


 École et observatoire  
des **sciences de la Terre**  
de l'Université de Strasbourg  
et du 

 Laboratoire  
Institut de **physique du globe**  
de Strasbourg | IPGS | UMR 7516  
de l'Université de Strasbourg  
et du 

---

# Support de cours – 1A – EOST

## Informatique I : Langage C

---

Christophe ZAROLI

December 1, 2020



# CONTENTS

Contents	1
1 Ordinateur, programme et langage	3
2 Les variables et l'instruction d'affectation	7
3 Les instructions d'affichage et de lecture	13
4 L'instruction if	23
5 Les structures de répétition	27
6 Quelques techniques usuelles	31
7 Les tableaux	37
8 Les fonctions	49
9 Les pointeurs	61
10 Les chaînes de caractères	69
11 Les structures	81
12 Allocation de mémoire dynamique	89





# CHAPTER 1

## ORDINATEUR, PROGRAMME ET LANGAGE

Un ordinateur comporte :

- \* la mémoire centrale : elle sert à mémoriser le programme en langage machine pendant le temps nécessaire à son exécution, ainsi que des informations temporaires qu'il manipule ; elle est composée d'une suite d'octets (groupes de 8 bits), repérés chacun par une adresse. (mémoire vive, RAM: random access memory)
- \* l'unité centrale : elle exécute une à une les différentes instructions du programme.
- \* les périphériques :
  - de communication : ils assurent l'échange d'informations entre l'homme et la machine (clavier, écran...)
  - d'archivage : ils permettent de conserver de façon permanente de grande quantité d'informations.

Pour des raisons purement technologiques, toutes les informations manipulées par un ordinateur sont codées en "binaire" (0, 1).

Binary digIT : BIT  $\begin{matrix} \rightarrow 0 \\ \rightarrow 1 \end{matrix}$

Il est possible de s'affranchir du langage machine en faisant appel à un langage de programmation dit "évolué" (t.q. le langage C), à condition de disposer d'un programme approprié de traduction qu'on nomme un "compilateur".

Les différents langages évolués reposent sur des concepts communs :

- \* la variable : il s'agit d'un nom qu'on donne à un emplacement de la mémoire destiné à contenir une information.
- \* les instructions de base : affectation, lecture et écriture.
- \* les structures de contrôle : elles permettent de programmer les choix et les répétitions.
- \* les structures de données, en particulier les tableaux.

Dans un langage évolué on distingue les instructions exécutables des instructions de déclaration.

Rappels:

- 1 octet = 8 bits → on peut représenter  $2^8$  valeurs = 256  
↳ [0, 255]
- 1 Ko =  $2^{10}$  octets = 1024 octets
- 1 Mo =  $2^{10}$  Ko =  $2^{20}$  octets
- 1 Go =  $2^{10}$  Mo = ...
- 1 To =  $2^{10}$  Go = ... (mémoire disque)
- 1 Po =  $2^{10}$  To (1 pentaoctet)



• coder les images de couleurs : (papier tricolore...)

$3 \text{ octets} = 1 \text{ pixel}$

rouge	vert	bleu
[0, 255]	[0, 255]	[0, 255]
1 octet	1 octet	1 octet

feuille A4

29 cm

21 cm

• couleur → 3 octets / pixel

600 dpi (pixel/pouce) (1 pouce = 2.54 cm)

\* combien d'octets occupe cette image en mémoire ?

$$\frac{21 \times 600}{2.54} \times \frac{29 \times 600}{2.54} \times 3 \approx 100 \text{ millions octets (100 Mo) ...}$$

Applications

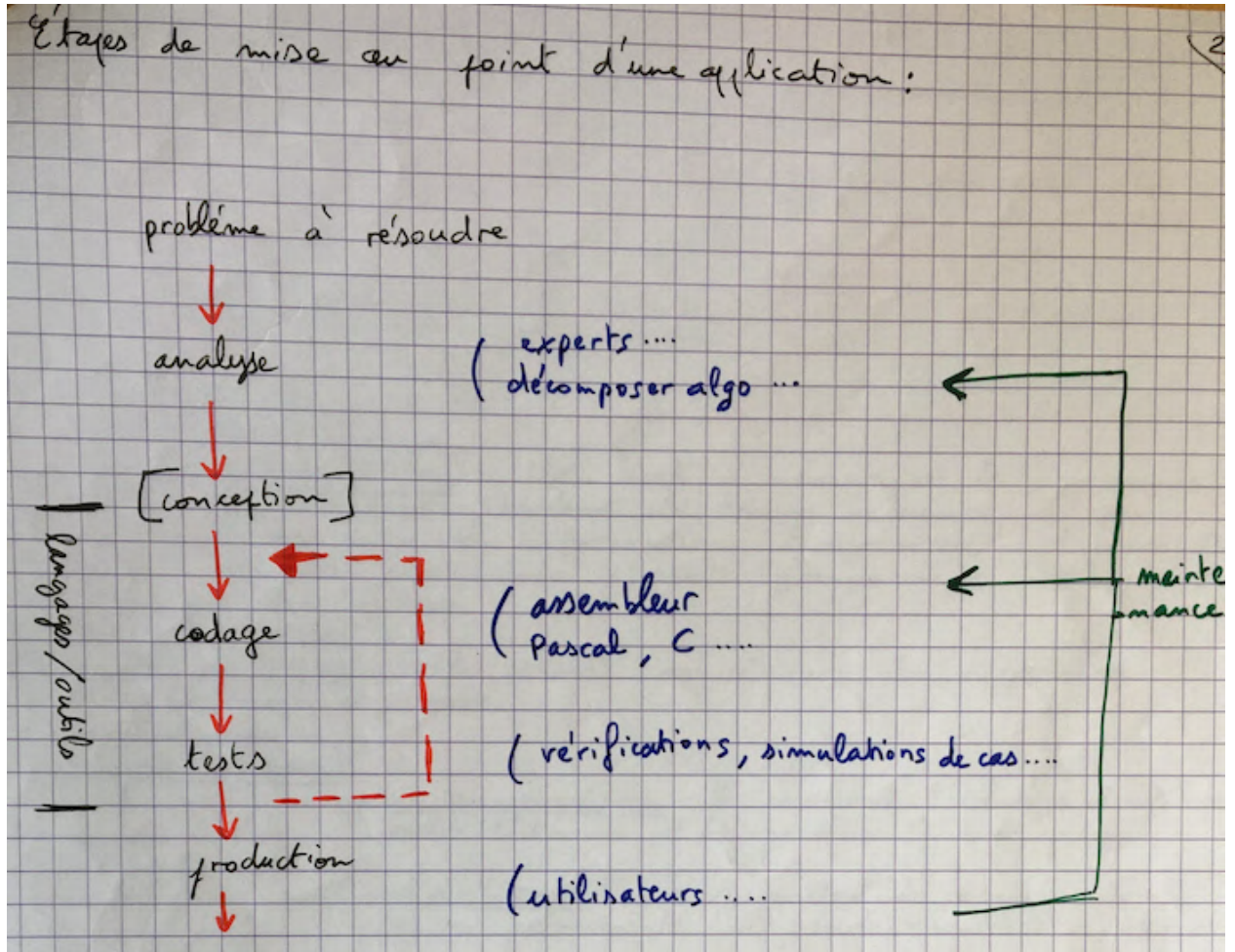
↔

utilisateur

matériel

↕

système d'exploitation



---

## CHAPTER 2

# LES VARIABLES ET L'INSTRUCTION D'AFFECTATION

### ★ Les noms de variables. (en C)

- les caractères utilisés doivent être obligatoirement choisis parmi les 26 lettres majuscules ou minuscules de l'alphabet, les chiffres de 0 à 9 et le caractère "sous-tilde" (`\_`). Les caractères accentués ne sont pas admis, pas plus que le "`".
- le 1<sup>er</sup> caractère doit obligatoirement être différent d'un chiffre.
- un nom ne doit pas comporter plus de 32 caractères.
- aucun espace ne peut figurer dans un nom
- les majuscules sont distinguées des minuscules



★ Type d'une variable et déclaration correspondante

Toute information doit être codée en binaire. Les variables servent à conserver différentes sortes d'informations : nombres entiers, nombres réels, caractères... Chaque sorte d'information devra disposer d'un codage approprié et on parlera de type d'une variable.

⚠ A un type donné correspond toujours un nombre d'octets (donc de bits) donné et, donc, un nombre limité de valeurs différentes.

Exemples:

{	int	: nombres entiers
	float	: approximation de nombres réels
	char	: caractères (lettres, chiffres, signes...)

Dans un programme, on choisira le type des variables à l'aide d'instructions de déclaration :

Exemples:

	int	n, p;
	float	valeur, res, x1, x2;
	char	reponse;

rg: la plupart des instructions en C sont terminées par ";"  
 • on n'est pas obligé de placer 1 instruction par ligne  
 • les instructions de déclaration doivent précéder les instructions exécutables

★ L'instruction d'affectation

rg:  $a = a + 1;$

↑ gauche, ↑ droite, "a" représente le contenu c'est le contenant

• exemple:

		a	b
(1)	$a = 1;$	1	-
(2)	$b = a + 3;$	1	4
(3)	$a = 3;$	3	4

exemple:

	$n = 10;$
	$p = 2 * n - 3;$

Il faut tenir compte du "type" des informations manipulées. En effet, il est très important de savoir que les différents opérateurs du langage C ne sont a priori définis que lorsqu'ils portent sur deux valeurs d'un même type : on sait additionner deux entiers ou deux flottants (flottants), mais pas un entier et un flottant.

