

# Mémoire et pointeurs

# Mémoire

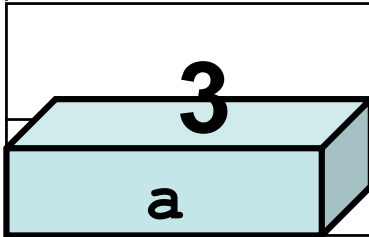
La mémoire d'un ordinateur est composée d'un grand nombre **d'octets** (une valeur entre 0 et 255) :

adresse	octet
0	
1	
2	
3	
...	

Chaque octet est repéré par une **adresse**, qui est *un nombre entier*.

Les variables sont stockées en mémoire. Par exemple:

```
int a = 3;
```

adresse	contenu
24996	
25000	
25004	
25008	?
25012	?
25016	?
25020	

# L'opérateur &

On peut obtenir l'adresse d'une variable grâce à l'opérateur & :

Par exemple :

```
int a = 3;  
printf(«%d\n», a);  
printf(«%u\n», (unsigned int) &a);
```

va afficher pour cet exemple:

```
3  
25004
```

# Stocker une adresse dans une variable

En C et en C++, on peut définir des variables pour qu'elles contiennent des adresses.

Pour déclarer une telle variable, Il faut ajouter une étoile \* entre le type et le nom de la variable:

```
int * p;
```

p est appelé un *pointeur*.

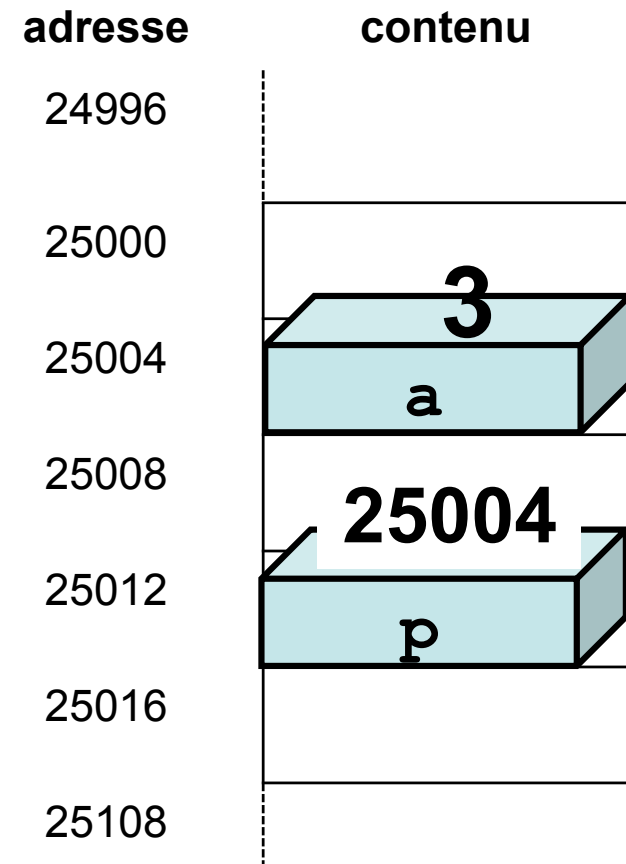
# Stocker une adresse dans une variable

Si on fait:

```
int * p;
```

```
p = &a;
```

p contient l'adresse de a.



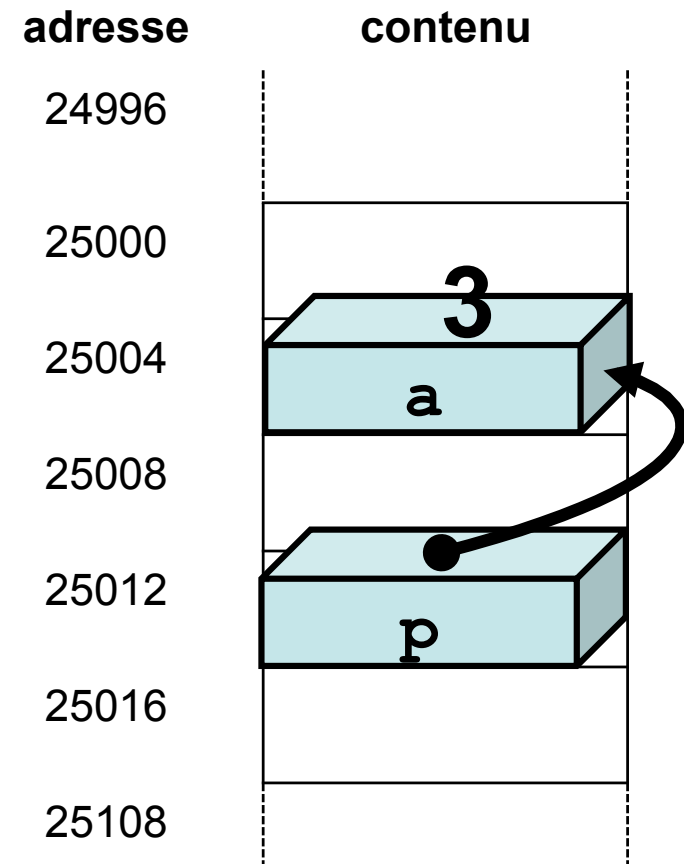
# Représentation graphique (1)

$p$  contient l'adresse de  $a$ .

On dit que  $p$  pointe sur  $a$ .

La valeur de l'adresse de  $a$  n'est pas importante en elle-même, ce qui est important est que  $p$  pointe sur  $a$ .

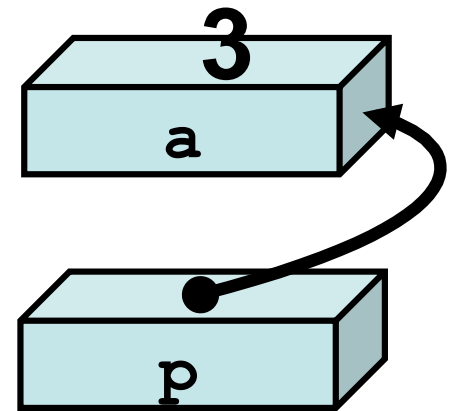
Ceci est généralement représenté par une flèche allant de  $p$  à  $a$ .



# Représentation graphique (2)

On n'a généralement pas besoin de la représentation de la mémoire, et on peut ne garder que la représentation des variables.

```
int a = 3;  
int * p;  
p = &a;
```





# L'opérateur \*

L'opérateur \* permet d'accéder à la valeur stockée à une adresse:

```
printf( «%d\n» , *p ) ;
```

affiche 3.

# Modifier la valeur pointée par un pointeur

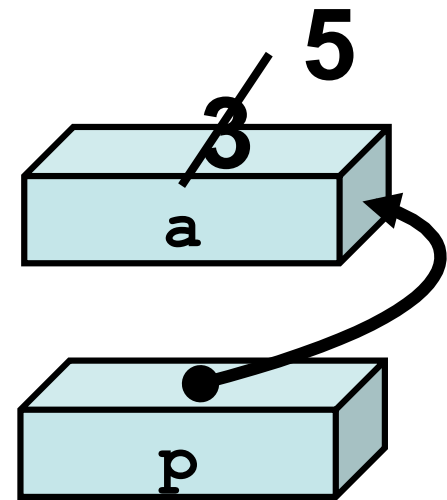
On peut aussi utiliser l'opérateur pour modifier une valeur pointée par un pointeur:

```
int a = 3;
```

```
int * p;
```

```
p = &a;
```

```
*p = 5;
```



# Attention à ne pas confondre

- l'étoile utilisée pour déclarer un pointeur:

`int *p;`

Même symbole (\*) mais  
signification différente.

- et l'étoile pour obtenir la valeur pointée par le pointeur:

`*p = 5;`

# Résumé

Un pointeur est une variable. On déclare un pointeur en mettant une étoile (\*) entre le type et le nom du pointeur:

```
int * p;
```

Un pointeur contient une adresse.

On peut obtenir l'adresse d'une variable avec l'**opérateur &**:

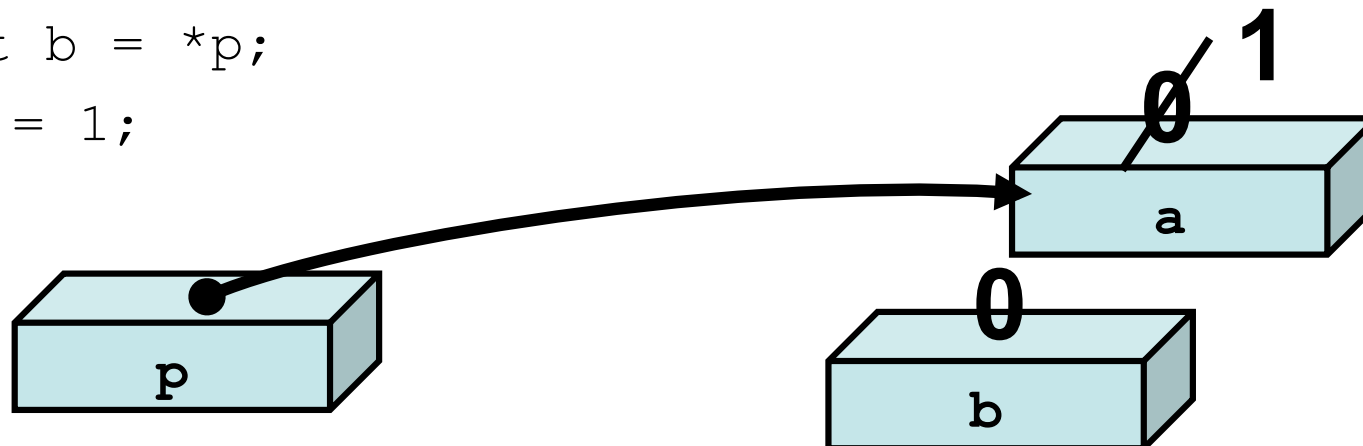
```
int a = 0;
```

```
p = &a;
```

L'**opérateur \*** permet d'accéder à la mémoire pointée par un pointeur:

```
int b = *p;
```

```
*p = 1;
```



# Déclaration d'un pointeur

*Un pointeur est une variable:*

On peut déclarer plusieurs pointeurs sur une même ligne, et initialiser un pointeur lors de sa déclaration. Exemples:

```
float * p1 = &a;  
int * p2, * p3;
```

**Attention:** pour déclarer plusieurs pointeurs sur une même ligne, il faut *répéter* l'étoile. Si on fait:

```
int * p2, p3; // !!
```

`p2` est de type pointeur sur `int`, mais `p3` est une variable classique, de type `int`.

# Le pointeur NULL ou 0

On utilise parfois la valeur 0, ou la constante `NULL` (de type pointeur, qui vaut 0) pour initialiser un pointeur:

```
int * p = 0;
```

ou

```
int * p = NULL;
```

L'adresse 0 n'est jamais utilisée pour stocker des variables, elle correspond donc toujours à une adresse invalide: Si on essaie d'accéder à la valeur pointée par `p` en faisant par exemple:

```
int a = *p;
```

```
*p = 0;
```

on provoque un:

```
Segmentation fault
```

L'intérêt du pointeur `NULL` est par exemple de signaler qu'une fonction n'a pas pu fonctionner correctement, ou qu'un pointeur ne contient pas une adresse valide.

# Remarques

- On peut voir l'opérateur `*` comme l'inverse de l'opérateur `&`:

```
b = *(&a);
```

est équivalent à

```
b = a;
```

- On peut mettre zéro, un ou plusieurs espaces avant et après l'étoile et le *et commercial*:

```
int *pa = & a;
```

```
b = * pa;
```

# Affectation de pointeurs

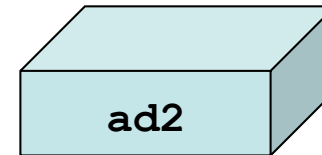
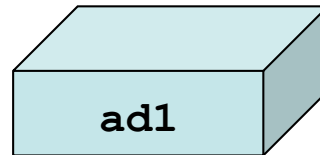
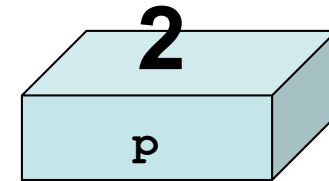
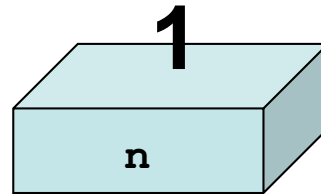
On peut affecter à un pointeur la valeur d'un autre pointeur de même type.

## Exemple:

```
int n = 1, p = 2;  
int * ad1, * ad2;
```

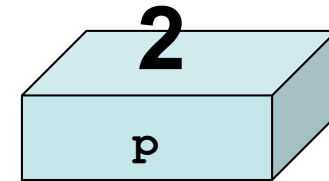
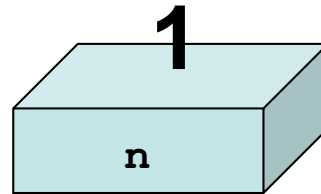
Que font les instructions:

```
ad1 = &n;  
ad2 = &p;  
*ad1 = *ad2 + 3;  
ad1 = ad2;  
*ad1 = *ad2 + 5;
```





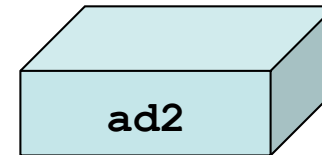
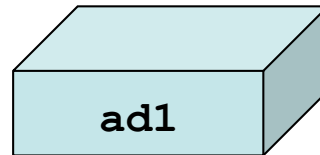
# Pas-à-pas



```
int n = 1, p = 2;  
int * ad1, * ad2;
```



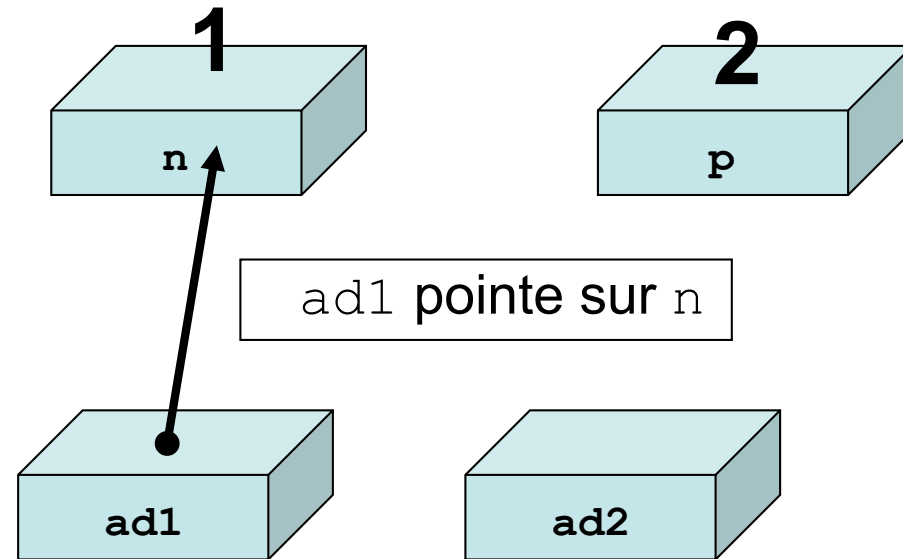
```
ad1 = &n;  
ad2 = &p;  
*ad1 = *ad2 + 3;  
ad1 = ad2;  
*ad1 = *ad2 + 5;
```



