Enseignant
Manuel clergue

4

Introduction au langage C
Tableaux & Pointeurs

2022

esiea

1 Tableaux

Pourquoi un tableau ?

Lorsqu'on doit utiliser plusieurs variables ayant des rôles similaires, on peut leur donner un même nom suivi d'un numéro.
Exemple : Entrer une série de notes ⇐ on peut les nommer Note1, Note2, Note3, etc...
C'est peu pratique, surtout que l'algorithme doit pouvoir convenir à un nombre quelconque de notes !

On utilisera alors **une seule** variable, de type tableau, que l'on nommera Note et qui contiendra plusieurs valeurs successives, rangées dans des cases repérées par un indice : Note[i]=val_i

Note	val_1	val_2	...	val_n
indice :	$i = 0$	$i = 1$	...	$i = n - 1$

Taille d'un tableau : définition

Un tableau contient nécessairement des éléments **de même type**.
Le nombre d'éléments (i.e. le nombre de cases) est la taille du tableau.

Cette taille doit être connue au moment de la déclaration d'une variable de type tableau, puisque la déclaration détermine la place mémoire à allouer
$\Rightarrow$ on doit en général surévaluer la taille des tableaux pour qu'ils puissent servir à des jeux de données différents.

Taille d'un tableau : usage

On travaille en général sur un tableau à l'aide de boucles.

Si l'on cherche à accéder à un élément de tableau d'indice supérieur à sa taille déclarée, il y a débordement. Ceci provoque en général une erreur (signalée ou non, selon le compilateur . . .et pas toujours d'erreur à l'exécution) : le fameux `segmentation fault`.

En C, les indices commencent à 0

WARNING :

En C, les indices de tableau commencent nécessairement à 0, ce qui implique que les valeurs autorisées vont de 0 à $nmax - 1$ (donc 99,
pour une variable de type tableau nommée **tablo** et ainsi déclarée) :

```
#define nmax 100    // commentaire : déclaration d'une constante
float tablo[nmax];
```

On peut initialiser un tableau à la déclaration :

```
int feu[4] = {3,2,1,0};
```

N.B. Pour ce tableau, on aura : `feu[0]=3` et `feu[3]=0` mais la taille du tableau vaut bien 4 (cf. sa déclaration).

Dimension d'un tableau

Les tableaux dont on vient de parler sont de dimension 1, c'est-à-dire qu'ils correspondent à des vecteurs. On peut aussi définir des variables correspondant à des matrices en utilisant des tableaux de dimension 2, voire même des tableaux multidimensionnels de dimension d.

La dimension d'un tableau est par définition le nombre d'indices nécessaire à décrire l'un de ses éléments ; par exemple : Matrix[i][j]$=m_{ij}$

Matrix

m_{11}	m_{12}	...	m_{1j}	...	m_{1n}
m_{21}	m_{22}	...	m_{2j}	...	m_{2n}
...	...	...	...	...	...
m_{i1}	m_{i2}	...	m_{ij}	...	m_{in}
...	...	...	...	...	...
m_{p1}	m_{p2}	...	m_{pj}	...	m_{pn}

où i est l'indice de ligne et j l'indice de colonne.

Pour déclarer un tableau de dimension 2, on écrit (pour la variable **Matrix** de l'exemple ci-dessus) :

int **Matrix**[p][n]

où p et n sont des entiers positifs, notés en chiffres ou définis par des constantes.

Chaque élément Matrix[i][j], pour un $i < p$ et un $j < n$ donnés, est alors une variable de type Entier et s'utilise comme telle.

N.B. On peut initialiser à la déclaration la liste des valeurs d'un tableau :

Entier **Matrix**[2][3]={ {1,2,3} , {4,5,6} }

pour créer le tableau bidimensionnel **Matrix**

1	2	3
4	5	6

dont on pourra ultérieurement modifier la valeur d'un ou plusieurs éléments.