Rq: nous verrons par la suite comment il est possible en C :  
 → d'écrire des "expressions mixtes" (variables et constantes... de type ≠)  
 → d'introduire des "conversions" de type en affectant à une variable d'un certain type une valeur d'un autre type.



★ Le type entier : int

→ permet de représenter des nombres entiers relatifs suivant un nombre d'octets (souvent 2 ou 4) qui dépend de l'ordinateur et du compilateur employés.

→ opérateurs usuels dont l'on dispose en C :

}	+	
	-	
	*	
	/	→ $\textcircled{\frac{\nabla}{0}}$ division entière : $11/4$ donne 2

$2.75000$

$11 = 2 \times 4 + 3$

→ opérateur "modulo" : qui correspond au reste de la division entière (ou euclidienne) →  $11 \% 4$  donne 3

→ règles de priorité :

$a + b * c$	.....	$a + (b * c)$
$a * b + c \% d$	.....	$(a * b) + (c \% d)$
$-c \% d$	.....	$(-c) \% d$
$-a + c \% d$	.....	$(-a) + (c \% d)$
$-a / -b + c$	.....	$(-a) / (-b) + c$
$-a / -(b+c)$	.....	$(-a) / (-(b+c))$

## ★ Le type réel : float

- permet de représenter, de manière approchée, des nombres réels.
- il y a des limites aux valeurs que l'on peut ainsi représenter (leur valeur absolue ne doit être ni trop grande, ni trop petite).

• Notation des constantes: → 12.43    -0.38    -.38    4.    .27

(manchette, exposant...)

$$\rightarrow \left\{ \begin{array}{l} 4.25E4 \Leftrightarrow 4.25e+4 \Leftrightarrow 42.5E3 \\ 54.27E-32 \Leftrightarrow 542.7E-33 \Leftrightarrow 5427e-34 \\ 48e13 \Leftrightarrow 48.e13 \Leftrightarrow 48.0E13 \end{array} \right.$$

- ~~⊗~~: → l'opérateur "<sup>modulo</sup>%" n'existe pas pour le type float
- l'opérateur "/" appliqué à 2 valeurs de type float fournit un résultat de type float
  - ↳  $5./2.$  donne une valeur de type float égale à 2.5 (environ, car les réels sont représentés de manière approchée!)
  - ↳  $5/2$  donne une valeur de type int égale à 2

## ★ Expressions "mixtes"

```
int m, p;
float x;
```

$m + x$  : pour calculer la valeur de cette expression, le compilateur commencera par prévoir une conversion de la valeur de  $m$  dans le type float avant de l'ajouter à  $x$ . Le résultat final sera de type float

$m * p + x$  :  $m * p$  est un "int", que le compilateur convertit en float pour pouvoir être ajouté à la valeur de  $x$ . Le résultat final est de type float.



★ Quand l'affectation impose une conversion de type

```
int n; float x; x = 3.4;
n = x + 5.3;
```

8.7  
 ↘ partie entière (car n est int)  
 8  
 donc n vaut 8 à la fin

• Dès que l'expression est d'un type ≠ de celui de la variable réceptrice, il y a une conversion.  
 Celle-ci peut être non dégradante (ex: int → float)  
 ou dégradante (ex: float → int)

★ Le type caractère: char

→ 'e' 'a' 'y' '+' '\$' 'e' 's' ':' '5'

→ un certain nombre de ces caractères sont "non imprimables". Ils possèdent une représentation conventionnelle utilisant le caractère "\" (antislash):

'\n'	saut de ligne
'\b'	retour arrière (tabulation)
'\a'	cloche, ou bip (ce caractère ne peut pas être lu, mais, si on cherche à l'afficher à l'écran, on obtient un ... bip)
'\\'	\
'\''	'
'\"'	"

exemple:

```
char c1, c2;
c1 = 's'; /* affecte à c1 le caractère s */
c2 = c1; /* affecte à c2 le contenu de c1 */
```

### ★ Initialisation de variables

```
int n = 5;  
float x = 5.25;
```

→ pour se prémunir du risque de rencontrer des variables non définies, mais pas super si l'on doit lire la valeur de x plus tard dans le programme. (risque de confusion pour le lecteur).

### ★ Résumé

Une variable est caractérisée par un nom et un type.  
Le nom sert à repérer un emplacement mémoire.  
Le type précise la taille de cet emplacement, la manière dont l'information y sera codée, les opérations autorisées ainsi que les valeurs représentables.



## CHAPTER 3

### LES INSTRUCTIONS D’AFFICHAGE ET DE LECTURE

★ L'instruction d'affichage: printf

<pre>int m = 20; printf("%d", m);</pre> <p>20</p> <p>(19): tous les codes de format commencent par %</p>	<pre>int m = 10, p = 25; printf("nombre: %d valeur: %d", m, p)</pre> <p>nombre: 10 valeur: 25</p>
<pre>printf("total: %d", m);</pre> <p>total: 20</p>	<pre>printf("la somme de %d et de %d est %d", m, p, m+p);</pre> <p>la somme de 10 et de 25 est 35</p> <p>(19) affichage de la valeur d'une expression</p>

★ autres codes de format

→ pour le type char : %c

```
int n = 15;
char c = 'S';
printf("nombre : %d type : %c",
      n, c);
nombre : 15 type : S
```

→ pour le type float : %e et %f

↑ notation exponentielle (mantisse et exposant)  
↑ notation décimale

```
float x = 1.23456e4;
printf("x notation expo : %e , x notation déci : %f", x, x);
```

x notation expo : 1.23456e+04 , x notation déci : 12345.599609

**Rq** La notation exponentielle utilise toujours une mantisse entre 1 et 10 avec 6 chiffres après le point décimal et 2 chiffres pour l'exposant.  
La notation décimale affiche toujours 6 chiffres après le point décimal. Elle ne convient donc que pour des nombres ni trop grands, ni trop petits.  
→ (On voit d'ailleurs sur l'ex. ci-dessus l'effet de la "précision machine" pour représenter les nombres).



★ Gabarit d'affichage

```
printf("%3d", m);
```

$m = 20$	→	20
$m = 3$	→	3
$m = -5$	→	-5
$m = 2358$	→	2358
$m = -5200$	→	-5200

Les flottants sont affichés avec 6 chiffres après le point décimal.

Un nombre placé après % dans le code de format précise un gabarit d'affichage, c-à-d un nombre minimal de caractères à utiliser.

---

```
printf("%10f", x);
```

pour matérialiser un espace

$x = 1.2345$	→	1.234500
$x = 12.345$	→	12.345000
$x = 1.2345E5$	→	123450.000000

6 chiffres obligatoirement après.

★ Précision de l'affichage

N.B. le gabarit indique un minimum qui peut être dépassé en cas de besoin.

```
printf("%10.3f", x);
```

$x = 1.2345$	→	1.235	10 cases
$x = 1.2345E3$	→	1234.500	3 chiffres après le .
$x = 1.2345E7$	→	12345000.000	3 chiffres

On peut spécifier le nombre de chiffres souhaités en le faisant figurer, précédé d'un point, avant le code de format et après le gabarit.

`printf("%12.4e", x);`

$x = 1.2345 \rightarrow$  `1.2345e+00`

4 chiffres après le .  
2 chiffres pour exposant.  
12 cases

$x = 123.456789E8 \rightarrow$  `1.2346e+10`

12 cases  
↑ toujours manqué entre 1 et 10  
↑ arrondi  
2 chiffres pour exposant

★ Changer de ligne

`int m = 15; char c = 's'; printf("nombre: %d", m); printf("type: %c", c);`

nombre: 15type: s

`printf("nombre: %d\n type: %c", m, c);`

nombre: 15  
type: s



### ★ Un premier programme complet

→ Tout programme C doit être construit suivant le "canevas" suivant:

```
main()  
{  
  instructions de déclaration  
  instructions exécutables  
}
```

### ★ Commentaires

**/\*** ne pas oublier de commenter les programmes **\*/**

Exemple :

```

main()
{
    float valeur = 12.85;
    float carre, cube; int ent;

    carre = valeur * valeur;
    cube = carre * valeur;

    printf("la valeur %f a pour carré %f et pour cube %f \n
           valeur, carre, cube);

    ent = cube;

} printf("la partie entière de son cube est: %d", ent);

