);
        cout<<i<<endl;
        h(i);
        cout<<i<<endl;}
```

(nom du fichier : `Chapitre2/prog15.cpp`*)*

Exercice 14 (Utilisation des pointeurs pour modifier les arguments d'une fonction)**.** *Écrire une fonction*

```
void echange_contenus(int *p, int *q)
```

qui a pour effet de permuter les contenus des pointeurs vers les entiers `p` *et* `q`*. La tester.*

Exercice 15 (Pointeurs versus références 1)**.** *a) Dans le programme suivant, remplacer les cases REMPLIR par le contenu adéquat. b) Réécrire le programme de façon à remplacer la première fonction par une fonction de prototype :*

```
void minmax(int i, int j, int& min, int& max)
```

```
#include<iostream>
using namespace std;

void minmax(int i, int j, int* min, int* max){
    if(i<j) { *min=i; *max=j; }
    else { *min=j; *max=i; }}

int main(){
    int a, b, w, x;
    cout << "Tapez la valeur de a : "; cin >> a;
    cout << "Tapez la valeur de b : "; cin >> b;

    minmax(a, b, REMPLIR, REMPLIR);
    cout << "Le plus petit vaut : " << w << endl;
    cout << "Le plus grand vaut : " << x << endl;
    return 0; }
```

(nom du fichier : `Chapitre2/prog16.cpp`*)*

Exercice 16 (Pointeurs versus références 2)**.** *a) Le programme suivant va-t-il fonctionner ? On choisit alors de ne laisser qu'une des trois définitions de* `ajouter`*. Pour laquelle/lesquelles des définitions le programme fonctionnera-t-il ? Qu'affichera-t-il alors sur la console ?*

b) On remplace, dans la fonction main, `ajouter(a,b,c);` *par* `ajouter(a,b,&c);`*. Répondre à nouveau aux questions du a).*

```
#include<iostream>
using namespace std;

void ajouter (int a, int b, int c) { c = a+b; }
void ajouter (int a, int b, int * c) { *c = a+b; }
void ajouter (int a, int b, int & c) { c = a+b; }

main() {
  int a=1; int b=2; int c=0;
  cout << "avant appel de ajouter" << endl;
  cout << "a = " << a << endl;
  cout << "b = " << b << endl;
  cout << "c = " << c << endl;
  ajouter(a,b,c);
  cout << "apres appel de ajouter" << endl;
  cout << "a = " << a << endl;
  cout << "b = " << b << endl;
  cout << "c = " << c << endl;}
```

(nom du fichier : `Chapitre2/prog17.cpp`*)*

Exercice 17 (Tableaux dynamiques et pointeurs)**.** *Écrire (et tester dans un programme) une fonction qui a comme paramètres un tableau d'entiers de taille quelconque, la taille du tableau, et 2 pointeurs vers des entiers min et max. La fonction doit renvoyer dans les entiers pointés par min et max respectivement les plus petits et les plus grands entiers du tableau.*

4 Tableaux et pointeurs

4.1 Le tableau comme pointeur sur son premier élément

Un tableau est en réalité un pointeur sur son premier élément. On peut incrémenter et décrémenter ce pointeur pour parcourir le tableau, comme le montre les programmes suivants. Plus généralement, en tant qu'entiers, les pointeurs sont munis de toute une arithmétique utilisée implicitement dans les formules `t[i]` à comprendre comme `*(t+i)` où le décalage se fait par multiple de la taille de la variable derrière le type de `t`.

En pratique, on utilise rarement cette arithmétique par peur des **erreurs de segmentation** qui correspondent à une opération du type `*p` lorsque `p` pointe vers une zone mémoire qui n'a jamais été allouée (par exemple si l'on regarde `t[-1]` ou `t[N]` lorsque `N` est la taille du tableau).