Tableaux de caractères

Un tableau de caractères peut être initialisé par :

- une liste de caractères (comme un tableau de nombres) :
  `char tab[n]={'e','x','e','m','p','l','e'};`
- une chaîne de caractères littérale :
  `char tab[n]="exemple";`

Le compilateur C complète toute chaîne de caractères avec un caractère nul '\0'. Il faut donc que le tableau ait au moins un élément de plus que
le nombre de caractères de la chaîne littérale.
⇒ il faut **n=8** même si le mot "exemple" ne comporte que 7 caractères.

Tableaux et affectation

En C, on ne peut pas affecter directement un tableau à une autre variable de type tableau (même si la dimension et la taille sont identiques) :

~~tablo2 = tablo1~~

Il faut passer par une boucle for :

```
#define nmax 5

int tablo1[nmax]={12,-7,5,14,23};
int tablo2[nmax];
main()
{
for (int i=0; i<nmax; i++)
tablo2[i]=tablo1[i];
for (int i=0; i<nmax; i++)    // plus clair que faire tout dans une seule boucle
printf("tablo1[%d]=%d\t tablo2[%d]=%d\n",i,tablo1[i],i,tablo2[i]);
}
```

Exemples

```c
#define N 256
// Déclarations et initialisations
int tab[N] = {0};
char mot[10] = "Bonjour";
int a;

// Utilisation d'un tableau statique dans une fonction
int f(int tab[]); // Prototype de la fonction
f(tab);           // Appel de la fonction

// Accès et modification
a = tab[10];
mot[5] = 'c';

// Parcours avec une boucle
for(int i=0;i<N;i++) T[i] = i + 1;
```

```c
#define WIDTH 2 // Définition de la largeur
#define HEIGHT 3 // Définition du nombre de lignes
int tabI[10][20];
char tabC[HEIGHT][WIDTH] = {{11,12},{21,22},{31,32}};
int i, j;

int a=3; b=2;
float tabF[a][b];

for (i=0; i<HEIGHT; i++)
        for (j=0; j<WIDTH; j++)
                tabC[i][j] = 2*i+j;

int fct1D (int tab[]){ . . . } // Prototype de fonction
int fct2D (int tab[][20]){ . . . } // Prototype de fonction
a = fct2D (tabC); // Appel de fonction
```

2 Les pointeurs

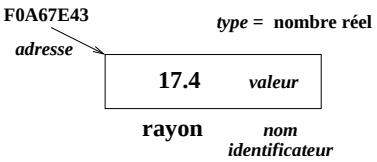
2 Les pointeurs

2.1 La notion de pointeur

Variable : rappel

Les **variables** servent à représenter les données, les résultats, etc...

Une **variable** correspond à :

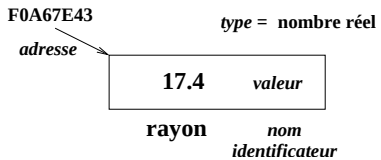


- (pour l'homme) un objet, une quantité
  - qui reçoit un **nom** ou **identificateur**,
  - qui est d'un certain **type** (selon la nature de l'information),
  - qui peut prendre des **valeurs** ;
- (pour la machine) un emplacement mémoire
  - caractérisé par une **adresse** (lieu où la variable est stockée)
  - et une taille (place utilisée) qui dépend du type de la variable.

Nom ou pointeur

La plupart du temps, on manipule les variables en utilisant leur **nom**.
Toutefois, comme on l'a constaté pour les passages de paramètres à des fonctions ou procédures, par exemple, il peut être utile de manipuler **directement** les **pointeurs**.

Si l'on reprend l'exemple, on pourra considérer :



- la variable réelle **rayon** dont la valeur est **17.4**
- un pointeur **ptr** sur un réel, qui pointera sur rayon, et dont la valeur sera **F0A67E43** (cette valeur est un **entier**, écrit ici en hexadécimal)

2 Les pointeurs

2.2 Pointeurs et affectations

Déclaration et affectation pour un pointeur

Il n'est cependant pas nécessaire (voire même impossible...) de connaître explicitement l'adresse à laquelle sera allouée la variable rayon.
On peut néanmoins déclarer une variable de nom **ptr**, avec, comme type, un "pointeur_sur_un_réel", et lui affecter la valeur de l'adresse de rayon.

```
float rayon = 17.4;
float *ptr;      // déclaration d'un pointeur_sur_un_réel
ptr = &rayon;    // affectation de l'adresse de rayon au pointeur
```

Précisions sur les adresses

DONC : un pointeur est une **adresse**.