```

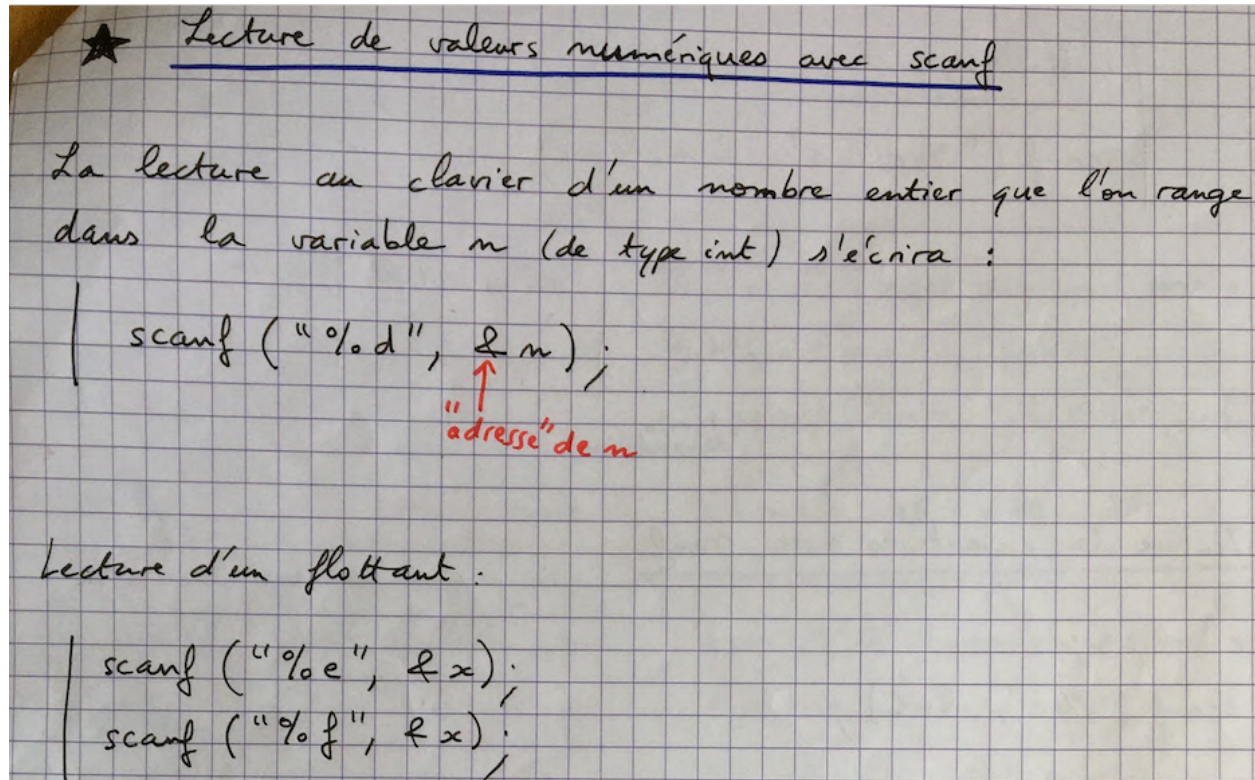
la valeur 12.850000 a pour carré 165.122516  
6 chiffres après.

et pour cube 2121.824463

↘ on va à la ligne \*/

la partie entière de son cube est 2121





Exemple :

```

main()
{
    float valeur, carre, cube; int ent;
    printf("donnez un nombre réel : ");
    scanf("%e", &valeur);
    carre = valeur * valeur; cube = carre * valeur;
    printf("la valeur %e a pour carre' %f et pour cube %f \n",
           valeur, carre, cube);
    ent = cube;
    printf("la partie entière de son cube est: %d", ent);
}
    
```

donnez un nombre réel : 12.85 <sup>à rentrer au clavier</sup>  
 la valeur 12.850000 a pour carre' 165.122500 et pour cube 2121.824463  
 la partie entière de son cube est: 2121

ex :

```

int m, p;
scanf("%d%d", &m, &p);
    
```

Au clavier, on peut taper : 15 @ | 15 341 @ | ...  
 341 @  
 ↑  
 touche de validation

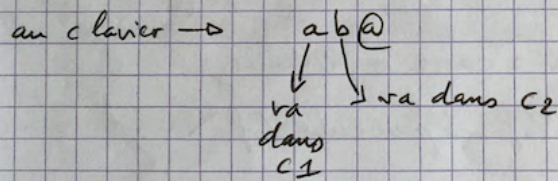


★ Lecture de caractères avec scanf

```
char c1;
scanf("%c", &c1);
```

Alors que les codes numériques (%d, %e, %f) sautent tous les séparateurs précédant une valeur, le code %c ne saute aucun séparateur et prend le premier caractère qui se présente, même s'il s'agit d'un espace ou d'une fin de ligne.

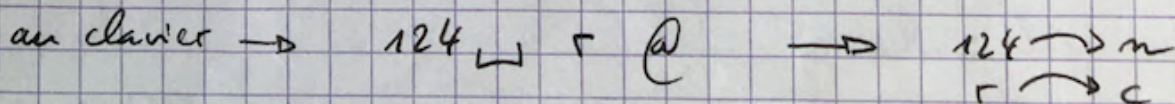
```
char c1, c2;
scanf("%c%c", &c1, &c2);
```



★ Pour forcer scanf à sauter les séparateurs

```
scanf("%d %c", &m, &c);
```

↑  
espace





---

## CHAPTER 4

### L'INSTRUCTION IF

ex.

```
main ()
{
    int n, p;
    printf (" donnez 2 nombres entiers : ");
    scanf ("%d %d", &n, &p);
    if (n < p)
    {
        printf ("croissant \n");
    }
    else
    {
        printf ("décroissant \n");
    }
} printf ("test finit");
```



ex:

```

main ( )
{
  int m, p, maxi;
  printf("donnez 2 entiers : ");
  scanf("%d %d", &m, &p);
  if (m < p)
  {
    maxi = p;
    printf("croissant \n");
  }
  else
  {
    maxi = m;
    printf("decroissant \n");
  }
  printf("le plus grand des deux nombres est : %d", maxi);
}

```

ex.

<pre> if (m &lt; p)   printf("croissant \n"); printf("fini \n"); </pre>	$\Leftrightarrow$	<pre> if (m &lt; p) {   printf("croissant \n"); } </pre>
---	-------------------	--

$\rightarrow$  on peut avoir un if tout seul (sans else)

★ Opérateurs de comparaison en C :

→ La signification d'un opérateur de comparaison n'est définie que pour des expressions de même type.

Opérateur	Signification "numérique"	Signification "caractère"
==	égal	identique
<	inférieur	de code inférieur
>	supérieur	de code supérieur
<=	≤	de code ≤
>=	≥	de code ≥
!=	≠	différent

N.B. pour la comparaison des caractères, on utilise en fait la valeur de leur code, c-à-d. la valeur obtenue en considérant que les 8 bits de leur code représentent un nombre entier.

- on a toujours 'a' < 'c', 'C' < 'S', '2' < '5'
- en revanche, aucune hypothèse ne peut être faite sur les places relatives des chiffres, des minuscules et des majuscules.

★ Les opérateurs logiques du langage C

opérateur	signification
&&	et
	ou (inclusif)



ex :  $(a < b) \text{ \&\& } (c < d)$  → prend la valeur "vrai" si les 2 expressions  $a < b$  et  $c < d$  sont toutes les 2 vraies, et la valeur "faux" dans le cas contraire.

$(a < b) \text{ \|\| } (c < d)$  → au moins un des deux conditions

$!(a < b)$  → vraie si  $a < b$  est faux

équivalent ↷  $a \geq b$

★ Les choix imbriqués

```

if (condition1)
{
    ..... /* réalise si condition1 est vraie */

    if (condition2)
    {
        ... /* si condition 1 et 2 vraies */
    }
    else
    {
        ... /* si cond 1 vraie et cond 2 fausse */
    }
}
else
{
    ... /* si condition 1 fausse */
}
..... /* réalise dans tous les cas */

```



---

## CHAPTER 5

### LES STRUCTURES DE RÉPÉTITION

ex:

```
main()
{
    int n;
    do
    {
        printf("donnez un entier: ");
        scanf("%d", &n);
        printf("son carre est: %d \n", n*n);
    }
    while (n != 0);

    printf("fin du programme");
}
```

```
do
{
  instructions;
}
while (expression);
```

→ la condition de poursuite est examinée à la fin de chaque tour.

★ L'instruction: while

ex:

```
while (cap <= 2 * cap_init)
{
  cap = cap * (1 + taux);
  printf("capital un an plus tard: %12.2f\n", cap);
}
```



## ★ Introduire un compteur dans une boucle.

ex:

```

main()
{
    int n;
    int i;          /* compteur */
    i = 0;
    while (i < 4)
    {
        printf("donnez un nlor entier: ");
        scanf("%d", &n);
        printf("son carre est: %d \n", n * n);
        i = i + 1;  /* +1 sur le compteur */
    }
}

```

## ★ L'instruction: for

```

for (i = 0; i < 4; i = i + 1)
{
    ...          /* suite des instructions à répéter */
}

```

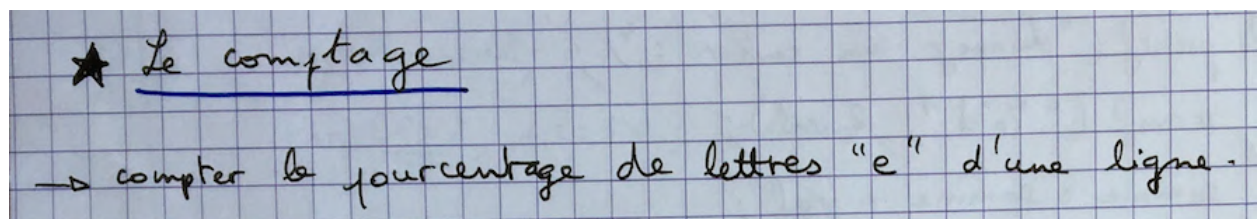


---

## CHAPTER 6

### QUELQUES TECHNIQUES USUELLES

Comptage, accumulation, recherche de maximum, répétitions imbriquées ...



```

main ()
{
  int n_e; n_e = 0;
  int n_c; n_c = 0;
  char c;
  float p;
  printf ("donnez une ligne de texte quelconque : \n");
  do
  {
    scanf ("%c", &c);
    n_c = n_c + 1;
    if (c == 'e')
    {
      n_e = n_e + 1;
    }
  }
  while (c != '\n');

  n_c = n_c - 1; // car fin de ligne comptée en trop */
  if (n_e == 0)
  {
    printf ("votre ligne ne comporte pas de e");
    // il ne faut pas écrire p = n_e / n_c * 100.0
    // car division entière donne 0 (sauf si n_e == n_c → 1)
  }
  else
  {
    p = (100.0 * n_e) / n_c;
    printf ("il y a %.2f pour cent de e", p);
  }
}

