Pointeurs, allocation de mémoire et passage par adresse

Pierre-Alain FOUQUE
Département d'Informatique
École normale supérieure

Plan

1 – Pointeurs

2 – Allocation dynamique

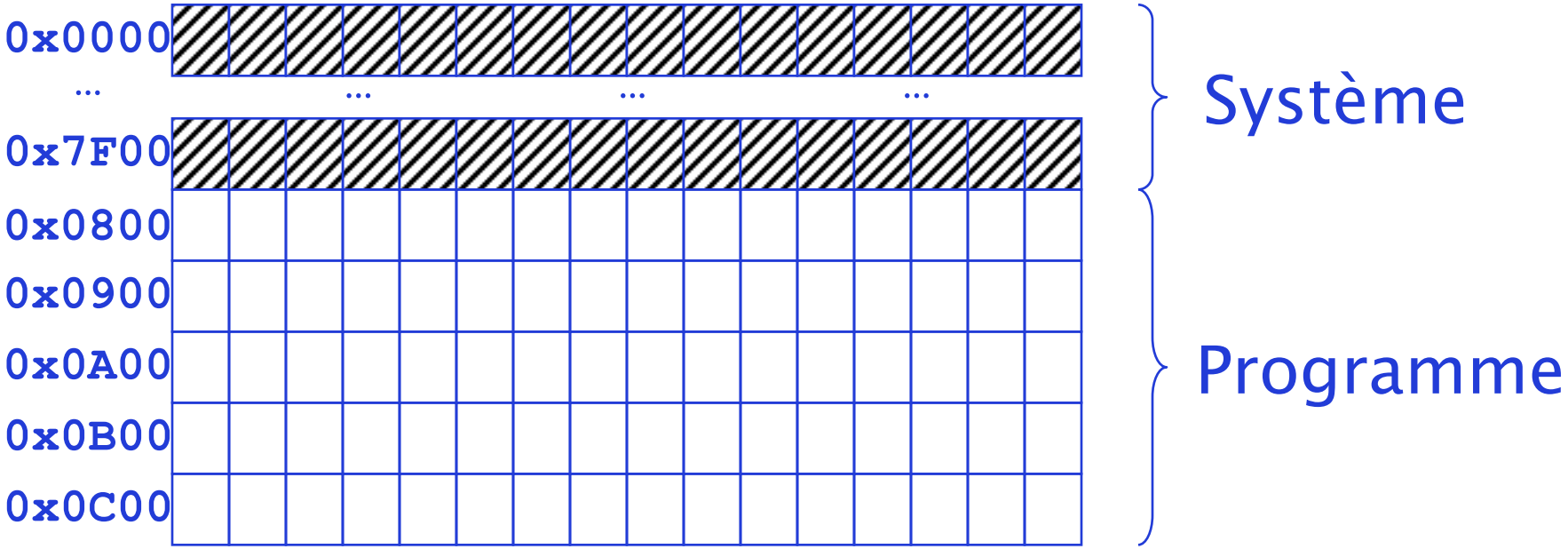
3 – Tableaux à plusieurs dimensions

4 – Passage par adresse

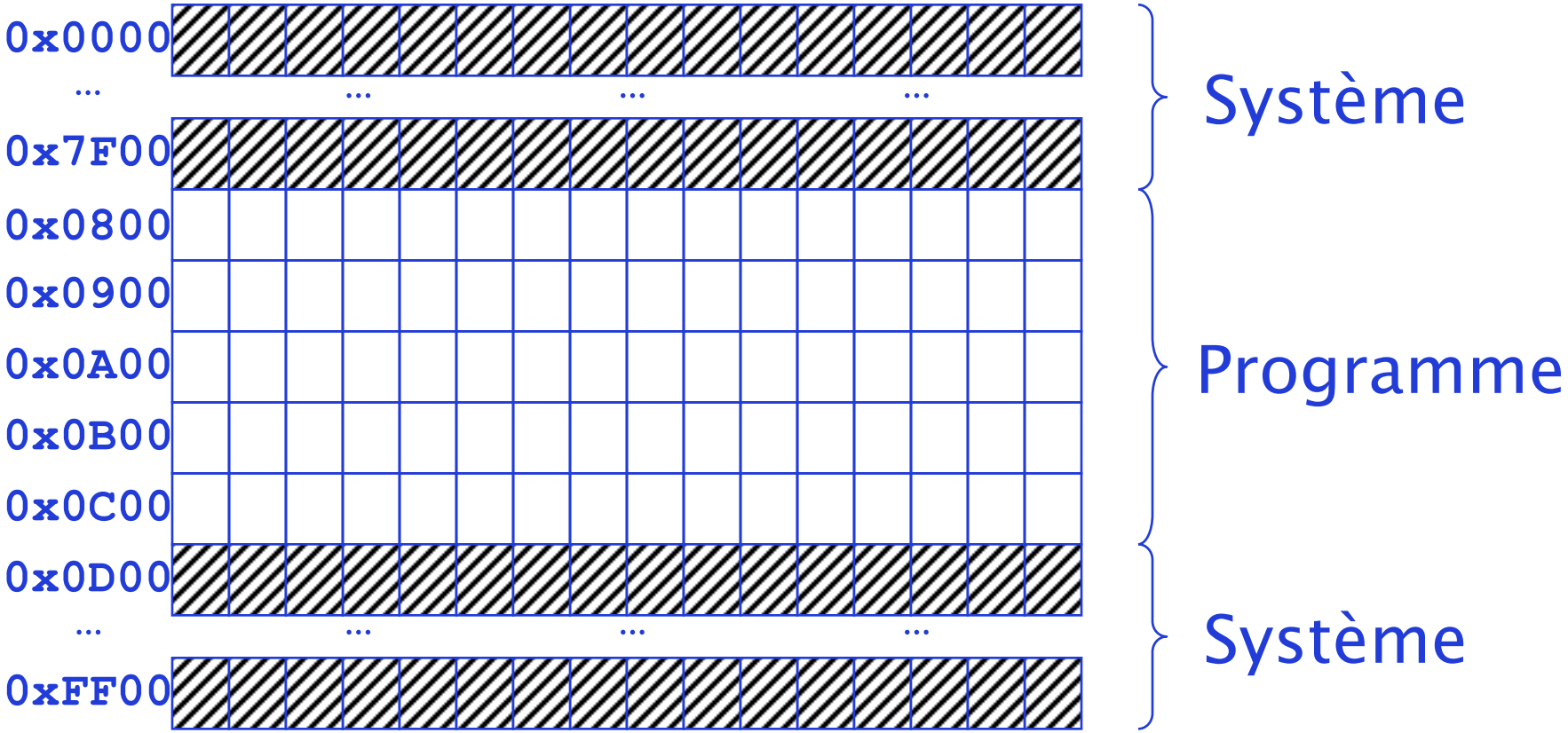
Mémoire

Mémoire

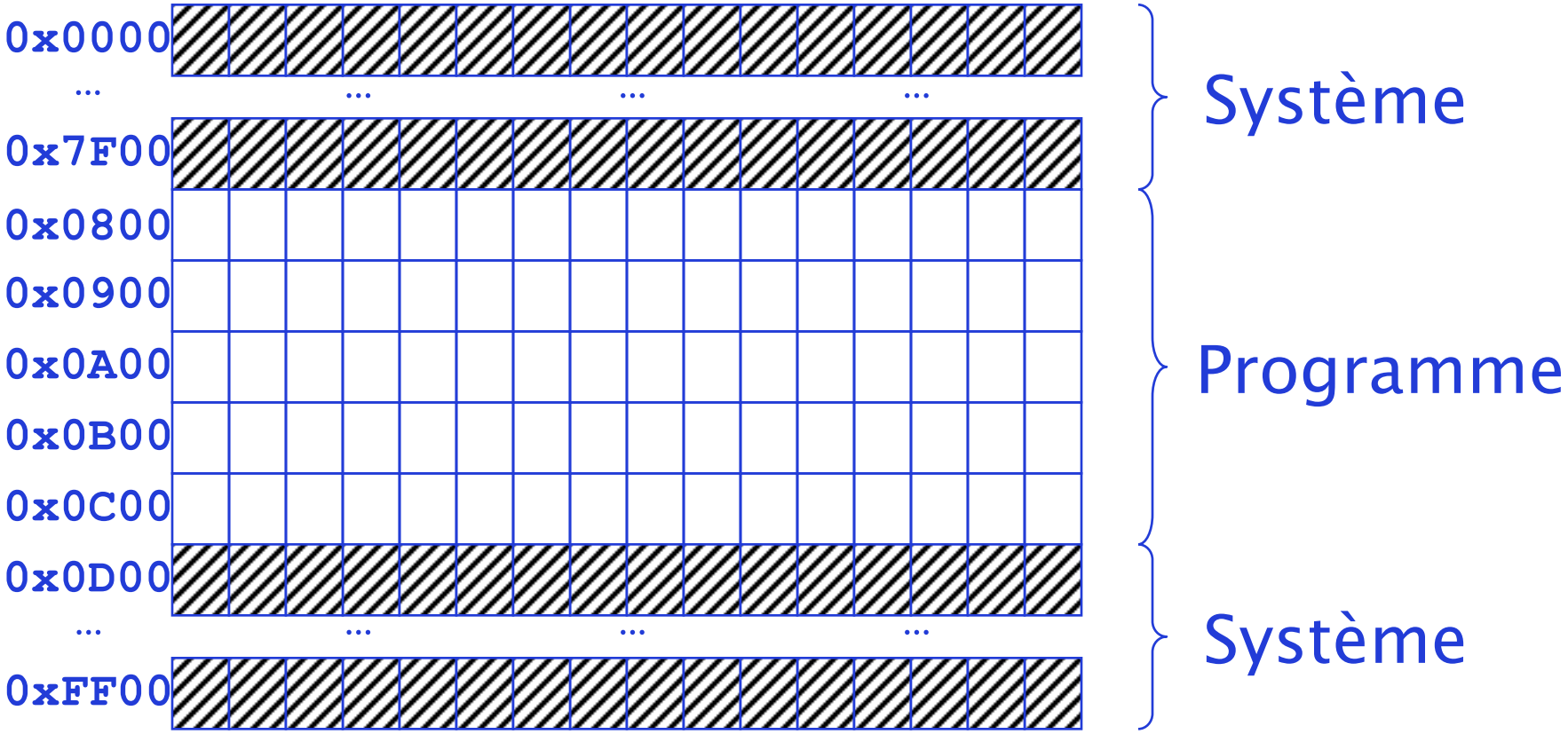
Mémoire


0x0000
...
0x7F00
0x0800
0x0900
0x0A00
0x0B00
0x0C00
Système
Programme

Mémoire

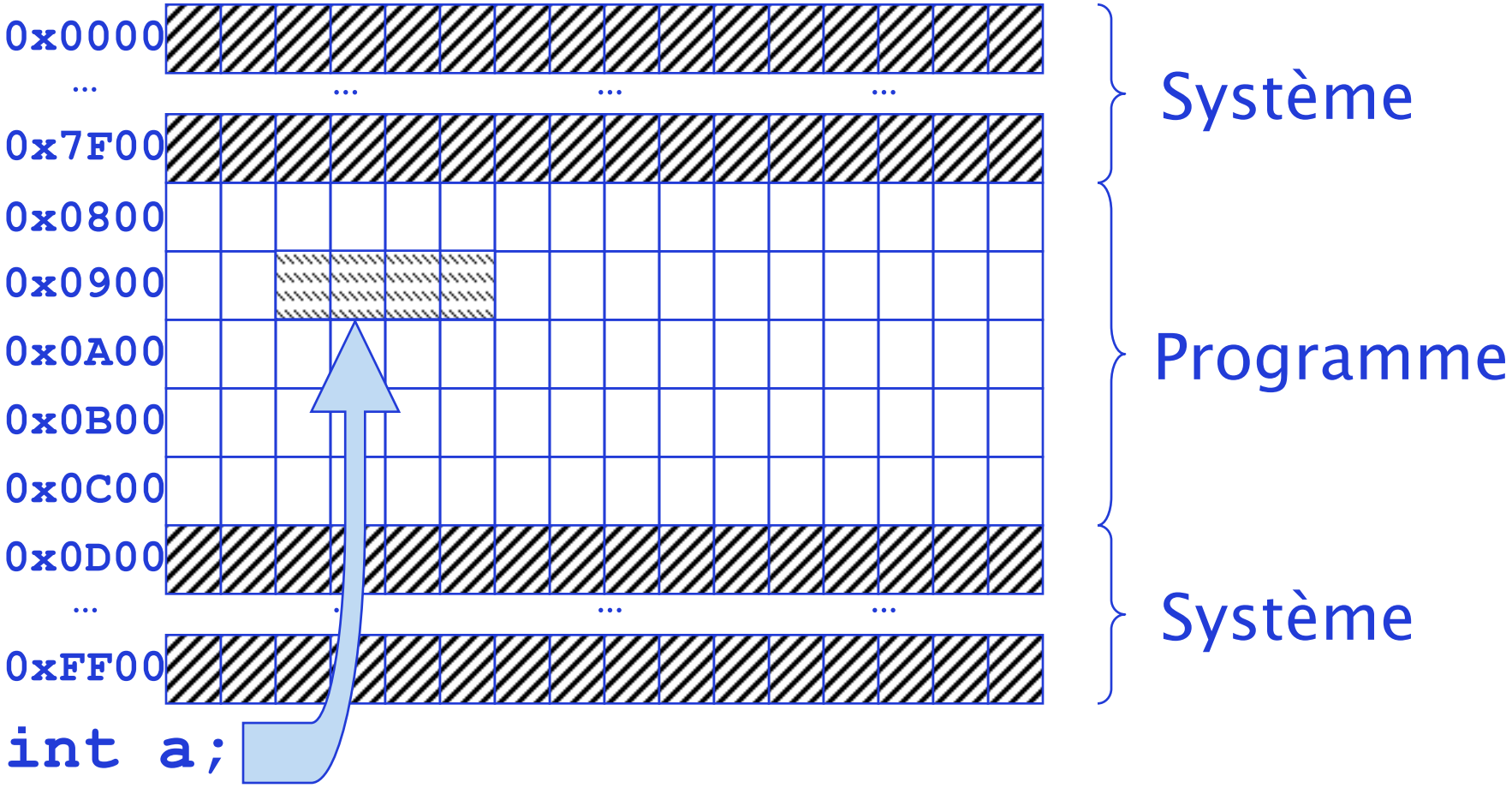

0x0000
...
0x7F00
0x0800
0x0900
0x0A00
0x0B00
0x0C00
0x0D00
...
0xFF00
Système
Programme
Système