Dans le codage informatique, une adresse est celle d'un octet.
Or la variable réelle rayon est codée sur plusieurs octets

$\Rightarrow$

la valeur du pointeur **ptr** sera l'adresse du premier octet de l'espace mémoire occupé par la variable rayon.

Cette valeur est un nombre entier ; toutefois **WARNING**
un pointeur n'est PAS de type `int`.

(voir suite de cette section)

L'opérateur d'indirection

Si l'on accède à une variable par un pointeur sur son adresse, on peut retrouver la **valeur** de la variable, en utilisant l'opérateur d'indirection *****

```
float rayon = 17.4;
float *ptr;      // déclaration d'un pointeur_sur_un_réel
ptr = &rayon;    // affectation de l'adresse de rayon au pointeur

printf("valeur pointée par p = %f",*ptr);    // valeur pointée
```

L'instruction `printf()` affichera **17.4** qui est la valeur pointée par **ptr**.

2 Les pointeurs

2.3 Allocation dynamique

Réservation d'un espace mémoire

Lorsqu'un pointeur est déclaré, sa valeur par défaut est (quel que soit le type pointé), une constante symbolique notée **NULL**. On peut initialiser le pointeur en lui affectant l'adresse d'une variable (cf. ci-dessus).

On peut aussi affecter directement une valeur à ***ptr**, mais il faut d'abord réserver un espace mémoire de taille adéquate. L'adresse du premier octet de cet espace sera la valeur de **ptr**.

Cette réservation d'espace mémoire s'appelle l'**allocation dynamique**.

La fonction malloc

En langage C, l'allocation dynamique se fait avec des fonctions de la bibliothèque `stdlib.h` (à inclure, donc). La plus courante est :

`malloc(nombre_d'octets)`

Cette fonction retourne un **pointeur générique de type** `void *` pointant vers un objet de taille nombre_d'octets mais de type non identifié a priori.

Pour spécifier le type, il faut faire un `cast`, i.e. une conversion explicite.

Exemple :

```
int *p = NULL;
p = malloc(sizeof(int));
*p = 27;
```

ou, mieux :

```
p = (int*)malloc(sizeof(int));   // type spécifié par un cast
*p = 27;
```

Les fonctions : `calloc` et `realloc`

La fonction `calloc` est une variante de `malloc` et s'appelle avec deux arguments (à la différence de `malloc`) :
`calloc(nombre_d'objets , taille_des_objets)`
outre la réservation d'un espace mémoire, elle initialise chaque objet à zéro.

La fonction `realloc` permet de ré-allouer au même pointeur (voire même à un autre) un espace mémoire de taille différente :
`realloc(nom_du_pointeur , nombre_d'octets)`
WARNING : si la nouvelle taille requise est supérieure à la taille de l'espace déjà réservé,

il se peut que **la valeur du pointeur change**.

Exemples et précisions

Supposons que l'on veuille réserver un espace mémoire pour 12 entiers :

```
ptr = (*int) malloc(12*sizeof(int));
```

Si l'on avait voulu faire la même chose, mais en s'assurant que l'espace réservé ait été nettoyé (le "c" vaut pour clear), i.e. que les entiers sont initialisés à zéro :

```
ptr = (*int) calloc(12, sizeof(int));
```

Et si l'on veut, ensuite, réduire à 8 entiers la taille de l'espace mémoire réservé :

```
ptr = (*int) realloc(ptr, 8*sizeof(int));
```

Pour toutes ces fonctions, en cas d'échec (ex. pas assez de place mémoire), la valeur **NULL** est renvoyée au pointeur demandant l'allocation.