```



## ★ L'accumulation

ex:

```
main()
{
  int val, i, somme = 0;
  for (i = 1; i <= 100; i++)
  {
    printf("donnez un entier :");
    scanf("%d", &val);
    somme = somme + val;
  }
  printf("somme des valeurs fournies : %d", somme);
}
```

```
ex: main ()
{ float val; int nval; float somme = 0; nval = 0;
  do
  { printf ("donnez une valeur (0 pour terminer): ");
    scanf ("%e", &val);
    somme = somme + val;
    nval ++;
  }
  while (val != 0);

  if (nval <= 1)
  { printf ("aucune valeur - pas de moyenne");
  }
  else
  { printf ("moyenne des %d valeurs : %e", nval-1,
           somme / (nval-1) );
  }
}
```





Recherche de maximum

→ déterminer la valeur maximale de 50 valeurs entières lues au clavier

ex:

```

main()
{
    int val, max, i;
    printf("rentrez 50 valeurs\n");
    scanf("%d", &val); /* la première valeur sert de
                        maximum provisoire */
    max = val;
    for (i=2; i<=50; i=i+1)
    {
        scanf("%d", &val);
        if (val > max)
        {
            max = val;
        }
    }
    printf("le max de vos 50 valeurs est %d", max);
}
    
```



Imbrication de répétitions

→ écrire un programme qui affiche les tables de multiplication des nombres de 1 à 9, comme suit:

```

TABLE DES 4
4 x 1 = 4
4 x 2 = 8
  ⋮
4 x 10 = 40
    
```

```
main()
{
    int i, j, prod;
    for (i = 1; i <= 9; i = i + 1)
    {
        printf ("TABLE des %d \n", i);
        for (j = 1; j <= 10; j = j + 1)
        {
            prod = i * j;
            printf ("%2d x %2d = %2d \n", i, j, prod);
        }
    }
}
```



# CHAPTER 7

## LES TABLEAUX

La "structure de données" la plus utilisée en C est le tableau. Il s'agit d'un ensemble ordonné d'éléments de même type, caractérisé par :

- un nom
- un type (celui commun à tous ses éléments)
- le nombre de dimensions
- la valeur de chacune de ses dimensions.

On déclare un tableau ainsi :

```
| int val[10]; /* tableau nommé val, à une dimension, */  
/* de 10 éléments de type int */
```

```
| char lettres[5][25]; /* tableau nommé lettres, à deux */  
/* dimensions (5,25), de 125 */  
/* éléments de type char */
```

★ schéma

La 1<sup>ère</sup> valeur d'un indice est 0, et non 1, en C.

---

★ affectation de valeurs à un tableau

```
int x[4]; /* déclaration */
x[0]=12; x[1]=5; x[2]=8; x[3]=20; /* affectations */
```

★ Remarque

Il n'existe pas en langage C d'instruction agissant directement sur toutes les valeurs d'un tableau.

Par exemple, si l'on souhaite placer la même valeur (1) dans chacun des éléments du tableau  $x$ , il faut répéter une instruction d'affectation de cette valeur à un élément de rang  $i$ , c-à-d. :

```
for (i=0; i<4; i=i+1)
{
    x[i]=1;
}
```



De la même manière, si  $t1$  et  $t2$  sont deux tableaux de même type et de même taille :

```
int t1[100], t2[100];
```

la seule façon de recopier toutes les valeurs de  $t2$  dans  $t1$  sera d'utiliser une répétition de la forme :

```
for (i=0; i < 100; i=i+1)
{
    t1[i] = t2[i];
}
```

Une instruction telle que  $t2 = t1$  n'aurait pas de sens en langage C (elle serait rejetée par le compilateur).

rq: cela serait néanmoins possible avec le langage Matlab.

★ Lecture d'éléments d'un tableau

attention range dans un tableau

```
int x[4]; /* déclaration */
printf("donnez 4 entiers: \n");
for (i=0; i < 4; i=i+1)
{
    scanf("%d", &x[i]); /* lire l'élément d'indice i */
}
```

adresse de x[i]



## ★ Écriture d'éléments d'un tableau

```
main ( )
```

```
{ int tab[6]; int i; tab[0] = 0;
```

```
  for (i = 1; i < 5; i = i + 1)
```

```
    { tab[i] = 1;
```

```
      }
```

```
  tab[5] = 2;
```

```
  for (i = 0; i < 6; i = i + 1)
```

```
    { printf ("%d", tab[i]);
```

```
      }
```

```
}
```

0 1 1 1 1 2

Exercice de tri par ordre croissant des éléments d'un tableau à 1 dimension.

```

main()
{
    int t[15]; int i, j, temp;
    printf("donnez 15 entiers : \n");
    for (i=0; i<15; i=i+1)
    {
        scanf("%d", &t[i]);
    }

    for (i=0; i<14; i=i+1) /* faire avec tous les éléments est
                           /* sauf le dernier */
    {
        for (j=i+1; j<15; j=j+1) /* pour comparer t[i] et
                                   /* avec tous les suivants */
        {
            if (t[i] > t[j])
            {
                temp = t[i];
                t[i] = t[j];
                t[j] = temp;
            }
        }
    }

    printf("valeurs triées par ordre croissant : \n");
    for (i=0; i<15; i=i+1)
        printf("%d", t[i]);
}

```



### ★ initialisation de tableaux à 1 dimension

```
int tab[5] = {10, 20, 5, 0, 3};
char voy[6] = {'a', 'e', 'i', 'o', 'u', 'y'};
```

### ★ pour faciliter la modification de la dimension d'un tableau

On veut que le nombre d'éléments du tableau n'apparaisse qu'à un seul endroit du programme, au cas où on veuille le changer.

1) on pourrait penser à faire:

```
int nel = 10; /* déclaration de la variable nel */
float t[nel]; /* incorrect car nel n'est pas une constante */
```

En effet, la dimension d'un tableau doit être connue du compilateur pour qu'il puisse réserver la place; certes, la valeur de `nel` semble "accessible" au compilateur, cependant, rien ne lui dit qu'elle ne risque pas d'évoluer au fil de l'exécution du programme.



2) la solution consiste à utiliser une instruction particulière `#define`

→ ex.:

```
#define NEL 10
main ()
{ float t[NEL];
  .....
  for (i=0; i < NEL; i=i+1)
  }
  ...
```

3) Conseil: écrire `NEL` en majuscule pour mieux repérer dans le programme de tels symboles (définis par `#define`)

4) Rq: les instructions `#define` ne portent que sur la partie du programme qui les suit; en général on les place avant le programme. (main)

Rq: les instructions suivantes sont correctes:

```
#define NEL 10
.....
float t1[NEL];
float t2[NEL+1];
int t3[2*NEL];
```

car là où une constante est requise (i.e. la dimension du tableau), le compilateur accepte aussi ce que l'on nomme une « expression constante » c'est-à-dire une expression arithmétique ne faisant intervenir que des constantes, dont il sait déterminer la valeur au moment de la compilation.

## ★ Tableaux à deux dimensions.

```
int notes [20] [10];
```

- réserve l'emplacement d'un tableau d'entiers de 200 valeurs (20x10).
- chaque élément sera repéré par deux indices :
  - ↳ notes [i] [j]

lecture : `scanf ("%d", &notes [1] [3]);`

initialisation : `int tab [3] [4] = { { 1, 2, 3, 4 },`  
`{ 5, 6, 7, 8 },`  
`{ 9, 10, 11, 12 } };`

`int tab [3] [4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };`

**Exemple** d'utilisation d'un tableau à 2 dimensions.



```

#define NEL 25      /* nbr d'élèves */
#define NPAT 5     /* nbr de matières */

main()
{
    float notes[NEL][NPAT]; /* notes des élèves par matière */
    int coeff[NPAT] = {1, 3, 2, 4, 1}; /* coefficients */
    int s_coeff, i, j;
    float somme, moyenne;
    /* lecture des notes */
    for (i=0; i<NEL; i=i+1)
    {
        printf("Notes de l'élève numéro %d dans les %d matières?\n",
               i+1, NPAT);
        for (j=0; j<NPAT; j=j+1)
        {
            scanf("%e", &notes[i][j]);
        }
    }
}

```



```

/* calcul de la somme des coefficients */
s_coeff = 0;
for (i=0; i < NPAT; i=i+1)
{
    s_coeff = s_coeff + coeff[i];
}

/* calcul et affichage des moyennes par élève */

```

```

for (i=0; i < NEL; i=i+1)
{
    somme = 0;
    for (j=0; j < NPAT; j=j+1)
    {
        somme = somme + coeff[j] * notes[i][j];
    }
    moyenne = somme / s_coeff; /* on suppose s_coeff non nul */
    printf("moyenne de l'élève numéro %d : %7.2f \n",
           i+1, moyenne);
}

```

```
/* calcul et affichage moyennes par matiere */
```

```
for (j=0; j<NMAT; j=j+1)
```

```
{  
  somme = 0;
```

```
  for (i=0; i<NEL; i=i+1)
```

```
    {  
      somme = somme + notes[i][j];  
    }
```

```
  moyenne = somme / NEL;
```

```
  printf ("moyenne dans la matiere numero %d : %.2f \n",  
         j+1, moyenne);
```

```
}
```

```
}
```





---

## CHAPTER 8

### LES FONCTIONS

```
/* exemple de programme utilisant notre fonction cube */  
main ()  
{  
    float cube(float) ;      /* prototype de la fonction cube */  
    float a, c ;  
    a = 1.5 ;  
    c = cube(a) ;  
    printf ("c : %f \n", c) ;  
    c = cube(a) * 3.0 ;      /* utilisation de la fonction dans une expression  
                             arithmétique */  
    printf ("c : %f \n", c) ;  
    printf ("cube de 2.2 : %f", cube(2.2)) ;  
}
```

```

/* définition de notre fonction cube */
type du resultat
float cube (float x)      type et nom des paramètres
{
    float y;             nom de la fonction
    y = x * x * x;
    return y;
}
/* en-tête de la fonction cube */

```

La "portée" des variables locales (comme y) et des paramètres (comme x) est limitée à la fonction où ils sont définis.