Mémoire

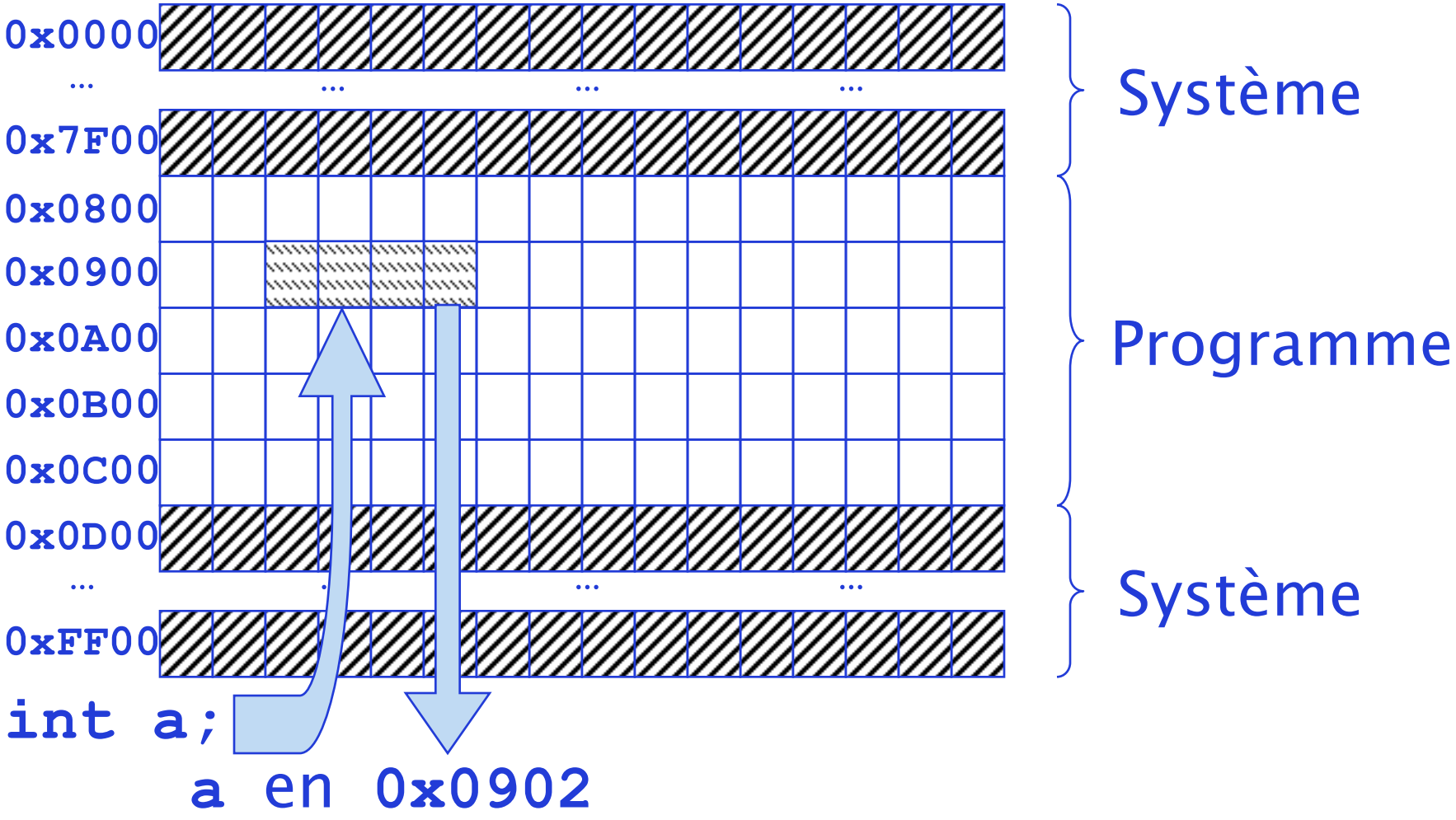



```
int a;
```

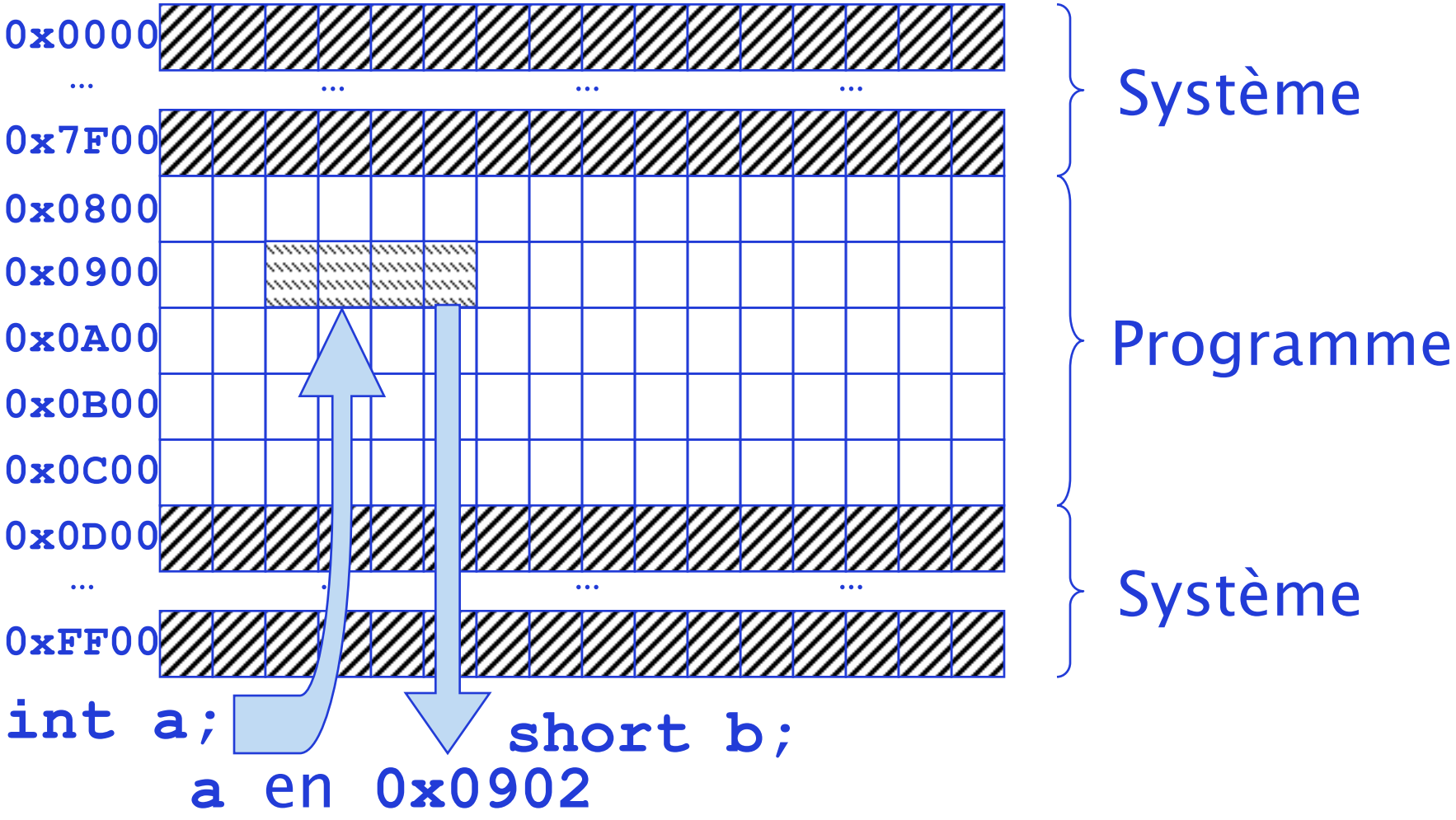
Mémoire


0x0000
0x7F00
0x0800
0x0900
0x0A00
0x0B00
0x0C00
0x0D00
0xFF00
int a;
Système
Programme
Système

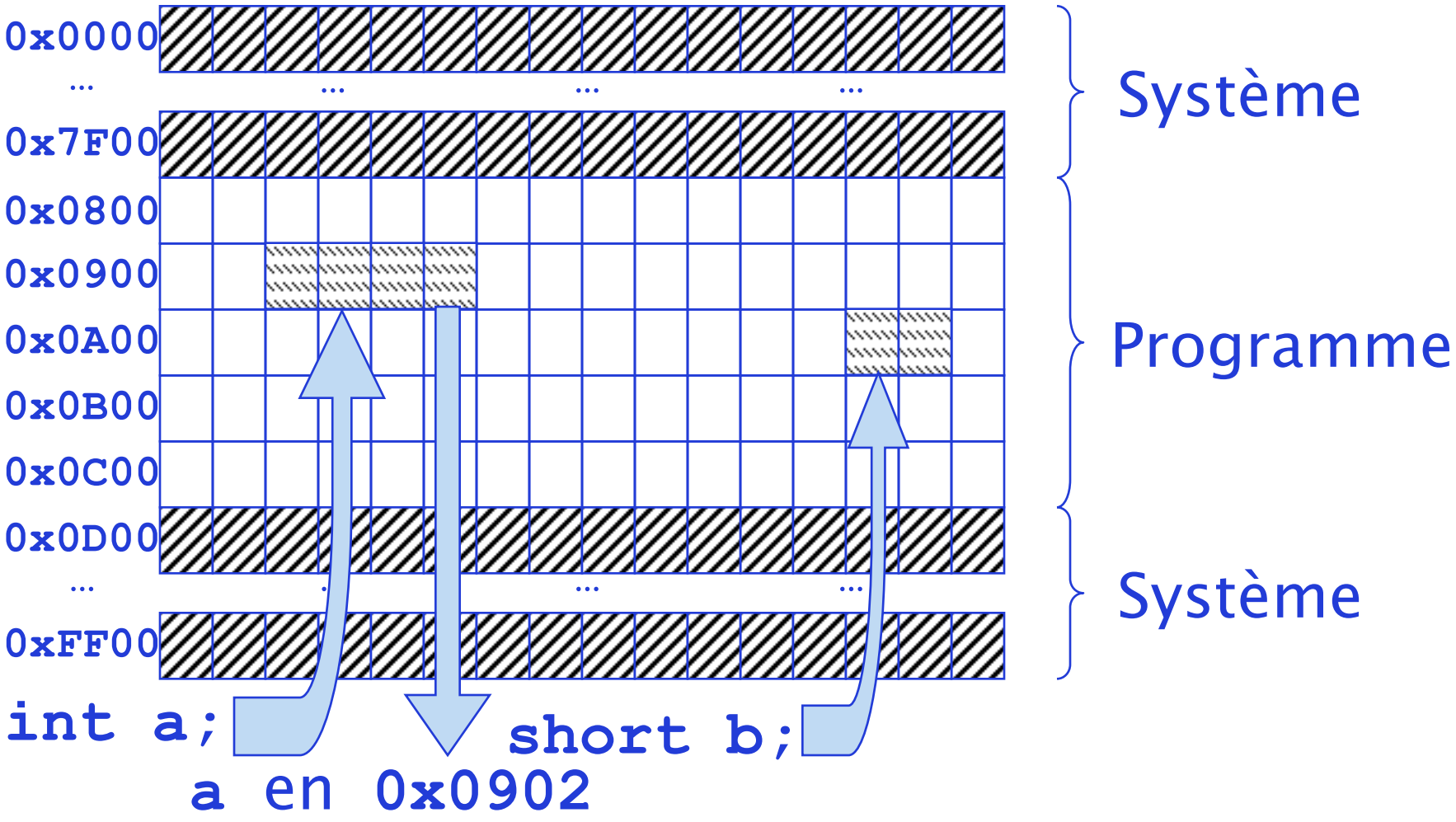
Mémoire


0x0000
...
0x7F00
0x0800
0x0900
0x0A00
0x0B00
0x0C00
0x0D00
...
0xFF00
Système
Programme
Système
int a;
a en 0x0902

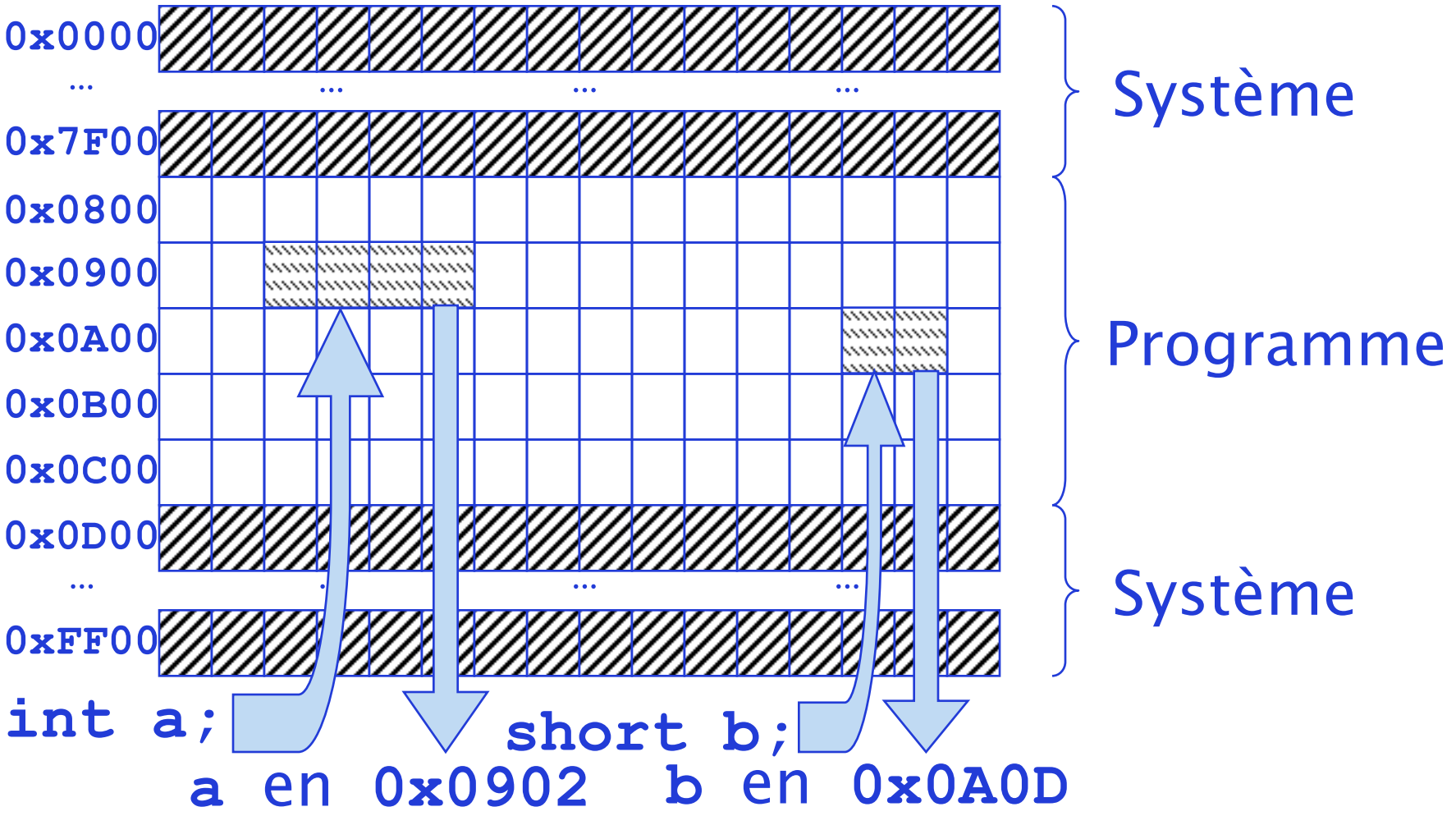
Mémoire


0x0000
...
0x7F00
0x0800
0x0900
0x0A00
0x0B00
0x0C00
0x0D00
...
0xFF00
Système
Programme
Système
int a;
short b;
a en 0x0902

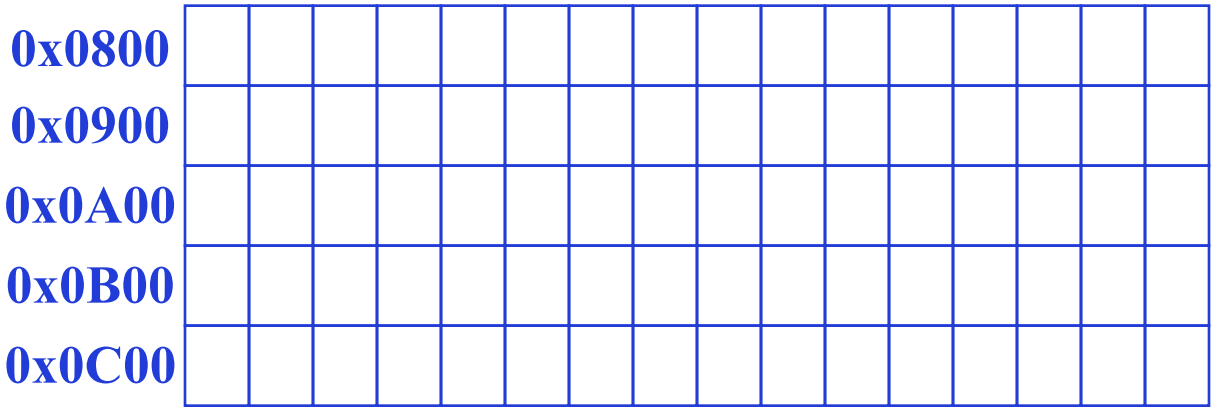
Mémoire


0x0000
...
0x7F00
0x0800
0x0900
0x0A00
0x0B00
0x0C00
0x0D00
...
0xFF00
Système
Programme
Système
int a;
short b;
a en 0x0902

