

## Les pointeurs

julien.brajard@upmc.fr revus par maria.potop-butucaru@lip6.fr

Polytech'UPMC

Octobre 2018

<https://moodle-sciences.upmc.fr> (cours Informatique Générale  
MAIN-ROB)

# Plan du cours

- 1 Adresse et pointeurs
- 2 Pointeurs et tableaux
- 3 Pointeurs et passage de paramètre
- 4 Entrées/Sorties
- 5 Les chaînes de caractères

- 1 Adresse et pointeurs
- 2 Pointeurs et tableaux
- 3 Pointeurs et passage de paramètre
- 4 Entrées/Sorties
- 5 Les chaînes de caractères

# Variable et adresse mémoire

La mémoire d'un ordinateur est organisée en suite de cases repérées par une adresse.

- Chaque case mémoire a une adresse et un contenu.

Les variables d'un programme sont stockées en mémoire et possèdent une valeur.

```
int n=3;
```

|         |     |      |      |      |     |
|---------|-----|------|------|------|-----|
| Adresse | ... | 4584 | 4585 | 4586 | ... |
| Valeur  | ... |      | 3    |      |     |

n

n est à l'adresse 4585

# L'opérateur &

L'opérateur & permet de retrouver l'adresse d'une variable.

```
int n=3;
```

|         |     |      |      |      |     |
|---------|-----|------|------|------|-----|
| Adresse | ... | 4584 | 4585 | 4586 | ... |
| Valeur  | ... |      | 3    |      |     |

n

n     contient 3

&n    contient 4585

# Les pointeurs

Un pointeur est une variable contenant l'adresse d'une case mémoire.

- Déclaration :

```
int *pn ;
```

|         |     |      |      |      |     |      |
|---------|-----|------|------|------|-----|------|
| Adresse | ... | 4584 | 4585 | 4586 | ... | 6208 |
| Valeur  | ... |      | 3    |      |     |      |
|         |     |      | n    |      |     |      |
|         |     |      |      |      |     | pn   |

- Affection :

```
pn = &n ;
```

|         |     |      |      |      |     |      |
|---------|-----|------|------|------|-----|------|
| Adresse | ... | 4584 | 4585 | 4586 | ... | 6208 |
| Valeur  | ... |      | 3    |      |     | 4585 |
|         |     |      | n    |      |     | pn   |

## L'opérateur \*

- L'opérateur `&` permet de retrouver l'adresse d'une variable.
- L'opérateur `*` permet de retrouver le contenu d'une adresse mémoire.  
(Opération de déréférencement)

|         |     |      |      |      |     |      |
|---------|-----|------|------|------|-----|------|
| Adresse | ... | 4584 | 4585 | 4586 | ... | 6208 |
| Valeur  | ... |      | 3    |      |     | 4585 |

$n$  $pn$

$n$      contient 3                       $\&n$      contient 4585  
 $pn$     contient 4585                   $*pn$     contient 3

Les égalités suivantes sont vraies :

$n == *pn$

$pn == \&n$

# Manipulation des pointeurs

```
int x=1, y=2, z=3 ;
```

| Adresse | 2158 | 2159 | 2160 |
|---------|------|------|------|
| Valeur  |      |      |      |
|         | x    | y    | z    |

```
int *p = &x ;
```

| Adresse | 2158 | 2159 | 2160 | 2161 |
|---------|------|------|------|------|
| Valeur  |      |      |      |      |
|         | x    | y    | z    | p    |

```
y = *p ;
```

| Adresse | 2158 | 2159 | 2160 | 2161 |
|---------|------|------|------|------|
| Valeur  |      |      |      |      |
|         | x    | y    | z    | p    |

```
*p = 0 ;
```

| Adresse | 2158 | 2159 | 2160 | 2161 |
|---------|------|------|------|------|
| Valeur  |      |      |      |      |
|         | x    | y    | z    | p    |

```
p =&z ;
```

| Adresse | 2158 | 2159 | 2160 | 2161 |
|---------|------|------|------|------|
| Valeur  |      |      |      |      |
|         | x    | y    | z    | p    |



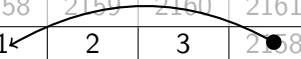
# Autre Notation

```
int x=1, y=2, z=3 ;
```

| Adresse | 2158 | 2159 | 2160 |
|---------|------|------|------|
| Valeur  | 1    | 2    | 3    |
|         | x    | y    | z    |

