# Introduction à la programmation
# TD 4 - Pointeurs

Polytech Marseille - 1ère année

Filière Informatique
Alexandra Bac - Benoît Favre

Cette fiche a pour but d'assimiler les pointeurs et structures (ainsi que quelques compléments de cours) puis à les pratiquer dans des cadres de plus en plus complexes.

## 1 TP

### 1.1 Exercices de base sur les pointeurs

**Exercice 1** (Analyse de code)**.** Considérons les fonctions :

```
int foo(int a)
{
  a = a*2;
  return a+3
}
```

```
int bar(int * pb)
{
  *pb = (*pb)*3;
  return *pb+7;
}
```

```
int foobar(int * pc)
{
  return foo(bar(pc));
}
```

```
int barfoo(int d)
{
  int t;
  t = foo(d);
  return bar(&t);
}
```

(i) En considérant seulement le type des variables et sans essayer de comprendre le résultat de l'exécution, indiquez quelles sont les instructions correctes dans le code suivant :

```
int a;
int * p = NULL;
a = 10;
*p = &a;
p = &a;
*a = 5;
*p = 5;
```

```
a = foo(a);
a = foo(p);
*p = foo(a);

a = bar(&a);
*p = bar(p);
```

```
a = foobar(*a);
p = foobar(*p);
*p = foobar(&a);

*p = barfoo(*p);
p = barfoo(p);
&p = barfoo(&p);
```

(ii) Donnez la valeur des différentes variables présentes après exécution du code suivant :

```
int a = 10, b = 20, c = 50, d = 70, ra, rb, rc, rd;
ra = foo(a);
rb = bar(&b);
rc = foobar(&c);
rd = barfoo(d);
```

**Exercice 2** (Allocation dynamique)**.** On veut simplifier la création de tableaux dynamiques et pour cela, créer deux fonctions "boîte à outil" :

(i) Ecrire une fonction de prototype :

```
int * create_fill_tab(int n)
```

créant et renvoyant un tableau de taille `n` contenant des valeurs entières aléatoires comprises entre 0 et une constante `M`. Pourquoi cette fonction qui renvoie un tableau retourne un `int *` ?

(ii) Ecrire une fonction de protytype :

```
int * my_realloc(int *T, int n, int new)
```

"modifiant la taille" du tableau `T` (de taille `n`) en réallouant un emplacement de taille `new`, en y copiant les données de `T` correspondantes et en retournant l'adresse de ce tableau.

(iii) Dans le `main`, lesquelles de ces séquences sont correctes ? Pourquoi la troisième ne fait-elle rien ?

```
int * T = create_fill(10) ;
T = my_realloc(T, 10, 20) ;
```

```
int *T ;
T = my_realloc(T, 10, 20) ;
```

```
int * T = create_fill(10) ;
my_realloc(T, 10, 20) ;
```

```
int * T = create_fill(10) ;
int * tmp ;
tmp = my_realloc(T, 10, 20) ;
free(T);
T = tmp ;
```

(iv) On veut écrire une version "à effet de bord" de la fonction de réallocation, c'est-à-dire une fonction modifiant le tableau `T` (au lieu de simplement retourner un nouveau tableau). Pourquoi son prototype est-il :

```
void my_realloc2(int **T, int n, int new)
```

Ecrire cette fonction (qui libèrera la mémoire de l'ancien emplacement devenu inutile).

(v) Enfin, pour terminer, pour explorer la différence (subtile) entre tableaux et pointeurs, tester le code suivant :

```
int T[5] = {1, 2, 3, 4, 5};
printf("--1--\n");
my_realloc(T, 5, 10) ;
printf("--2--\n");
T = my_realloc(T, 5, 10);
printf("--3--\n");
```

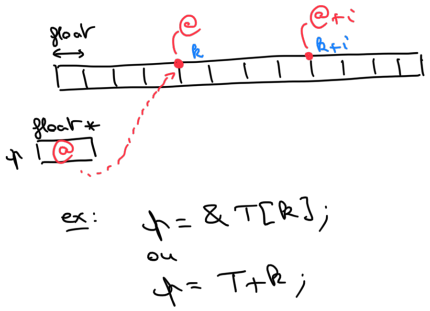
Que se passe-t-il ?