# Pas-à-pas

```
int n = 1, p = 2;  
int * ad1, * ad2;
```

→ `ad1 = &n;`  
`ad2 = &p;`  
`*ad1 = *ad2 + 3;`  
`ad1 = ad2;`  
`*ad1 = *ad2 + 5;`



# Pas-à-pas

```
int n = 1, p = 2;  
int * ad1, * ad2;
```

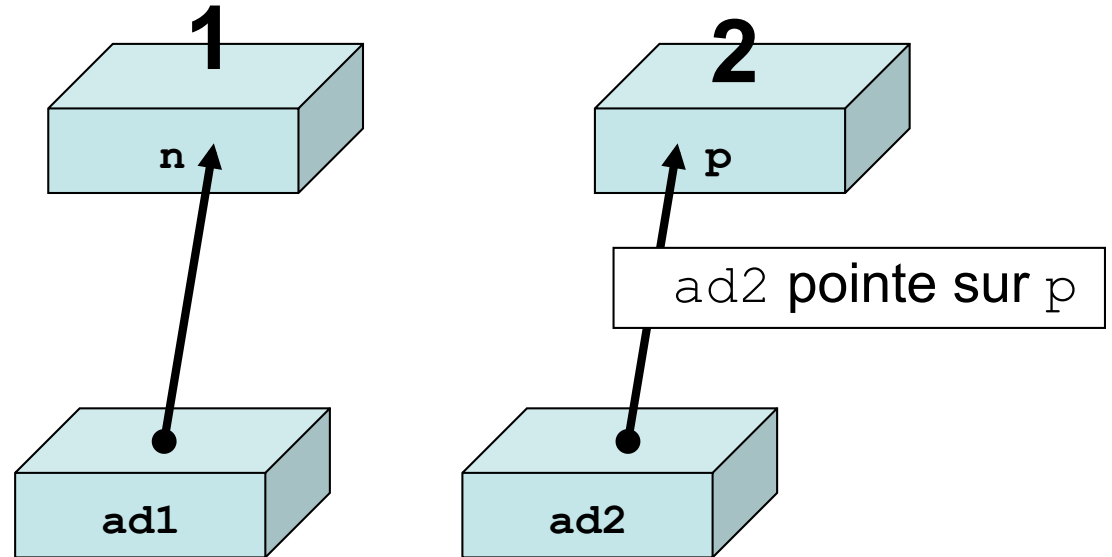
```
ad1 = &n;
```

```
→ ad2 = &p;
```

```
*ad1 = *ad2 + 3;
```

```
ad1 = ad2;
```

```
*ad1 = *ad2 + 5;
```



# Pas-à-pas

```
int n = 1, p = 2;  
int * ad1, * ad2;
```

```
ad1 = &n;
```

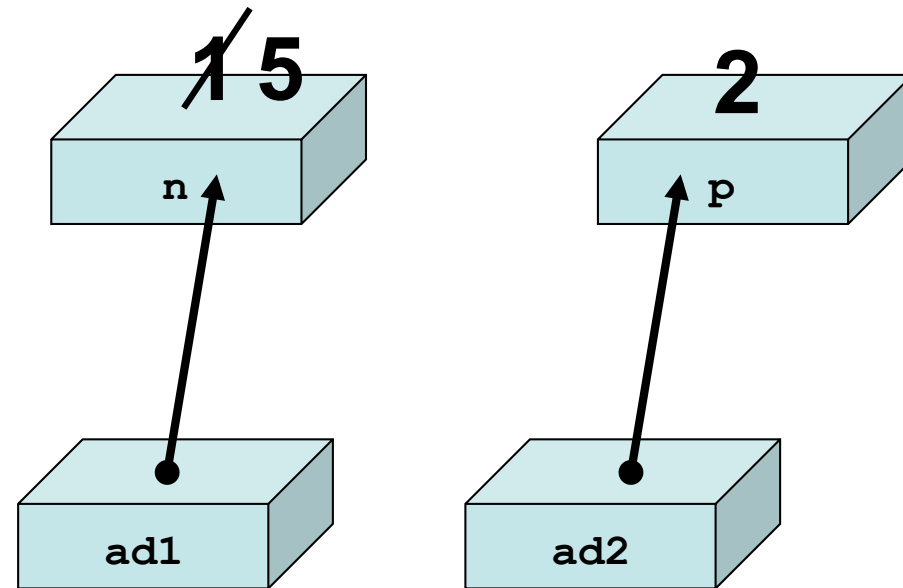
```
ad2 = &p;
```

→ 

```
*ad1 = *ad2 + 3;
```

```
ad1 = ad2;
```

```
*ad1 = *ad2 + 5;
```



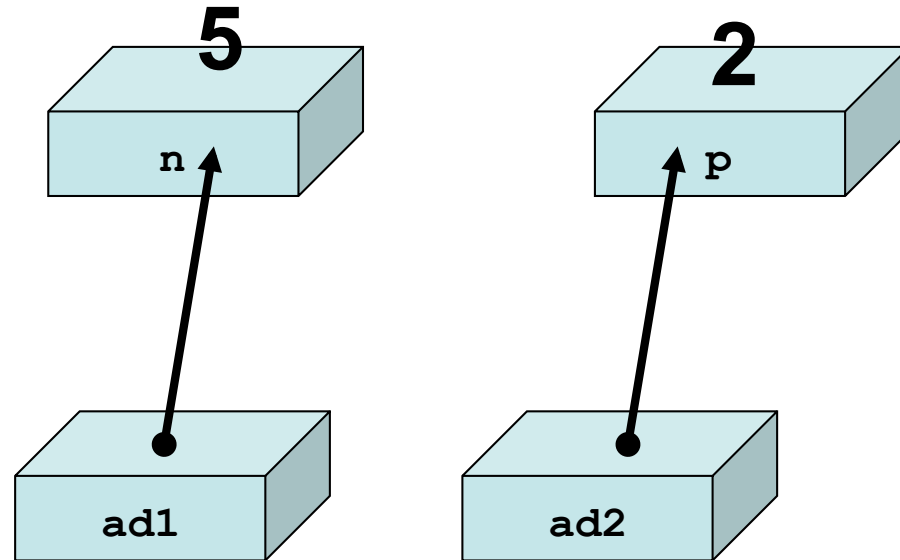
exactement comme  $n = p + 3$ ;  
`n` contient maintenant 5

# Pas-à-pas

```
int n = 1, p = 2;  
int * ad1, * ad2;
```

```
ad1 = &n;  
ad2 = &p;  
*ad1 = *ad2 + 3;
```

```
→ ad1 = ad2;  
*ad1 = *ad2 + 5;
```

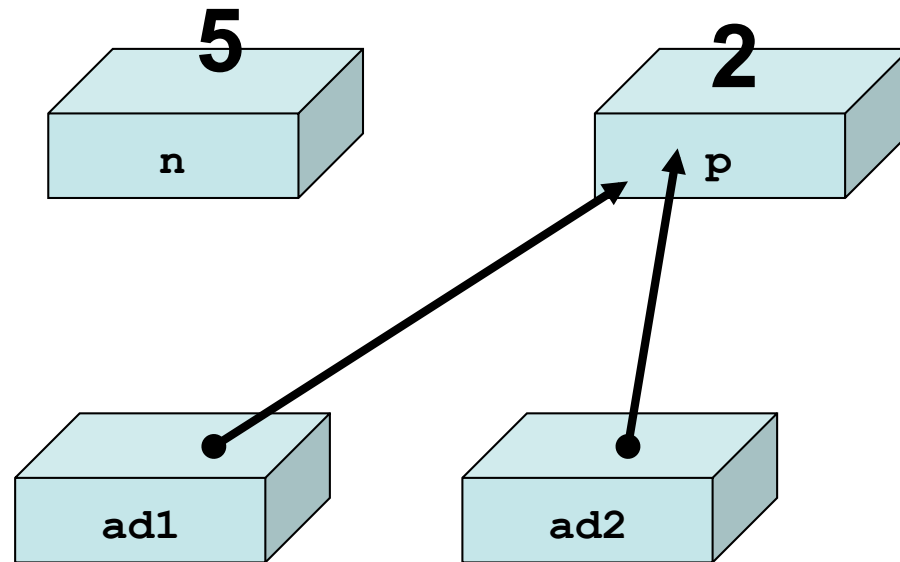


# Pas-à-pas

```
int n = 1, p = 2;  
int * ad1, * ad2;
```

```
ad1 = &n;  
ad2 = &p;  
*ad1 = *ad2 + 3;
```

```
→ ad1 = ad2;  
*ad1 = *ad2 + 5;
```



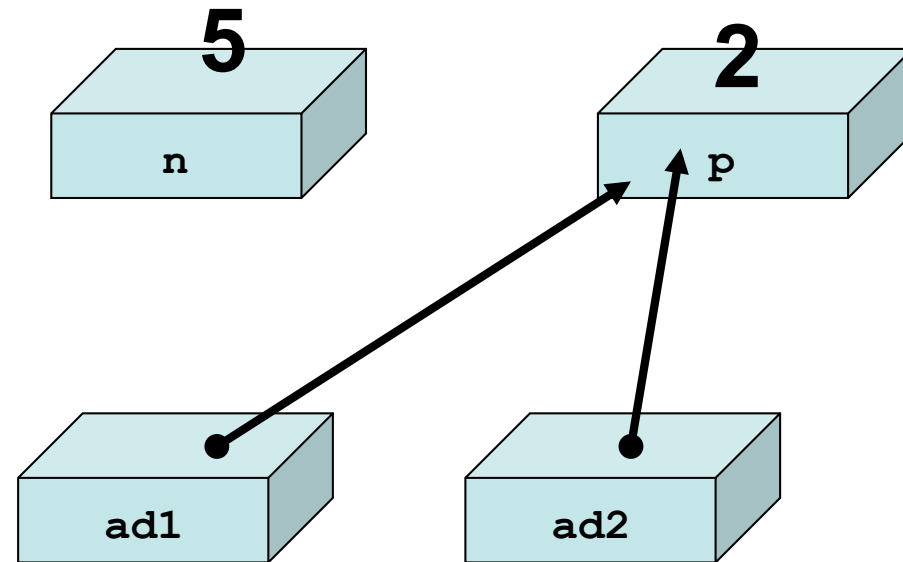
ad1 et ad2 pointent maintenant tous les deux sur p.

# Pas-à-pas

```
int n = 1, p = 2;  
int * ad1, * ad2;
```

```
ad1 = &n;  
ad2 = &p;  
*ad1 = *ad2 + 3;  
ad1 = ad2;
```

→ `*ad1 = *ad2 + 5;`



exactement comme `p = p + 5;`  
`p` contient donc maintenant 7

# **Première application**

Fonction modifiant une variable  
passée en paramètre



# Rappel

Si on exécute:

```
void mis_a_zero(int a)
{
    a = 0;
}
```

```
int main(void)
{
    int n = 10;
    mis_a_zero(n);
    printf(«apres appel: n = %d\n», n);
}
```

on obtient:

apres appel: n = 10

**La fonction `mis_a_zero` ne peut pas changer la valeur de `n` !**

# Une fonction modifiant une variable passée en paramètre

Pour que la fonction puisse modifier la variable `n`, il faut passer en paramètre l'adresse de la variable plutôt que sa valeur:

```
void mis_a_zero(int * pa)
{
    *pa = 0;
}

int main(void)
{
    int n = 10;
    mis_a_zero(&n);
    printf(«apres appel: n = %d\n», n);
}
```

on obtient cette fois:

apres appel: n = 0

# Recette

1. Ajouter une étoile (\*) quand la fonction doit modifier la variable passée en paramètre:

```
void mis_a_zero(int * pa)
```

`pa` est maintenant un pointeur sur la variable passée en paramètre.

2. Dans le code de la fonction, on peut accéder à la variable passée en paramètre en ajoutant une étoile devant le nom du paramètre:

```
*pa = 0;
```

3. La fonction attend maintenant un paramètre de type pointeur. A l'appel de la fonction, il faut ajouter un `&` commercial devant la variable passée en paramètre:

```
mis_a_zero(&n);
```

MAIS ATTENTION...

# ...Attention à l'appel

Supposons que l'on ait une fonction qui ajoute 1 à une variable passée en paramètre:

```
void ajoutel(int * pn)
{
    *pn = *pn + 1;
}
```

et qu'on veuille utiliser cette fonction pour écrire une fonction qui ajoute 2 à une variable passée en paramètre.

# ...Attention à l'appel

La recette précédente ne marche pas dans ce cas.

La fonction `ajoute2` appelle deux fois la fonction `ajoute1` pour ajouter 2 à la variable passée en paramètre.

Mais il faut faire attention à l'expression exacte du paramètre. La version ci-dessous ne marche pas:

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    ajoute1(&pm);
    ajoute1(&pm);
}
```

`pm` contient déjà l'adresse de la variable à modifier.

`&pm` correspond à l'adresse de `pm` !

```
...
int a = 0;

ajoute2(&a);
```

# ...Attention à l'appel

Dans ce cas, il faut faire:

```
void ajoutel(int * pn)
{
    *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    ajoutel(pm);
    ajoutel(pm);
}
```

← pm contient déjà l'adresse de la variable à modifier.