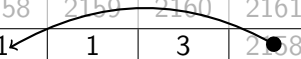
```
int *p = &x ;
```

| Adresse | 2158 | 2159 | 2160 | 2161 |
|---------|------|------|------|------|
| Valeur  | 1    | 2    | 3    | ●    |
|         | x    | y    | z    | p    |



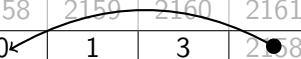
```
y = *p ;
```

| Adresse | 2158 | 2159 | 2160 | 2161 |
|---------|------|------|------|------|
| Valeur  | 1    | 1    | 3    | ●    |
|         | x    | y    | z    | p    |



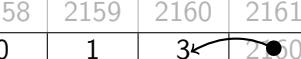
```
*p = 0 ;
```

| Adresse | 2158 | 2159 | 2160 | 2161 |
|---------|------|------|------|------|
| Valeur  | 0    | 1    | 3    | ●    |
|         | x    | y    | z    | p    |



```
p = &z ;
```

| Adresse | 2158 | 2159 | 2160 | 2161 |
|---------|------|------|------|------|
| Valeur  | 0    | 1    | 3    | ●    |
|         | x    | y    | z    | p    |



## Autre exemple

```
#include <stdio.h>

int main()
{
    int u1, u2, v=3;
    int *p1;
    int *p2;
    u1 = 2*(v+5);
    p1 = &v;
    p2 = p1;
    *p2 = 5;
    p2=&u1;
    u2 = 2*(*p1+5);
    *p2=*p2+1;
    printf("u1=%d u2=%d v=%d", u1, u2, v);
    return (0);
}
```

Test d'exécution

u1=... u2=... v=...

- 1 Adresse et pointeurs
- 2 Pointeurs et tableaux**
- 3 Pointeurs et passage de paramètre
- 4 Entrées/Sorties
- 5 Les chaînes de caractères

# Qu'est-ce qu'un tableau

## Point de vue algorithmique

Un tableau est un ensemble de données de même type.

# Qu'est-ce qu'un tableau

## Point de vue algorithmique

Un tableau est un ensemble de données de même type.

## Point de vue langage C

Un tableau est un **pointeur** sur le premier élément d'un ensemble de données de même type.

# Qu'est-ce qu'un tableau

## Point de vue algorithmique

Un tableau est un ensemble de données de même type.

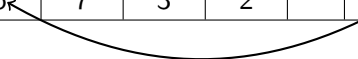
## Point de vue langage C

Un tableau est un **pointeur** sur le premier élément d'un ensemble de données de même type.

```
int Tab[4] = {5, 7, 3, 2};
```

|         |      |      |      |      |     |      |
|---------|------|------|------|------|-----|------|
| Adresse | 4583 | 4584 | 4585 | 4586 | ... | 6208 |
| Valeur  | 5    | 7    | 3    | 2    |     | 4583 |

Tab



Les égalités suivantes  
sont vraies :

`Tab == &Tab[0]`

`*Tab == Tab[0]`

# Allocation dynamique de mémoire pour un tableau

## Allocation

On peut affecter dynamiquement la mémoire d'un tableau :

- fonction `malloc` de la bibliothèque `stdlib.h` :  
`nom_pointeur = (type*)malloc(espace_memoire);`
- `espace_memoire = taille du tableau × taille du type`

Exemple pour un tableau de 10 entiers :

```
int *Tab ;  
Tab = (int *) malloc (10*sizeof(int));
```

## Libération

Lorsque la mémoire n'est plus utilisée, il faut penser à la libérer :

```
free(nom_pointeur);
```

## Exemple

```
#include <stdio.h>
#include <stdlib.h>

void init (int *Tab, int n);

int main()
{
    int *Tab;
    int n;
    printf("Entrez la taille du tableau : ");
    scanf("%d",&n);
    Tab=(int *)malloc(n*sizeof(int));
    init(Tab, n);
    ...
    free(Tab);
    return (0);
}
```



# Problèmes d'allocation mémoire

Le système ne peut plus allouer de mémoire ?

- Des zones réservées n'ont pas été libérées ;
- La taille des zones demandées est trop grande.