## 1.2 Complément de cours 1 : arithmétique des pointeurs

En C, les pointeurs sont à la fois importants et très utilisés du fait, entre autre, de cette assimilation pointeurs-tableaux. Le langage offre donc un ensemble d'opération arithmétiques sur les pointeurs permettant d'exploiter cette identification.

Toutes ces opérations partent de l'idée qu'un pointeur `Type *` peut désigner l'adresse de la première case d'un tableau d'éléments de type `Type`. L'arithmétique sur les pointeurs tiendra donc automatiquement compte du type de données considéré et permettra de se déplacer dans ce tableau via les adresses.

Ainsi, si `p` désigne l'adresse de la case `k` d'un tableau, `p+i` (resp. `p-i`) désignera l'adresse de la case `k+i` (resp. `k-i`). On aura de même accès aux opérations `p++`, `p--` (avancer, reculer d'une case), `p+=...`, etc. ...




Attention, cette arithmétique tient compte du type de `p` et incrémente donc l'adresse du bon nombre d'octets en fonction du type des données (et donc de la taille des cases) du tableau.

**Exercice 3** (Arithmétique des pointeurs)**.** On considère les déclarations suivantes :

```
int a[8] = {5, 15, 34, 54, 14, 2, 52, 72};
int *p = &a[1];
int *q = &a[5];
```

(i) Quelle est la valeur de `*(p+3)` ?

(ii) Quelle est la valeur de `q-p` ?

(iii) La condition `p<q` est-elle vraie ou fausse ?

(iv) La condition `*p<*q` est-elle vraie ou fausse ?

(v) Réécrivez le code `int *p = &a[1], *q = &a[5];` sans utiliser de crochets.

## 1.3 Complément de cours 2 : chaînes de caractères en C

Comme nous l'avons vu, le C ne comporte pas de type `string`. Pourtant, vous avez tous écrit `''Hello world''` ... En fait, les chaînes de caractères sont codées comme des tableaux de caractères terminés par le caractère spécial `\0`. Attention à ce caractère spécial qui permet des opérations impossibles sur des tableaux standards (comme par exemple calculer la longueur d'une chaîne de caractères par recherche du caractère `\0` justement).

Une chaîne de $n$ caractères est donc stockée dans un tableau de taille $n+1$ ! Et votre `"Hello world"` est donc codé par :

| H | e | l | l | o | | w | o | r | l | d | \0 |
|---|---|---|---|---|---|---|---|---|---|---|---|

**Exercice 4** (Sur le type `char *`)**.** Les prototypes des fonctions de la bibliothèque C sur les chaînes de caractères utilisent le type `char *` au lieu de `char [ ]`.

(i) Expliquez pourquoi cela est possible.

(ii) Dans la suite de l'exercice, on parlera de chaîne de caractères pour un objet de type `char *` (qui devra être associé à une chaîne) et on utilisera la notation pointeur (aucun crochet).

(a) Écrivez une fonction `int length_of_string(char * ch)` qui renvoie la longueur de la chaîne de caractères `ch`.

(b) Écrivez une fonction `int nb_occ_in_string(char c, char * ch)` qui compte le nombre d'occurrences du caractère `c` dans la chaîne `ch`.

(c) Pour tester la fonction `length_of_string` on utilise la fonction :

```
int main () {
  char * ch = NULL;
  int lg;
```

```
  printf("entrer une chaine\n");
  scanf("%s",ch);
  lg=length_of_string(ch);
  printf("longueur %d\n",lg);
  return 0;
}
```

Le compilateur signale `warning: ‘ch’ is used uninitialized in this function` et l'exécution donne `Erreur de segmentation` après que l'utilisateur a rentré une chaîne de caractères. Expliquez ce qui se passe et suggérez une correction. Indication : représentez la mémoire à l'exécution.