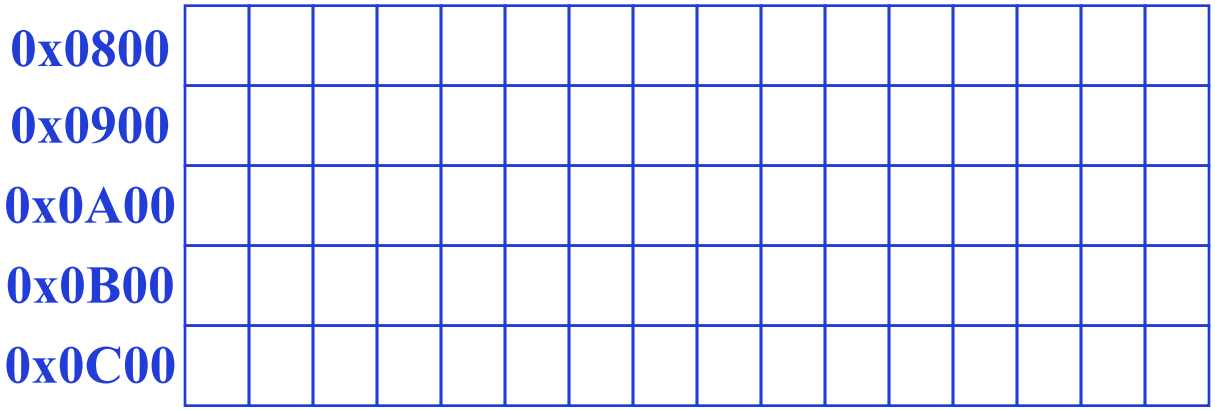
Mémoire


0x0000
...
0x7F00
0x0800
0x0900
0x0A00
0x0B00
0x0C00
0x0D00
...
0xFF00
Système
Programme
Système
int a;
short b;
a en 0x0902
b en 0x0A0D

Pointeur

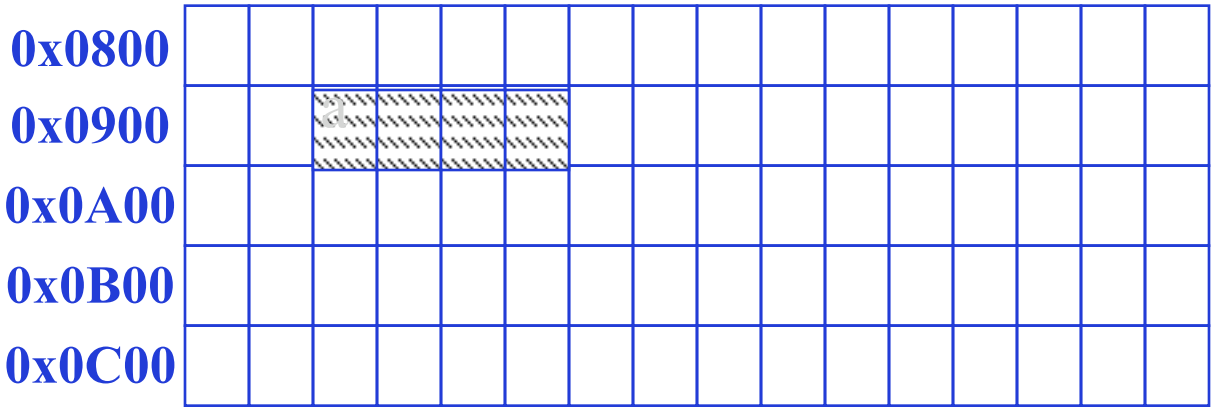


Pointeur



Une variable est stockée dans une zone mémoire réservée lors de la déclaration

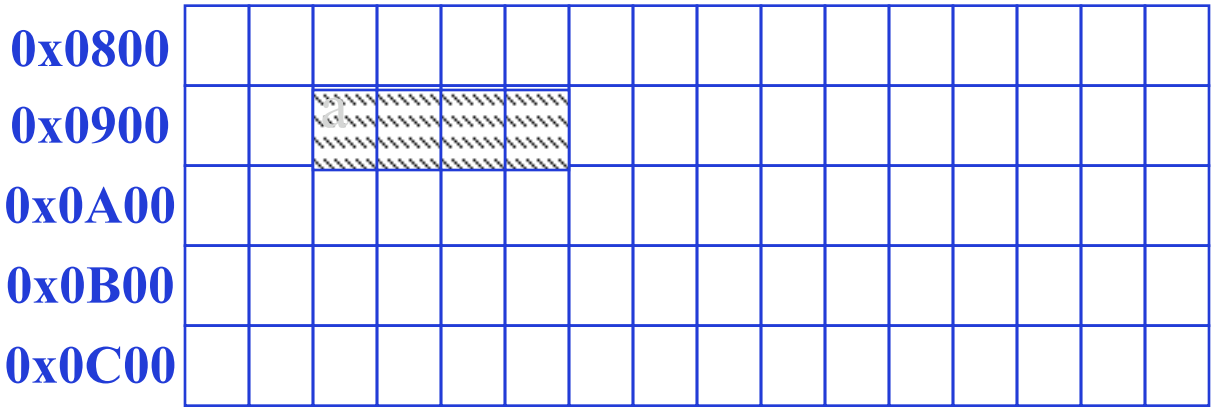
Pointeur



Une variable est stockée dans une zone mémoire réservée lors de la déclaration

```
int a;
```

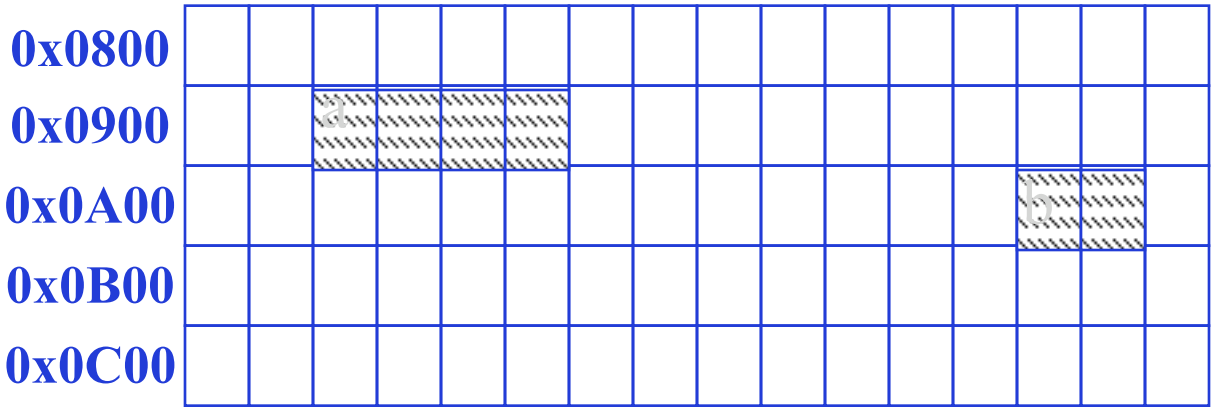
Pointeur



Une variable est stockée dans une zone mémoire réservée lors de la déclaration

`int a;` **0x0902** pointeur sur `(int) a`

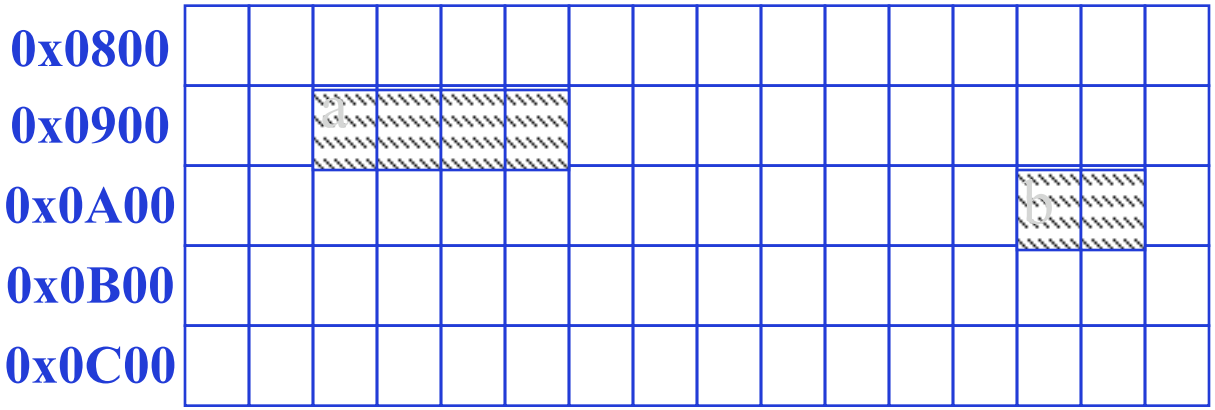
Pointeur



Une variable est stockée dans une zone mémoire réservée lors de la déclaration

```
int a;    0x0902 pointeur sur (int) a
short b;
```

Pointeur

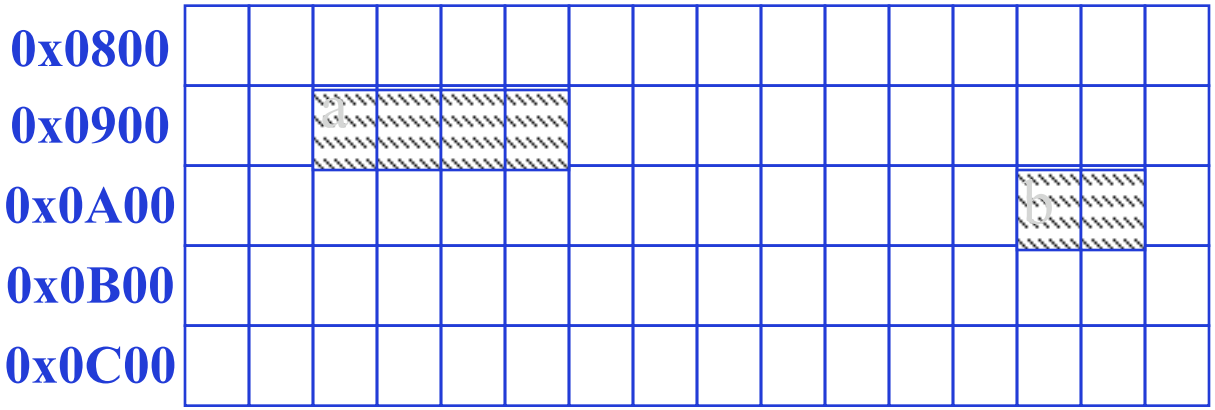


Une variable est stockée dans une zone mémoire réservée lors de la déclaration

`int a;` **0x0902** pointeur sur `(int) a`

`short b;` **0x0A0D** pointeur sur `(short) b`

Pointeur



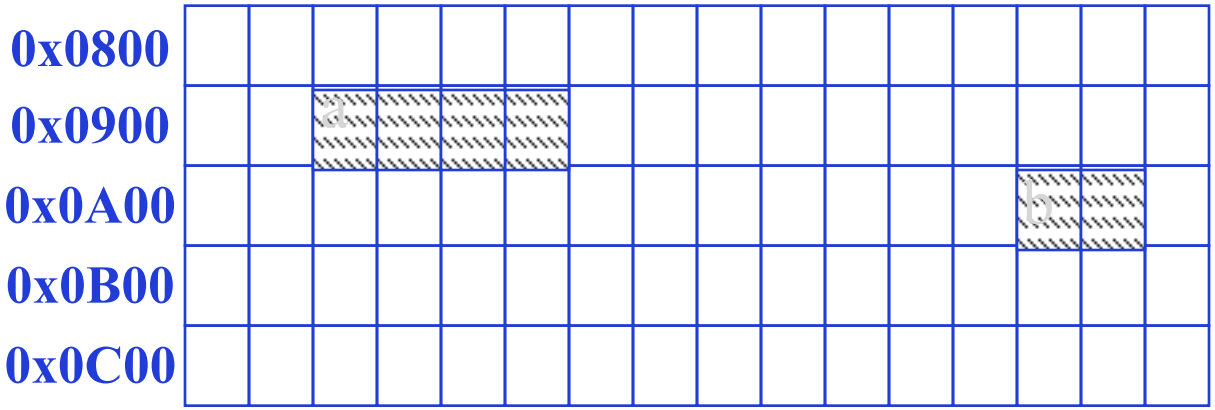
Une variable est stockée dans une zone mémoire réservée lors de la déclaration

& opérateur de référencement (adresse)

`int a;` **0x0902** pointeur sur **`(int) a`**

`short b;` **0x0A0D** pointeur sur **`(short) b`**

Pointeur



&a : 0x0902

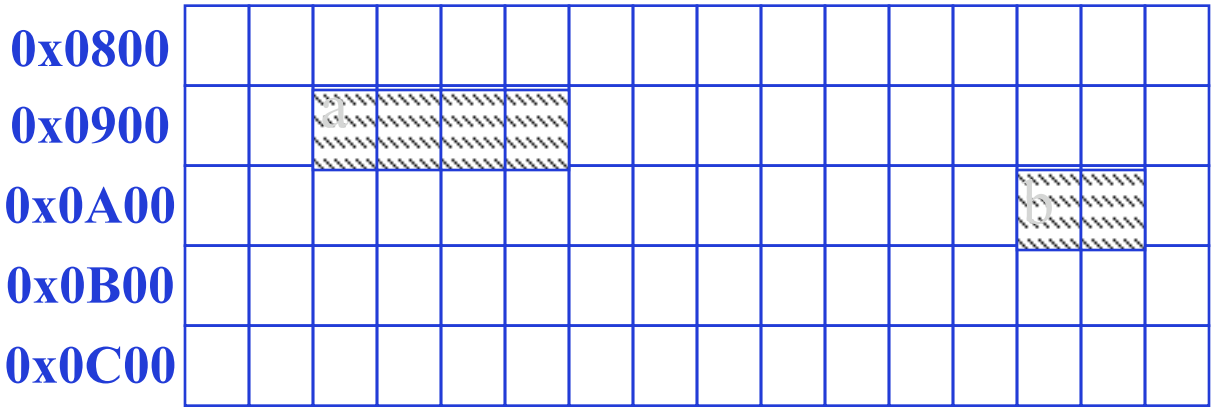
Une variable est stockée dans une zone mémoire réservée lors de la déclaration

& opérateur de référencement (adresse)

`int a;` 0x0902 pointeur sur `(int) a`

`short b;` 0x0A0D pointeur sur `(short) b`

Pointeur




```
&a : 0x0902
&b : 0x0A0D
```

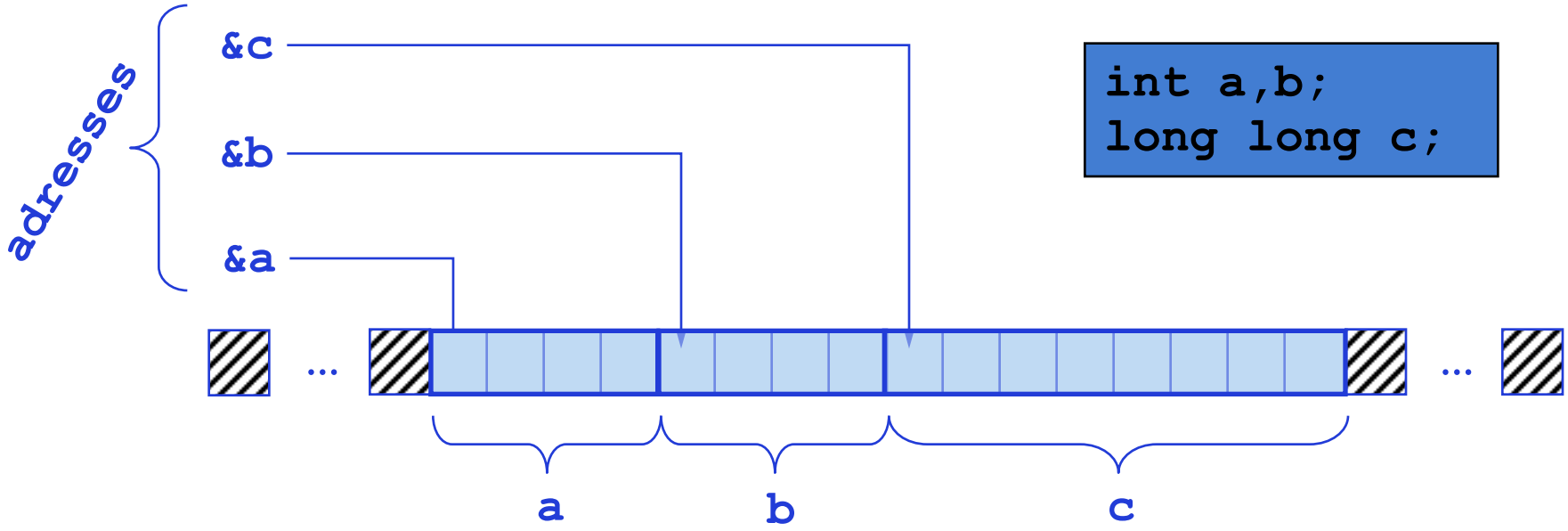
Une variable est stockée dans une zone mémoire réservée lors de la déclaration