Il faut vérifier que la zone mémoire est bien valide (test sur la nullité du pointeur sur cette zone).

```
Tab=(int *) malloc (n*sizeof(int));  
if (Tab == NULL) {  
    //Traitement de l'erreur  
    printf("Erreur dans l'allocation mémoire");  
}
```

# Allocation statique Vs Allocation dynamique

## Statique

- La mémoire est spécifiée dans le code ;
- Elle est réservée lors de la compilation ;
- Il n'y a pas d'allocation à l'exécution (pas de problème d'allocation lors de l'exécution)

## Dynamique

- Le programmeur gère sa mémoire ;
- Il n'y a pas besoin de connaître à l'avance (à la compilation) la taille du tableau.

# Les tableaux en 2 dimensions

Un tableau en deux dimensions peut être vu comme un tableau de tableau et donc un pointeur sur un pointeur...

```
int **Tab2D ;
```

Il faut donc allouer la mémoire pour le tableau principal et pour chacun des "sous-tableaux".

```
int nblignes=3,nbcolonnes=4;
```

```
Tab2D=(int **) malloc ( nblignes*sizeof (int *));
```

```
for (i=0;i<nblignes;i++) {  
    Tab2D[i]=(int *) malloc (nbcolonnes*sizeof (int));  
}
```

## Exemple

Initialiser le tableau

|    |    |    |
|----|----|----|
| 1  | 2  | 3  |
| 11 | 12 | 13 |

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int **Tab2D;
    int i, j, nblignes=2, nbcolonnes=3;

    Tab2D=(int **) malloc ( nblignes*sizeof(int *));

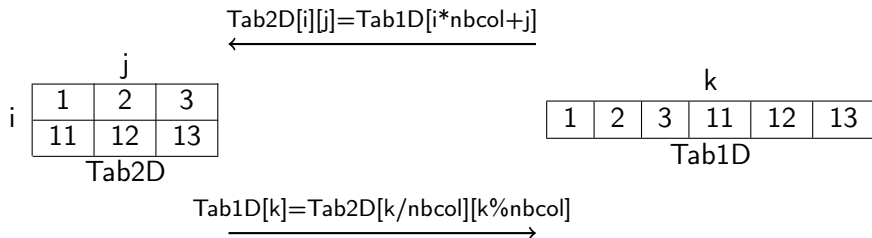
    for (i=0; i<nblignes; i++) {
        tab2D[i]=(int *) malloc (nbcolonnes*sizeof(int));
    }

    for (i=0; i<lignes; i++) {
        for (j=0; j<colonnes; j++) {
            Tab2D[i][j]=10*i + j+1;
        }
    }
}
```

## 2D ou 1D ?

La gestion de la mémoire des tableaux en 2 dimensions peut être compliquée.

On peut "aplatir" des tableaux en deux dimensions sur 1 dimension, quitte à faire des calculs sur les indices



- 1 Adresse et pointeurs
- 2 Pointeurs et tableaux
- 3 Pointeurs et passage de paramètre**
- 4 Entrées/Sorties
- 5 Les chaînes de caractères

# Passage par valeur

Argument formel

```
int fct (int a);
```

Argument effectif

```
int n = 3 ;  
fct(n);
```

Dans la fonction

```
return (a+1);
```

# Un exemple de passage par valeur

```
#include <stdio.h>

void fct(int x)
{
    x = x+2;
}

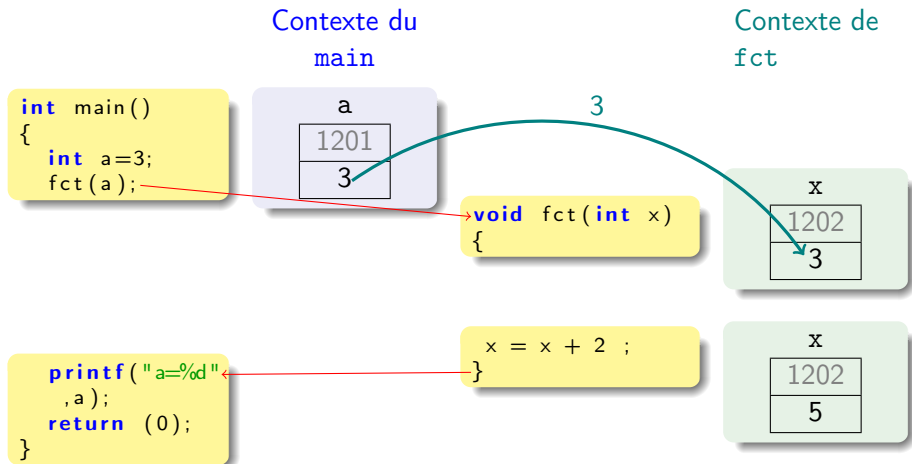
int main()
{
    int a=3;
    fct(a);
    printf(" \na=%d",a);
    return (0);
}
```

Test d'exécution

a = ...



# Suivi de la mémoire



## Passage par adresse (ou par référence)

Argument formel