# 2 Et maintenant, entraînement grandeur nature ...

Étant donnée la longueur de l'énoncé :
- la partie 2.1 sur les vecteurs est la partie de base à préparer puis à travailler en séance ;
- ceux qui sont à l'aise pourront faire la partie 2.2 sur les matrices, les autres pourront s'en servir dans leur travail personnel et leurs révisions de l'UE ;
- la partie 2.3 reprend largement le contenu du TD 6 sur les polynômes afin d'utiliser à la fois les notions d'allocation dynamique vue en cours et TD, et de création de type vue dans les séances précédentes.

Vous complèterez les fichier `vectors.c`, `matrices.c` et `polynomials.c` fournis avec leurs `.h` respectifs et utiliserez les programmes de test également fournis.

## 2.1 Vecteurs

En mathématiques, un vecteur réel $v$ de dimension $N$ peut s'écrire $v = (v_1, \ldots, v_N)$, où $v_n \in \mathbb{R}$ est le $n$-ième coefficient de $v$ pour tout $n = 1, \ldots, N$. On définit ici les vecteurs mathématiques de dimension $N$ comme des tableaux de `double` de taille $N$ alloués dynamiquement (cf. figure 1).

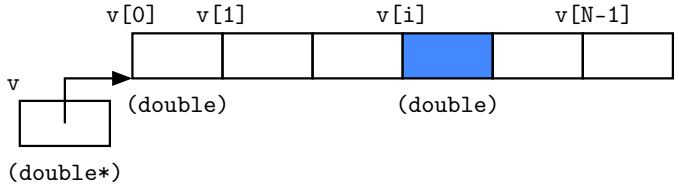



FIGURE 1 – Représentation mémoire d'un vecteur.

**Exercice 5** (Création d'un vecteur par allocation dynamique*)**.** En reprenant la technique d'allocation dynamique pour les tableaux à une dimension, écrivez une fonction de prototype

```
double* create_vector(int N);
```

qui crée et renvoie un vecteur de taille `N` et dont les éléments sont de type `double`.

**Exercice 6** (Suppression d'un vecteur*)**.** Dans le même esprit, écrivez une fonction de prototype

```
void free_vector(double *v);
```

qui libére la mémoire occupée par un vecteur `v`. Pourquoi n'a-t-on pas besoin de passer la taille du vecteur en argument ?

**Exercice 7** (Remplissage d'un vecteur*)**.** Écrivez une fonction de prototype

```
void fill_vector(double *v, int N);
```

qui remplit un vecteur `v` de taille `N` avec les entiers de `1` à `N`.

**Exercice 8** (Affichage d'un vecteur*)**.** Écrivez une fonction de prototype

```
void print_vector(double *v, int N);
```

qui affiche un vecteur `v` de taille `N`.

**Exercice 9** (Produit scalaire*)**.** Écrivez une fonction de prototype

```
double dot_product(double *u, double *v, int N);
```

qui calcule et renvoie le produit scalaire entre un vecteur `u` et un vecteur `v` de même taille `N`.
Pour rappel, $\forall u, v \in \mathbb{R}^N, uv = \sum_{n=1}^{N} u_n v_n$.

**Exercice 10** (Orthogonalité)**.** Écrivez une fonction de prototype

```
int is_orthogonal(double *u, double *v, int N);
```

qui renvoie `1` si le vecteur `u` et le vecteur `v` de même taille `N` sont orthogonaux et `0` sinon.

**Exercice 11** (Norme 2, optionnel)**.** Écrivez une fonction de prototype

```
double norm2(double *u, int N);
```

qui renvoie la norme euclidienne d'un vecteur `u` de taille `N`.
Pour rappel, $\forall u \in \mathbb{R}^N, u = \sqrt{uu} = \sqrt{\sum_{n=1}^{N} u_n^2}$.