& opérateur de référencement (adresse)

`int a;` **0x0902** pointeur sur **`(int) a`**

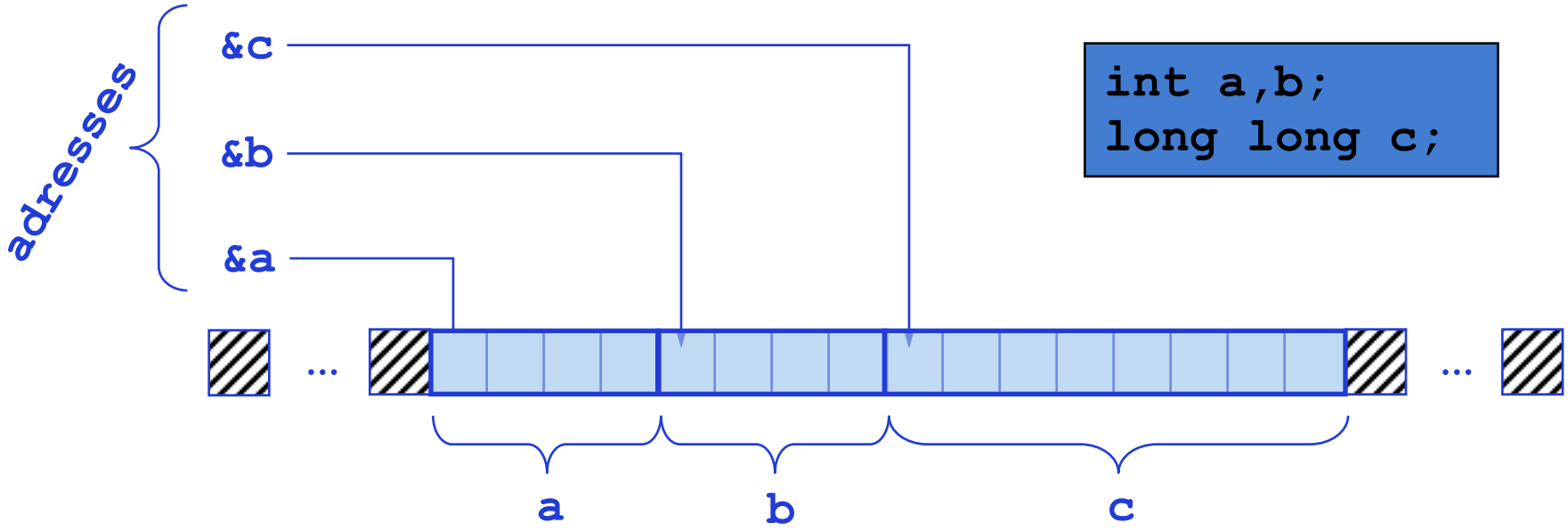
`short b;` **0x0A0D** pointeur sur **`(short) b`**

Pointeurs = Adresse



Pointeurs = Adresse

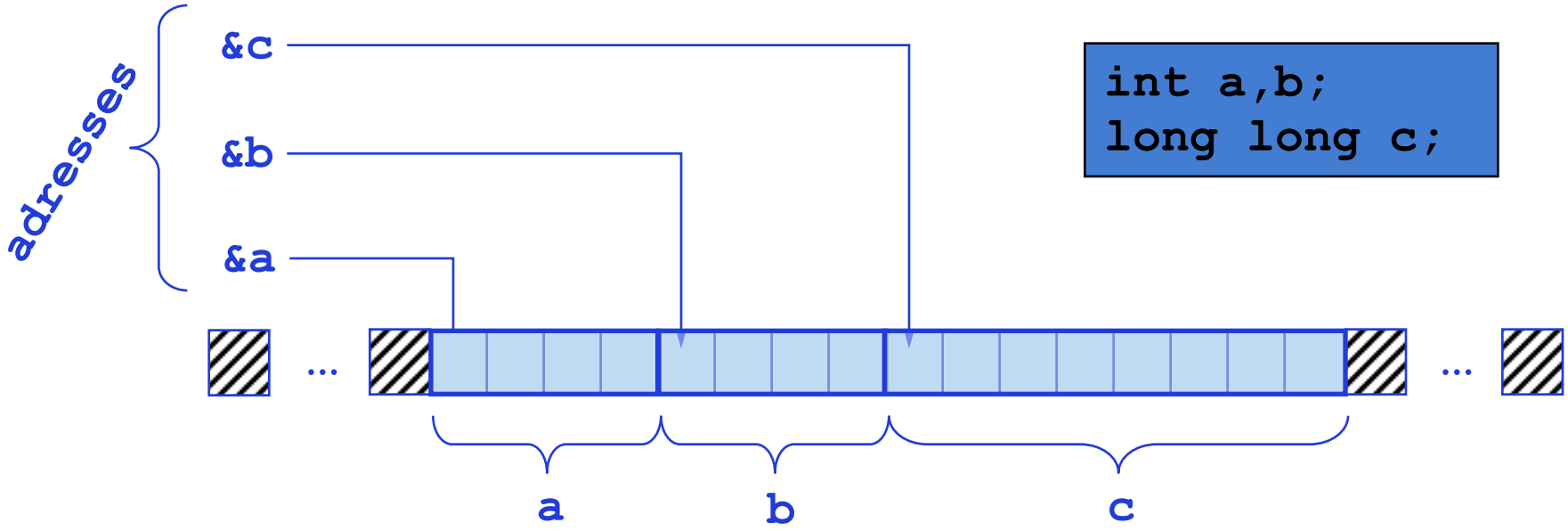
& opérateur d'adresse
(de référencement)



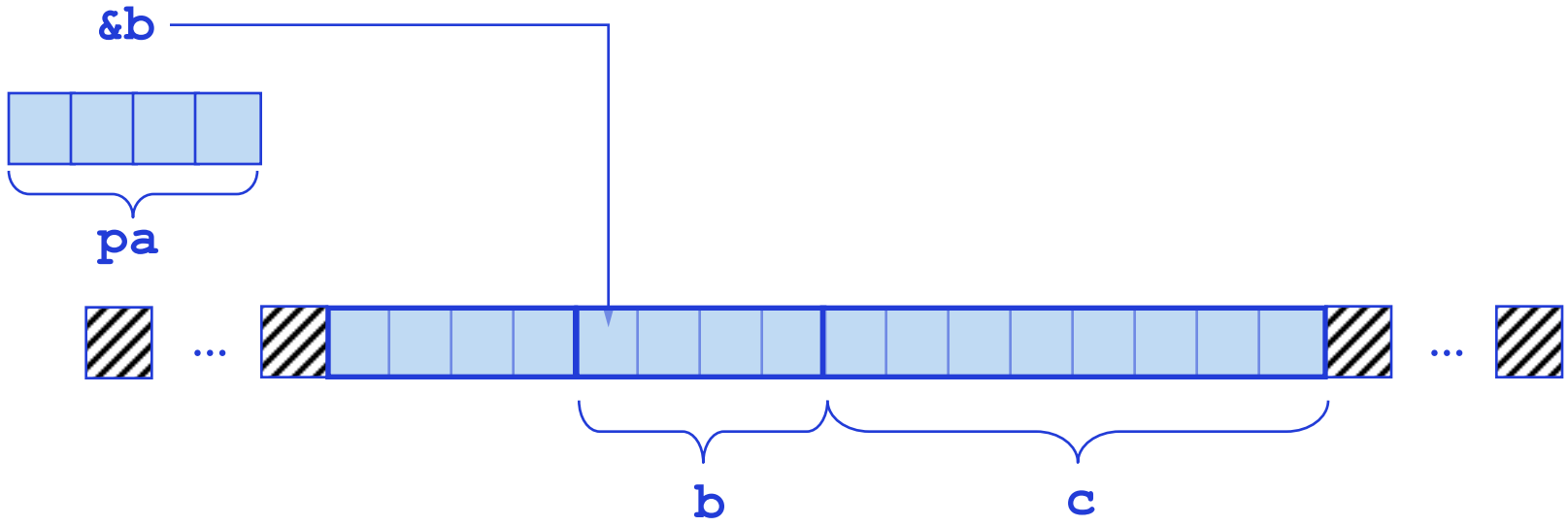
Pointeurs = Adresse

& opérateur d'adresse
(de référencement)

&a fournit l'adresse mémoire de la variable **a**

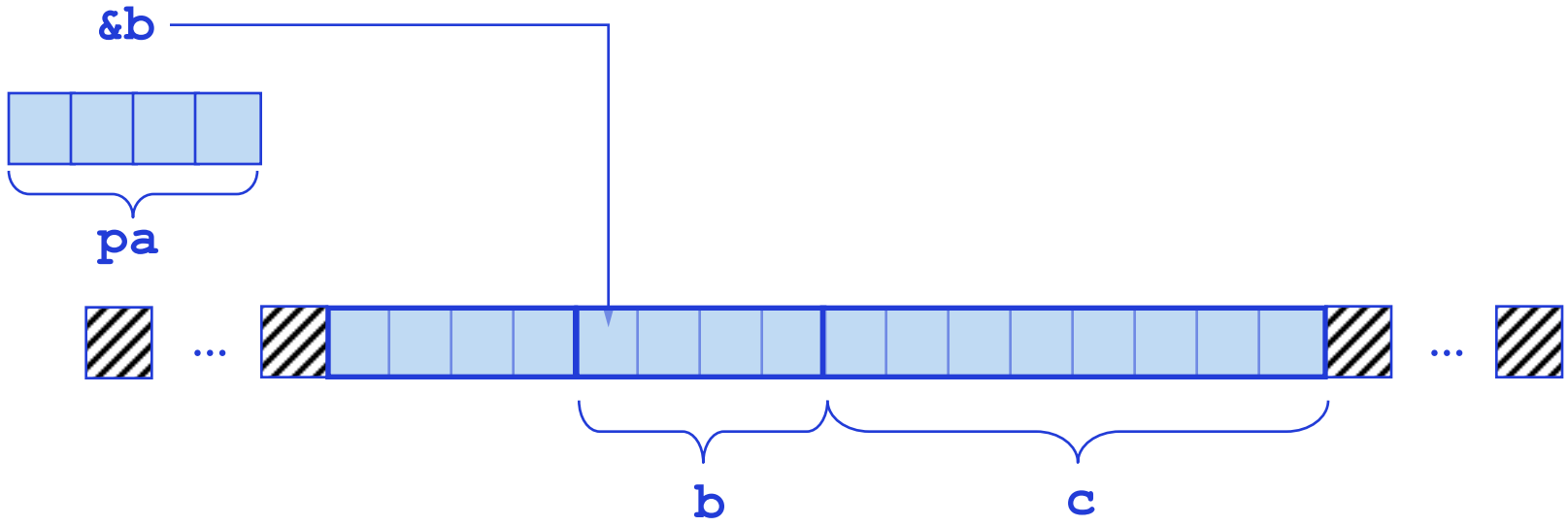


Opérateur de contenu



Opérateur de contenu

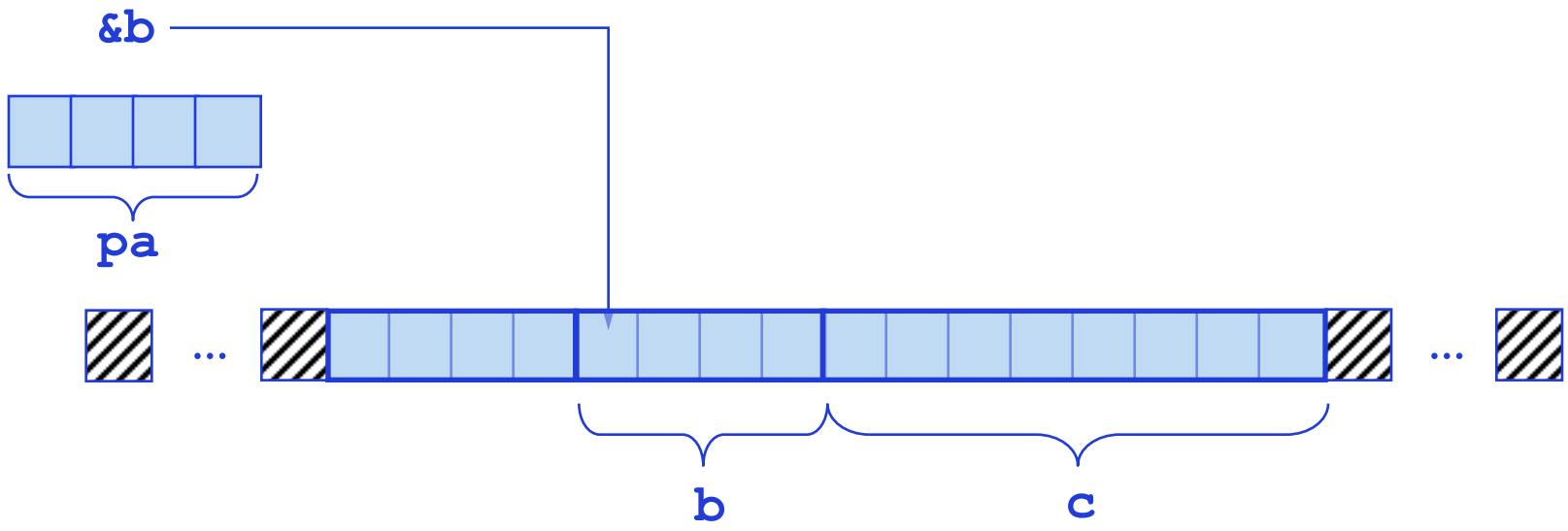
pa pointeur sur un entier **int *pa;**



Opérateur de contenu

`pa` pointeur sur un entier **`int *pa;`**

`*` opérateur de déréférencement (contenu)