```
int fct (int *pa);
```

Argument effectif

```
int n = 3 ;  
fct(&n);
```

Dans la fonction

```
return (*pa + 1);
```

# Un exemple de passage par adresse

```
#include <stdio.h>

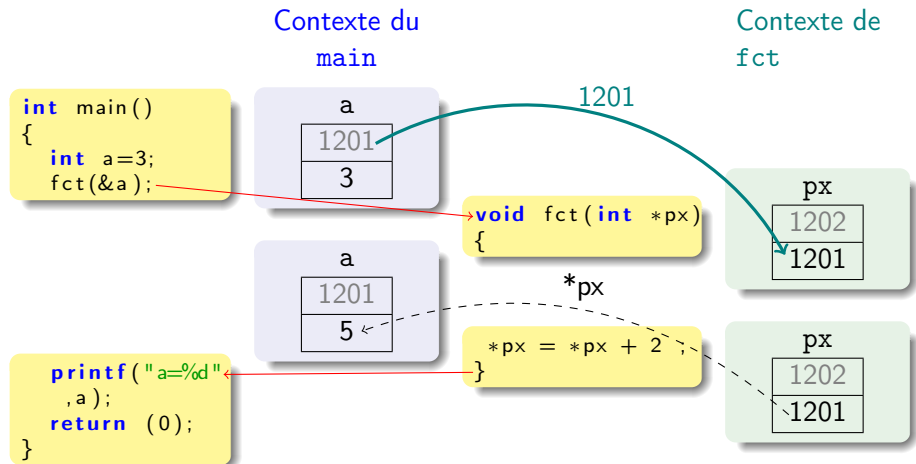
void fct(int *px)
{
    *px = *px+2;
}

int main()
{
    int a=3;
    fct(&a);
    printf(" \na=%d",a);
    return (0);
}
```

Test d'exécution

a = ...

# Suivi de la mémoire



# Passage par valeur Vs adresse

```
include <stdio.h>
int fonction(int x)
{
    int a=2;
    printf("\n2) a=%d, x=%d", a, x);
    x += a;
    printf("\n3) a=%d, x=%d", a, x);
    return (x);
}
int main()
{
    int a=0, x=4;
    printf("\n1) a=%d, x=%d", a, x);
    a = fonction(x);
    printf("\n4) a=%d, x=%d", a, x);
    return (0);
}
```

- 1) a=..., x=...
- 2) a=..., x=...
- 3) a=..., x=...
- 4) a=..., x=...

```
include <stdio.h>
int fonction(int *px)
{
    int a=2;
    printf("\n2) a=%d, *px=%d", a, *px);
    *px += a;
    printf("\n3) a=%d, *px=%d", a, *px);
    return (*px);
}
int main()
{
    int a=0, x=4;
    printf("\n1) a=%d, x=%d", a, x);
    a = fonction(&x);
    printf("\n4) a=%d, x=%d", a, x);
    return (0);
}
```

- 1) a=..., x=...
- 2) a=..., \*px=...
- 3) a=..., \*px=...
- 4) a=..., x=...

## Exemple : échange du contenu de 2 variables

```
#include <stdio.h>
void exchange(int * x, int * y)
{
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
void main()
{
    int a=4, b=5;
    printf("\n1) a= %d, b=%d", a, b);
    exchange(&a,&b);
    printf("\n2) a= %d, x=%d", a, b);
    return (0);
}
```

### Test d'exécution

- 1) a=..., b=...
- 2) a=..., b=...

# Ça explique

- L'utilisation du `&` dans le `scanf`

```
scanf("%d",&n) ;
```

`&n` est une adresse

- Le statut particulier des chaînes de caractères dans `scanf`

```
char chaine[50];  
scanf("%s",chaine);
```

`chaine` est un tableau de caractères, c'est à dire un pointeur (donc il contient déjà une adresse).

- Le fait que les tableaux passés en argument des fonctions étaient modifiés.