**Exercice 12** (Norme 1, optionnel)**.** Écrivez une fonction de prototype

```
double norm1(double *u, int N);
```

qui renvoie la norme 1 d'un vecteur `u` de taille `N`.
Pour rappel, $\forall u \in \mathbb{R}^N, u = \sum_{n=1}^{N} u_n$.

**Exercice 13** (Norme infinie, optionnel)**.** Écrivez une fonction de prototype

```
double norm_infty(double *u, int N);
```

qui renvoie la norme infinie d'un vecteur `u` de taille `N`.
Pour rappel, $\forall u \in \mathbb{R}^N, u = \max_n u_n$.

## 2.2 Matrices

**Exercice 14** (Création d'une matrice par allocations multiples)**.** En reprenant la technique d'allocations dynamiques multiples pour les tableaux à deux dimensions (cf. cours 5 et figure 2), écrivez une fonction de prototype

```
double** create_matrix(int I, int J);
```

qui crée et renvoie une matrice de taille `I`×`J` et dont les éléments sont de type `double`.

**Exercice 15** (Suppression d'une matrice)**.** Dans le même esprit, écrivez une fonction de prototype

```
void free_matrix(double **m, int I);
```

qui libére la mémoire occupée par une matrice `m` de `I` lignes. Pourquoi n'a-t-on pas besoin du nombre de colonnes ?

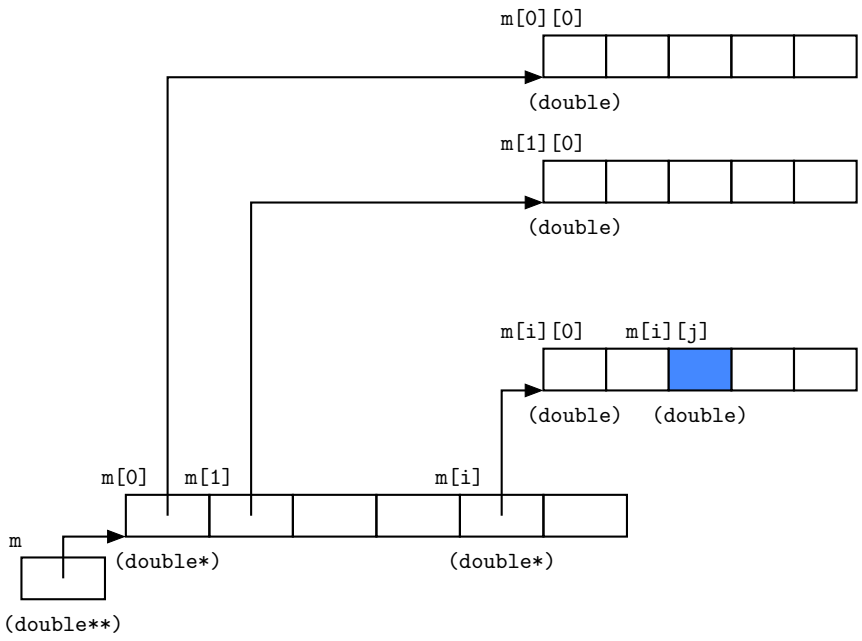



FIGURE 2 – Représentation mémoire d'une matrice construite par allocations multiples.

**Exercice 16** (Remplissage d'une matrice)**.** Écrivez une fonction de prototype

```
void fill_matrix(double **m, int I, int J);
```

qui remplit une matrice `m` de taille `I`×`J` avec les entiers de `1` à `IJ`.

**Exercice 17** (Affichage d'une matrice)**.** Écrivez une fonction de prototype

```
void print_matrix(double **m, int I, int J);
```

qui affiche une matrice `m` de taille `I`×`J`.

**Exercice 18** (Matrice identité)**.** Écrivez une fonction de prototype

```
double ** create_identity(int I);
```

qui crée et renvoie une matrice identité de taille `I`×`I`.