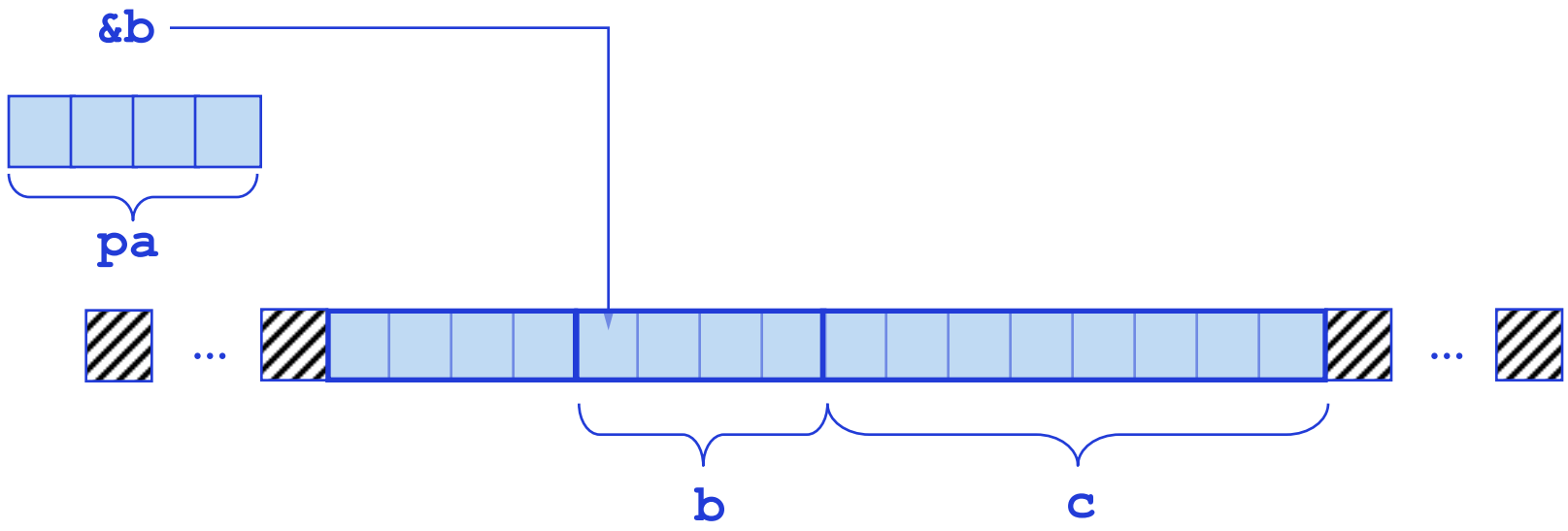


Opérateur de contenu

pa pointeur sur un entier **int *pa;**

***** opérateur de déréférencement (contenu)

***pa** est la valeur contenue à l'adresse **pa**

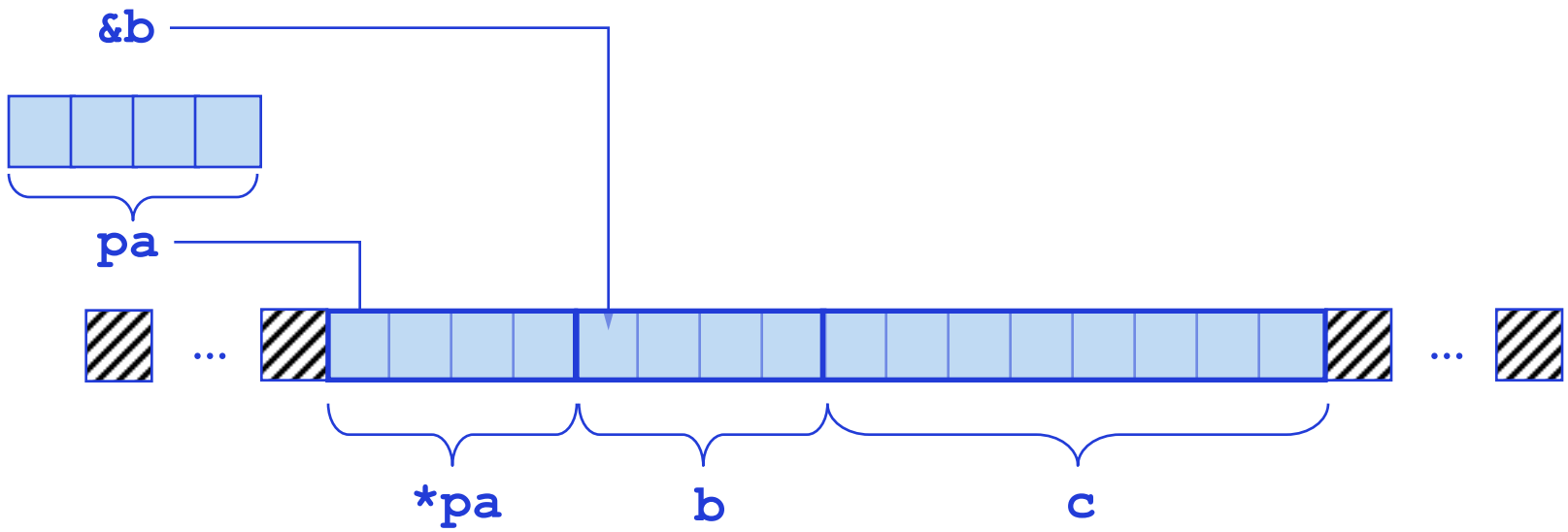


Opérateur de contenu

pa pointeur sur un entier **int *pa;**

***** opérateur de déréférencement (contenu)

***pa** est la valeur contenue à l'adresse **pa**

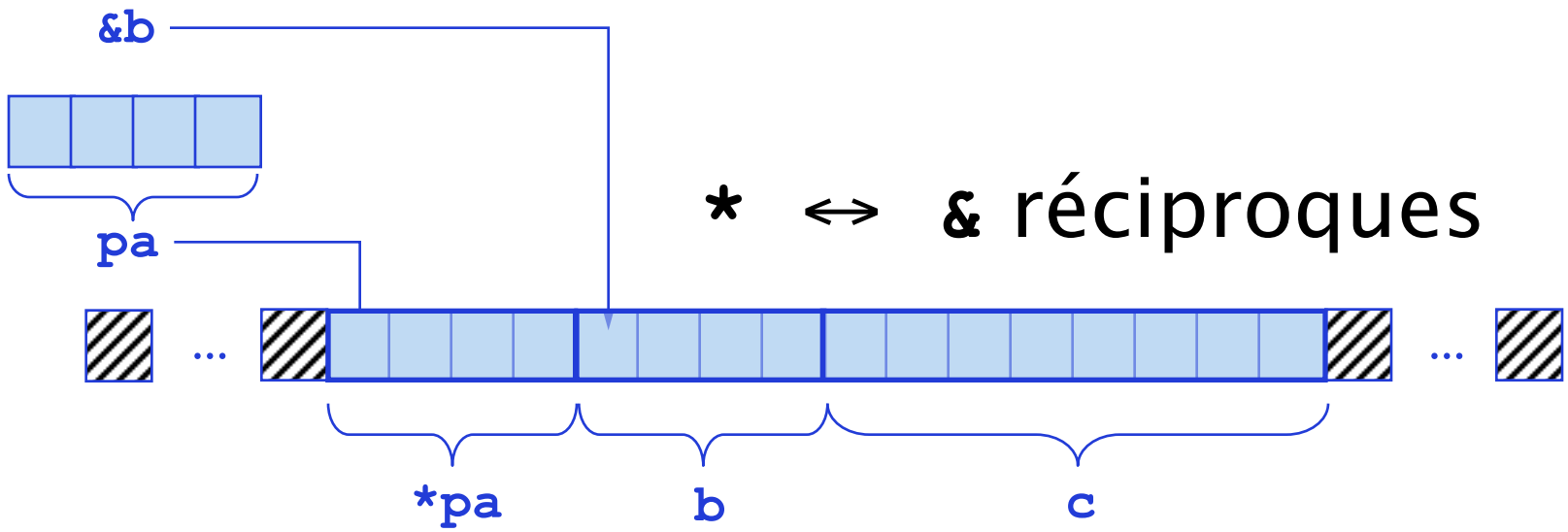


Opérateur de contenu

pa pointeur sur un entier **int *pa;**

***** opérateur de déréférencement (contenu)

***pa** est la valeur contenue à l'adresse **pa**



***** ↔ **&** réciproques

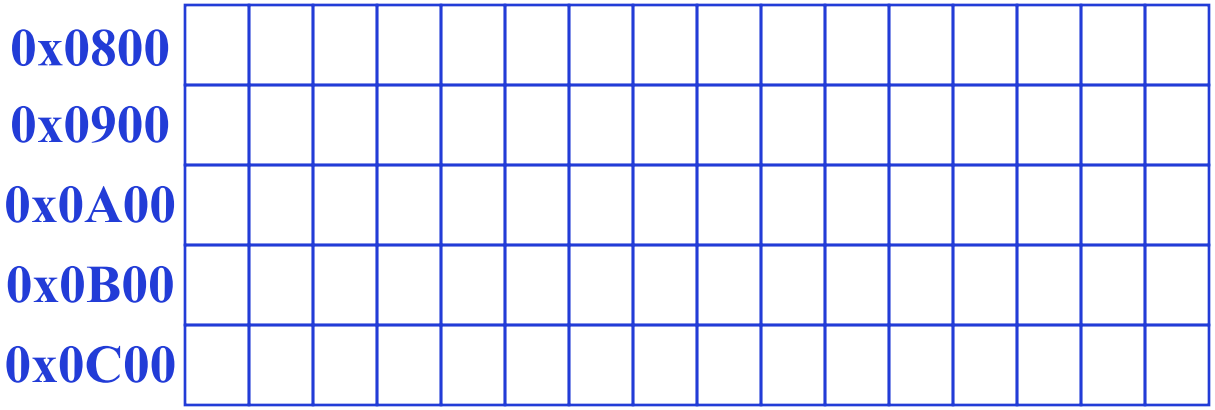
Pointeurs de tous types

```
int *pa;
```
⇒ **pa** pointeur sur un **int**

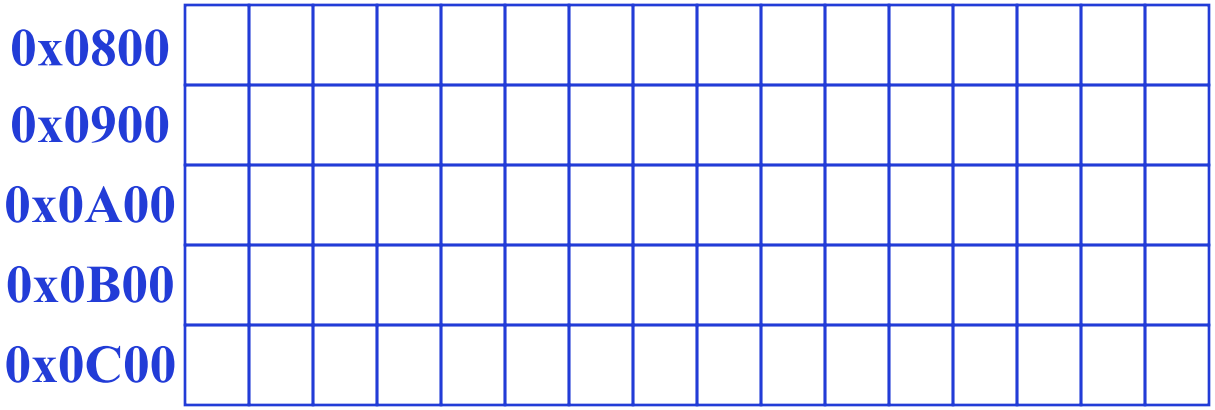
```
float *pb;
```
⇒ **pb** pointeur sur un **float**

```
char *pc;
```
⇒ **pc** pointeur sur un **char**