```
void init (int *Tab);
```

```
void init (int Tab[]);
```

- 1 Adresse et pointeurs
- 2 Pointeurs et tableaux
- 3 Pointeurs et passage de paramètre
- 4 Entrées/Sorties
- 5 Les chaînes de caractères



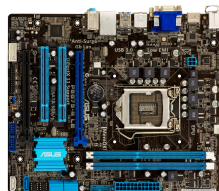


# Les entrées/sorties

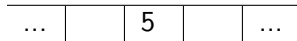
- Affichage de caractères à l'écran :  
`printf(chaine_de_caracteres , variables );`
- Lecture de caractères au clavier :  
`scanf(format , adresses );`
- Pour utiliser ces fonctions, il est nécessaire d'inclure au programme la bibliothèque `stdio.h` gérant les entrées sorties :

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

# Comment fonctionne le `scanf` ?



5



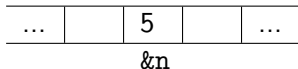
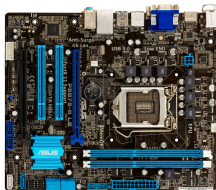
`&n`

```
scanf("%d",&n);
```

La valeur 5 est affectée à la variable `n`.

La case mémoire adressée par `&n` contient la valeur 5

# Comment fonctionne le `printf` ?



```
printf( "%d" , n );
```

Affiche la valeur de la variable `n`.  
La case mémoire adressée par `&n` est affiché.

# Les codes formats

|                                      |                                              |
|--------------------------------------|----------------------------------------------|
| <code>%c</code>                      | Caractère                                    |
| <code>%d</code>                      | Entier signé                                 |
| <code>%u</code>                      | Entier non signé                             |
| <code>%x</code> ou <code>%X</code>   | Entier en hexadécimal                        |
| <code>%o</code>                      | Entier en octal                              |
| <code>%ld</code>                     | Entier long signé                            |
| <code>%lu</code>                     | Entier long non signé                        |
| <code>%lx</code> ou <code>%lX</code> | Entier long en hexadécimal                   |
| <code>%f</code>                      | Réel simple précision avec virgule flottante |
| <code>%e</code> ou <code>%E</code>   | Réel simple précision avec exposant e ou E   |
| <code>%lf</code>                     | Réel double avec virgule flottante           |
| <code>%le</code> ou <code>%lE</code> | Réel double avec exposant e ou E             |
| <code>%Lf</code>                     | Réel long double avec virgule flottante      |
| <code>%Le</code> ou <code>%LE</code> | Réel long double avec exposant e ou E        |
| <code>%s</code>                      | Chaîne de caractères                         |
| <code>%p</code>                      | Pointeur                                     |

# Compléments sur les formats

- Autres formats d'affichage :

`\n` : Passage à la ligne.

`\t` : Tabulation.

`\0` : Fin de chaîne.

`\%` : Signe pourcentage.

- On peut précéder les codes des réels de deux entiers : `%n.p`

`n` : largeur minimale

`p` : précisions

- On peut précéder les codes des entiers d'un entier :

`%n` : largeur minimale de n.

`%0n` : largeur minimal de n avec les blancs comblés par des zéros.

# Exemples

```
int n=34 ;  
float x = 3.1 , y=1.525;  
printf( " _%d_ %3d_ %1d_ %03d_ \n" , n , n , n , n );  
printf( " _%f_ %3.2f_ \n" , x , x );  
printf( " _%f_ %5.2f_ %2.2f_ \n" , y , y , y );
```

## Test d'exécution

```
_34_ 34_34_034_  
_3.100000_3.10_  
_1.525000_ 1.52_1.52_
```

## scanf : comment ça se passe ?

```
int a; float x ;  
scanf( "%d %f ", &a, &x );
```

- 1 Si le tampon est vide, le programme s'interrompt pour laisser l'utilisateur taper au clavier.

## scanf : comment ça se passe ?

```
int a; float x ;  
scanf( "%d %f" ,&a,&x );
```

- ① Si le tampon est vide, le programme s'interrompt pour laisser l'utilisateur taper au clavier.
- ② Dès que la touche Entrée est frappée, la séquence des éléments entrés est stockée dans une zone mémoire appelée **tampon**.



## scanf : comment ça se passe ?