```
...
int a = 0;

ajoute2(&a);
```

# Pas-à-pas (version correcte)

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    ajoute1(pm);
    ajoute1(pm);
}
```

```
...
int a = 0;
```



```
ajoute2(&a);
```

**adresse**

**contenu**

24996

25000

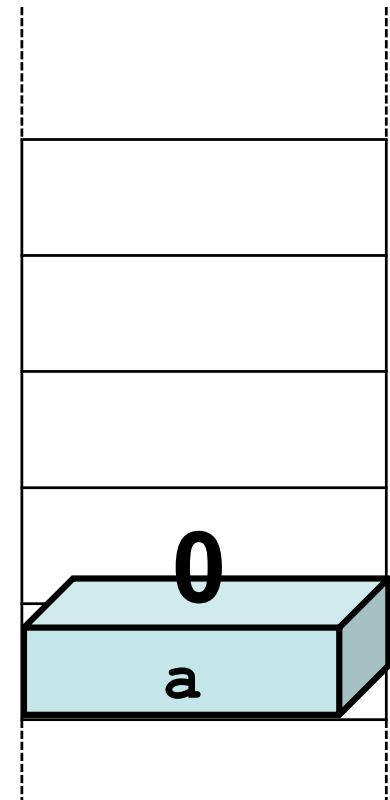
25004

25008

25012

25016

25108



# Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    ajoute1(pm);
    ajoute1(pm);
}
```

```
...
int a = 0;
```

➔ `ajoute2(&a);`

adresse	contenu
24996	
25000	
25004	
25008	
25012	
25016	0 a
25108	



# Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

➔

```
void ajoute2(int * pm)
{
    ajoute1(pm);
    ajoute1(pm);
}
```

```
...
int a = 0;
```

➡

```
ajoute2(&a);
```

adresse	contenu
24996	
25000	
25004	
25008	
25012	
25016	0
25108	a

# Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

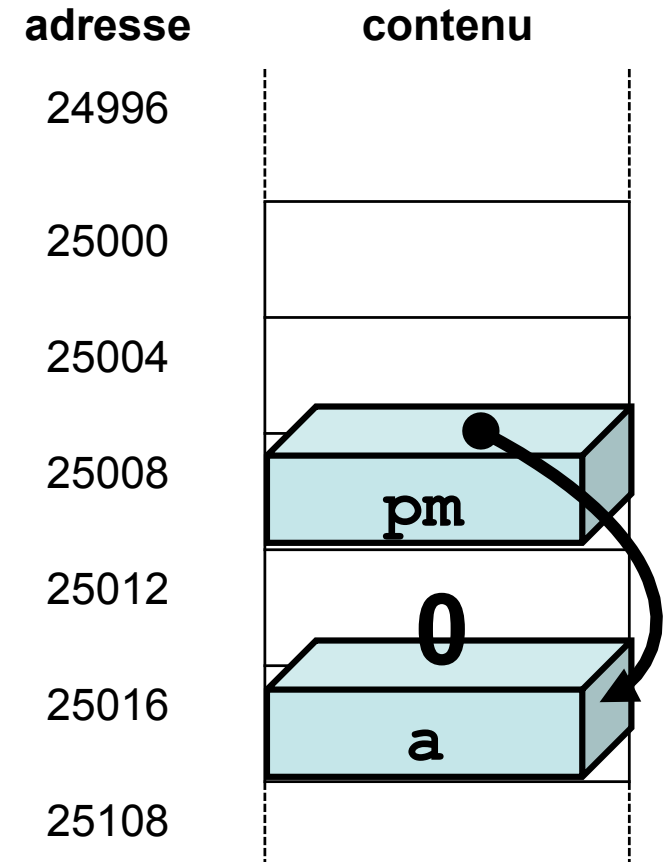
➔

```
void ajoute2(int * pm)
{
    ajoute1(pm);
    ajoute1(pm);
}
```

```
...
int a = 0;
```

➡

```
ajoute2(&a);
```



# Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    ajoute1(pm);
    ajoute1(pm);
}
```

```
...
int a = 0;
```

→ `ajoute2(&a);`

**adresse**

**contenu**

24996

25000

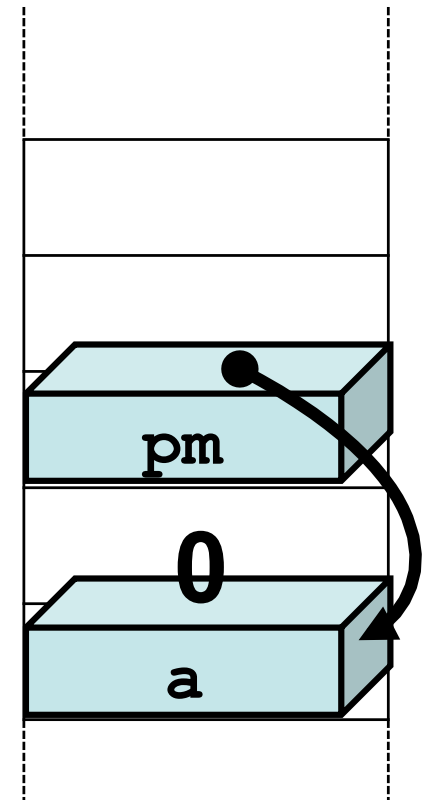
25004

25008

25012

25016

25108



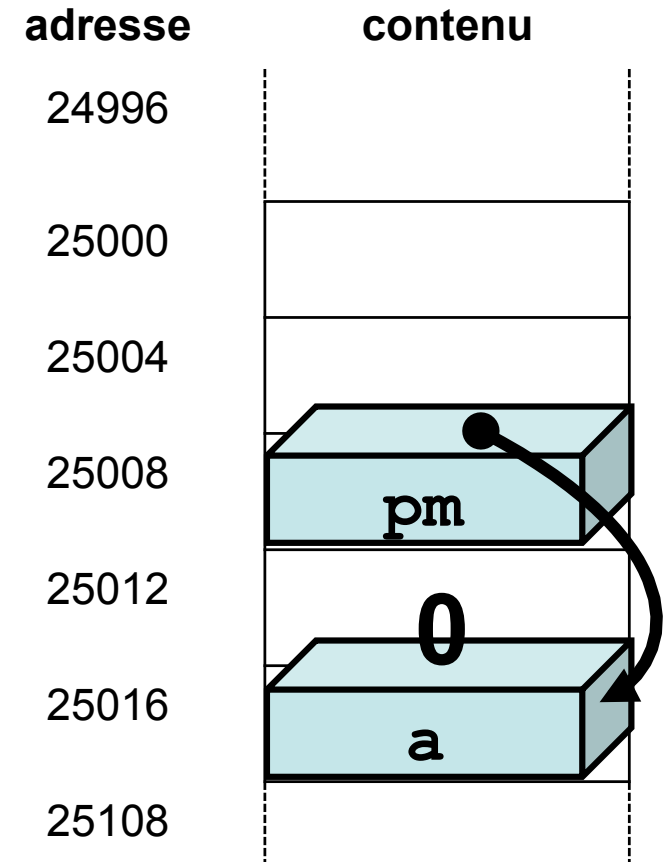
# Pas-à-pas

```
→ void ajoute1(int * pn)
{
    *pn = *pn + 1;
}

void ajoute2(int * pm)
{
    → ajoute1(pm);
    ajoute1(pm);
}

...
int a = 0;

→ ajoute2(&a);
```



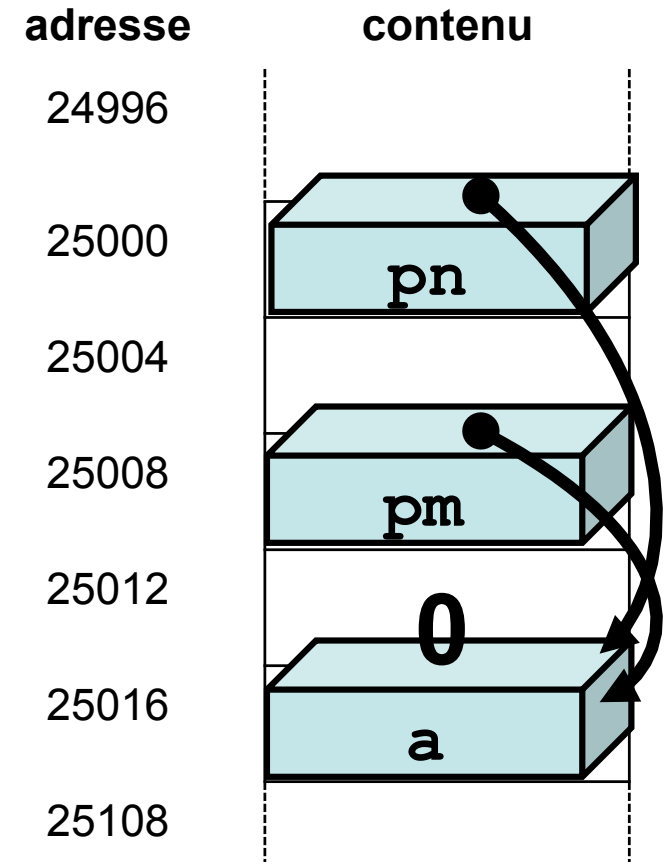
# Pas-à-pas

```
→ void ajoute1(int * pn)
{
    *pn = *pn + 1;
}

void ajoute2(int * pm)
{
    → ajoute1(pm);
    ajoute1(pm);
}

...
int a = 0;

→ ajoute2(&a);
```



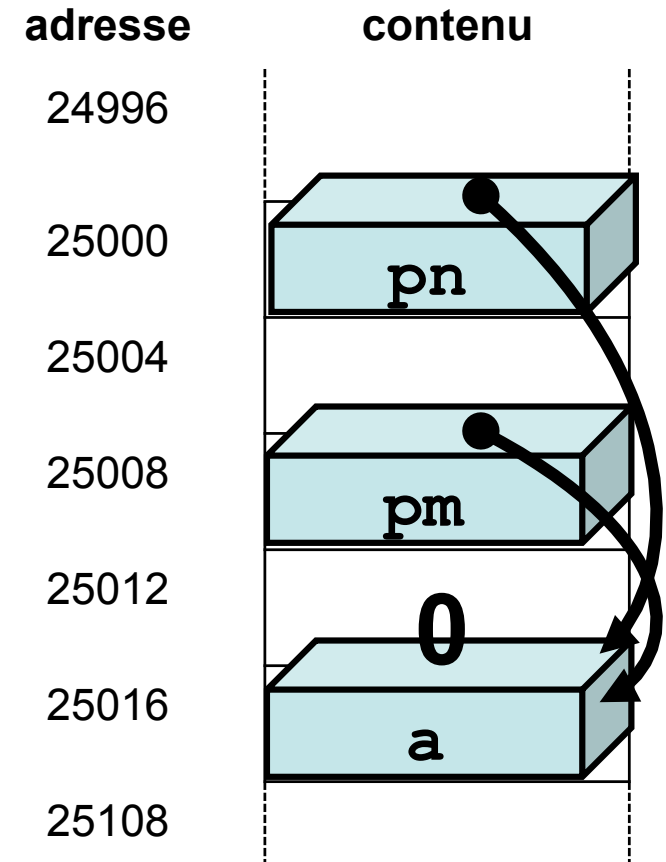
# Pas-à-pas

```
void ajoute1(int * pn)
{
    → *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    → ajoute1(pm);
    ajoute1(pm);
}
```

```
...
int a = 0;
```

```
→ ajoute2(&a);
```



# Pas-à-pas

```
void ajoute1(int * pn)
{
    → *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    → ajoute1(pm);
    ajoute1(pm);
}
```

```
...
int a = 0;
```

```
→ ajoute2(&a);
```

adresse

contenu

24996

25000

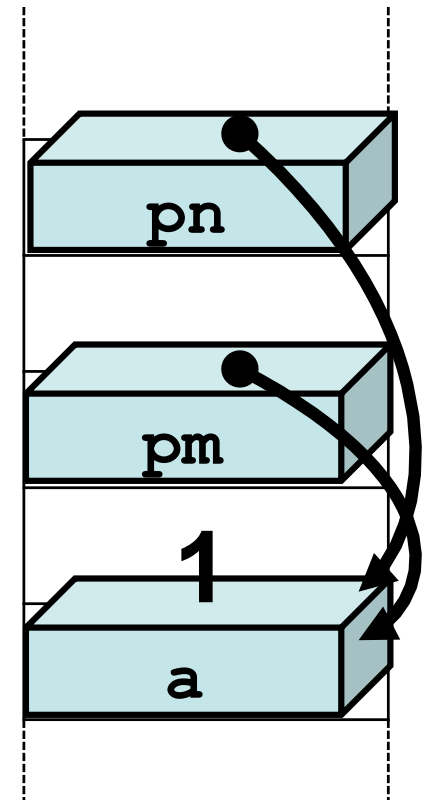
25004

25008

25012

25016

25108



# Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    ajoute1(pm);
    ajoute1(pm);
}
```

```
...
int a = 0;
```

→ `ajoute2(&a);`

**adresse**

**contenu**

24996

25000

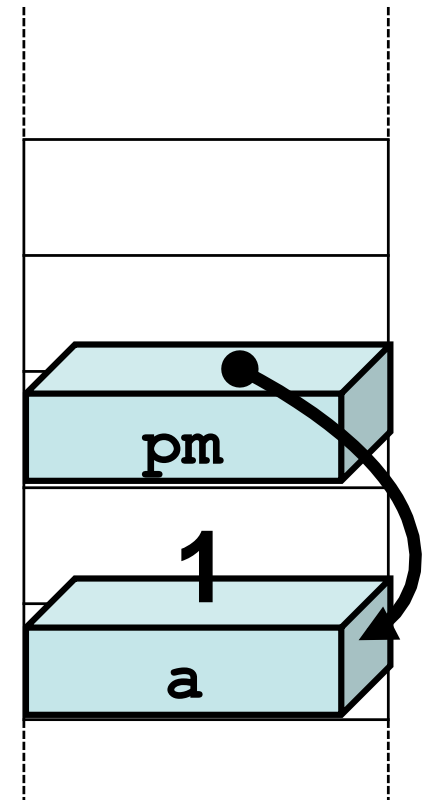
25004

25008

25012

25016

25108





# Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    ajoute1(pm);
    ajoute1(pm);
}
```



```
...
int a = 0;
```

→ `ajoute2(&a);`

**adresse**

**contenu**

24996

25000

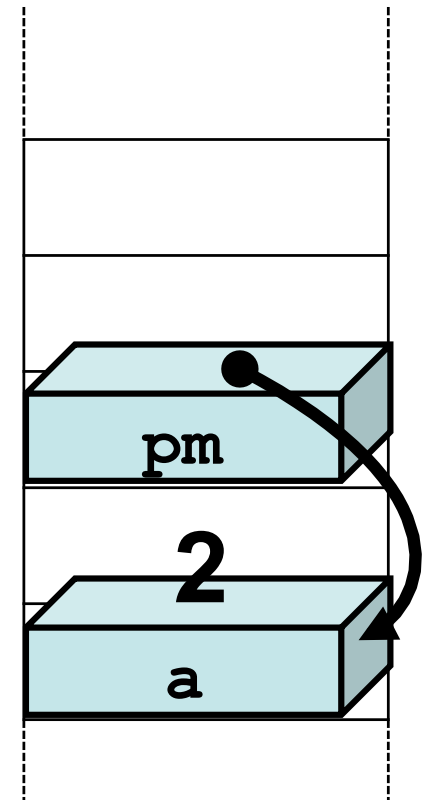
25004

25008

25012

25016

25108



# Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    ajoute1(pm);
    ajoute1(pm);
}
```

```
...
int a = 0;
```

```
ajoute2(&a);
```



**adresse**

**contenu**

24996

25000

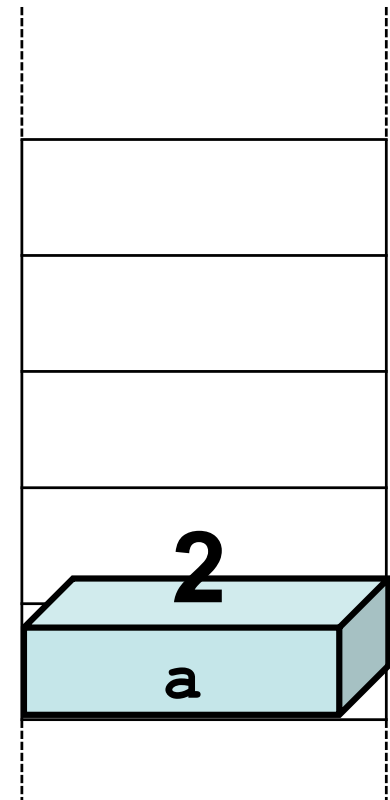
25004

25008

25012

25016

25108



# Fonction échange

Pour échanger la valeur de deux variables `a` et `b`, on peut faire:

```
int tmp;  
tmp = a;  
a = b;  
b = tmp;
```

On veut écrire une fonction qui exécute ce code pour inverser deux variables passées en paramètre.

On doit donc utiliser des pointeurs pour ces paramètres.

La fonction a donc comme en-tête:

```
void échange(int * pa, int * pb)
```

# Fonction échange

Pour échanger la valeur de deux variables `a` et `b`, on peut faire:

```
int tmp;  
tmp = a;  
a = b;  
b = tmp;
```

La fonction `échange` s'écrit:

```
void échange(int * pa, int * pb)  
{  
    int tmp;  
    tmp = *pa;  
    *pa = *pb;  
    *pb = tmp;  
}
```

et s'appelle:

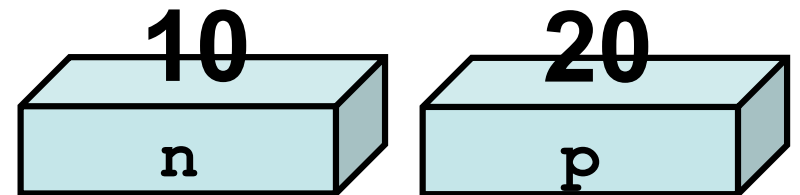
```
int n = 10, p = 20;  
échange(&n, &p);
```

# Pas-à-pas

```
void exchange(int * pa, int * pb)
{
    int tmp;
    tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}
```

...

→ `int n = 10, p = 20;`  
`exchange(&n, &p);`

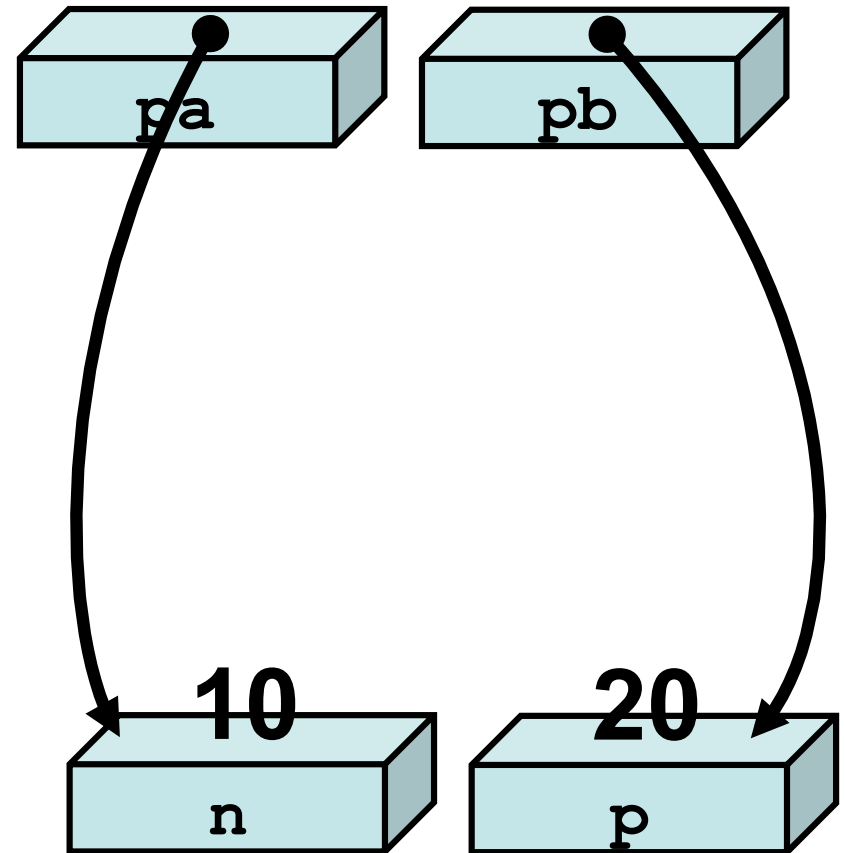


# Pas-à-pas

→ `void exchange(int * pa, int * pb)`  
{  
 int tmp;  
 tmp = \*pa;  
 \*pa = \*pb;  
 \*pb = tmp;  
}

...

→ `int n = 10, p = 20;`  
`exchange(&n, &p);`

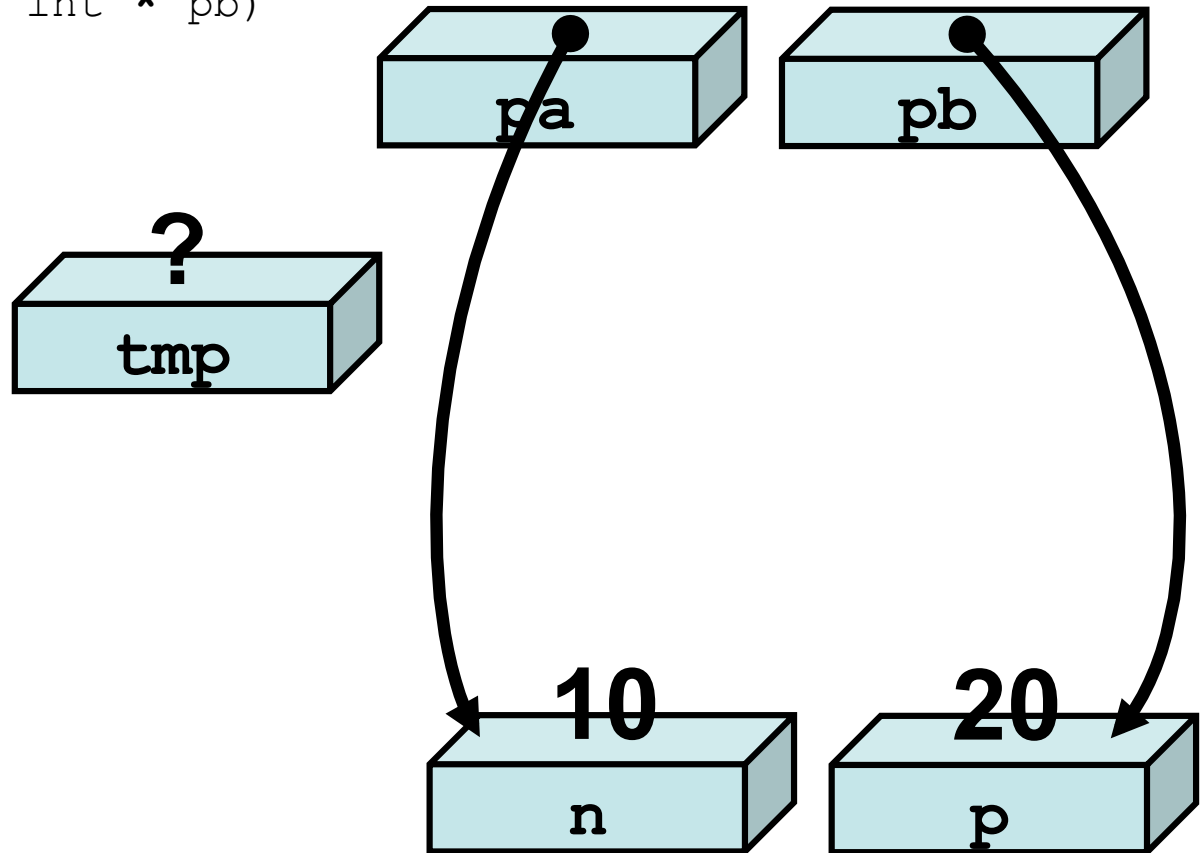


# Pas-à-pas

```
void exchange(int * pa, int * pb)
{
    → int tmp;
      tmp = *pa;
      *pa = *pb;
      *pb = tmp;
}
```

...

```
→ int n = 10, p = 20;
   exchange(&n, &p);
```

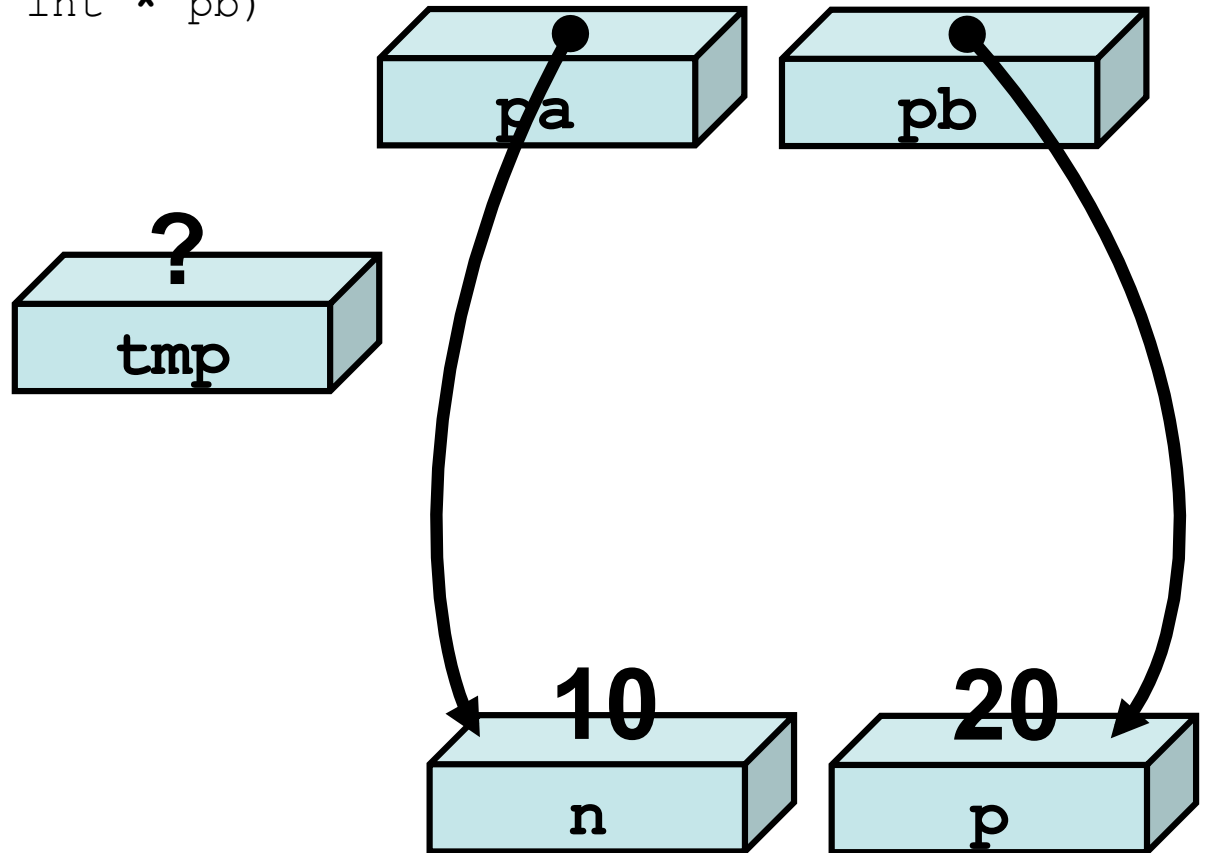


# Pas-à-pas

```
void exchange(int * pa, int * pb)
{
    int tmp;
    → tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}
```

...

```
→ int n = 10, p = 20;
   exchange(&n, &p);
```



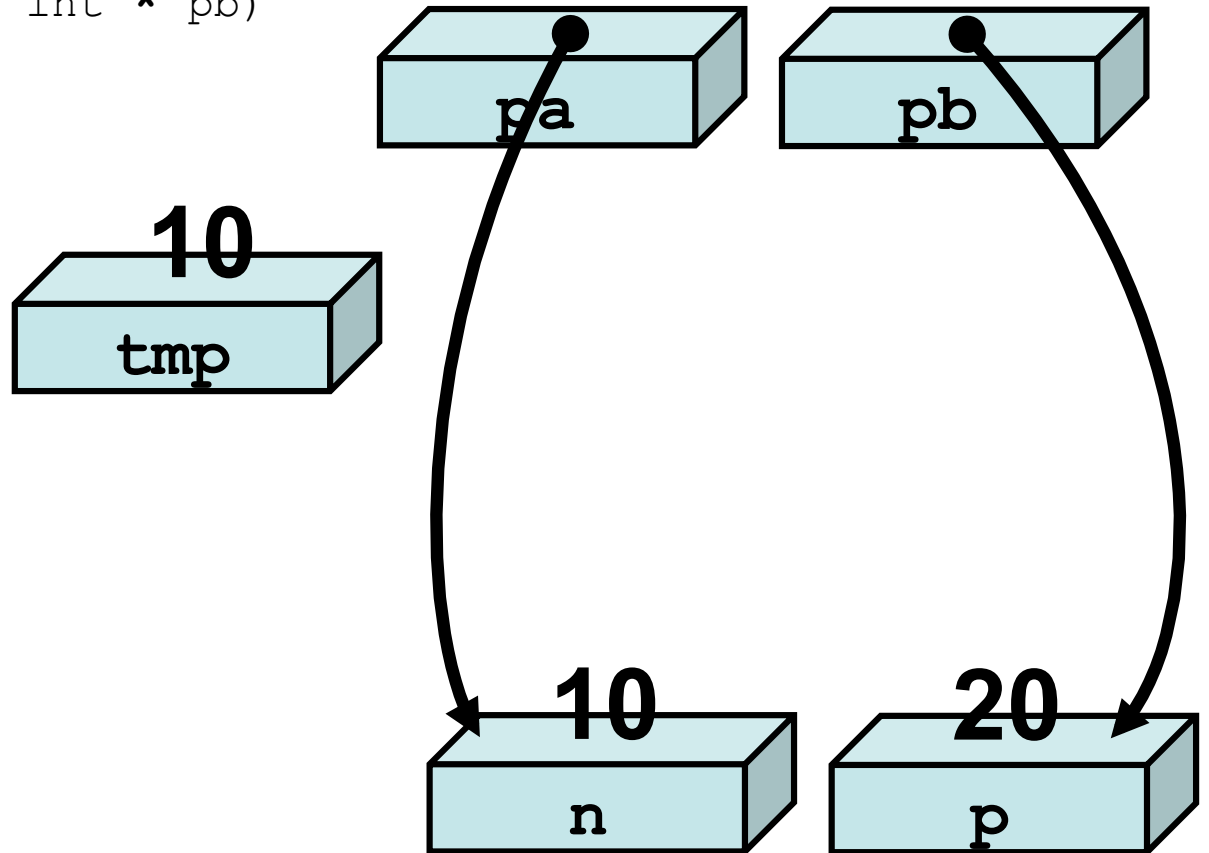


# Pas-à-pas

```
void exchange(int * pa, int * pb)
{
    int tmp;
    → tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}
```

...