Indispensable `free` pour libérer l'espace

Après utilisation, il est **indispensable** de libérer l'espace mémoire que l'on avait alloué, en faisant appel à la fonction `free` :

```
free(nom_du_pointeur)
```

WARNING : si une erreur se produit lors d'un appel à realloc alors on perd le lien vers l'espace mémoire initialement réservé ⇒ on ne pourra plus le libérer ! Solution : utiliser un pointeur auxiliaire lors de la ré-allocation.

Pour plus d'informations sur ces fonctions, on peut consulter, par exemple, le site web :

https ://en.wikipedia.org/wiki/C_dynamic_memory_allocation
ou OpenClassrooms

2 Les pointeurs

2.4 Pointeurs et opérations

Incrémentation et décrémentation

Bien qu'un pointeur ne soit pas de type `int`, son type est **discret**, i.e. il existe un successeur et un prédécesseur à toute valeur d'un pointeur.

WARNING

Le successeur n'est pas la valeur entière +1, mais :
la valeur entière + le nombre d'octets sur lequel est codé le type pointé.

Exemples (avec des variantes possibles, selon la taille des codages) :
`ptr++`
vaut l'ancienne valeur de **ptr** + 4 s'il s'agit d'un pointeur_sur_un_entier. `ptr-`
vaut l'ancienne valeur de **ptr** - 8 s'il s'agit d'un pointeur_sur_un_réel.

Additions et soustractions

On peut aussi ajouter et soustraire des pointeurs entre eux (utilisation délicate...), sous réserve qu'ils pointent sur des objets de même type, ou des pointeurs avec des constantes.

La même règle que ci-dessus est respectée : la constante **1** représente une unité de codage du type pointé (exemple : 4 pour un pointeur_sur_un_entier).

`ptr + i`
prend pour valeur entière celle de **ptr** à laquelle s'ajoute **i * sizeof(type1)** si **ptr** est un pointeur_sur_type1.
`p - q`
prend pour valeur entière celle de **(p - q) * sizeof(type2)** si **p** et **q** sont tous deux de type pointeur_sur_type2.

Comparaisons

Les opérateurs relationnels et logiques sont aussi applicables à des pointeurs, afin de faire des comparaisons et d'évaluer des tests, mais toujours avec les mêmes restrictions :

les pointeurs intervenant dans l'expression logique doivent toujours
pointer sur des objets de même type.

2 Les pointeurs

2.5 Pointeurs et types composés

Un tableau 1D correspond en fait à un pointeur vers son premier élément :
tab équivaut à &tab[0], l'adresse du premier élément.
Ainsi, on peut parcourir les éléments d'un tableau en passant d'un pointeur au suivant : l'incrémentation **ptr++** revient à se déplacer vers l'espace mémoire situé "taille_des_éléments" plus loin.

```
#define N 5
int tab[N] = {1, 2, 6, 0, 7};
void main()
{
int *ptr;
printf("\n lecture de tab, selon l'ordre croissant des indices:\n");
for (ptr = &tab[0]; ptr <= &tab[N-1]; ptr++)
printf(" %d \n",*ptr);
}
```

WARNING : le test NE peut PAS être : `ptr < &tab[N]` car tab[N] n'existe pas !

Quelques différences importantes

Toutefois, un tableau **n'est PAS** un pointeur

i.e. les manipulations autorisées sur les pointeurs ou sur les tableaux sont différentes :

- un pointeur doit toujours être initialisé, soit par une allocation dynamique, soit par affectation d'une expression adresse,
  par exemple : `p = &i;` ou `p = tab;`
- un tableau ne peut pas figurer à gauche d'un opérateur d'affectation. En particulier, un tableau ne supporte pas l'arithmétique (on ne peut **pas écrire** `tab++;`).