```
#include <iostream>
using namespace std;

int main(){int N=4;
        char T[4] = {'a','b','c','d'};
        for(char *p=T; p<T+N; p++) cout<<*p<<endl;}
```

(nom du fichier : `Chapitre2/prog18.cpp`*)*

```
#include<iostream>
#include <iomanip>
#include <cstdlib>
//bibliotheques permettant d'affiner l'affichage sur la console
using namespace std;

int main(){
    double t[10];

    for ( int i=0 ; i<5 ; i++) {t[i] = i*i + 1/3.;}
    for ( int j=5 ; j<10 ; j++) {t[j] = -j * j-1/3.;}

    double* x;
    x=t; // pointeur vers le premier element
    cout << fixed << setprecision(4); //"fixed" permet que les nombres
    //  apparaissent sur la meme colonne (qu'ils soient positifs ou negatifs)
    //  "setprecision": nombre de chiffres apres la virgule

    for ( int k=0 ; k<10 ; k++){
        cout << setw(8) <<  *x << endl;
        //"setw(8) " est le nombre max de caracteres dans l'affichage
        x++; // pointeur vers l'element suivant
        }
    cout << endl;}
```

(nom du fichier : `Chapitre2/prog19.cpp`*)*

Explications :

- Dans ce dernier exemple, `t` est un tableau statique de 10 entiers et `x` est un pointeur vers un entier.
- L'instruction `x=t;` permet de faire pointer `x` vers la première case du tableau `t`. Il est équivalant d'écrire `x=t;` que d'écrire `x=&t[0];`
- À l'intérieur d'une boucle for, on va afficher `*x`, c'est-à-dire l'entier pointé par `t` et à chaque étape, on écrit `x++`, ce qui incrémente la valeur de `t` de la taille d'un entier.
- `x` va donc pointer successivement `t[0]`, `t[1]`,..., `t[9]`. On va donc afficher une à une toutes les cases du tableau.

Exercice 18 (Tableaux multidimensionnels, incrémentation des pointeurs et précision de l'affichage)**.** *Écrire une fonction de prototype*

`void aff_ligne(double T[],int p, int prec_dig)`

qui a pour effet d'afficher (en ligne) le tableau `T` *de taille* `p` *avec* `prec_dig` *chiffres après la virgule, en utilisant le principe d'incrémentation des pointeurs utilisé ci-dessus. S'en servir dans un programme qui demande à l'utilisateur un entier* m*, un nombre de chiffes après la virgule, et affiche (en deux dimensions) la matrice* $[i/j]_{1\leq i,j\leq m}$*.*

4.2 Allocation dynamique de mémoire

Un pointeur ne peut accueillir que l'adresse que d'une zone déjà allouée de la mémoire. En conséquence, le programme suivant

```
int main(){ double *p;
        p=&5;}
```

(nom du fichier : `Chapitre2/prog20.cpp`*)*

ne pourra être compilé, le message d'erreur `error: invalid lvalue in assignment` étant rendu lorsque l'on essayera de le compiler. En effet, 5 est une valeur qui peut être stockée n'importe où et n'a pas fait l'objet d'un stockage dans une variable préalablement allouée. Une solution pour résoudre ce problème de compilation est donnée par la variante suivante, qui, elles, se compilent bien.

```
int main(){ double *p;
        double x=5;
        p=&x;}
```

(nom du fichier : `Chapitre2/prog21.cpp`*)*

Pour éviter d'avoir une variable auxiliaire `x` à construire, on peut utiliser le mécanisme d'*allocation dynamique de mémoire*, qui permet d'initialiser (*i.e.* construire) des objets sans leur donner de nom, via leur adresse. La commande *ad hoc* s'appelle `new`. Son fonctionnement est présenté dans les programmes suivants.

```
#include <iostream>
using namespace std;


int main(){ double *p=new double(5);
        cout<<*p<<endl;
        delete p;}
```

(nom du fichier : `Chapitre2/prog22.cpp`*)*

```
#include <iostream>
using namespace std;

int main () {int x=1; int *p, *q, *r;
        p=&x;
        q=new int;
        *q=4;
        r=new int(5);
        cout<<*p+*q+*r<<endl;
        delete q;
        delete r;}
```

(nom du fichier : `Chapitre2/prog23.cpp`*)*

La mémoire allouée dynamiquement (i.e. avec `new`) doit absolument être désallouée avec `delete`. En effet, lorsque l'on sort d'un bloc d'instructions (*i.e.* tout ensemble de commandes délimité par des accolades, comme l'appel d'une fonction), toutes les variables construites dans ce bloc sont détruites (sauf si elles sont `static`). Mais les variables créées implicitement sans avoir de nom propre via `new` ne font pas partie de cet ensemble. Si le pointeur qui les contient est détruit, elles deviennent inaccessibles, inutilisables, et il convient de les détruire aussi, afin de libérer de la mémoire vive.

On prendra également garde à ne jamais réallouée l'adresse d'une zone allouée par `new` sans avoir soit utilisé `delete` si l'information de la zone est devenue inutile, soit avoir copié au préalable l'adresse dans une autre variable.

4.3 Allocation dynamique de tableaux

Il s'agit du maniement de tableaux dont la taille est connue à l'*exécution* et non à la *compilation*, motivation principale pour l'utilisation des pointeurs. Dans de très nombreux programmes en effet, on ne sait pas d'avance quelle place une donnée (un tableau par exemple) va devoir utiliser. Les tableaux déclarés par des commandes du type