```
→ int n = 10, p = 20;
   exchange(&n, &p);
```

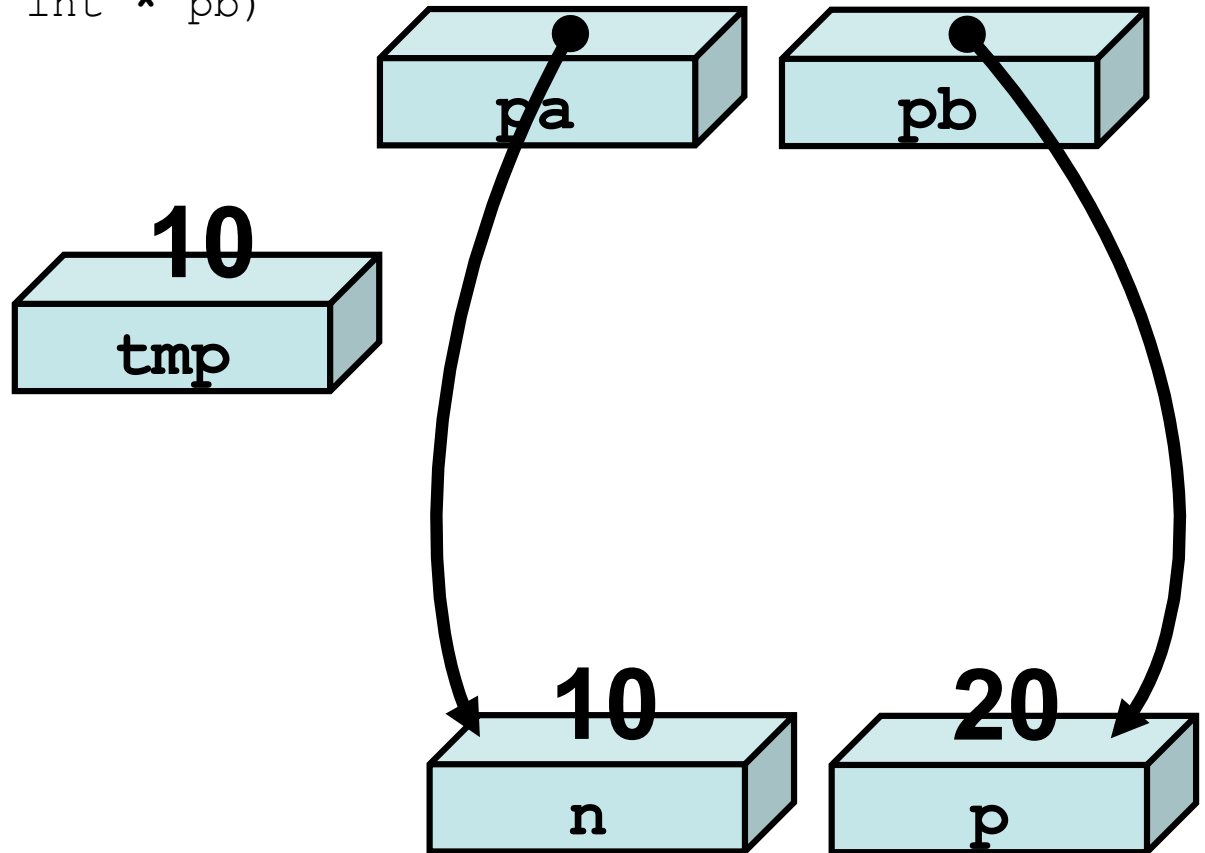


# Pas-à-pas

```
void exchange(int * pa, int * pb)
{
    int tmp;
    tmp = *pa;
    → *pa = *pb;
    *pb = tmp;
}
```

...

→ int n = 10, p = 20;  
exchange(&n, &p);

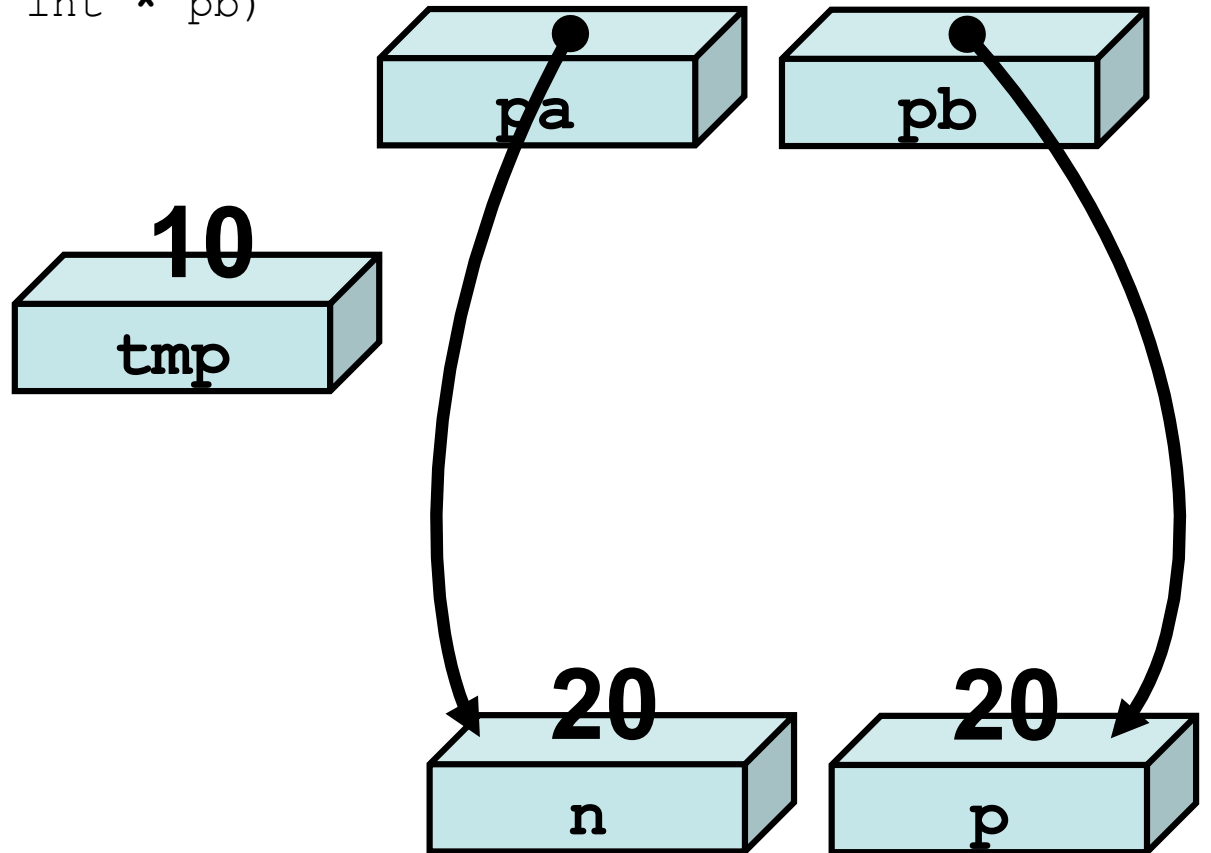


# Pas-à-pas

```
void exchange(int * pa, int * pb)
{
    int tmp;
    tmp = *pa;
    → *pa = *pb;
    *pb = tmp;
}
```

...

→ int n = 10, p = 20;  
exchange(&n, &p);



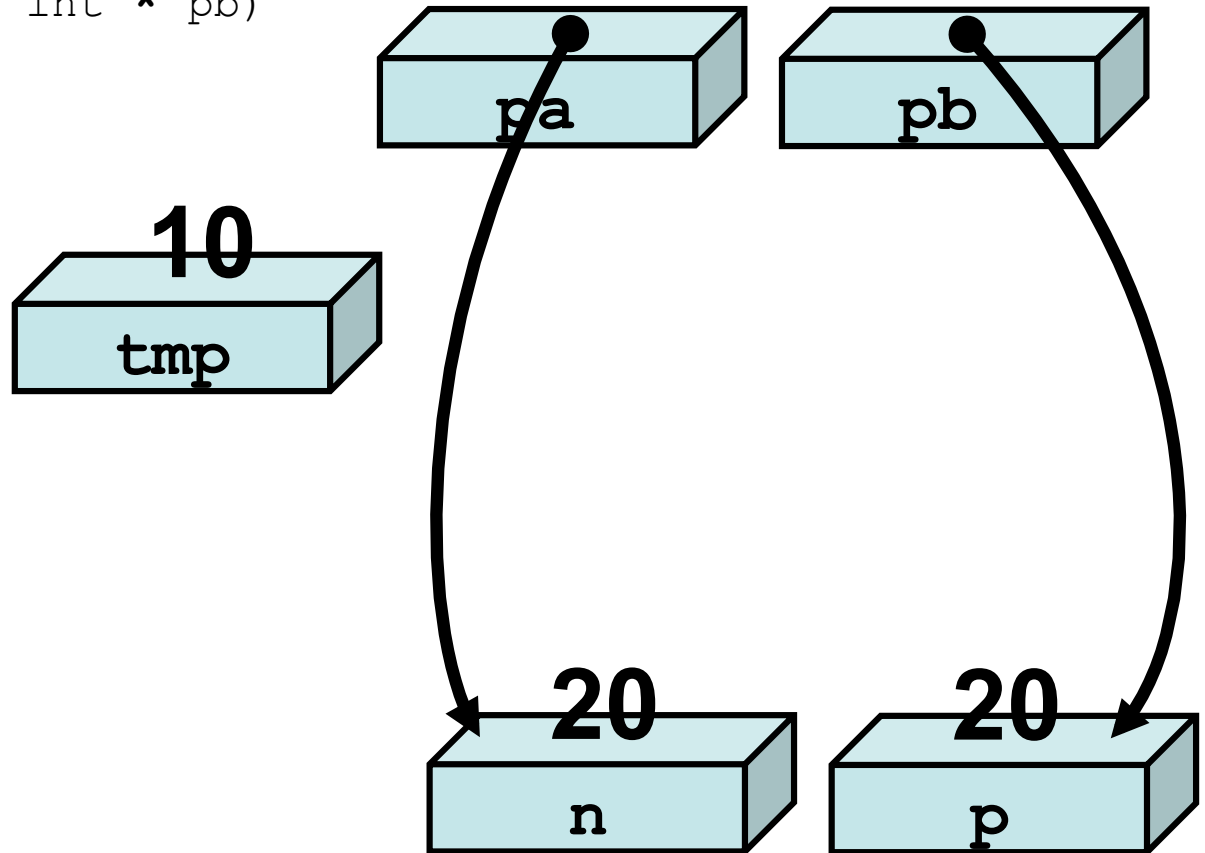
# Pas-à-pas

```
void exchange(int * pa, int * pb)
{
    int tmp;
    tmp = *pa;
    *pa = *pb;
    → *pb = tmp;
}
```

...

→ 

```
int n = 10, p = 20;
exchange(&n, &p);
```



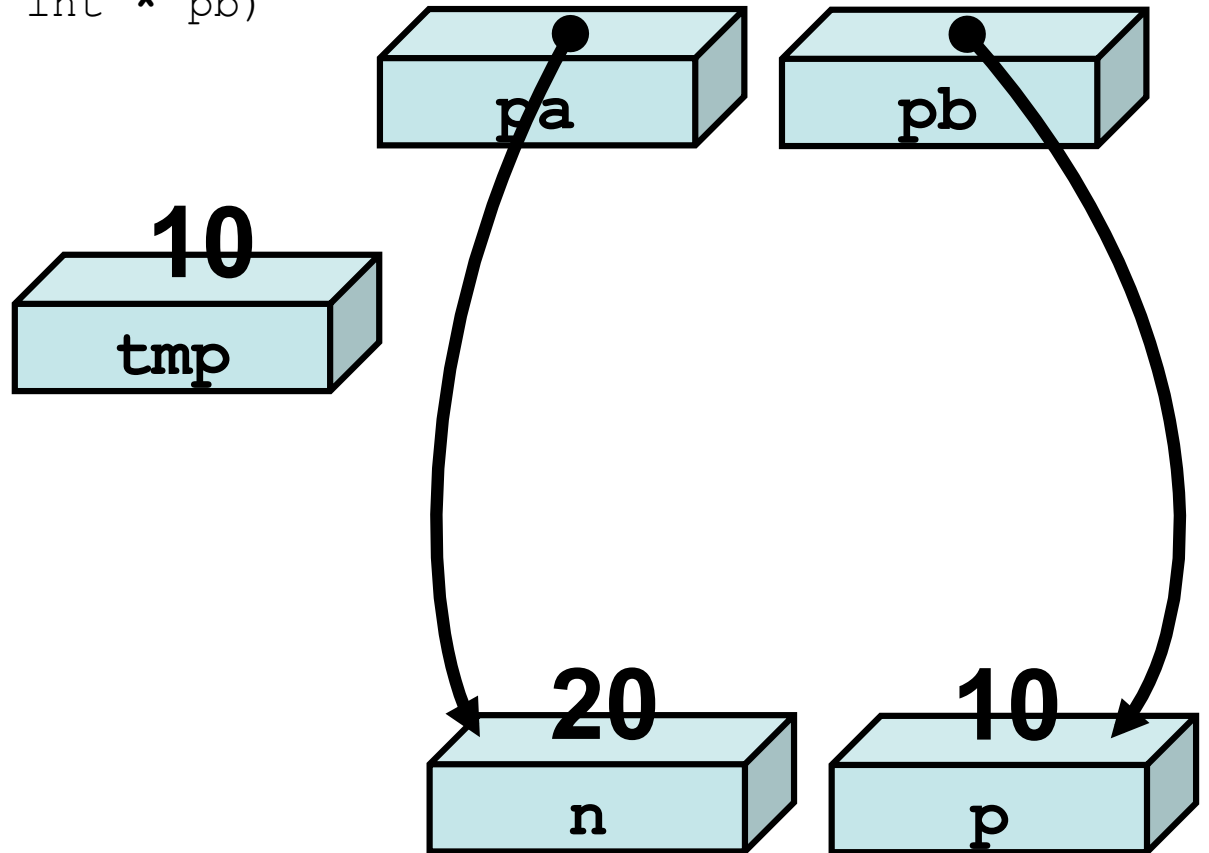
# Pas-à-pas

```
void exchange(int * pa, int * pb)
{
    int tmp;
    tmp = *pa;
    *pa = *pb;
    → *pb = tmp;
}
```

...

→ 

```
int n = 10, p = 20;
exchange(&n, &p);
```

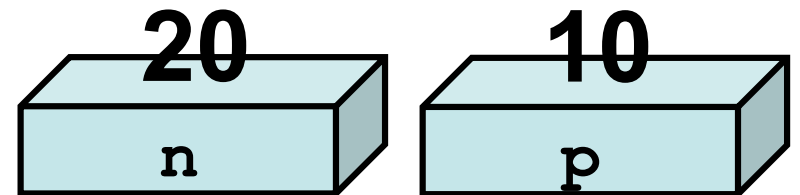


# Pas-à-pas

```
void exchange(int * pa, int * pb)
{
    int tmp;
    tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}
```

...