★ exemple de fonction à plusieurs paramètres.

```

int max (int a, int b, int c)
{
    int m;
    m = a;
    if (b > m)
    {
        m = b;
    }
    if (c > m)
    {
        m = c;
    }
    return m;
}

```



★ exemple de fonction sans résultat  
 indique que la fonction ne fournit pas de résultat → il n'y a donc pas de "return"

```

void optimist (int n)
{
  int i;
  for (i=0; i < n; i=i+1)
  {
    printf ("il fait beau \n");
  }
}

```

y = optimist (k); /\* incorrect \*/  
 optimist (k); /\* correct \*/

★ L'instruction "return" → peut mentionner : un nom de variable, à l'importe quelle expression :

```

float cube (float x)
{
  return (x * x * x);
}

```

/\* les parenthèses ne sont pas indispensables \*/



### ★ Le cas des fonctions sans paramètres

Si une fonction ne possède aucun paramètre, son en-tête et donc sa déclaration (prototype) doivent comporter le mot void à la place de la liste des paramètres.

```
| int func1(void)      /* en-tête d'une fonction */
```

Sa déclaration (prototype) serait :

```
| int func1(void);    /* prototype */
```

Autre exemple :

```
| void func2(void)    /* en-tête */
```

```
| void func2(void);  /* prototype */
```

L'appel d'une fonction sans paramètres doit quand même comporter des parenthèses vides.

Par exemple, l'appel de `func1` se fera : `func1()`

★ En langage C : les paramètres d'une fonction sont toujours transmis "par valeur" .



main ()

```
{ void echange (int a, int b);
  int n = 10, p = 20;
  printf ("avant appel : %d %d \n", n, p);
  echange (n, p);
} printf ("après appel : %d %d", n, p);
```

```
void echange (int a, int b)
```

```
{ int c;
  printf ("début échange : %d %d \n", a, b);
  c = a;
  a = b;
  b = c;
  printf ("fin échange : %d %d \n", a, b);
}
```

avant appel : 10 20

début échange : 10 20

fin échange : 20 10

après appel : 10 20

### ★ Les variables globales (en langage C)

↳ variables qui sont simultanément accessibles à toutes les fonctions du programme (y compris le programme principal).

exemple :

```
int n ; // attention à l'emplacement de cette déclaration //  
main()  
{  
    void optimist(void);  
    n = 2; optimist();  
    n = 3; optimist();  
}  
  
void optimist(void)  
{  
    int i;  
    for (i = 0; i < n; i = i + 1)  
    {  
        printf("il fait beau!\n");  
    }  
}
```



### ★ Une variable globale peut être cachée

Le langage C autorise d'utiliser un nom déjà attribué à une variable globale pour une variable locale ou un paramètre. Mais dans ce cas ce nom (local) "cache" la variable globale qui ne peut alors plus être utilisée.

En général, ce genre de chose est déconseillé!

```
int m, p;
```

```
main ()
```

```
{ int m; /* ici m correspond à la variable m locale au prog principal
        tandis que p correspond à la variable globale */
}
```

```
void func (float x, float p)
```

```
{ /* ici m correspond à la variable globale tandis que p correspond au
    2ème paramètre formel de func */
}
```

### ★ Les fonctions pré-définies.

↳ "fonctions de la bibliothèque standard". (ex: printf, scanf ...)

```
[ #include <stdio.h> ]
```

stdio: STAnDard Input Output → tous les prototypes des fonctions liées aux opérations d'entrées-sorties.

```
[ #include <math.h> ]
```

```
→
sin
cos
tan
exp
log
log 10
sqrt
fabs
...
```



★ Cas des tableaux à une dimension transmis en paramètre d'une fonction

a) cas des paramètres tableau à une dim. de taille fixe

<p style="text-align: center;">remise à zéro ↓</p> <pre>void raz(int v[5]) {     int i;     for (i=0; i&lt;5; i=i+1)     {         v[i] = 0;     } }</pre>	<pre>main() {     ...     int t1[5];     ...     void raz(int [5]); /*probleme*/     ...     raz(t1); // t1 =&gt; {t1[0]}     ... }</pre>
--	---

**Rq** Si l'on s'intéresse de plus près à la manière dont le tableau est transmis en paramètre à la fonction, il faut savoir que :

- pour le compilateur, un nom de tableau (par ex.  $t_1$ ) est identique à son adresse, c-à-d. l'adresse de son premier élément ( $\&t_1[0]$ )
- l'appel  $\text{raz}(t_1)$  provoque la transmission à la fonction  $\text{raz}$  de la valeur du paramètre  $t_1$ , c-à-d. en fait de l'adresse du tableau  $t_1$ .
- dans la fonction  $\text{raz}$ , à chaque appel, on a :
 

```
v[i] = 0;
```

↳ est traduite par : "affecter au  $i$ -ième entier, à partir de l'adresse  $v$ , la valeur 0".

b) Cas des paramètres tableau à une dimension de taille variable.

→ on transmet également la taille du tableau en paramètre.

```
void raz (int v[], int nb)
{
    int i;
    for (i=0; i < nb; i=i+1)
    {
        v[i] = 0;
    }
}
```

ex:

```
main()
{
    int t1[10], t2[15], t3[100];
    void raz (int [], int); // prototype de raz //
    ...
    raz (t1, 10);
    raz (t2, 15);
    raz (t3, 100);
    ...
}
```



★ Cas des tableaux à deux dimensions transmis en paramètres d'une fonction

remise à un  
↓

```
void raun (int t[5][4])
{
    int i, j;
    for (i=0; i<5; i=i+1)
    {
        for (j=0; j<4; j=j+1)
        {
            t[i][j] = 1;
        }
    }
}
```

Exemples d'utilisation de cette fonction:

```
main ()
{
    int tab[5][4];
    raun (tab); /* => raun (&tab[0][0]); */
}
           adresse du début
           du tableau
```

Ex: | float t[5][3]; /\* 5 lignes de 3 éléments \*/

↳ les éléments se succèdent ainsi:

```
t[0][0]
t[0][1]
t[0][2]
t[1][0]
t[1][1]
t[1][2]
...
t[4][2]
```





# CHAPTER 9

## LES POINTEURS

★ Notion de pointeurs *de la zone mémoire*

En langage C, il est possible de définir une variable destinée à contenir une adresse. On pourrait dire qu'une telle variable est de type pointeur, mais, ceci n'est pas assez précis, car C distingue plusieurs types de pointeurs en se basant sur le type des informations qu'on trouvera à l'adresse indiquée : caractère, entiers, flottants...

a) La déclaration d'une variable pointeur

`int * adi;`    // l'\* sert à définir un pointeur, mais aussi, dans une expression, la valeur de l'objet pointé.

↑ *variable nommée adi (adresse int) destinée à contenir l'adresse d'un entier* } on dit qu'il s'agit d'une variable du type pointeur sur des entiers

{ nous verrons plus loin que \* est un opérateur particulier qui désigne la valeur située à l'adresse qui le suit



## b) L'opérateur &

Nous pouvons affecter une valeur à `adi`, pourvu que cette valeur soit de type "pointeur sur un entier".

Par ex., si nous avons déclaré :

```
int m = 20;
```

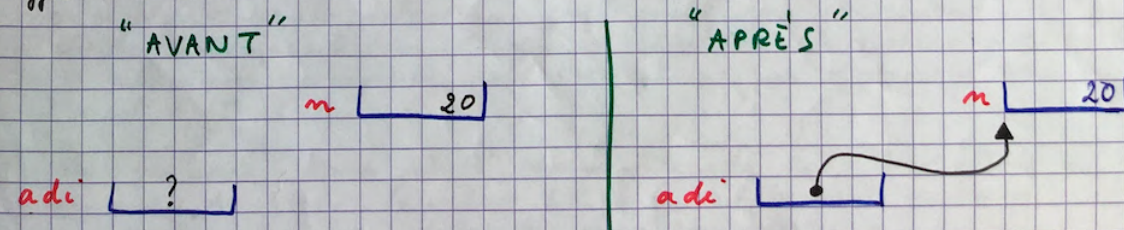
l'exécution de l'instruction :

```
adi = &m;
```

va affecter à `adi` l'adresse de notre variable `m` (rappel: `&` est un opérateur qui fournit l'adresse de la variable figurant à sa suite).

On dit alors que `adi` pointe sur `m`.

On peut schématiser la situation "avant" et "après" l'exécution de notre précédente affectation :



c) L'opérateur \*

L'opérateur \* joue un rôle symétrique de l'opérateur &. Il s'applique à une variable d'un type pointeur quelconque (c-a-d. pointant sur une information de type quelconque), et il désigne l'information ainsi pointée.