Quelques exemples

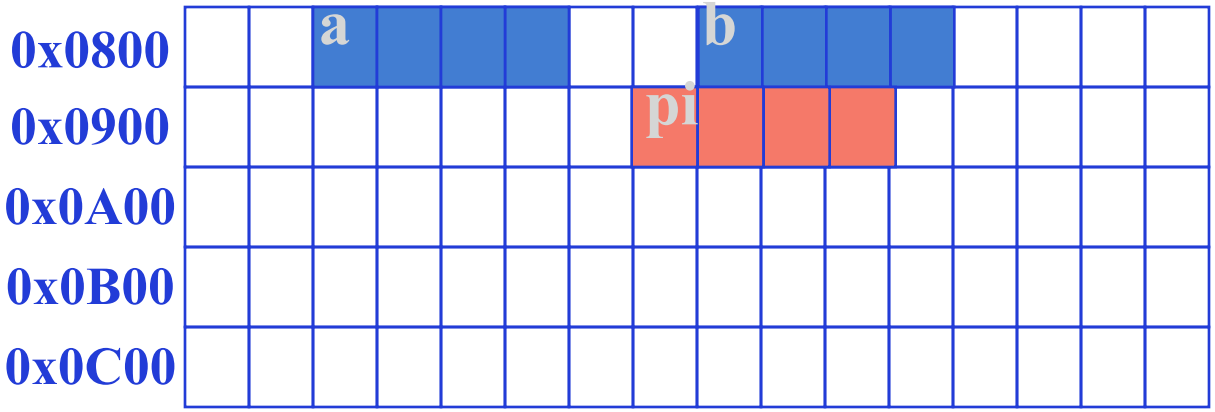


Quelques exemples



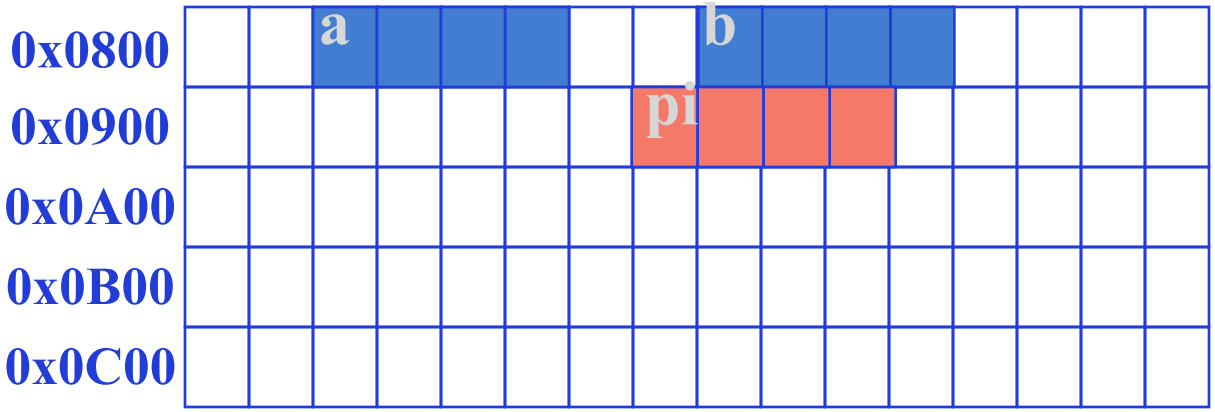

```
int a,b,*pi;
```

Quelques exemples



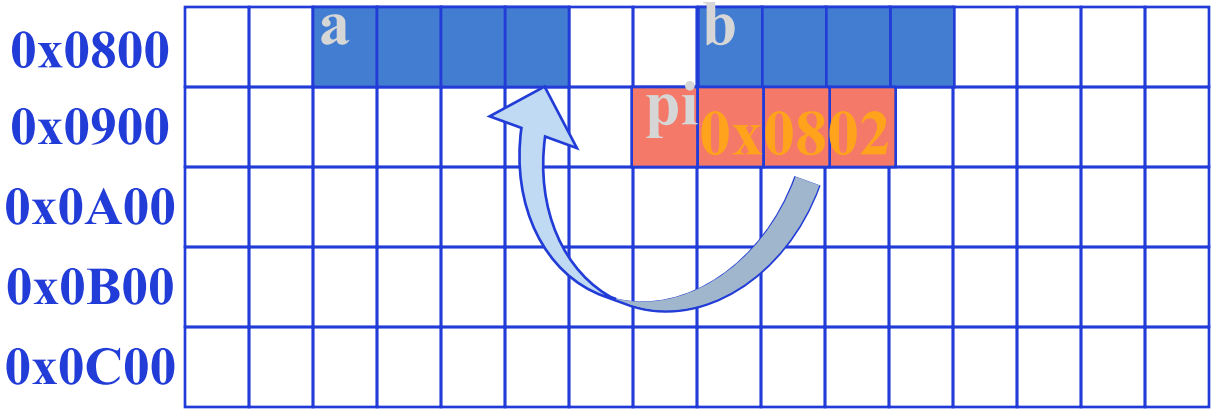

```
int a,b,*pi;
```

Quelques exemples



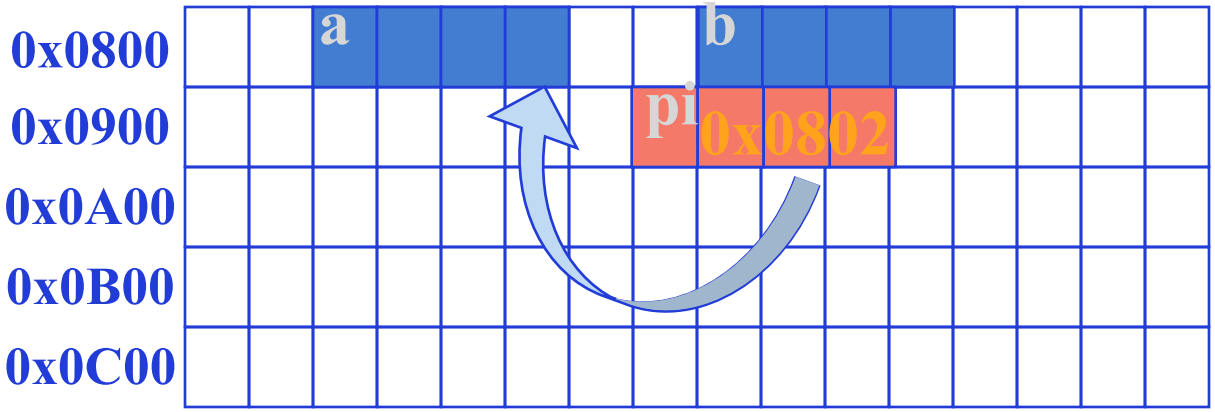

```
int a,b,*pi;
pi = &a;
```

Quelques exemples



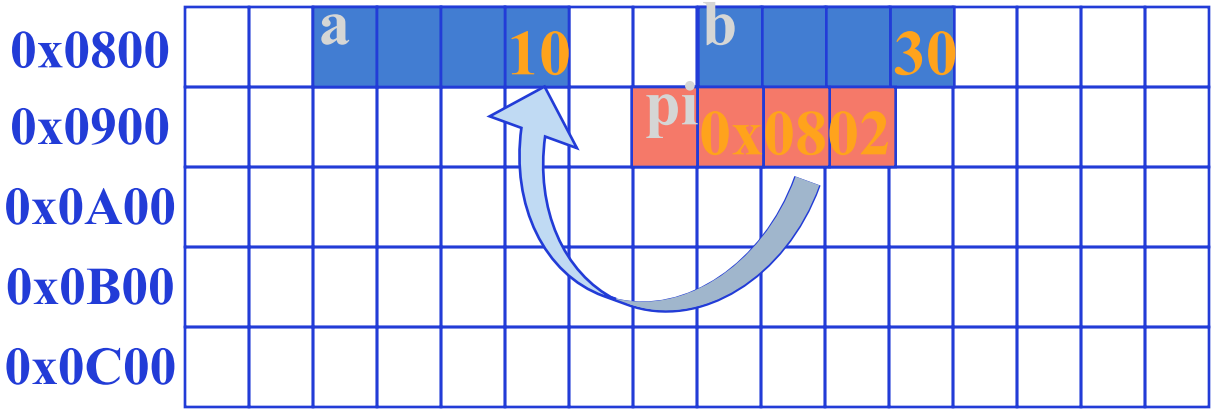

```
int a,b,*pi;
pi = &a;
```

Quelques exemples



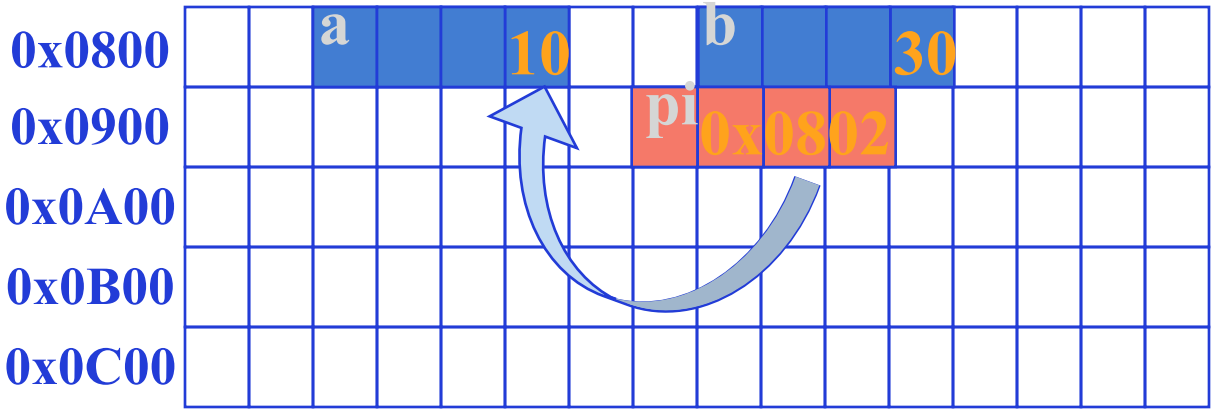

```
int a,b,*pi;
pi = &a;
a = 10; b = 30;
```

Quelques exemples



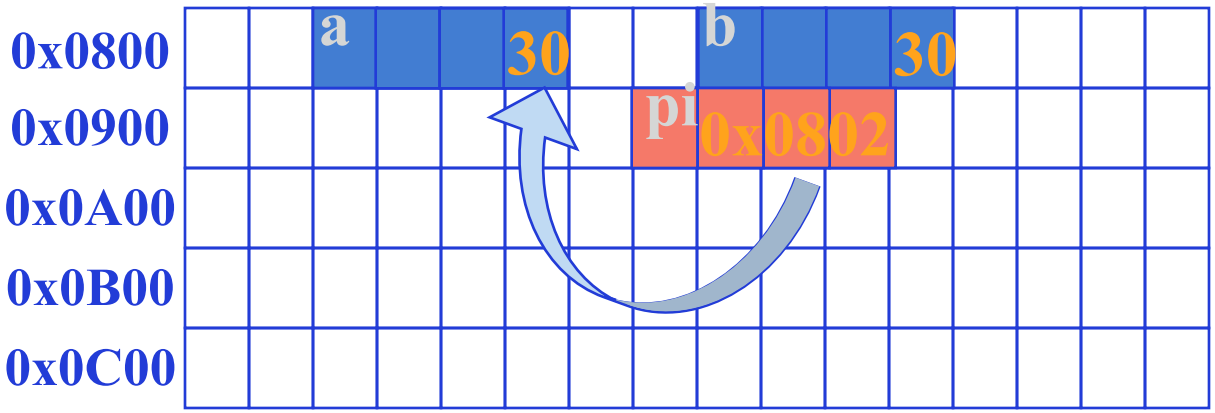

```
int a,b,*pi;
pi = &a;
a = 10; b = 30;
```

Quelques exemples



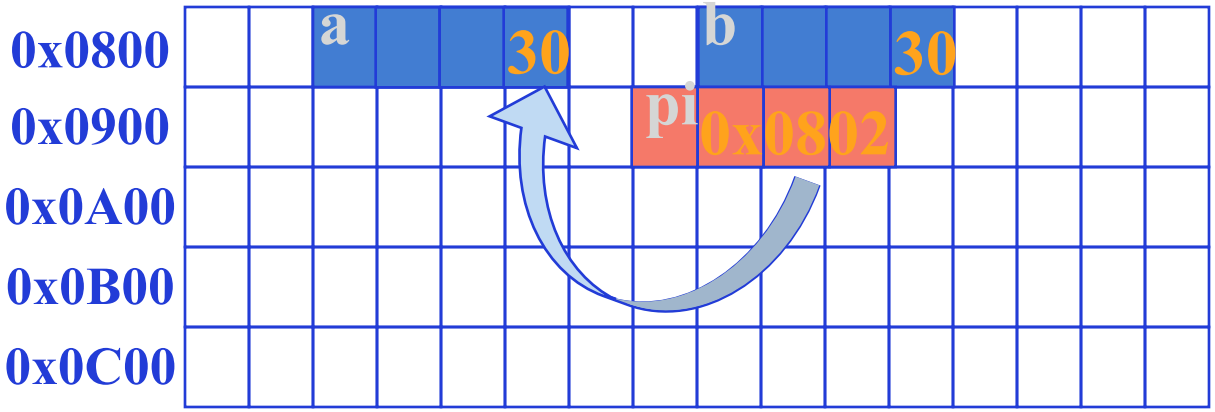

```
int a,b,*pi;
pi = &a;
a = 10; b = 30;
*pi = b;
```

Quelques exemples




```
int a,b,*pi;
pi = &a;
a = 10; b = 30;
*pi = b;
```

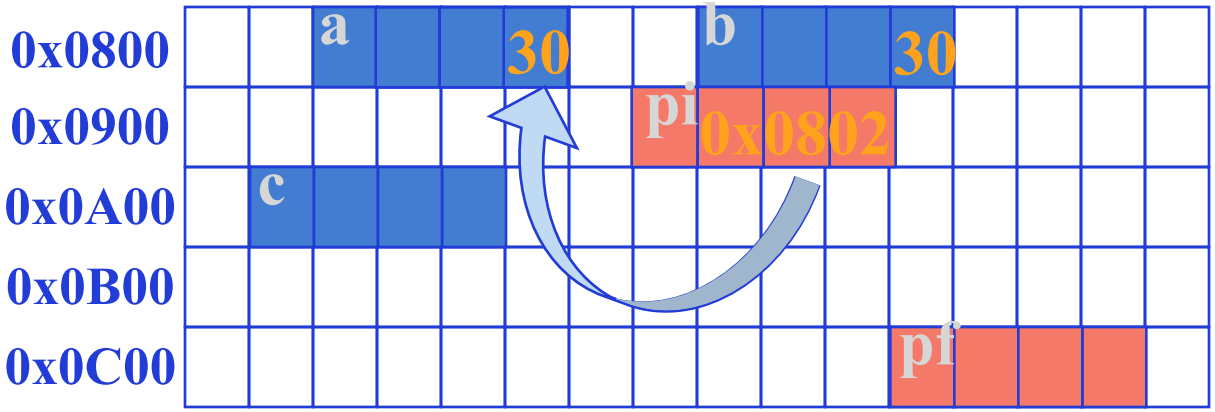
Quelques exemples




```
int a,b,*pi;
pi = &a;
a = 10; b = 30;
*pi = b;
```

```
float c,*pf;
```

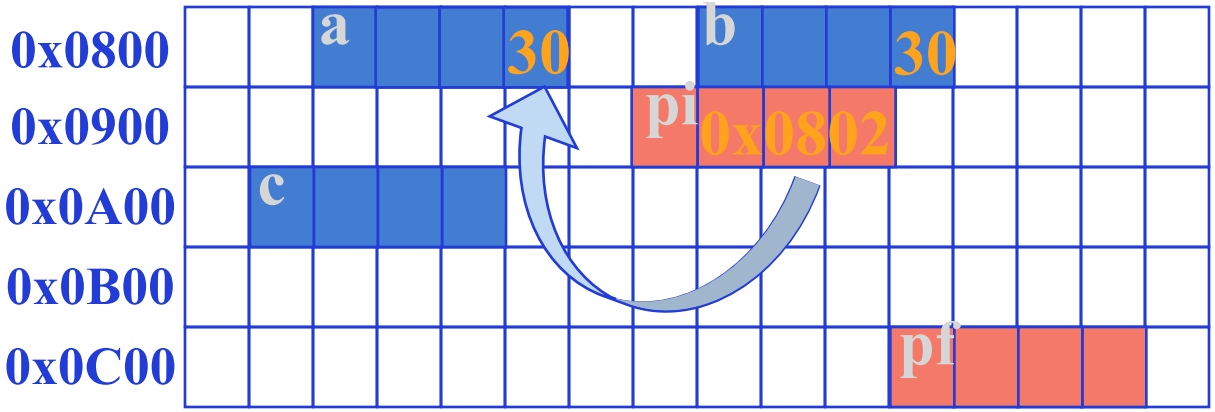
Quelques exemples




```
int a,b,*pi;
pi = &a;
a = 10; b = 30;
*pi = b;
```

```
float c,*pf;
```

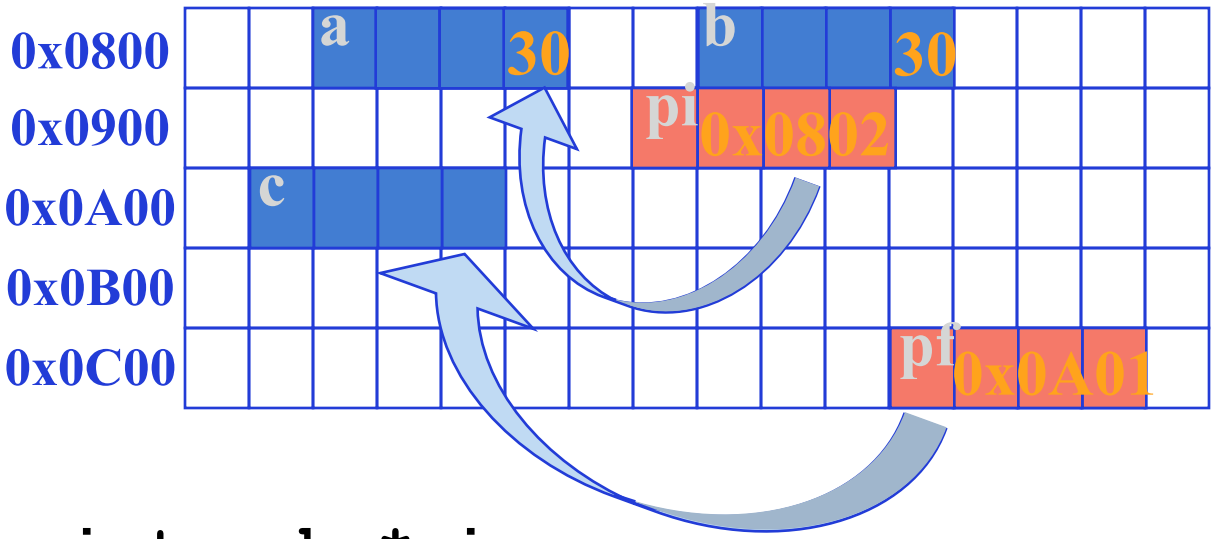
Quelques exemples




```
int a,b,*pi;
pi = &a;
a = 10; b = 30;
*pi = b;
```

```
float c,*pf;
pf = &c;
```

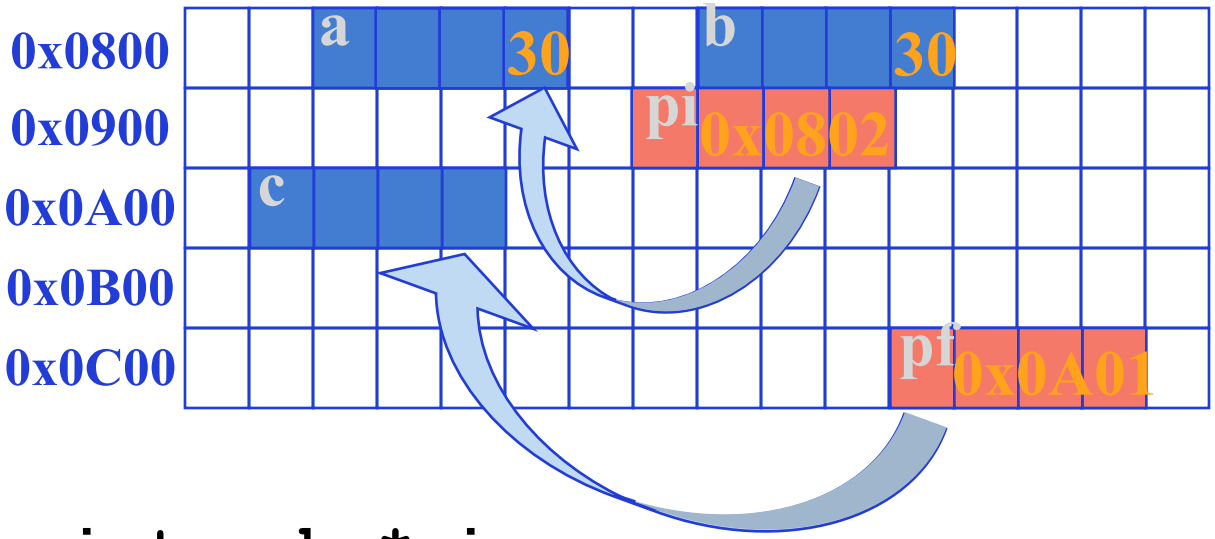
Quelques exemples




```
int a,b,*pi;
pi = &a;
a = 10; b = 30;
*pi = b;
```

```
float c,*pf;
pf = &c;
```

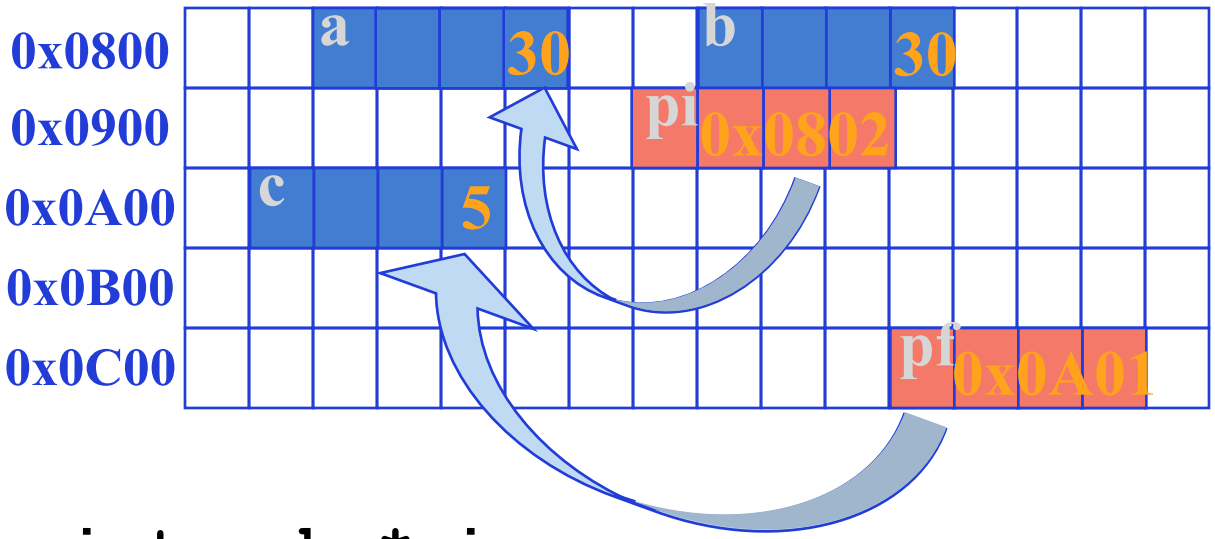
Quelques exemples




```
int a,b,*pi;
pi = &a;
a = 10; b = 30;
*pi = b;
```

```
float c,*pf;
pf = &c;
c = 5;
```