```
int n = 10, p = 20;
exchange(&n, &p);
```



**Deuxième application**  
Une fonction peut accéder  
aux éléments d'un tableau  
passé en paramètre

# Tableaux et pointeurs

Utilisé seul, le nom du tableau correspond à l'adresse du premier élément.

Par exemple, on peut faire:

```
int T[5];  
int * tab = T;
```

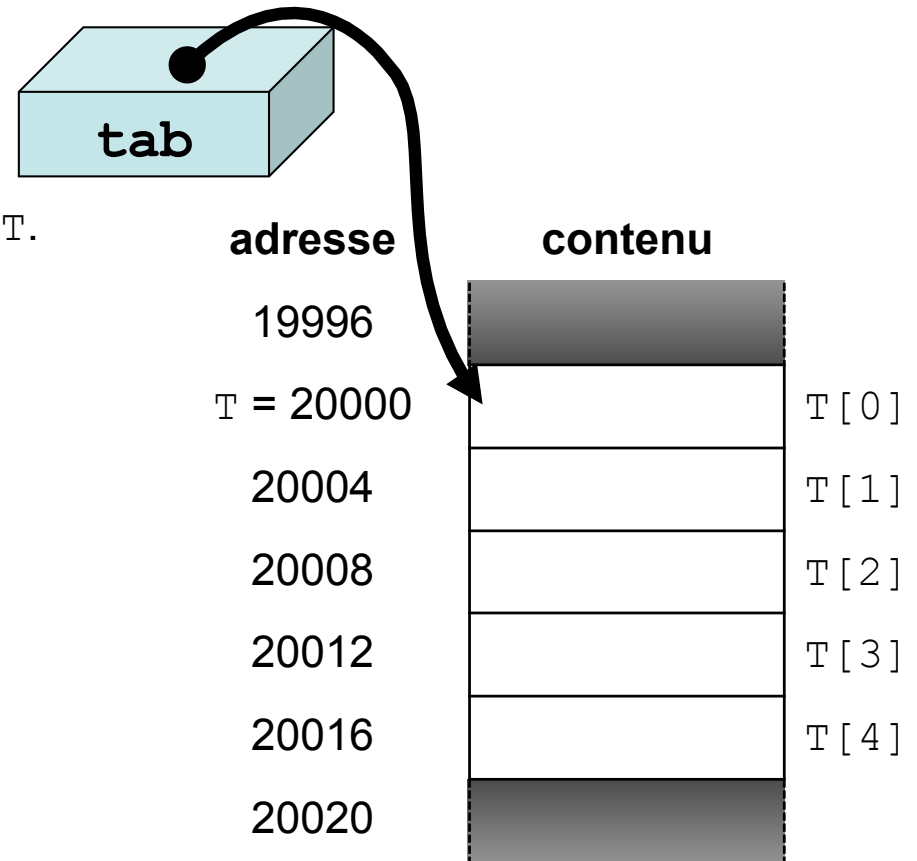
et `tab` pointe alors sur le premier élément de `T`.

Attention, `T` n'est pas pour autant un pointeur.

On ne peut pas faire par exemple:

```
T = &a;
```

si `T` a été déclaré comme un tableau.





# Tableaux et pointeurs

Après:

```
int T[5];  
int * tab = T;
```

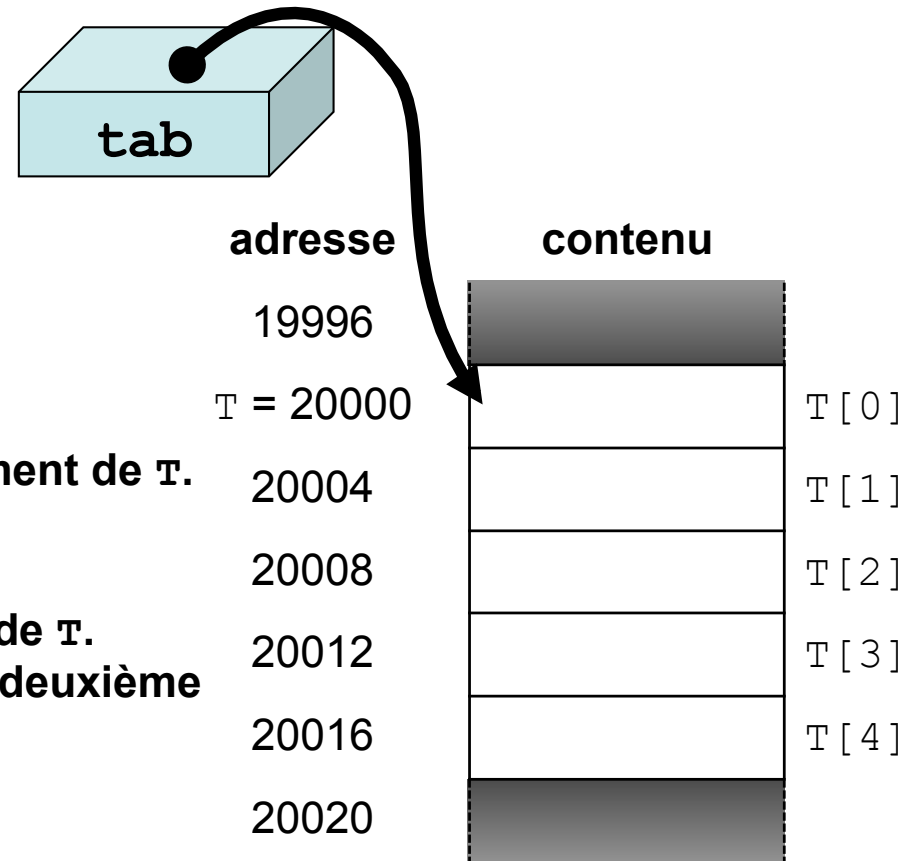
`tab` pointe sur le premier élément de `T`.

**`*tab` correspond donc à `T[0]`, le premier élément de `T`.**

**`tab + 1` est l'adresse du deuxième élément de `T`.**

**`*(tab + 1)` correspond donc à `T[1]`, le deuxième élément de `T`.**

etc...



Notez que:

Si

`tab` vaut 20000,  
et si `tab` est un pointeur sur `int`

Alors

`tab + 1` vaut 20004

pour pointer sur le `int` suivant: un `int` est représenté sur 4 octets, et les adresses sont définies en octet.

Ce mécanisme de calcul simplifie l'accès aux éléments du tableau: il n'y a pas à savoir qu'il faut multiplier par 4 dans le cas des `int`, par 8 dans le cas des `double`, etc...

# Tableaux et pointeurs

Après:

```
int T[5];  
int * tab = T;
```

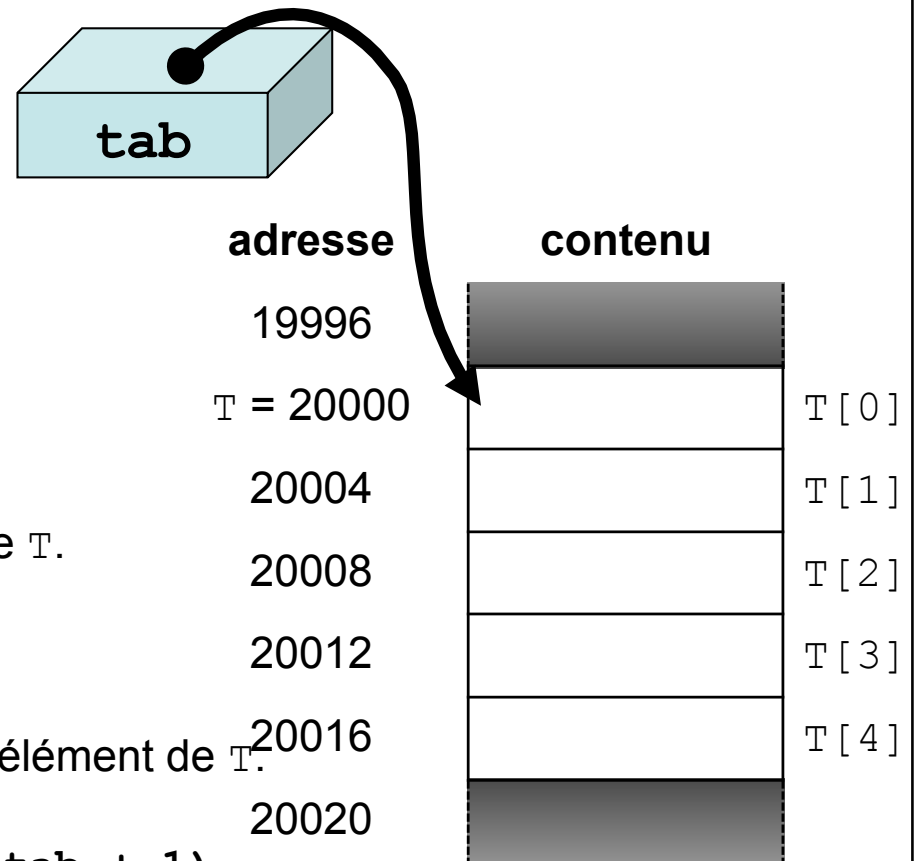
`tab` pointe sur le premier élément de `T`.

`*tab` correspond donc à `T[0]`, le premier élément de `T`.

`tab + 1` est l'adresse du deuxième élément de `T`.

`*(tab + 1)` correspond donc à `T[1]`, le deuxième élément de `T`.

On peut également écrire `tab[1]` à la place de `*(tab + 1)`.



En C et en C++, on peut écrire

```
tab[i]
```

à la place de

```
*(tab + i)
```

ce qui permet d'utiliser `tab` comme on utiliserait un tableau!

# Accès aux éléments d'un tableau

Supposons qu'on veut écrire une fonction qui ajoute 1 aux éléments d'un tableau. On peut donc écrire cette fonction comme ça:

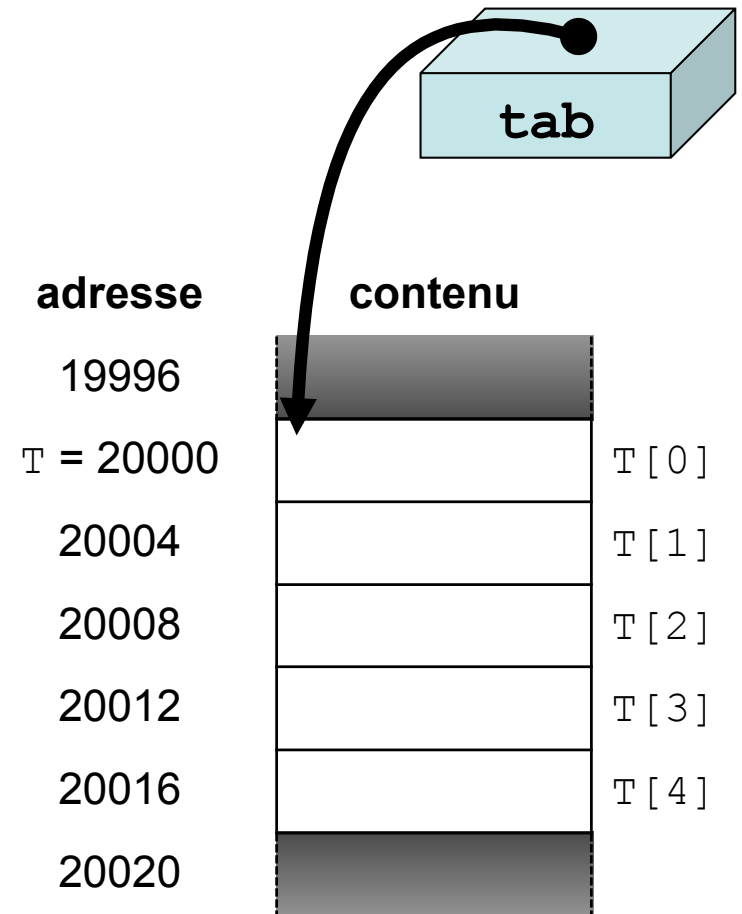
```
void ajoutez1_aux_elements(int * tab)
{
    for(int i = 0; i < 5; i++)
        *(tab + i) = *(tab + i) + 1;
}
```

ou comme ça:

```
void ajoutez1_aux_elements(int * tab)
{
    for(int i = 0; i < 5; i++)
        tab[i] = tab[i] + 1;
}
```

et on peut appeler cette fonction ainsi:

```
int T[5];
ajoutez1_aux_elements(T);
```



# Attention au nombre d'éléments

La fonction

```
void ajoute1_aux_elements(int * tab)
```

ne connaît PAS LA TAILLE du tableau passé en paramètre.

# Passer le nombre d'éléments en paramètre

Pour que la fonction puisse être utilisée pour n'importe quel tableau, quelle que soit sa taille, on peut passer la taille du tableau en paramètre.

Exemple:

```
void ajoute1_aux_elements(int * tab, int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
        tab[i] = tab[i] + 1;
}
```

Exemple d'utilisation:

```
int T[5], U[101];
```

...

```
ajoute1_aux_elements(T, 5);
ajoute1_aux_elements(U, 101);
```

# Passer l'adresse d'un élément d'un tableau en paramètre

Supposons maintenant qu'on veuille écrire la fonction `ajoute1_aux_elements()` en utilisant la fonction `ajoute1()` vue précédemment:

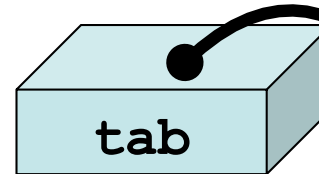
```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

On peut écrire:

```
void ajoute1_aux_elements(int * tab, int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
        ajoute1(&(tab[i]));
}
```



# Passer l'adresse d'un élément d'un tableau en paramètre



`&(tab[i])`

correspond à l'adresse de l'élément `i` de `tab`.

Une autre façon d'obtenir cette adresse est:

`tab + i`

`&(tab[i])` est équivalent à `tab + i`

adresse	contenu	
19996		
$T = 20000$		<code>T[0]</code>
20004		<code>T[1]</code>
20008		<code>T[2]</code>
20012		<code>T[3]</code>
20016		<code>T[4]</code>
20020		

# Passer l'adresse d'un élément d'un tableau en paramètre

```
void ajoute1_aux_elements(int * tab, int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
        ajoute1(tab + i);
}
```

est donc équivalent à (mais plus élégant que):

```
void ajoute1_aux_elements(int * tab, int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
        ajoute1(&(tab[i]));
}
```

# Quelques exemples

```
void fonction1(int * tab, int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
        printf(«%d\n», tab[i]);
}
```

```
int fonction2(int * tab, int nb_elements)
{
    int f = 0;
    for(int i = 0; i < nb_elements; i++)
        f = f + tab[i];
    return f;
}
```

```
void fonction3(float * tab, int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
        tab[i] = 0.0;
}
```

```
void fonction4(float * X, float * Y, int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
        Y[i] = cos(X[i]);
}
```

# Structures et pointeurs

# Déclaration d'une structure

Voici un exemple de déclaration de structure:

```
struct Chat
{
    float poids;
    char sexe; // 'M' ou 'F'
    int numero_tatouage;
};
```

Il s'agit d'un *nouveau type* appelé `struct Chat`

Ce type peut maintenant être utilisé pour déclarer des variables:

```
struct Chat felix;
struct Chat garfield;
```

Chacune des variables `felix` et `garfield` contiendra un flottant (poids), un caractère (sexe) et un entier (numero\_tatouage).

# Déclaration d'une structure

`struct` est un mot-clé indiquant la déclaration d'une structure

`Chat` est le nom de la structure.

`struct Chat`

```
{  
    float poids;  
    char sexe;  
    int numero_tatouage;  
}
```

`poids` est un des champs de la structure, de type `float`.

La liste des champs de la structure est délimitée par des accolades `{` et `}`.

L'accolade fermante `}` est suivie d'un point-virgule `;` (contrairement à l'accolade fermante du corps d'une fonction, d'une boucle ou d'un `if`).

# Déclaration d'une structure

```
struct Chat
```

```
{  
    float poids;  
    char sexe;  
    int numero_tatouage;  
};
```

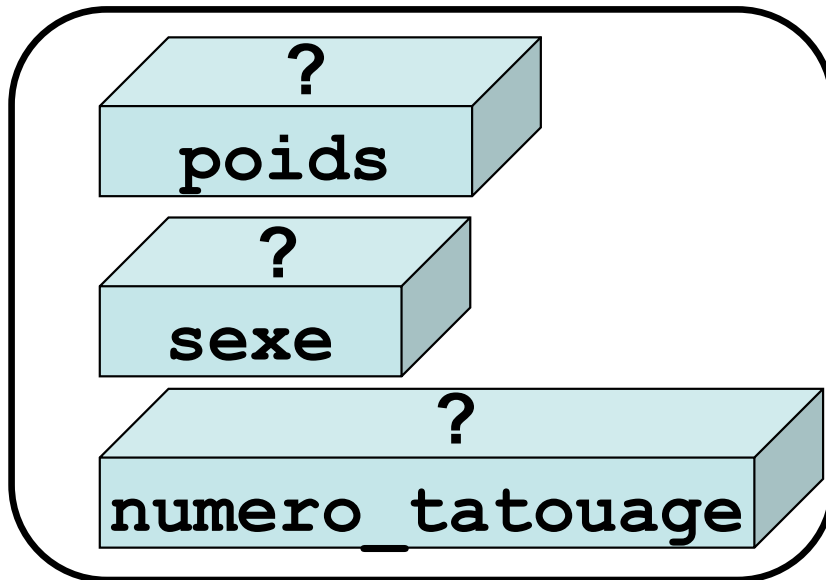
***Champs de la structure***

# Déclaration d'une variable de type structure

`Chat` est le nom de la structure.