Ex: `int p;`  
`p = *adi;` → affecte à p la valeur désignée par \*adi, c'est-à-dire ici la valeur de n, soit 20  
contient  
`printf("%d", *adi);` → affiche la valeur 20

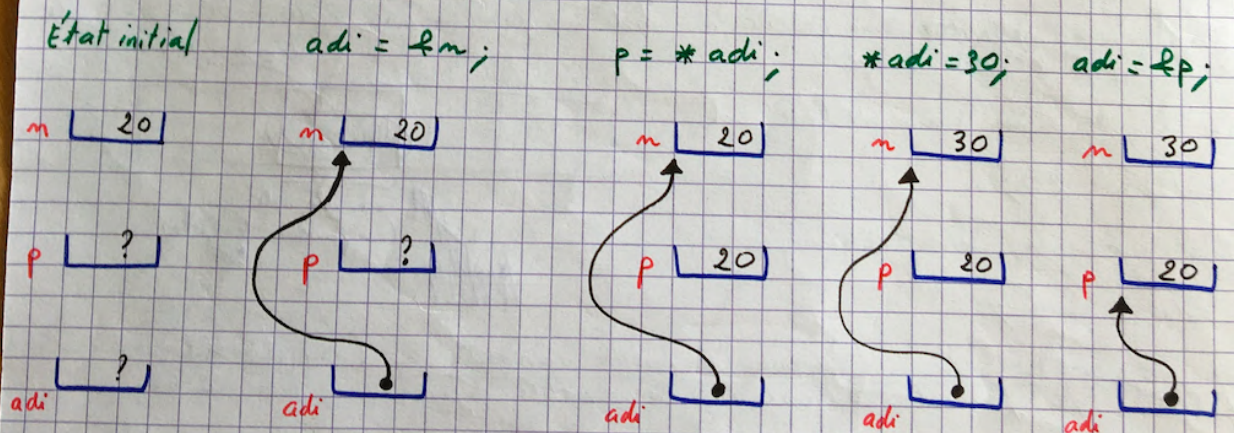
D'une façon similaire:

`*adi = 30;` → affectera à l'entier pointé par adi la valeur 30  
contient

Attention: on ne modifie pas ici la valeur de adi, mais bien celle de l'information pointée

Schema récapitulatif:

`int n = 20;`  
`int p;`  
`int *adi;`





LEMANQUES

① on ne peut pas attribuer une signification unique à une notation telle que  $*adi$ , en effet:

- $p = *adi;$   
 $*adi$  désigne la valeur de l'information pointée par  $adi$  (ici on est à droite, donc c'est le contenu)
- $*adi = 30;$   
 $*adi$  désigne l'information pointée par  $adi$  (on est à gauche de  $=$ , donc on désigne le contenant)

② il y a bien équivalence entre  $*adi$  et une variable entière:

- $*adi = 4.6;$   
 → il y a bien conversion de la valeur flottante  $4.6$  en entier, et le résultat ( $4$ ) est affecté à  $*adi$
- de même, si  $x$  est supposé de type float, avec  
 $x = *adi;$   
 → il y a conversion en flottant de la valeur entière pointée par  $adi$ , avant affectation à la variable  $x$

③ La priorité de l'opérateur unaire  $*$  est plus élevée que celle des opérateurs arithmétiques:

$2 * *adi$  est interprété comme  $2 * (*adi)$

④ Déclaration simultanée de plusieurs pointeurs:

$char * adc1, adc2;$	FAUX
$char * adc1, *adc2;$	CORRECT



⑤ } Réserver un pointeur ne réserve pas un emplacement pour une information pointée !

→ lorsqu'on réserve l'emplacement pour une variable pointeur

```
int * ad1;
```

on ne réserve pas pour autant un emplacement pour un entier.  
D'ailleurs, la valeur (donc l'adresse) contenue alors dans `ad1` est imprévisible.  
Si, suite à notre déclaration, nous introduisons une affectation telle que :

```
*ad1 = 12;
```

alors que nous n'avons affecté aucune valeur à `ad1`, nous allons en fait demander de placer la valeur 12 à un emplacement quelconque. Ce genre d'anomalie n'est pas détectée par le compilateur.

★ Affectations de pointeurs

Nous avons déjà vu comment affecter une valeur à une variable de type pointeur (nous parlerons souvent de "pointeur" tout court), à l'aide de l'opérateur `&`, comme dans `ad1 = &n`.

D'une manière générale, on peut affecter à un pointeur la valeur d'un autre pointeur de même type.

Exo:

```
int * ad1, * ad2;
int n=1, p=2, q=3;
ad1 = &n; /* ad1 pointe sur n */
ad2 = &p; /* ad2 pointe sur p */

*ad1 = *ad2 + q; /* m rôle ici que n=p+3.
                 n contient maintenant 5 */
ad1 = ad2; /* ad1 et ad2 pointent maintenant tous les deux
            sur p */
*ad1 = *ad2 + 5; /* m rôle ici que p=p+5; */
```

Il n'est pas permis d'affecter la valeur d'un pointeur d'un certain type à une variable pointeur d'un type différent.

Par exemple avec :

```
char * adc ;
int * adi ;
```

les affectations suivantes seraient incorrectes et provoqueraient une erreur de compilation :

```
adi = adc ;
adc = adi ;
```

} FAUX

Autre ex. : L'opérateur  $\&$ , appliqué à une variable, fournit en fait une valeur (constante) d'un type "pointeur sur des informations du type de cette variable".

```
int n ;
```

```
char * adc ;
```

```
adc = &n ;
```

FAUX :  $\&n$  est du type `int *`  
et `adc` est du type `char *`

★ Comment "simuler" une transmission par adresse avec un pointeur

→ but : écrire une fonction permettant des valeurs de deux variables



```
#include <stdio.h>
main ()
{
    void echange (int * ad1, int * ad2); /* prototype */
    int a = 10, b = 20;
    printf ("avant appel %d %d\n", a, b);
    echange (&a, &b);
    printf ("après appel %d %d", a, b);
}
```

ici on pourrait mettre ad1 et ad2, cf. chapitre précédent sur les fonctions et leurs prototypes.

les paramètres effectifs sont les adresses des variables a et b (et non plus leurs valeurs)

Notez bien que la transmission se fait toujours par valeur, à savoir que l'on transmet à la fonction echange les valeurs des expressions &a et &b

```
void echange (int * ad1, int * ad2)
{
    int x;
    x = * ad1;
    * ad1 = * ad2;
    * ad2 = x;
}
```

la case pointeur → contenant

Dans echange, nous avons indiqué comme paramètres muets deux variables pointeurs destinées à recevoir ces adresses

Rq: il n'aurait pas fallu se contenter d'échanger simplement les valeurs de ces pointeurs en écrivant:

```
int * x;
x = ad1;
ad1 = ad2;
ad2 = x;
```

cela n'aurait conduit qu'à échanger (localement) la valeur de ces deux adresses alors qu'il a fallu échanger les valeurs situés à ces adresses.

avant appel	10	20
après appel	20	10





## CHAPTER 10

# LES CHAÎNES DE CARACTÈRES

Une chaîne de caractères est une suite de caractères quelconques.  
Le langage C ne dispose pas d'un véritable type "chaîne". Il existe cependant une convention de représentation des chaînes qui consiste à indiquer leur fin par un caractère particulier de code nul qui peut se noter `\0`.

### ★ Comment lire ou écrire des chaînes

#### a) Avec les fonctions usuelles scanf ou printf

Soit un tableau de caractères déclaré par : `char nom[30];`

L'instruction : `scanf("%s", nom);`  
/\* ou bien : `scanf("%s", &nom[0]);` \*/

va lire une suite de caractères au clavier pour les ranger dans le tableau `nom`, en commençant à partir de `nom[0]` et en ajoutant automatiquement, à la suite, un caractère de fin de chaîne (caractère de code nul).

L'opérateur `%s` commence par sauter les délimiteurs éventuels (espace ou fin de ligne) et il s'interrompt à la rencontre de l'un de ces délimiteurs ; il ne permet donc pas de lire des chaînes contenant des espaces (pour cela, on utilisera la fonction `gets`).

REMARQUES: • lorsqu'on lit une chaîne de  $n$  caractères, il faut prévoir un emplacement d'au moins  $n+1$  caractères, compte tenu du caractère supplémentaire de fin de chaîne.

b) Avec les fonctions spécialisées gets et puts

| gets(nom);

→ lit une suite de caractères en la rangeant dans le tableau nom, terminée par un caractère de fin de chaîne. Aucun délimiteur n'est sauté avant la lecture, les espaces sont lus comme les autres caractères. la lecture se termine à la rencontre d'un caractère de fin de ligne.

De même: | puts(nom);

→ affiche les caractères trouvés à partir de nom[0], en s'interrompant à la rencontre du caractère de fin de chaîne et réalisant un changement de ligne.

c) Exemple

```
#include <string.h>
```

```
#include <stdio.h> + string
```

```
main()
```

```
{ char nom[20], prenom[20], ville[25];
```

```
printf("Quelle est votre ville?");
```

```
gets(ville);
```

```
printf("Nom et prénom?");
```

```
scanf("%s %s", &nom, &prenom);
```

```
printf("bonjour cher %s %s qui habitez à", prenom, nom);
```

```
puts(ville);
```

```
}
```

Quelle est votre ville? Paris

Nom et prénom? Bertrand Delanoë

bonjour cher Bertrand Delanoë qui habitez à Paris



★ Pour comparer des chaînes : la fonction `strcmp` → le prototype figure dans `string.h`

`strcmp (chaîne1, chaîne2);`

↓ fournit en résultat une valeur entière qui est :

$\left\{ \begin{array}{l} > 0 \text{ si chaîne 1 arrive "après" chaîne 2 au sens de l'ordre défini par le code des caractères} \\ = 0 \text{ si chaîne 1 est égale à chaîne 2 (si les 2 chaînes contiennent exactement la même suite de caractères)} \\ < 0 \text{ si chaîne 1 arrive "avant" chaîne 2, ...} \end{array} \right.$