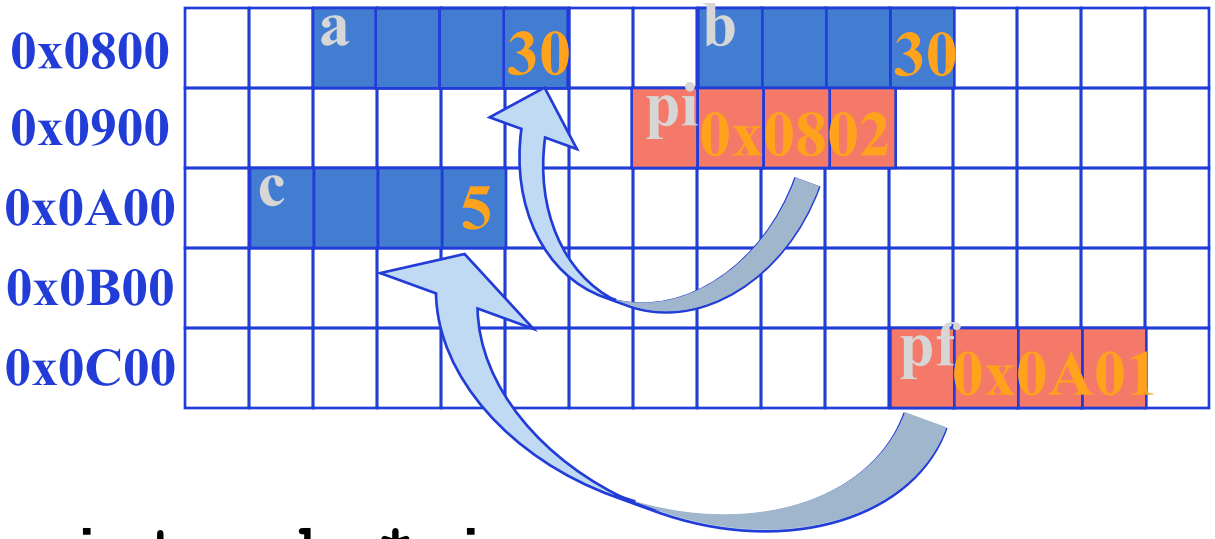
Quelques exemples




```
int a,b,*pi;
pi = &a;
a = 10; b = 30;
*pi = b;
```

```
float c,*pf;
pf = &c;
c = 5;
```

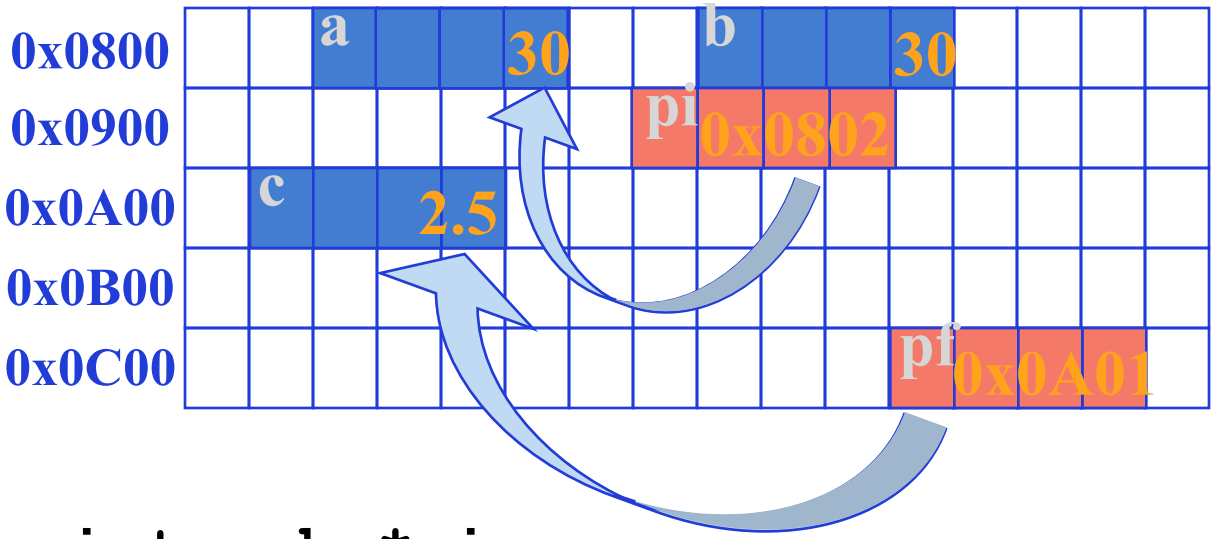
Quelques exemples




```
int a,b,*pi;
pi = &a;
a = 10; b = 30;
*pi = b;
```

```
float c,*pf;
pf = &c;
c = 5;
*pf = *pf/2;
```

Quelques exemples




```
int a,b,*pi;
pi = &a;
a = 10; b = 30;
*pi = b;
```

```
float c,*pf;
pf = &c;
c = 5;
*pf = *pf/2;
```

Taille d'un pointeur

En général, les pointeurs sont typés :

- `int *pa;` pointeur sur un `int`
- `float *pb;` pointeur sur un `float`

Mais tout pointeur est une adresse mémoire, soit 32 bits (GCC Linux)

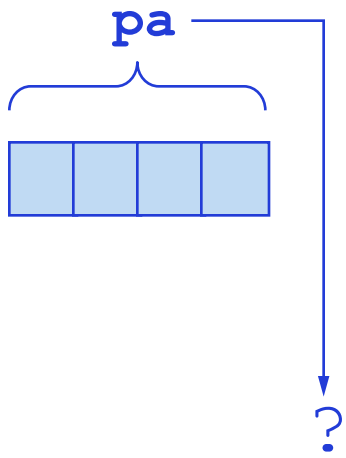
⇒ dans la plupart des cas, tous les pointeurs seront équivalents

Allocation dynamique

```
int *pa;
```

Allocation dynamique

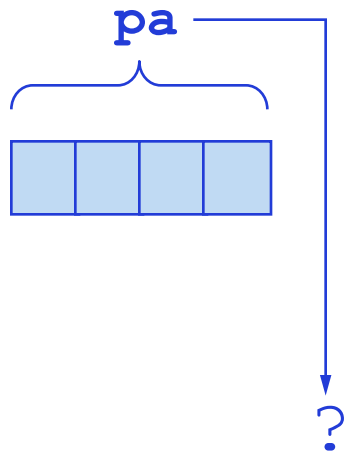
```
int *pa;
```



Allocation dynamique

```
int *pa;
```

⇒ déclaration d'un pointeur **pa** sur un **int**

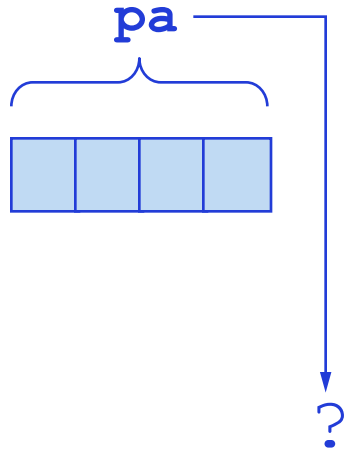


Allocation dynamique

```
int *pa;
```

⇒ déclaration d'un pointeur **`pa`** sur un **`int`**

- **réservation** d'une zone mémoire pour stocker une adresse

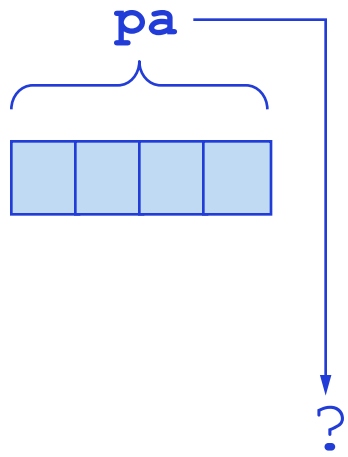


Allocation dynamique

```
int *pa;
```

⇒ déclaration d'un pointeur **`pa`** sur un **`int`**

- **réservation** d'une zone mémoire pour stocker une adresse
- pas d'initialisation

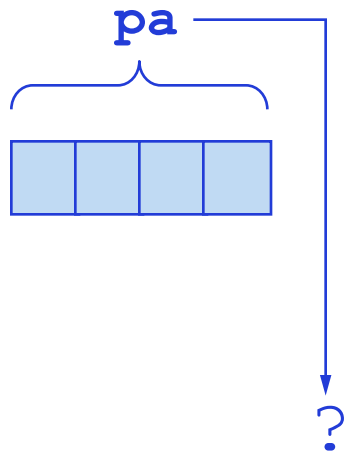


Allocation dynamique

```
int *pa;
```

⇒ déclaration d'un pointeur **`pa`** sur un **`int`**

- **réservation** d'une zone mémoire pour stocker une adresse
- pas d'initialisation
- pas d'entier pointé



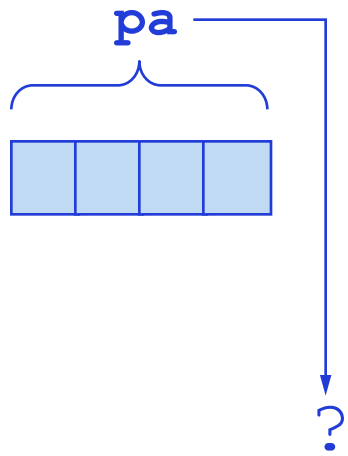
Allocation dynamique

```
int *pa;
```

⇒ déclaration d'un pointeur **`pa`** sur un **`int`**

- **réservation** d'une zone mémoire pour stocker une adresse
- pas d'initialisation
- pas d'entier pointé

  ⇒ il faut réserver la zone mémoire pour cet entier : allocation puis affecter le pointeur



Allocation dynamique : `malloc`

```
int *pa;
```

⇒ déclaration d'un pointeur **`pa`** sur un **`int`**

Allocation dynamique : `malloc`

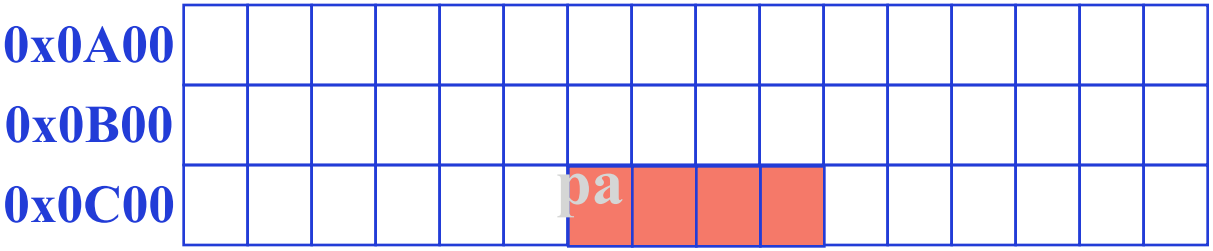
```
int *pa;
```

⇒ déclaration d'un pointeur **`pa`** sur un **`int`**

Allocation dynamique : `malloc`

```
int *pa;
```

⇒ déclaration d'un pointeur **`pa`** sur un **`int`**

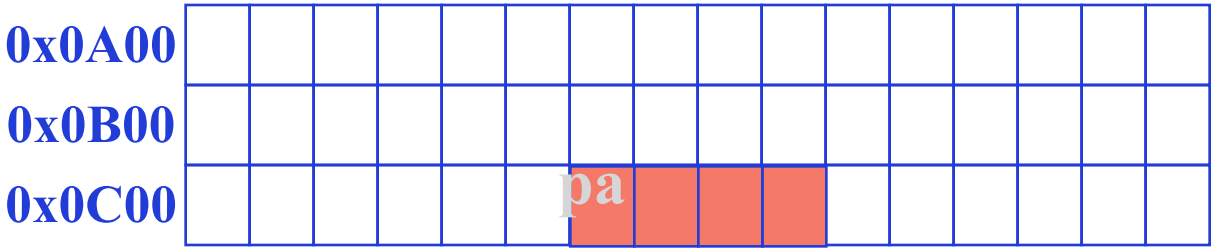


Allocation dynamique : `malloc`

```
int *pa;
```

⇒ déclaration d'un pointeur **`pa`** sur un **`int`**

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

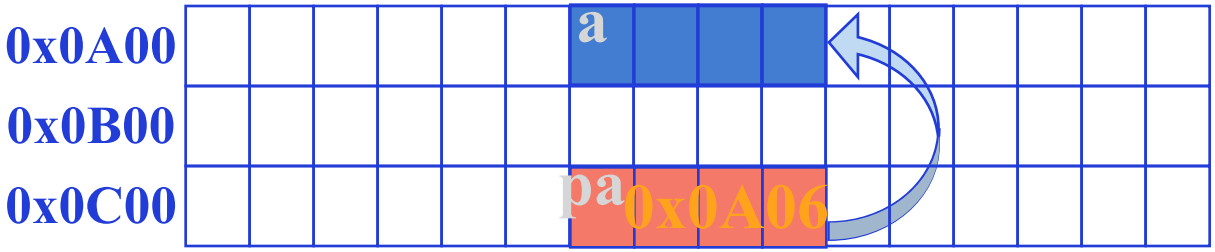


Allocation dynamique : `malloc`

```
int *pa;
```

⇒ déclaration d'un pointeur **`pa`** sur un **`int`**

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

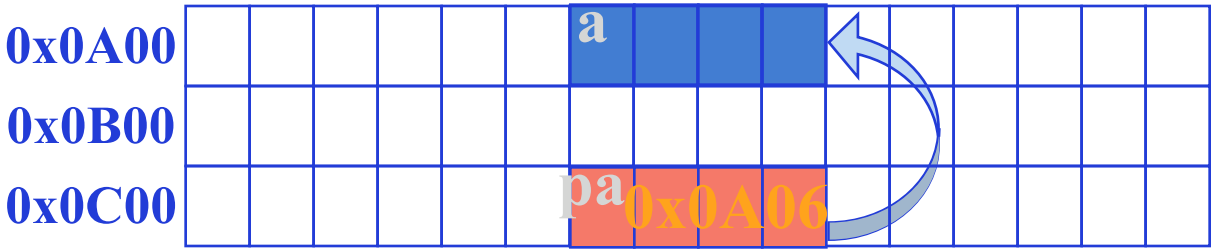


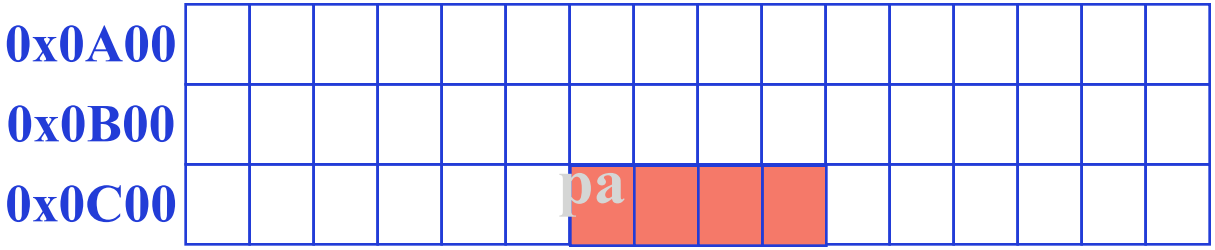
Allocation dynamique : `malloc`

```
int *pa;
```