```
double Tab[10];
```

sont des pointeurs dont la taille est connue *statiquement* (i.e. à la compilation, donc lors de l'écriture du programme). Il est cependant possible de définir des tableaux de taille *dynamique*, i.e. dont la taille est connue au cours de l'exécution du programme. L'allocation (i.e. la définition) de tels tableaux se fait via l'opérateur `new` encore et la désallocation d'un tableau `Tab` créé avec `new` se fait par la commande

```
delete [] Tab;
```

Exemple :

```
#include <iostream>
using namespace std;

int main () {int n;
        cin>>n;
        int *T=new int[n];
        for(int i=0; i<n;i++) T[i]=i*i;
        cout<<T[n-1]<<endl;
        delete [] T;}
```

(nom du fichier : `Chapitre2/prog24.cpp`*)*

L'allocation dynamique de tableaux multidimentionnels se fait selon le modèle suivant.

```
#include <iostream>
using namespace std;

int main () {int n;
        cin>>n;
        int **T=new int*[n];
        for(int i=0; i<n;i++) {
                T[i]=new int[n];
                for(int j=0;j<n;j++) T[i][j]=(i+j)%2;
        }
        int somme=0;
        for(int i=0; i<n;i++)
                for(int j=0;j<n;j++) somme+=T[i][j];
        cout<<somme<<endl;

        for(int i=0; i<n; i++) delete [] T[i];
        delete [] T;}
```

(nom du fichier : `Chapitre2/prog25.cpp`*)*

Exercice 19 (Allocation dynamique et maniement de tableaux)**.** *Dans tout cet exercice, α est un réel strictement positif fixé arbitrairement. On placera donc en préambule de son programme (ou dans le fichier d'en tête) la ligne*

`const double alpha =` *mettre ici la valeur que l'on choisit*

(on essayera, à la fin de l'exercice, de l'ajuster selon un critère qui sera présenté plus loin)

a) Écrire une fonction de prototype

```
void produit_matriciel(double **A, double **B, double **P, int n)
```

qui, pour 3 matrices carrées `A`*,* `B`*,* `P` *de taille* `n`*, affecte à* `P` *la valeur* `A`$\times$`B`*.*

b) Écrire une fonction de prototype

```
double coordonnee_max(double** A, int n)
```

qui, à une matrice carrée de taille `n`*, associe sa coordonnée de plus grande valeur absolue.*

c) Écrire une fonction de prototype

```
void matrice_aleatoire(double** A, int n)
```

qui pour un entier `n` *et une matrice carrée* `A` *de taille* `n` *affecte aux coordonnées de* `A` *des valeurs aléatoires indépendantes de loi uniforme sur l'intervalle*

$$\left[-\frac{1}{n^\alpha}, \frac{1}{n^\alpha}\right].$$

NB : Avec la bibliothèque `cmath`*, x^a s'écrit* `pow(x,a)`*.*

d) Tester ces fonctions en écrivant un programme principal qui demande à l'utilisateur deux entiers n et p, construit une matrice carrée M de taille n, à coordonnées aléatoires indépendantes de loi uniforme sur l'intervalle

$$\left[-\frac{1}{n^\alpha}, \frac{1}{n^\alpha}\right].$$

puis fait apparaître sur une colonne les coordonnées de valeur absolue maximale de M^i pour i variant de 1 à p (essayer de prendre n de l'ordre de 100 et de faire tendre p vers l'infini, raisonnablement).

e) (Facultatif) Essayer d'ajuster la valeur de $\alpha > 0$ de façon à ce que, pour une grande valeur de n fixée (encore une fois, prendre n de l'ordre de 100) et une grande valeur de p fixée, lorsque i varie de 1 à p, la coordonnée de valeur absolue maximale de M^i aie tendance à se "stabiliser", i.e. à ne devenir ni trop proche de 0, ni trop grande.

NB : Cette question n'a pas de réponse lorsque l'on ne se fixe pas de borne pour i !