Exemple : programme qui lit deux mots de moins de 30 lettres supposés ne contenir que des lettres minuscules, et qui indique s'ils sont ou non dans l'ordre alphabétique :

```

#define LG_MOT 30 // longueur max des chaînes
#include <stdio.h>
#include <string.h>

main()
{
    char mot1[LG_MOT+1]; /* +1 pour tenir compte du caractère de
                          fin de chaîne */
    char mot2[LG_MOT+1];
    int comp; /* pour le résultat de la comparaison des 2 mots */
    printf("donnez deux mots en minuscules :\n");
    scanf("%s %s", mot1, mot2);
    comp = strcmp(mot1, mot2);
    if (comp < 0) printf("dans l'ordre alphabétique\n");
    if (comp == 0) printf("identiques\n");
    if (comp > 0) printf("pas dans l'ordre alphabétique\n");
}

```

donnez deux mots en minuscules :  
bonjour hello  
dans l'ordre alphabétique



★ Pour recopier des chaînes : la fonction `strcpy` (`string.h`)

```
| strcpy (destination, source);
```

Exemple : (variante de l'ex. précédent)

```
# define LG_MOT 30
# include <stdio.h>
# include <string.h>

main ()
{
    char mot1 [LG_MOT+1]; char mot2 [LG_MOT+1];
    char mot [LG_MOT+1]; /* pour procéder à l'échange éventuel
                           de mot1 et mot2 */

    printf ("donnez deux mots en minuscules : \n");
    scanf ("%s %s", mot1, mot2);
    if (strcmp (mot1, mot2) > 0)
        {
            strcpy (mot, mot1);
            strcpy (mot1, mot2);
            strcpy (mot2, mot);
        }
    printf ("voici vos deux mots ordonnés : %s %s", mot1, mot2);
}
```

donnez deux mots en minuscules:  
 hello bonjour  
 voici vos deux mots ordonnés: bonjour hello

★ Pour obtenir la longueur d'une chaîne : la fonction strlen (string.h)

exemple :

```
#define LG_MOT 30
#include <stdio.h>
#include <string.h>
main()
{ char mot[LG_MOT+1]; int i;
  int ne; /* compteurs du nombre de e */
  printf("donnez un mot: \n");
  scanf("%s", mot);
  ne = 0;
  for (i = 0; i < strlen(mot); i = i + 1)
  { if (mot[i] == 'e')
    { ne = ne + 1;
    }
  }
  printf("votre mot de %d lettres comporte %d fois la lettre e",
        strlen(mot), ne);
}
```

donnez un mot:  
 terre  
 votre mot de 5 lettres  
 comporte 2 fois la  
 lettre e

★ Les constantes chaîne

exemple :

|"bonjour"|

strlen("bonjour") → 7

1 2 3 4 5 6 7



★ Concaténation de chaînes : strcat (string.h)

| strcat (destination, source) // attention au pb de place suffisante dans destination... (V)

La fonction strcat concatène à la chaîne destination tous les caractères qu'elle trouve dans la chaîne source.

ex // strcat (char \*d, char \*s) // concatène la chaîne pointée par s en fin de la chaîne pointée par d.

★ Manipulation d'une partie d'une chaîne

| strcat (mot1, &mot2 [3]).

→ concatène à mot1 la partie de mot2 qui commence à son 3<sup>ème</sup> caractère

Autre exemple de manipulation d'une partie d'une chaîne.

Exemple :

```
#include <stdio.h>
#include <string.h>
main ()
{
    char mot [30];
    printf ("donnez un mot se terminant par er :");
    gets (mot);
    if (strcmp (&mot [strlen (mot) - 2], "er") == 0)
    {
        printf ("le mot %s se termine bien par er", mot);
    }
    else
    {
        printf ("le mot %s ne se termine pas par er", mot);
    }
}
```

on passe l'adresse de 'er' à strcmp, et après elle "compare" jusqu'à ce qu'elle trouve '\0'

donnez un mot se terminant par er : programme  
 le mot program ne se termine pas par er

### ★ Exemple: inverser un mot

But: écrire un programme qui lit un mot et qui l'affiche à l'envers, et nous voulons que le mot inversé soit effectivement créé en mémoire

solution: on crée la chaîne contenant le mot inversé, caractère par caractère, en ajoutant au bon endroit le caractère de fin de chaîne.

```
# define LG_MOT 30
# include <stdio.h>
# include <string.h>

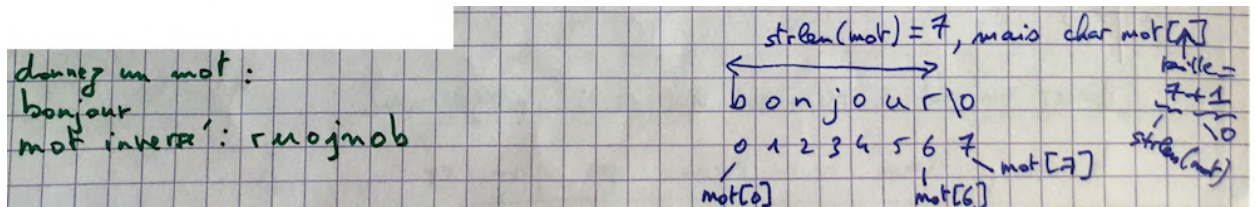
main()
{
  char mot [LG_MOT + 1]; /* pour le mot lui */
  char motinv [LG_MOT + 1]; /* pour le mot inverse */
  int i;
  int longueur; /* pour la longueur du mot */
  /* lecture du mot */
  printf ("donnez un mot : \n");
  scanf ("%s", mot);
```



```

/* création en mémoire du mot inversé dans motinv */
longueur = strlen(mot);
for (i=0 ; i < longueur ; i=i+1)
    { motinv [longueur - i - 1] = mot [i] ;
    }
motinv [longueur] = '\0' ; /* pour indiquer la fin de chaîne */

/* affichage du mot inversé */
printf ("mot inversé : %s", motinv);
}
    
```



★ Initialisation rapide de tableaux de caractères

ex: | char t[40] = "bonjour";

| char t[40] = { 'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0' };;

★ Tableaux de chaînes (de caractères)

a) Comment "simuler" des tableaux de chaînes

| char noms [10][25]; /\* pour ranger 10 chaînes d'au plus  
24 caractères \*/

| scanf ("%s", &noms [3][0]); /\* cela correspond à la 4<sup>ème</sup>  
ligne du tableau \*/

| strcpy (&noms [5][0], "hello"); /\* va placer dans la 6<sup>ème</sup> ligne du  
tableau noms, la chaîne "hello", i.e. les caractères h, e, l, l, o, et le fin de chaîne '\0'.

b) Exemple:

↳ voici un programme qui lit 5 noms d'au plus 26 caractères et qui les réaffiche dans l'ordre alphabétique.



```
#define LG_MOTS 26
#define NB_MOTS 5
#include <stdio.h>
#include <string.h>

main()
{
    char mots[NB_MOTS][LG_MOTS+1];
    char tempo[LG_MOTS+1];
    int i, j;
    printf("donnez %d mots \n", NB_MOTS);
    /* lecture des mots a' trier */
    for (i=0; i < NB_MOTS; i=i+1)
    {
        scanf("%s", &mots[i][0]);
    }
}
```

```

/* tri des mots */
for (i=0; i < NB_MOTS-1; i=i+1)
{
    for (j=i+1; j < NB_MOTS; j=j+1)
    {
        if (strcmp(&mots[i][0], &mots[j][0]) > 0)
        {
            strcpy(destination tempo, source &mots[i][0]);
            strcpy(&mots[i][0], &mots[j][0]);
            strcpy(&mots[j][0], tempo);
        }
    }
}

```

```

/* affichage des mots triés */
printf("voici vos mots triés: \n");
for (i=0; i < NB_MOTS; i=i+1)
{
    puts(&mots[i][0]);
}

```



# CHAPTER 11

## LES STRUCTURES

En langage C, outre le tableau, il existe une seconde "structure de données", qu'on appelle une structure.

### ★ Déclaration d'une structure

```
struct enreg  
{  
    int numero;  
    int qte;  
    float prix;  
};
```

Ceci définit un modèle de structure, mais ne réserve pas de "variables" correspondant à cette structure.

Ce modèle s'appelle ici enreg et il précise le nom et le type de chacun des "champs" constituant la structure (numero, qte, prix).

Une fois un tel modèle défini, nous pouvons déclarer des "variables" du type correspondant. Par exemple:

```
struct enreg article1;
```

↳ réserve un emplacement nommé article1 de type enreg destiné à contenir deux entiers et un flottant.

```
struct enreg art1, art2, art3;
```

## ★ Utilisation d'une structure

### a) Utilisation des champs d'une structure

exemples:

```
art1.numero = 15; /* affecte la valeur 15 au champ numero de la
                  structure art1 */

printf("%e", art1.prix); /* affiche, suivant le code format %e, la
                          valeur du champ prix de la structure art1 */

scanf("%le", &art2.prix);

art1.numero = art1.numero + 1;
```

### b) Utilisation globale d'une structure (pas conseillé)

Si, et seulement si, art1 et art2 ont été déclarés suivant le même modèle enreg, on peut faire une affectation "globale": `art1 = art2;`

En dehors de l'affectation, il n'y a d'autres possibilités d'utilisation globale d'une structure.

**Rq:** pour les tableaux, on ne dispose d'aucune possibilité d'utilisation globale, pas même au niveau de l'affectation.