```
int a; float x ;  
scanf( "%d %f", &a, &x );
```

- ❶ Si le tampon est vide, le programme s'interrompt pour laisser l'utilisateur taper au clavier.
- ❷ Dès que la touche Entrée est frappée, la séquence des éléments entrés est stockée dans une zone mémoire appelée **tampon**.
- ❸ La directive **scanf** consomme la mémoire tampon en fonction du format indiqué et stocke les éléments convertis dans les variables :

## scanf : comment ça se passe ?

```
int a; float x ;  
scanf( "%d %f" ,&a,&x );
```

- ❶ Si le tampon est vide, le programme s'interrompt pour laisser l'utilisateur taper au clavier.
- ❷ Dès que la touche Entrée est frappée, la séquence des éléments entrés est stockée dans une zone mémoire appelée **tampon**.
- ❸ La directive **scanf** consomme la mémoire tampon en fonction du format indiqué et stocke les éléments convertis dans les variables :
  - ▶ Si le tampon est vide avant que le format du **scanf** ne soit entièrement converti, on retourne à l'étape 1.

## scanf : comment ça se passe ?

```
int a; float x ;
scanf("%d %f",&a,&x);
```

- ❶ Si le tampon est vide, le programme s'interrompt pour laisser l'utilisateur taper au clavier.
- ❷ Dès que la touche Entrée est frappée, la séquence des éléments entrés est stockée dans une zone mémoire appelée **tampon**.
- ❸ La directive **scanf** consomme la mémoire tampon en fonction du format indiqué et stocke les éléments convertis dans les variables :
  - ▶ Si le tampon est vide avant que le format du **scanf** ne soit entièrement converti, on retourne à l'étape 1.
  - ▶ Si le format dans le **scanf** ne correspond pas au prochain caractère de la séquence entrée au clavier, **la conversion s'interrompt et le programme continue**.

## scanf : comment ça se passe ?

```
int a; float x ;  
scanf( "%d %f" ,&a,&x );
```

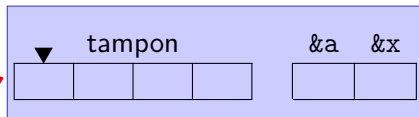
- ❶ Si le tampon est vide, le programme s'interrompt pour laisser l'utilisateur taper au clavier.
- ❷ Dès que la touche Entrée est frappée, la séquence des éléments entrés est stockée dans une zone mémoire appelée **tampon**.
- ❸ La directive **scanf** consomme la mémoire tampon en fonction du format indiqué et stocke les éléments convertis dans les variables :
  - ▶ Si le tampon est vide avant que le format du **scanf** ne soit entièrement converti, on retourne à l'étape 1.
  - ▶ Si le format dans le **scanf** ne correspond pas au prochain caractère de la séquence entrée au clavier, **la conversion s'interrompt et le programme continue**.
  - ▶ Si la conversion est entièrement terminée, le programme continue mais **il peut rester encore des éléments dans le tampon**.

# Illustration

```
int a; float x ;  
scanf( "%d %f", &a, &x );
```

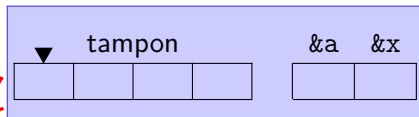
# Illustration

```
int a; float x ;  
scanf( "%d %f", &a, &x );
```



# Illustration

```
int a; float x ;  
scanf( "%d %f", &a, &x );
```



au clavier :

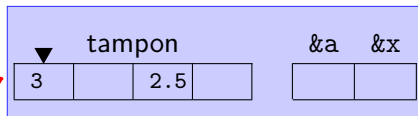
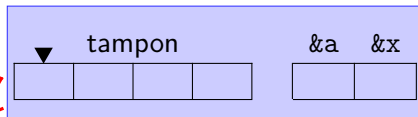
3 2.5 (Entrée)

# Illustration

```
int a; float x ;  
scanf( "%d %f", &a, &x );
```

au clavier :

3 2.5 (Entrée)



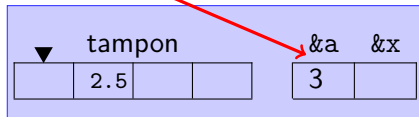
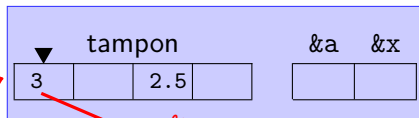
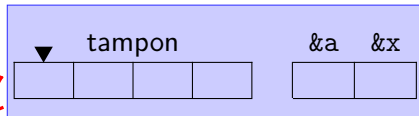


# Illustration