3 Du bon usage des pointeurs

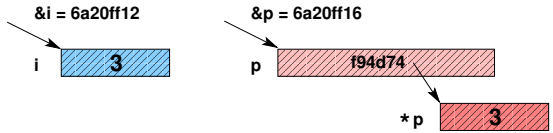
3 Du bon usage des pointeurs

3.1 Les instructions de base

L'instruction *p = i

CAS 1 : quels sont les effets de l'instruction ***p=i** ?

```
int i = 3;
int *p;
p = (int*)malloc(sizeof(int));
*p = i;
```



adresse de i = 0x6a20ff12 valeur de i = 3

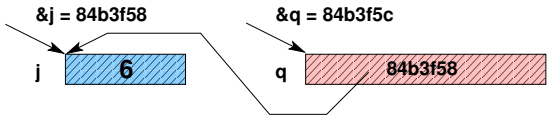
avec p seulement déclaré, adresse de p = 0x6a20ff16 valeur de p = NULL
après allocation dynamique, valeur de p = 0xf94d74 [N.B. zone mémoire différente]
valeur de *p, AVANT affectation = 0 . . .voire même. . .n'importe quoi !
valeur de *p, APRES affectation = 3

*p a la **même valeur** que i MAIS la valeur de p n'est PAS l'adresse de i
⇐ La valeur 3 est stockée à DEUX endroits différents de la mémoire (**recopie ; clonage**) : l'endroit pointé par &i et aussi celui pointé par p.

L'instruction q = &j

CAS 2 : quels sont les effets de l'instruction **q=&i** ?

```
int j = 6;
int *q;
q = &j;
```



adresse de j = 0x84b3f58 valeur de j = 6

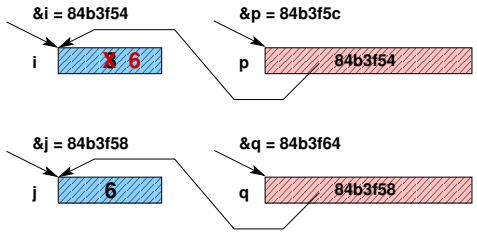
avec q seulement déclaré, adresse de q = 0x84b3f5c valeur de q = NULL
APRES affectation : valeur de q = 0x84b3f58 valeur de *q = 6

Cette fois-ci, on constate que : la valeur de q EST l'adresse de j.
Il en résulte (nécessairement !) que la valeur pointée *q EST la valeur de j.
En effet : ce 6 n'est stocké qu'à UN SEUL endroit dans la mémoire, endroit vers lequel pointent à la fois &j et q.
j et *q sont **identiques**, pas des clones (contrairement au cas 1).

L'affectation *p = *q

CAS 3 : quels sont les effets de l'affectation ***p = *q** ?

```
int i = 3, j = 6;
int *p, *q;
p = &i;
q = &j;
*p = *q; // affectation
```



AVANT l'affectation de pointeurs, valeurs pointées : *p = 3 et *q = 6

APRES l'affectation de pointeurs, valeurs pointées : *p = 6 et *q = 6

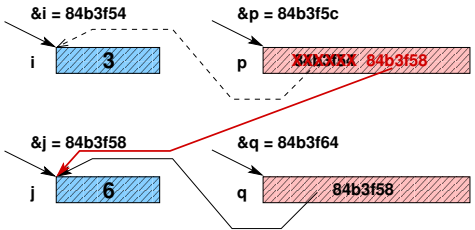
Mais comme p pointait sur i, alors *p n'était autre que la valeur de i ...

⇒ **la valeur de i** a AUSSI été **modifiée** !

L'affectation p = q

CAS 4 : quels sont les effets de l'affectation **p = q** ?

```
int i = 3, j = 6;
int *p, *q;
p = &i;
q = &j;
p = q; // affectation
```



AVANT l'affectation de pointeurs, valeurs pointées : $*p = 3$ et $*q = 6$

APRES l'affectation de pointeurs, valeurs pointées : $*p = 6$ et $*q = 6$

Au niveau des valeurs pointées, mêmes effets que dans le cas 3 ; pourtant :

1. ici, la valeur de i n'a PAS été modifiée ;
2. en revanche la **valeur de p** a changé : elle est devenue l'adresse de j (comme q) et non plus celle de i.

3 Du bon usage des pointeurs

3.2 Les tableaux dynamiques

Inconvénients des tableaux statiques

Les tableaux étudiés précédemment étaient **statiques** : ils devaient être surdimensionnés pour que la réservation de place mémoire se fasse correctement lors de la déclaration de la variable de type tableau.