`felix` est le nom de la variable.

`Chat felix;`



**`felix`**



# Exemple

```
struct Chat
{
    float poids;
    char  sexe; // 'M' ou 'F'
    int   numTatouage;
};

int main()
{
    struct Chat felix;

    felix.numTatouage = 65328;
    felix.sexe = 'M';

    printf(«Indroduire le poids du chat de tatouage %d: », felix.numTatouage);
    scanf(«%f», &(felix.poids));

    printf(«Le chat %d pese %f\n», felix.numTatouage, felix.poids);

    return 0;
}
```

# Utilisation des champs d'une structure

Chaque **champ** d'une structure peut être manipulé comme une variable:

```
felix.numTatouage = 65328;  
felix.sexe = 'M';
```

Un **point** sépare le nom de la variable du nom du champ.

`felix`: nom de la variable de type `struct Chat`

`numTatouage`: nom d'un champ de la structure `Chat`

`felix.numTatouage` se manipule alors comme toute variable de type `int`.

# L'opérateur $\rightarrow$

La notation

$(*c) . nom$

n'est pas très lisible.

Il existe un opérateur, noté  $\rightarrow$  (un signe moins  $-$ , un signe supérieur  $>$ ), qui permet d'écrire des expressions équivalentes mais plus lisibles:

$c \rightarrow nom$

**$c \rightarrow nom$  est équivalent à  $(*c) . nom$**

Dorénavant, utilisez  $\rightarrow$  plutôt que  $(*)$ .

# Structure en paramètre d'une fonction

```
struct Chat
{
    char nom[100];
    float poids;
    char sexe;
    int numero_tatouage;
};

void affiche(struct Chat c)
{
    printf(«%s est », c.nom);

    if (c.sexe == 'M')
        printf(«un mâle»);
    else
        printf(«une femelle»);

    printf(« et pèse %f kg.\n», c.poids);
}

int main()
{
    struct Chat chat1;
    (...on remplit les champs de chat1...)
    affiche(chat1);
    return 0;
}
```

# Structure en paramètre d'une fonction

```
struct Chat
{
    char nom[100];
    float poids;
    char sexe;
    int numero_tatouage;
};

void affiche(struct Chat *c)
{
    printf(«%s est », (*c).nom);

    if ((*c).sexe == 'M')
        printf(«un mâle»);
    else
        printf(«une femelle»);

    printf(« et pèse %f kg.\n», (*c).poids);
}

int main()
{
    struct Chat chat1;
    (...on remplit les champs de chat1...)
    affiche(&chat1);
    return 0;
}
```

# Structure en paramètre d'une fonction

```
struct Chat
{
    char nom[100];
    float poids;
    char sexe;
    int numero_tatouage;
};

void affiche(struct Chat *c)
{
    printf(«%s est », c->nom);

    if (c->sexe == 'M')
        printf(«un mâle»);
    else
        printf(«une femelle»);

    printf(« et pèse %f kg.\n», c->poids);
}

int main()
{
    struct Chat chat1;
    (...on remplit les champs de chat1...)
    affiche(&chat1);
    return 0;
}
```

# . ou -> ?

A gauche de -> on trouve forcément un *pointeur* sur structure:

```
struct Chat *c;  
c = (struct Chat *) malloc(sizeof(struct Chat));  
c->poids = 2.5;
```

A gauche de . on trouve forcément une *variable* de type structure:

```
struct Chat c;  
c.poids = 2.5;
```

# Manipulation des nombres complexes

```
struct NombreComplexe
{
    float re; // partie reelle
    float im; // partie imaginaire
};

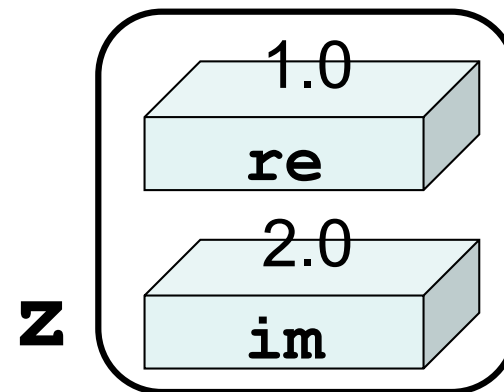
typedef Complexe struct NombreComplexe;
```



# Fonctions d'initialisation

Pour créer une variable de type `Complexe` à partir de ses parties réelle et imaginaire, on pourrait faire:

```
Complexe z;  
z.re = 1;  
z.im = 2;
```



Nous allons voir comment mettre ces instructions dans une fonction.

# Structure en résultat d'une fonction: Fonction d'initialisation (première version)

```
Complexe init_comp(float re, float im)
{
    Complexe C;

    C.re = re;
    C.im = im;

    return C;
}
...
Complexe z = init_comp(1, 2);
```

# Pas-à-pas

```
Complexe init_comp(float re, float im)
{
    Complexe C;

    C.re = re;
    C.im = im;

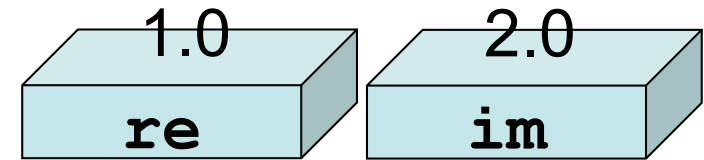
    return C;
}
```

```
...
→ Complexe z = init_comp(1, 2);
```

# Pas-à-pas

→ `Complexe init_comp(float re, float im)`  
`{`  
    `Complexe C;`  
  
    `C.re = re;`  
    `C.im = im;`  
  
    `return C;`  
`}`

→ `...  
Complexe z = init_comp(1, 2);`



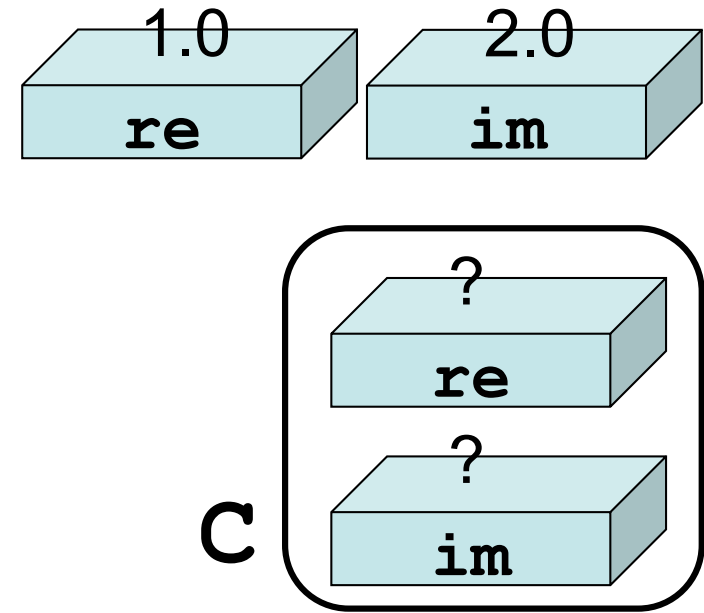
# Pas-à-pas

```
Complexe init_comp(float re, float im)
{
    → Complexe C;

    C.re = re;
    C.im = im;

    return C;
}
```

```
...
→ Complexe z = init_comp(1, 2);
```



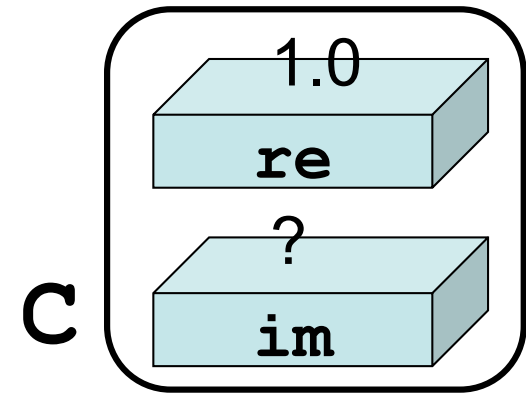
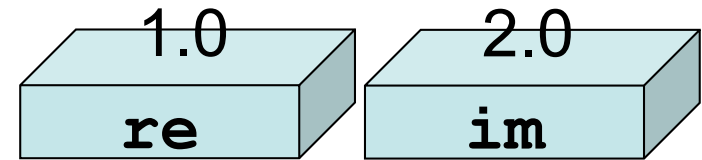
# Pas-à-pas

```
Complexe init_comp(float re, float im)
{
    Complexe C;

    → C.re = re;
      C.im = im;

    return C;
}
```

```
→ ... Complexe z = init_comp(1, 2);
```



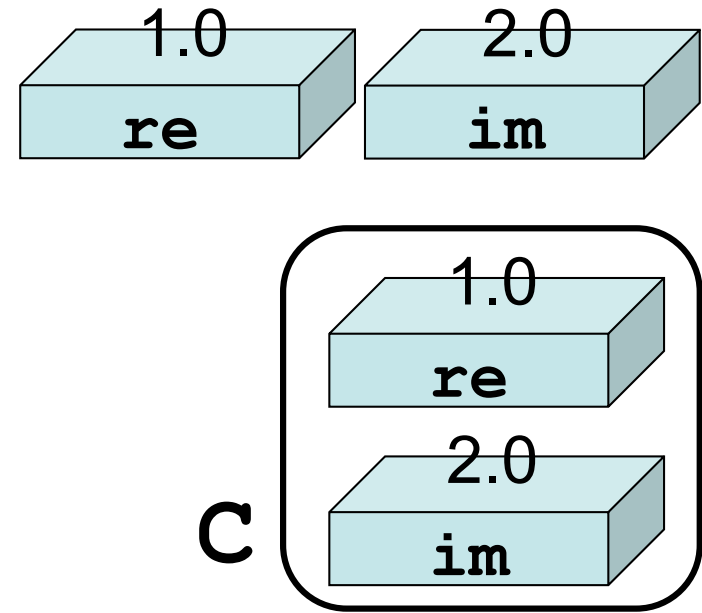
# Pas-à-pas

```
Complexe init_comp(float re, float im)
{
    Complexe C;

    C.re = re;
    → C.im = im;

    return C;
}
```

```
→ ... Complexe z = init_comp(1, 2);
```

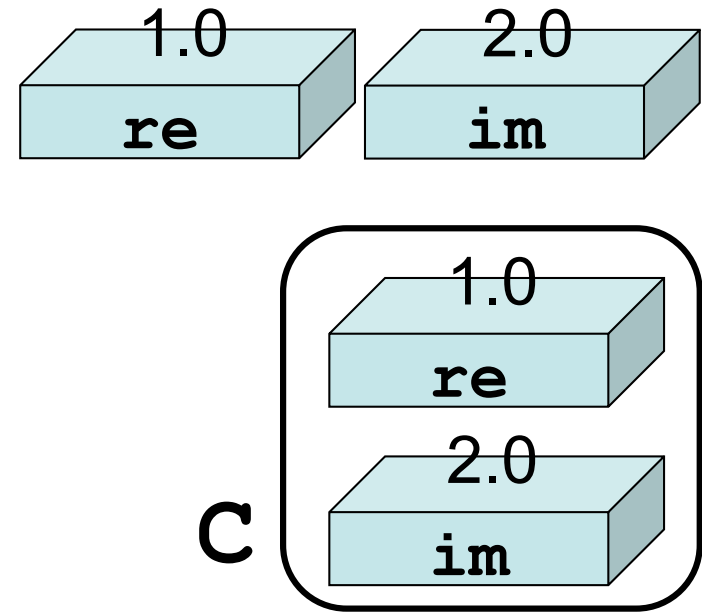


# Pas-à-pas

```
Complexe init_comp(float re, float im)
{
    Complexe C;

    C.re = re;
    C.im = im;
    → return C;
}
```

```
→ ...
Complexe z = init_comp(1, 2);
```





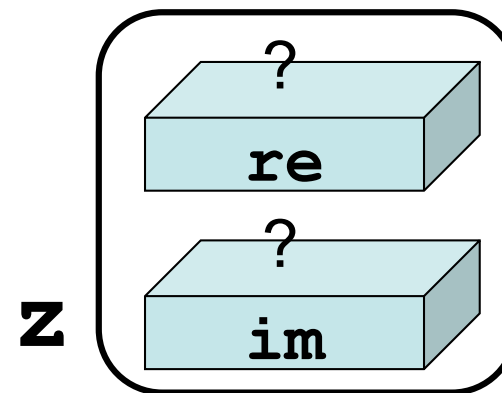
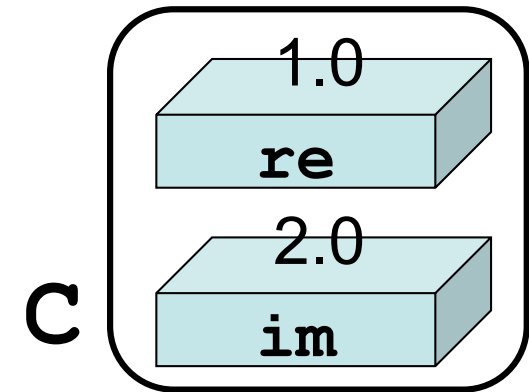
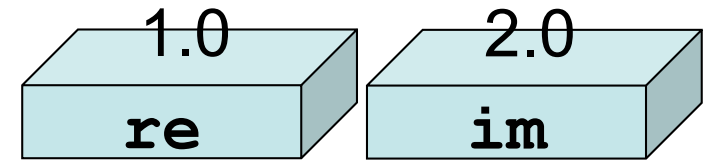
# Pas-à-pas

```
Complexe init_comp(float re, float im)
{
    Complexe C;

    C.re = re;
    C.im = im;

    → return C;
}
```

```
→ ...
Complexe z = init_comp(1, 2);
```



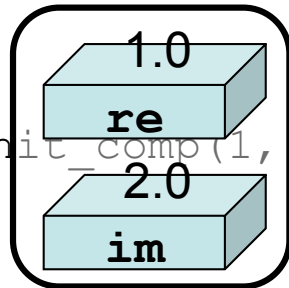
# Pas-à-pas

```
Complexe init_comp(float re, float im)
{
    Complexe C;

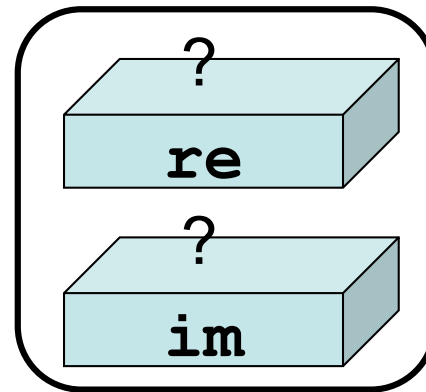
    C.re = re;
    C.im = im;

    return C;
}
```

→ ... Complexe z = init\_comp(1, 2);