```
int a; float x ;  
scanf( "%d %f", &a, &x );
```

au clavier :

3 2.5 (Entrée)



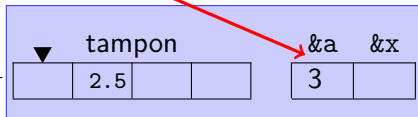
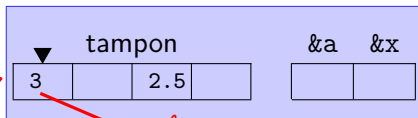
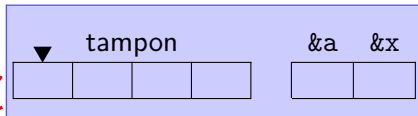
# Illustration

```
int a; float x ;  
scanf( "%d %f", &a, &x );
```

au clavier :

3 2.5 (Entrée)

L'espace est ignoré  
**sauf** si le format **%c**  
(caractère) est spécifié.



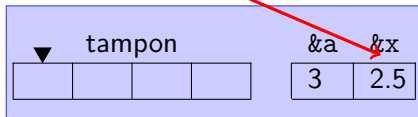
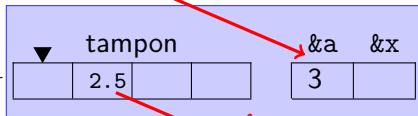
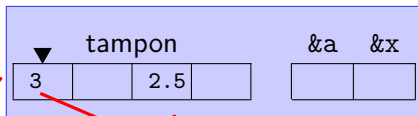
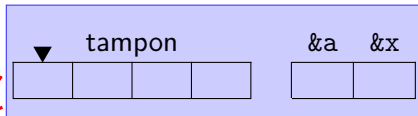
# Illustration

```
int a; float x ;  
scanf( "%d %f", &a, &x );
```

au clavier :

3 2.5 (Entrée)

L'espace est ignoré  
**sauf** si le format **%c**  
(caractère) est spécifié.



# Faire attention

- Les caractères peuvent rester dans le tampon (Exemple 1).
- Le caractère espace est pris en compte seulement dans le cas du formateur `%c` (Exemple 2).
- La conversion peut s'arrêter et ainsi des variables peuvent ne pas être attribuées (Exemple 3).

## Exemple 1

```
int a; float x ;  
printf("Ecrire a : ");  
scanf("%d",&a);  
printf("Ecrire x :");  
scanf("%f",&x);  
printf("Fin\n");
```

### Test d'exécution

Ecrire a : 32 (Entrée)  
Ecrire x : 2.1 (Entrée)  
Fin

Ce fonctionnement est normal : Le 1er `scanf` consomme l'entier 32 et le 2ème consomme le réel 2.1.

### Test d'exécution

Ecrire a : 3 2 (Entrée)  
Ecrire x :  
Fin

On a séparé (par erreur) le 3 et le 2. Le premier `scanf` consomme le 3. Le deuxième `scanf` consomme le 2 sans redonner la main à l'utilisateur.

## Solution

On peut écrire une petite procédure qui vide le tampon après chaque scanf.

```
int a ; float x ;
char c;
printf("Ecrire a : ");
scanf("%d",&a);
do {
    c = getchar();
} while (c != '\n' && c != EOF);
printf("Ecrire x : ");
scanf("%f",&x);
```

`getchar` lit un caractère `c` dans le tampon tant que (`while`) `c` est différent de `\n` ou EOF (fin du tampon).

## Exemple 2

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

un espace

Test d'exécution

A ( Entrée )

La variable c contient 'A'.

Test d'exécution

A ( Entrée )

La variable c contient ' '.

# Solution

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

L'espace indique qu'on ignore les espaces avant le caractère

un espace

Test d'exécution

A (Entrée)

La variable c contient 'A'.

Test d'exécution

A (Entrée)

La variable c contient 'A'.



## Exemple 3

```
int a ; float x ;  
scanf( "%d %f ", &a, &x );
```

Test d'exécution

3 k (Entrée)

La variable `a` a bien été initialisée à 3, mais la variable `x` n'a pas été initialisée (impossible de convertir `k` en `float`).

## Solution