### c) Initialisation de structures

```
struct enreg art1 = { 100, 285, 49.95 };
```



## d) La portée du modèle de structure

Ex:

```
struct enreg
{
    int numero;
    int qte;
    float prix;
};
```

```
main()
{
    struct enreg x;
    ...
}
```

```
fonction lambda(... )
{
    struct enreg y, z;
    ...
}
```

## ★ Imbrication de structures

### a) Structure comportant des tableaux

```

struct personne
{
    char nom [30];
    char prenom [20];
    float heures [31];
};

```

← peut accueillir une chaîne d'au plus 29 caractères

← float heures

```

struct personne employe;

```

ex: employe.heures [4] désigne le 5<sup>ème</sup> élément du tableau heures de la structure employe

employe.nom [0] représente le 1<sup>er</sup> caractère du champ nom de la structure employe

← employe.heures [4] représente l'adresse du cinquième élément du tableau heures de la structure employe

### b) Tableaux de structures

```

struct point {
    char nom;
    int x;
    int y;
};

```

```

struct point courbe [50];

```

i est un entier ici

→ courbe [i].x désigne la valeur du champ x de l'élément de rang i du tableau courbe

→ struct point courbe [50] = { {'A', 10, 25}, {'M', 12, 28}, {'P', 18, 2} };



c) Structures comportant d'autres structures

```

struct date
{
    int jour;
    int mois;
    int annee;
};

```

```

struct personne
{
    char nom [30];
    char prenom [20];
    float heures [31];
    struct date date_embauche;
    struct date date_poste;
};

```

```

struct personne employe1, employe2;

```

La notation `employe1.date_embauche.annee;` représente l'année d'embauche correspondant à la structure `employe1`. Il s'agit d'une valeur de type `int`.

★ Transmission d'une structure en paramètre d'une fonctiona) Transmission de la valeur d'une structure

```

#include <stdio.h>

struct enreg { int a;
               float b;
               };

main()
{
    struct enreg x;
    void fct(struct enreg y); /* prototype */
    x.a = 1; x.b = 12.5;
    printf("In Avant appel fct: %d %e", x.a, x.b);
    fct(x);
    printf("In Au retour dans main: %d %e", x.a, x.b);
}

```

*optionnel (pas nécessaire)*

```

void fct(struct enreg s)
{
    s.a = 0; s.b = 1;
    printf("In Dans fct: %d %e", s.a, s.b);
}

```

Avant appel fct : 1 1.25000e+01  
 Dans fct : 0 1.00000e+00  
 Au retour dans main: 1 1.25000e+01



NUST

1) quand on passe un tableau  $t$  dans une fonction, celui-ci est modifiable dans la fonction, car celle-ci connaît son adresse (car  $t$  représente d'office l'adresse du tableau).  
 cf. perso → (quand on exécute le prog.,  $t$  est remplacé par son adresse, il n'est pas dans la table des variables)

2) par contre, quand on passe une structure dans une fonction, celle-ci est dupliquée et donc non modifiable par la fonction (on dit que c'est comme un scalaire, c-à-d. int, char...).

N.B. Il faudrait donc donner l'adresse de la structure dans une fonction pour la modifier à l'intérieur de la fonction.

b) Pour transmettre l'adresse d'une structure : l'opérateur  $\rightarrow$

Cherchons à modifier notre précédent programme pour que la fonction  $fct$  reçoive l'adresse d'une structure et non plus sa valeur.

L'appel de  $fct$  devra donc se présenter sous la forme :  $fct(\underline{p});$

Cela signifie que son en-tête (ou prototype) devra préciser que son unique paramètre est du type "pointeur sur des informations de type struct enreg".

Il sera donc de la forme :

```
void fct ( struct enreg * ads );
```

*pas nécessaire pour le prototype*

Nous allons devoir, au sein de la définition de  $fct$ , désigner les différents champs de "la structure ayant pour adresse  $ads$ ".

Il n'est plus question d'utiliser la notation "point" qui s'applique à un nom de structure et non à une adresse.

Il faut en fait recourir à une autre et nouvelle notation, dans laquelle on remplace le point par le symbole  $\rightarrow$ . Ainsi,  $ads \rightarrow b$  désignera le second champ de la structure reçue en paramètre.

(c'est une *comme dite* facilité d'écriture : en fait :

```
{ (*p).nom } ⇔  
  p->nom
```

*(\*)p).nom  
 ↑↑ ↑↑ : 4 symboles  
 p->nom  
 ↑↑ : 2 symboles*

```

ex:
#include <stdio.h>
struct enreg { int a;
              float b;
            };

main()
{ struct enreg x;
  void fct (struct enreg x); x.a = 1; x.b = 12.5;
  printf ("\ avant : %d %e", x.a, x.b);
  fct (x);
  printf ("\ apres : %d %e", x.a, x.b);
}

```

```

void fct (struct enreg *ads)
{ ads->a = 0; ads->b = 1; // => (*ads).a = 0;
  printf ("\ pendant : %d %e", ads->a, ads->b);
}

```

```

avant : 1 1.25000e+01
pendant : 0 1.00000e+00
apres : 0 1.00000e+00 ←

```



## CHAPTER 12

# ALLOCATION DE MÉMOIRE DYNAMIQUE

objectif : apprendre à demander de la mémoire manuellement.

On doit inclure la bibliothèque `<stdlib.h>`, qui contient les fonctions dont nous allons avoir besoin :

- malloc (memory allocation) : demande au système d'exploitation la permission d'utiliser de la mémoire.
- free : permet d'indiquer à l'OS (operating system) que l'on n'a plus besoin de la mémoire qu'on avait demandée. La place en mémoire est libérée, un autre programme peut maintenant s'en servir au besoin.

Idéalement, quand on fait une allocation manuelle de mémoire, on doit suivre 3 étapes :

- 1) appeler `malloc` pour demander de la mémoire ;
- 2) vérifier la valeur retournée par `malloc` pour savoir si l'OS a bien réussi à allouer la mémoire.
- 3) une fois qu'on a fini d'utiliser la mémoire, on doit la libérer avec `free`. Si on ne le fait pas, on s'expose à des fuites de mémoire, c-à-d. que votre programme risque au final de prendre bcp de mémoire par rapport aux vrais besoins.  
trop

Prototype de la fonction malloc :

```
void * malloc (size_t nombreOctetsNecessaires);
```

La fonction prend en paramètre le nombre d'octets à réserver.  
Ainsi, il suffira d'écrire sizeof(int) dans ce paramètre pour réserver suffisamment d'espace pour stocker un int.

La fonction renvoie un void \*.  
pointeur sur n'importe quel type  
("pointeur universel").

En effet, malloc ne sait pas quel type de variable on cherche à créer puisqu'on ne lui donne en paramètre qu'un nombre d'octets en mémoire dont on a besoin.

Comme malloc ne sait pas quel type elle doit retourner, elle renvoie le type void \*.

Rq : malloc renvoie un pointeur indiquant l'adresse que l'OS a réservée pour votre variable. Si l'OS a trouvé de la place pour vous à l'adresse 1600, la fonction renvoie donc un pointeur contenant l'adresse 1600.

(ici 1600 représente juste une certaine adresse en mémoire, ce n'est pas le nombre 1600 !)



Exemple :

```
int * memoireAllouee = NULL; // on crée un pointeur sur int
memoireAllouee = malloc(sizeof(int)); // la fonction malloc
// 4 octets inscrit dans notre
// pointeur l'adresse qui
// a été réservée.
```

À la fin de ce code, `memoireAllouee` est un pointeur contenant une adresse qui vous a été réservée par l'OS, par exemple l'adresse 1600 pour reprendre les schémas d'avant.

Tester le pointeur

La fonction `malloc` a donc renvoyé dans notre pointeur `memoireAllouee` l'adresse qui a été réservée pour vous en mémoire.

Deux possibilités :

- 1) Si l'allocation a marché, notre pointeur contient une adresse.
- 2) Si l'allocation a échoué, notre pointeur contient l'adresse `NULL`.

```

int main (int argc, char * argv[])
{
    int * memoireAllouee = NULL;

    memoireAllouee = malloc (sizeof (int));
    if (memoireAllouee == NULL) // si l'allocation a échoué
    {
        exit (0); // on arrête immédiatement le programme
    }

    // on peut continuer le prog. normalement sinon

    return 0;
}

```

Free : libérer de la mémoire.

prototype : void free (void\* pointeur);

Où:

```

int main (int argc, char * argv[])
{
    // ...
    free (memoireAllouee); // on n'a plus besoin de la mémoire,
                           // on la libère.
    // ...
    return 0;
}

```

} pareil que ci-dessus.



Et voici un exemple complet d'un petit programme dans lequel on a besoin de faire de l'allocation dynamique de mémoire (ici pour un tableau de int):

Ex. Allocation dynamique d'un tableau.

```
int main(int argc, char * argv[])
{
    int nombreDAmis = 0;
    int i = 0;
    int * ageAmis = NULL; // ce pointeur va servir de tableau
                          // après l'appel du malloc
    printf("Combien d'amis avez-vous?");
    scanf("%d", &nombreDAmis);
```

```
if (nombreDAmis > 0)
{
    ageAmis = malloc(nombreDAmis * sizeof(int)); // on alloue de
                                                    // la mémoire
                                                    // pour le tableau
    if (ageAmis == NULL) // on vérifie si l'allocation
                          // a bien marché
    {
        exit(0);
    }
    for (i = 0; i < nombreDAmis; i++)
    {
        printf("quel âge a l'ami numéro %d?", i+1);
        scanf("%d", &ageAmis[i]);
    }
}
```

```
// on affiche ...  
printf ("\n Vos amis ont les âges suivants :\n");  
for (i=0; i < nombreAmis; i++)  
{ printf ("%d ans\n", ageAmis[i]);  
}  
free (ageAmis); // on libère la mémoire allouée avec  
                malloc, on n'en a plus besoin  
}  
return 0;
```

✂ FIN ✂