**z**



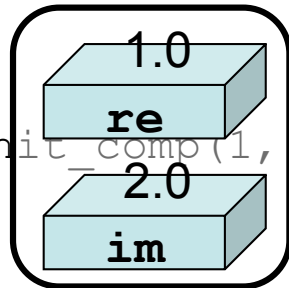
# Pas-à-pas

```
Complexe init_comp(float re, float im)
{
    Complexe C;

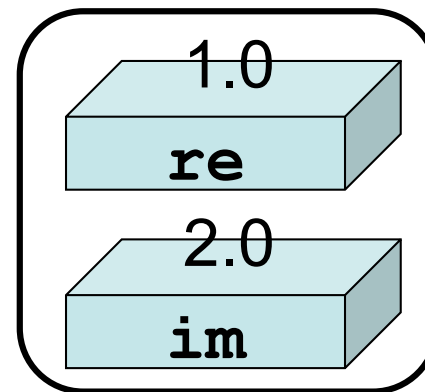
    C.re = re;
    C.im = im;

    return C;
}
```

→ ... Complexe z = init\_comp(1, 2);



**z**



# Pas-à-pas

```
Complexe init_comp(float re, float im)
{
    Complexe C;

    C.re = re;
    C.im = im;

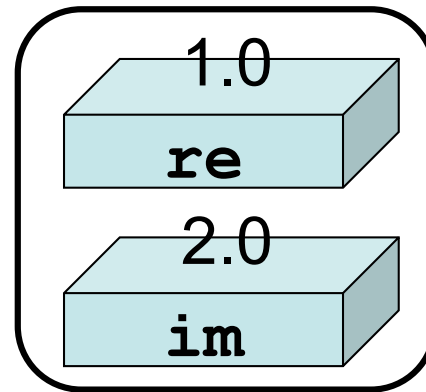
    return C;
}
```

...

```
Complexe z = init_comp(1, 2);
```



**z**

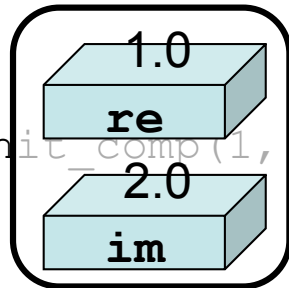


```
Complexe init_comp(float re, float im)
{
    Complexe C;

    C.re = re;
    C.im = im;

    return C;
}
```

→ ... Complexe z = init\_comp(1, 2);



Attention, les champs du résultat de `init_comp` sont copiés dans ceux de `z`, ce qui peut être coûteux comme pour le passage par valeur d'une structure.

# Fonction d'initialisation

## Deuxième version

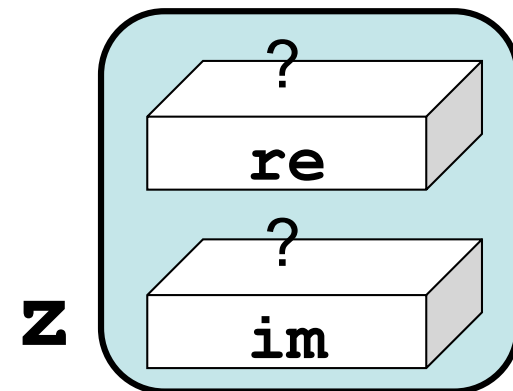
```
void init_comp(Complexe * c, float re, float im)
{
    c->re = re;
    c->im = im;
}
...
Complexe z;
init_comp(&z, 1, 2);
```

# Pas-à-pas

```
void init_comp(Complexe * c, float re, float im)
{
    c->re = re;
    c->im = im;
}
```

...

→ `Complexe z;`  
`init_comp(&z, 1, 2);`



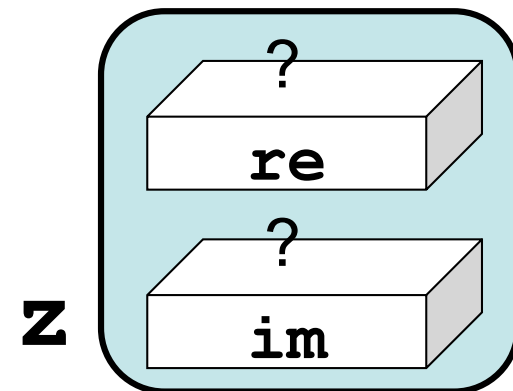
# Pas-à-pas

```
void init_comp(Complexe * c, float re, float im)
{
    c->re = re;
    c->im = im;
}
```

...

```
Complexe z;
```

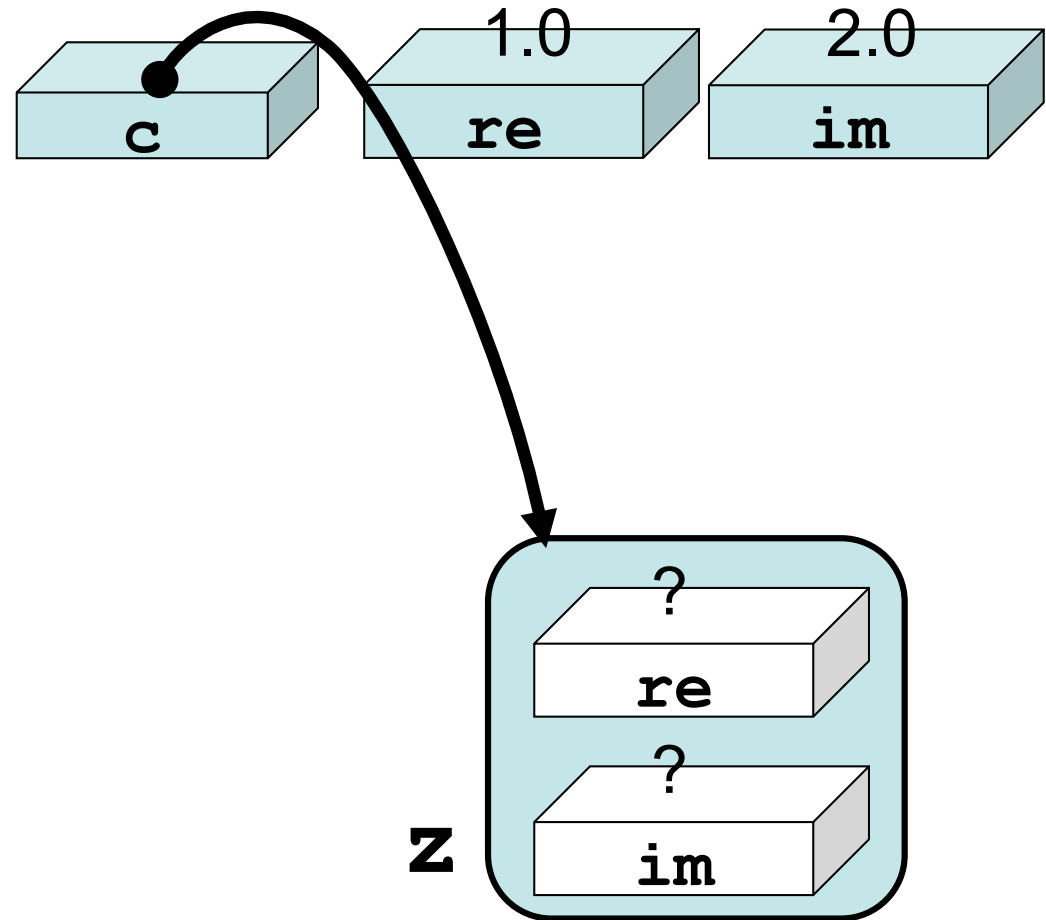
→ `init_comp(&z, 1, 2);`





# Pas-à-pas

```
→ void init_comp(Complexe * c, float re, float im)
{
    c->re = re;
    c->im = im;
}
...
Complexe z;
→ init_comp(&z, 1, 2);
```



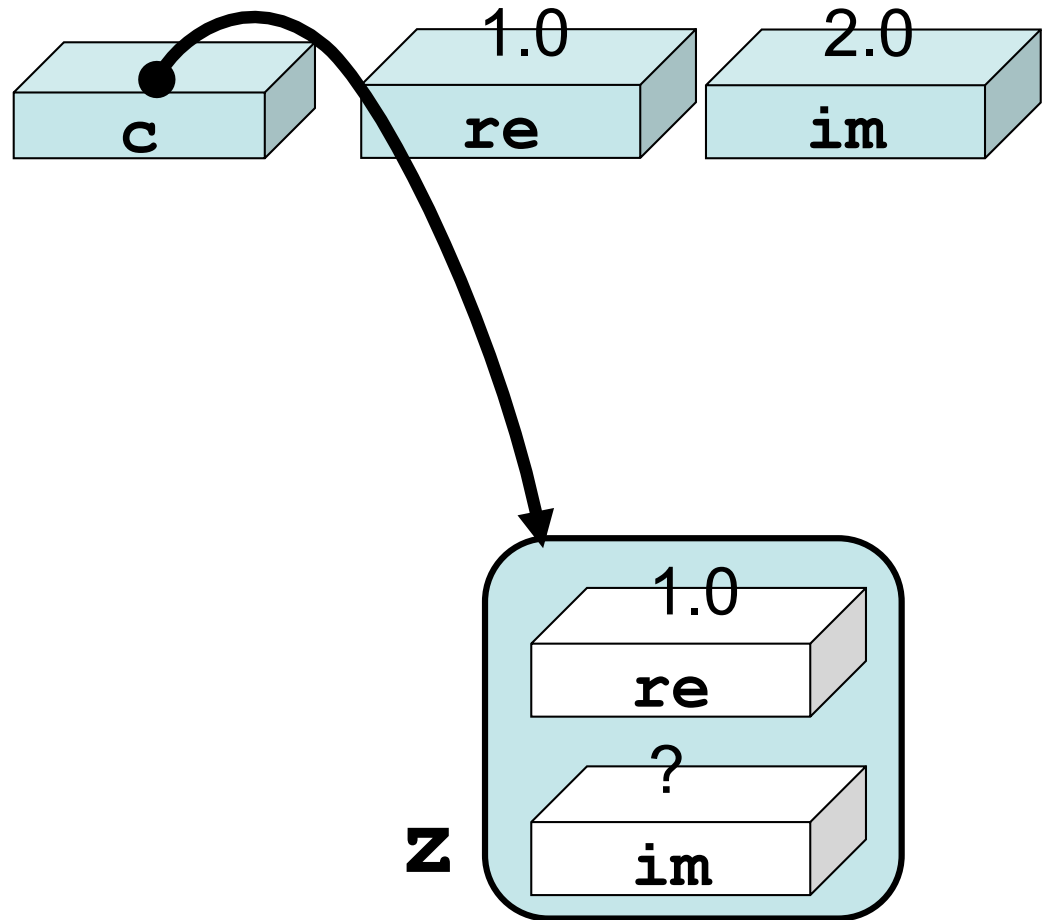
# Pas-à-pas

```
void init_comp(Complexe * c, float re, float im)
{
  → c->re = re;
    c->im = im;
}
```

...

```
Complexe z;
```

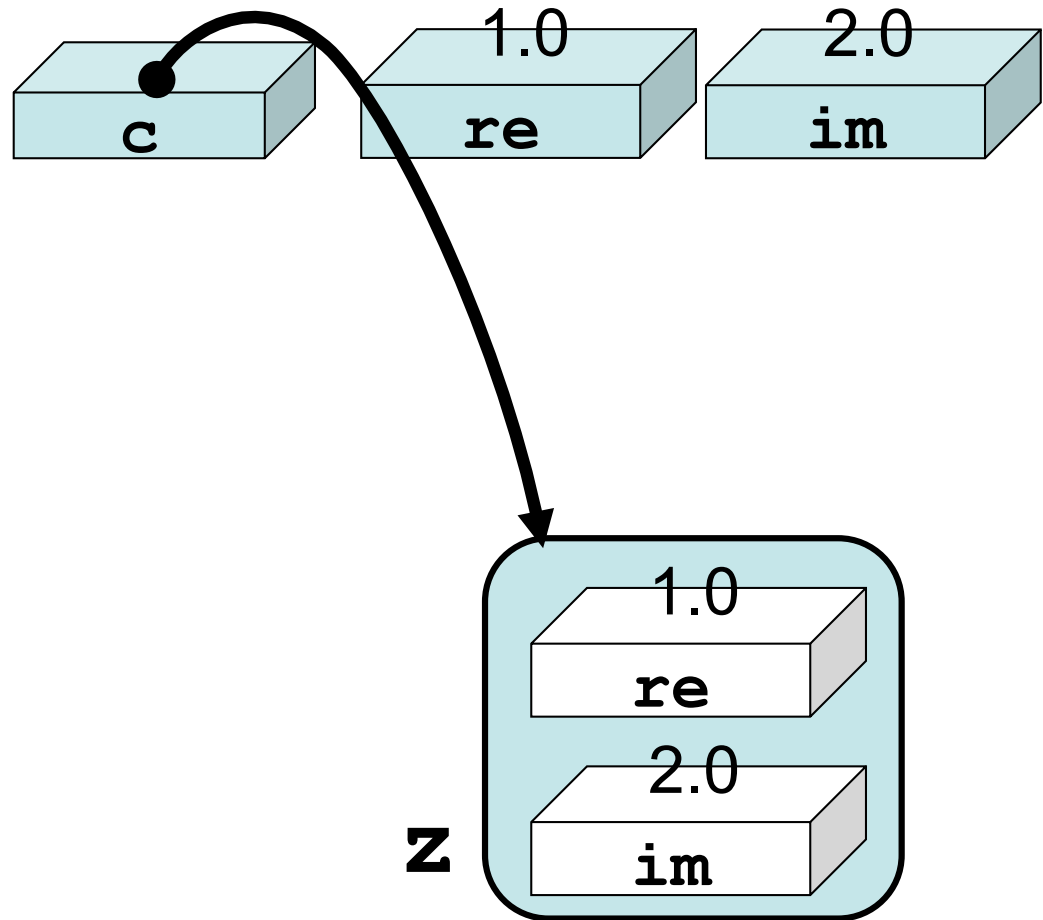
```
→ init_comp(&z, 1, 2);
```



# Pas-à-pas

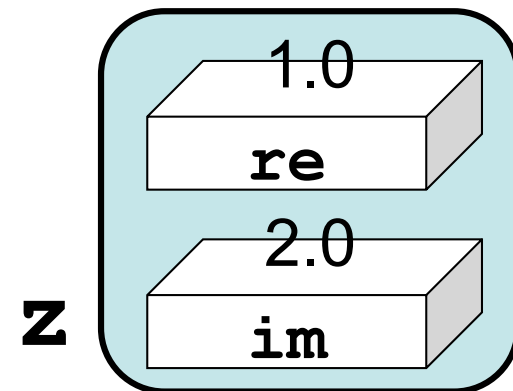
```
void init_comp(Complexe * c, float re, float im)
{
    c->re = re;
    c->im = im;
}
```

```
...
Complexe z;
→ init_comp(&z, 1, 2);
```



# Pas-à-pas

```
void init_comp(Complexe * c, float re, float im)
{
    c->re = re;
    c->im = im;
}
...
Complexe z;
init_comp(&z, 1, 2);
```



# Fonction d'initialisation, 3è version

```
Complexe * init_comp(float re, float im)
{
    Complexe * res = (Complexe *) malloc(sizeof(Complexe));
    res->re = re;
    res->im = im;
    return res;
}

...
Complexe * z = init_comp(1, 2);
```

z est maintenant un pointeur sur Complexe.

Nous verrons plus en détail cette autre forme dans la suite.