```
int a ; float x ;  
int status ;  
status = scanf( "%d %f", &a, &x );  
if ( status != 2 )  
    { ...
```

### Test d'exécution

3 k (Entrée)

La variable `status` sera égale à 1 (et non comme attendu à 2). On peut donc prévoir un traitement spécifique à l'aide d'une structure `if`.

- 1 Adresse et pointeurs
- 2 Pointeurs et tableaux
- 3 Pointeurs et passage de paramètre
- 4 Entrées/Sorties
- 5 Les chaînes de caractères**

# Les chaînes de caractères

- Les chaînes de caractères sont des tableaux de caractères ;
- Chaque case du tableau contient un caractère ;
- Le dernier élément est **toujours** `'\0'` ;
- Pour stocker une chaîne de  $n$  éléments, il faut  $n+1$  emplacements (à cause du `'\0'`).

# Initialisation des chaînes de caractères

On peut utiliser les doubles quotes " pour initialiser une chaîne de caractères.

```
char tab[] = { 'i', 'n', 'f', 'o', 'r', 'm', 'a', 't', 'i', 'q', 'u', 'e', '\0' };
```

ou

```
char tab[] = "informatique";
```

# Le `printf` et les chaînes de caractères

Le formateur de la chaîne de caractères dans le `printf` est `%s`.

Exemple :

```
char tab[] = "informatique";  
printf("Nous sommes en cours d'%s\n", tab);
```

# Saisie de chaînes de caractères au clavier

Il existe deux fonctions en langage C permettant de saisir des chaînes de caractères :

- `scanf` (déjà vu) ;
- `fgets` (spécifique pour les chaînes de caractères).

# Les `scanf` et les chaînes de caractères

- Le formateur de la chaîne de caractère dans le `scanf` est `%s`.
- Pensez à déclarer un tableau de caractères d'une taille suffisante avant de l'affecter dans le `scanf`.
- Il ne faut **pas** mettre le `&`.

```
char nom[20];  
printf("\n Entrez votre nom : ");  
scanf("%s", nom);  
printf("\n Bonjour %s !\n", nom);
```



## Rappel : le caractère espace

Le caractère espace est considéré comme un délimiteur. La conversion de la mémoire tampon en chaîne de caractères s'arrête donc à l'espace et ce dernier n'est pas consommé : il reste disponible pour la prochaine lecture.

### exemple.c

```
char nom[20];  
printf("\n Entrez votre nom : ");  
scanf("%s", nom);  
printf("\n Bonjour %s !\n", nom);
```

#### test 1

```
» ./exemple  
Entrez votre nom : Dupont  
Bonjour Dupont !
```

#### test 2

```
» ./exemple  
Entrez votre nom : Dupont toto  
Bonjour Dupont !
```

## Conclusion

Avec `scanf` on ne peut pas saisir de chaîne de caractères avec des espaces.

## Autre solution : `fgets`

`fgets(string, taille, stdin);`

### exemple.c

```
char nom[20];  
printf("\n Entrez votre nom : ");  
fgets (nom,19,stdin) ;  
printf("\n Bonjour %s !\n",nom);
```

### test

```
» ./exemple  
Entrez votre nom : Dupont toto  
Bonjour Dupont toto  
!
```

- Avec `fgets`, seule la fin de ligne sert de délimiteur.
- `taille` est le nombre maximum de caractères qui peuvent être lus.
- `stdin` indique qu'on doit lire les caractères dans "l'entrée standard", c'est à dire dans ce que vous tapez au clavier. On peut aussi lire les caractères dans un fichier (cf cours n° 7)
- **ATTENTION** : la chaîne de caractère lue contiendra le retour à la ligne. Dans l'exemple précédent `nom='D','u','p','o','n','t',' ','t','o','t','o','\n','\0'`

# Égalité de chaînes de caractères

Pour vérifier l'égalité de deux chaînes de caractères, il faut vérifier un par un tous les caractères (comme pour un autre tableau).

## Exemple :

Demander à l'utilisateur d'entrer pluie ou soleil, et afficher "Prenez un parapluie" si l'utilisateur a mis pluie.

# La bibliothèque `string.h`

exemple.c

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

Contient des fonctions standards pour gérer les chaînes de caractères

Exemples :

| Prototype de la fonction                       | Utilisation                                                              |
|------------------------------------------------|--------------------------------------------------------------------------|
| <code>int strcmp(char s1[], char s2[]);</code> | Renvoie 0 si les chaînes <code>s1</code> et <code>s2</code> sont égales. |
| <code>int strlen(char s[]);</code>             | Renvoie la taille de <code>s</code>                                      |

Pour le détail de toutes les fonctions :

```
» man string
```