Outre le gâchis de place mémoire, les tableaux statiques ont (au moins) trois inconvénients :

- on ne peut pas créer de tableaux dont la taille est une variable du programme, (**SI**)
- on ne peut pas créer de tableaux bidimensionnels dont les lignes n'auraient pas toutes le même nombre d'éléments.
- la taille des tableaux est limitée par la taille de la pile

Les tableaux dynamiques

L'usage de pointeurs et l'allocation dynamique va permettre de résoudre ces problèmes en créant des **tableaux dynamiques**.

```
#include <stdlib.h>
void main()
{
int n;
int *tabdyn;
...// instructions diverses, dont lecture ou affectation de n

  tabdyn = (int*)malloc(n * sizeof(int));   // allocation dynamique d'un
// espace mémoire pour tableau de n éléments exactement
...// instructions concernant le tableau
free(tabdyn);   // libération de la place mémoire allouée
}
```

Tableaux multidimensionnels dynamiques

Un tableau 2D statique est un tableau de tableaux, déclaré sous la forme :

```
int tab[p][n]
```

On peut le voir comme un pointeur vers un pointeur, et donc :

- `tab` équivaut à `&tab[0][0]`, l'adresse du premier élément
- pour tout i de 0 à $p-1$, `tab[i]` équivaut à `&tab[i][0]`

Pour rendre dynamiques ces tableaux, on utilise des **pointeurs de pointeurs**

Tableaux multidimensionnels dynamiques

```
1  void main(){
2  int i, p, n;
3  int **tab; //  double indirection}
4  scanf("\%d \%d",&p,&n) ;
5  tab = (int**)malloc}(p * sizeof(int*)); // tableau 2D~: p lignes de  tableaux
6  for (i = 0; i < p; i++)
7    tab[i] = (int*)malloc}(n * sizeof(int)); // n cases dans chaque tableau 1D
8  //  ...
9  for (i = 0; i < p; i++)
10   free(tab[i]); // lib\'eration de chaque tableau 1D, i.e. de chaque ligne
11 free(tab); // lib\'eration de la 1\`ere  colonne, donc de l'ensemble du tableau
12 }
```

3 Du bon usage des pointeurs

3.3 Les chaînes de caractères

Chaîne ou tableau de caractères

Une chaîne de caractères est un tableau 1D de caratères dont la taille est éminemment variable

$\Rightarrow$ l'utilisation de pointeurs et d'allocation dynamique est recommandée.

Chaîne ou tableau de caractères

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main(){
int i;
char *chaine1, *chaine2, *res, *p;
chaine1 = "chaine ";
chaine2 = "de caracteres";
res = (char*)malloc((strlen(chaine1) + strlen(chaine2)) * sizeof(char));
p = res;
for (i = 0; i < strlen(chaine1); i++, p++)
  *p = chaine1[i];
for (i = 0; i < strlen(chaine2); i++, p++)
  *p = chaine2[i];
printf("%s\n",res);
}
```

Tableau de pointeurs

Très utilisés pour manipuler des chaînes de caractères, les tableaux de pointeurs permettent de stocker des chaînes de dimensions différentes.

Exemple : on veut créer un tableau des noms des jours de la semaine `char *tabJour[] = {"Lundi","Mardi","Mercredi","Jeudi","Vendredi","Samedi","Dimanche"} ;`

Chaque élément `tabJour[K]` du tableau est de type pointeur sur caractère.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define NKar 100
#define NJ 7
char ch[NKar], *tabJour[NJ] ;
int main(){
 for (i = 0 ; i < NJ ; i++){
  fgets(ch,NKar,stdin); // saisie de cha\^ine s\'ecuris\'ee
  printf("\%s\n", ch);
  k =strlen(ch);
  printf("\%d\n", k);
  tabJour[i] = (char*)malloc((k)*sizeof(char));
  strncpy(tabJour[i], ch, k);
  printf("tabJour[\%d] = \%s$\backslash$n", i, tabJour[i]);
 }
}
```