**Exercice 19** (Produit de matrices)**.** Écrivez une fonction de prototype

```
double ** compute_product(double **m1, double **m2, int I, int J, int K);
```

qui calcule et renvoie le produit des matrices `m1` de taille `I`×`J` et `m2` de taille `J`×`K`.
Pour rappel, $\forall P \in \mathbb{R}^{I\times J}, \forall Q \in \mathbb{R}^{J\times K}, [PQ]_{ik} = \sum_{j=1}^{J} P_{ij}Q_{jk}$.

**Exercice 20** (Puissance d'une matrice)**.** Écrivez une fonction de prototype

```
double ** compute_power(double **m, int I, int n);
```

qui calcule et renvoie la puissance `n`-ième d'une matrice carrée `m` de taille `I`×`I`.

**Exercice 21** (Test d'une matrice diagonale)**.** Écrivez une fonction de prototype

```
int is_diagonal(double **m, int I);
```

qui renvoie `1` si la matrice carrée `m` de taille `I`×`I` est diagonale et `0` sinon.

**Exercice 22** (Matrice diagonale)**.** Écrivez une fonction de prototype

```
double **m create_diagonal(double **m, int I);
```

qui crée et renvoie une matrice diagonale dont la diagonale est égale à celle de la matrice carrée `m` de taille `I`×`I` passée en argument.

**Exercice 23** (Matrice symetrique)**.** Écrivez une fonction de prototype

```
int is_symetric(double **m, int I);
```

qui renvoie `1` si la matrice carrée `m` de taille `I`×`I` est symétrique et `0` sinon.

## 2.3 Amélioration du TD

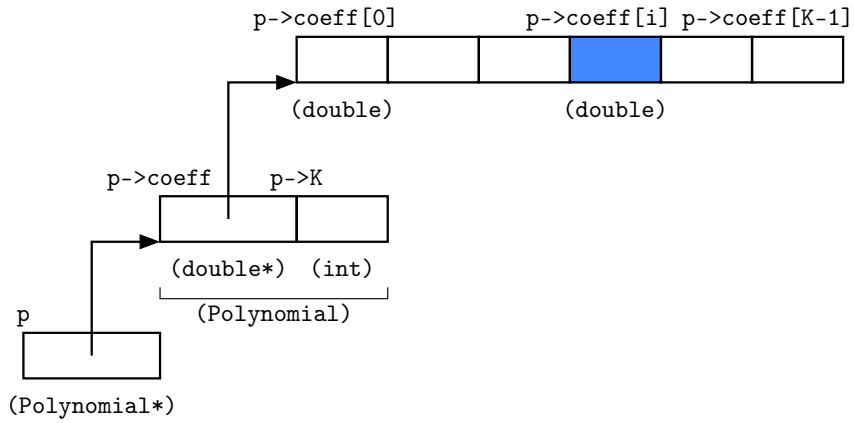



FIGURE 3 – Représentation mémoire d'une variable `p` de type `Polynomial*`.

**Exercice 24** (Amélioration des exercices du TD avec un type polynôme)**.** Nous adoptons une autre approche en introduisant le type `Polynomial` (cf. figure 3 ) :

```
typedef struct {
    double * coeff; /* Coefficients */
    int K; /* Ordre du polynôme */
} Polynomial;
```

Créez les fonctions suivantes, en veillant à une bonne gestion de la mémoire :

- `Polynomial * create_polynomial(int K)` : crée un polynôme d'ordre $K$ initialisé à 0.
- `void free_polynomial(Polynomial * p)` : supprime un polynôme en libérant l'espace mémoire correspondant.
- `void set_coefficient(Polynomial * p, int k, double c)` : donne la valeur `c` au coefficient d'ordre `k`.
- `double eval_polynomial(Polynomial * p, double x)` : calcule la valeur de `p` en `x`.
- `Polynomial * diff_polynomial(Polynomial * p)` : dérive un polynôme.
- `Polynomial * diff_n_polynomial(Polynomial * p, int n)` : dérivée $n$-ième.