⇒ déclaration d'un pointeur **`pa`** sur un **`int`**

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



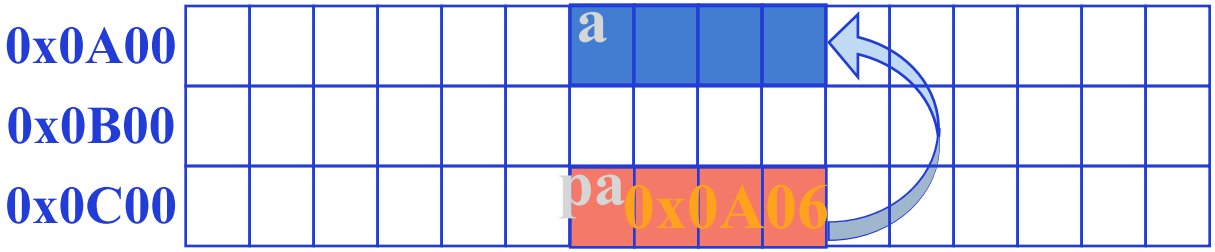


Allocation dynamique : `malloc`

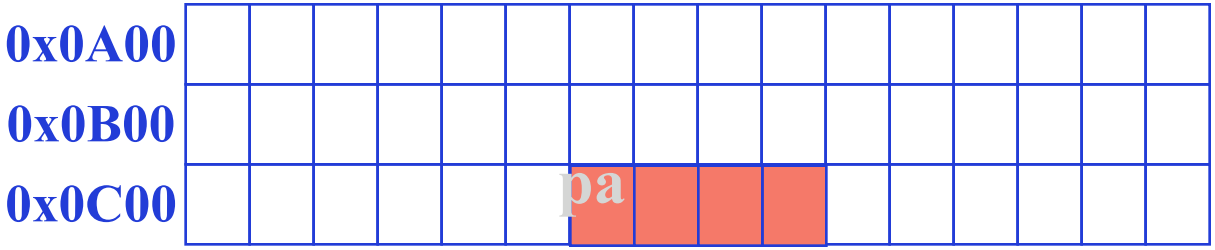
```
int *pa;
```

⇒ déclaration d'un pointeur **`pa`** sur un **`int`**

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




```
pa = malloc(sizeof(int));
```

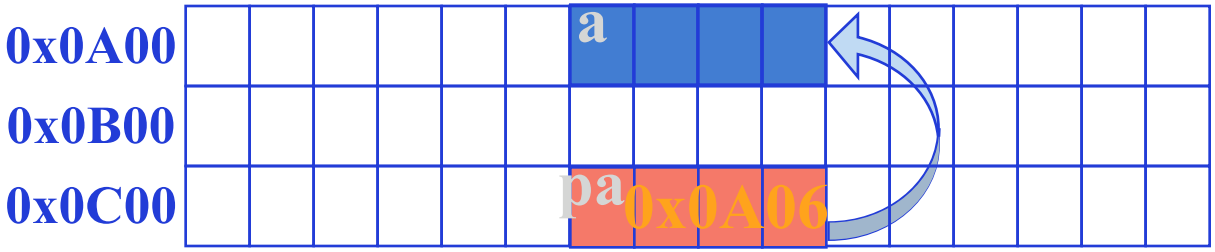


Allocation dynamique : `malloc`

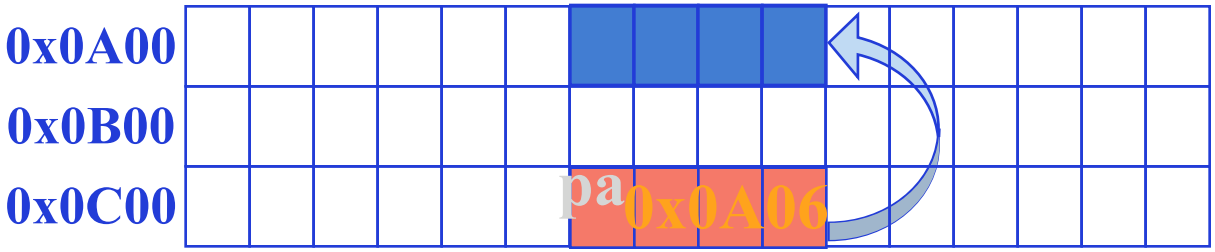
```
int *pa;
```

⇒ déclaration d'un pointeur **`pa`** sur un **`int`**

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




```
pa = malloc(sizeof(int));
```



Utilisation de `malloc`

```
int *pa;
pa = malloc(sizeof(int));
```

Utilisation de `malloc`

```
int *pa;
pa = malloc(sizeof(int));
```

```
void *malloc(int n)
```

Utilisation de `malloc`

```
int *pa;
pa = malloc(sizeof(int));
```

`void *malloc(int n)`

- allocation de n octets de mémoire, l'adresse de cette zone est retournée

Utilisation de `malloc`

```
int *pa;
pa = malloc(sizeof(int));
```

`void *malloc(int n)`

- allocation de n octets de mémoire, l'adresse de cette zone est retournée
- pointeur générique, **`void *`**

Utilisation de `malloc`

```
int *pa;
pa = malloc(sizeof(int));
```

`void *malloc(int n)`

- allocation de n octets de mémoire, l'adresse de cette zone est retournée
- pointeur générique, **`void *`**

  $\Rightarrow$ il est automatiquement « casté » en pointeur typé (dès que possible)

Taille d'un objet `sizeof`

La taille d'un type n'est pas standardisée

⇒ pour la portabilité d'un programme C, utiliser **`sizeof(<type>)`**

GCC Linux:

- **`sizeof(int)`** → **4**
- **`sizeof(char)`** → **1**
- …

Allocation dynamique : tableaux

```
int *pa;
```

⇒ déclaration d'un pointeur **`pa`** sur un **`int`**

Allocation dynamique : tableaux

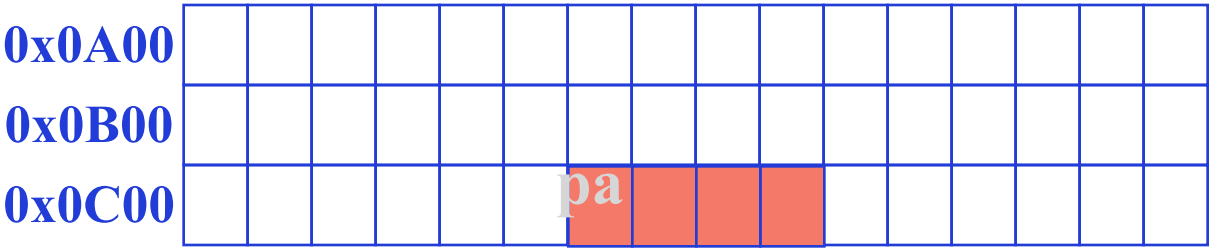
```
int *pa;
```

⇒ déclaration d'un pointeur **pa** sur un **int**

Allocation dynamique : tableaux

```
int *pa;
```

⇒ déclaration d'un pointeur **pa** sur un **int**

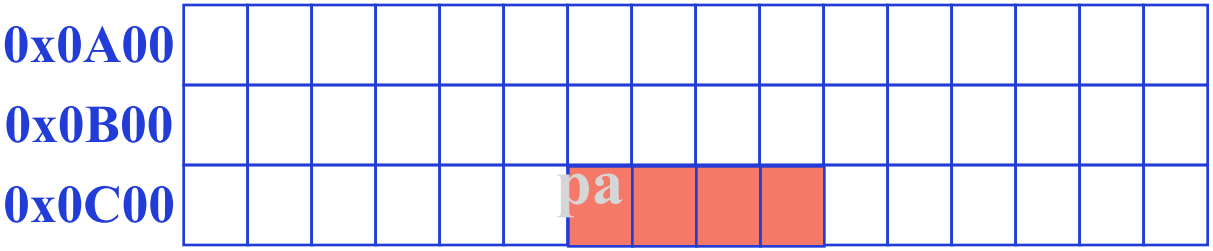


Allocation dynamique : tableaux

```
int *pa;
```

⇒ déclaration d'un pointeur **pa** sur un **int**

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

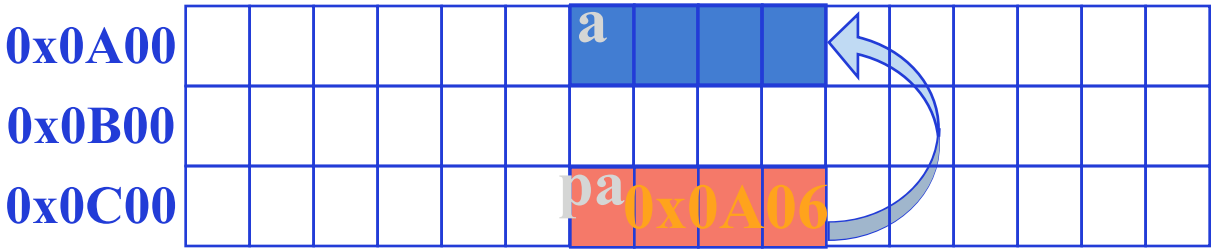


Allocation dynamique : tableaux

```
int *pa;
```

⇒ déclaration d'un pointeur **pa** sur un **int**

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

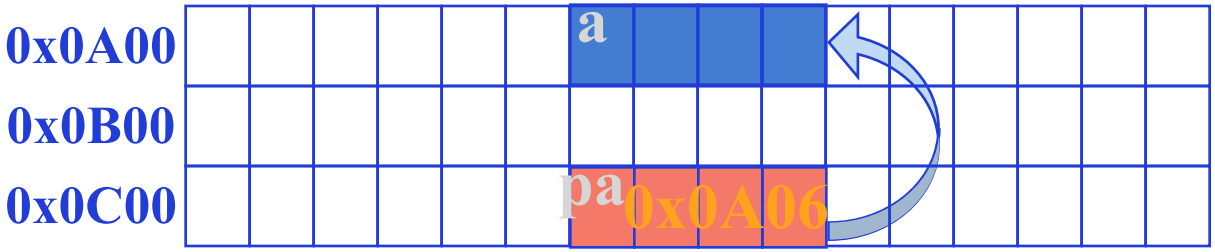


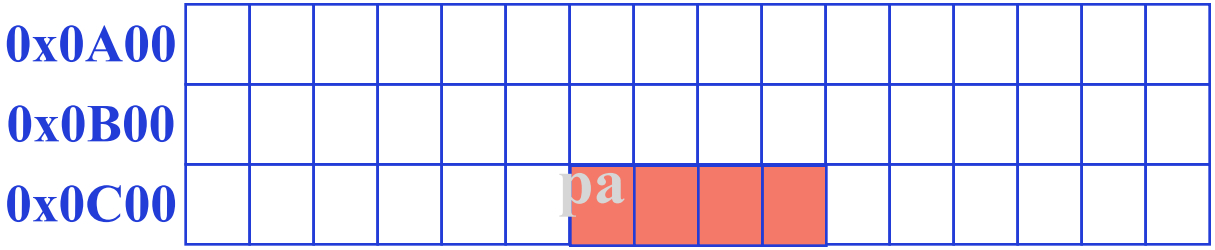
Allocation dynamique : tableaux

```
int *pa;
```

⇒ déclaration d'un pointeur **pa** sur un **int**

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



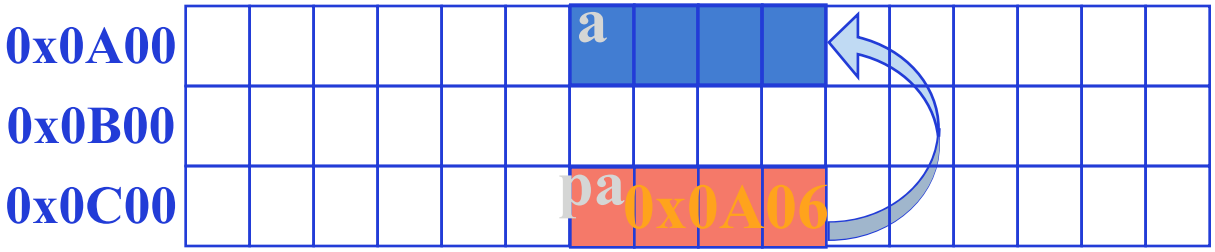


Allocation dynamique : tableaux

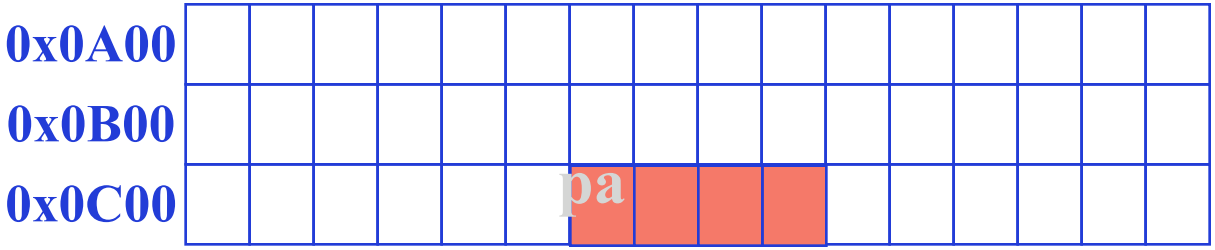
```
int *pa;
```

⇒ déclaration d'un pointeur **pa** sur un **int**

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




```
pa = malloc(3*(sizeof(int)));
```

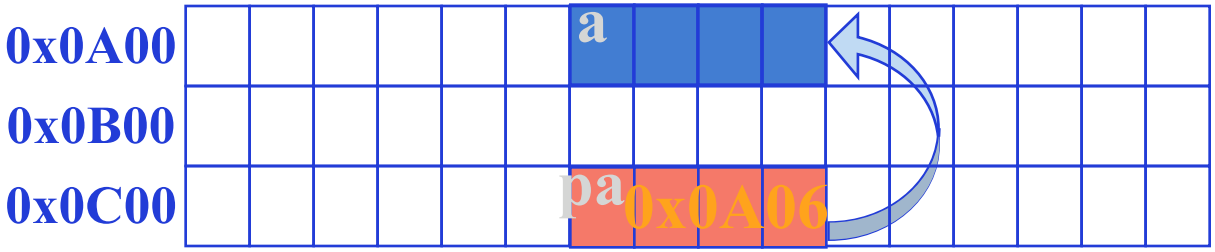


Allocation dynamique : tableaux

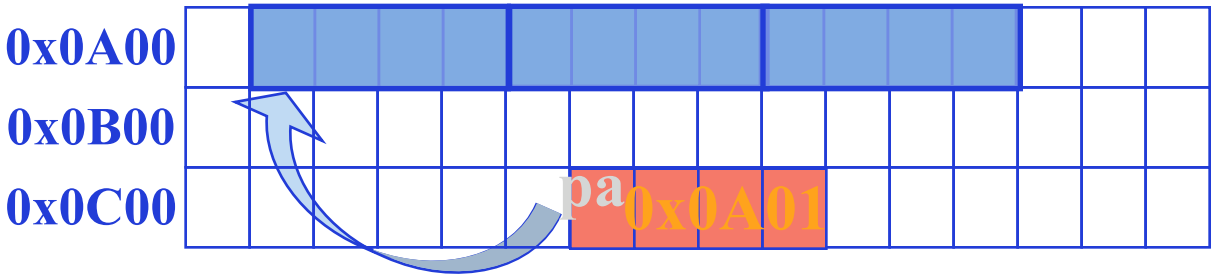
```
int *pa;
```

⇒ déclaration d'un pointeur **pa** sur un **int**

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




```
pa = malloc(3*(sizeof(int)));
```



Tableaux dynamiques

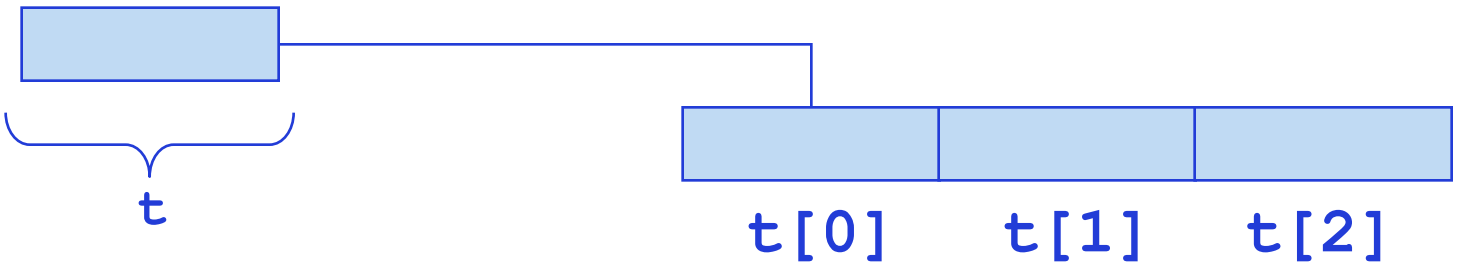
```
int t[3];
```

⇒ tableau de 3 entiers

Tableaux dynamiques

```
int t[3];
```

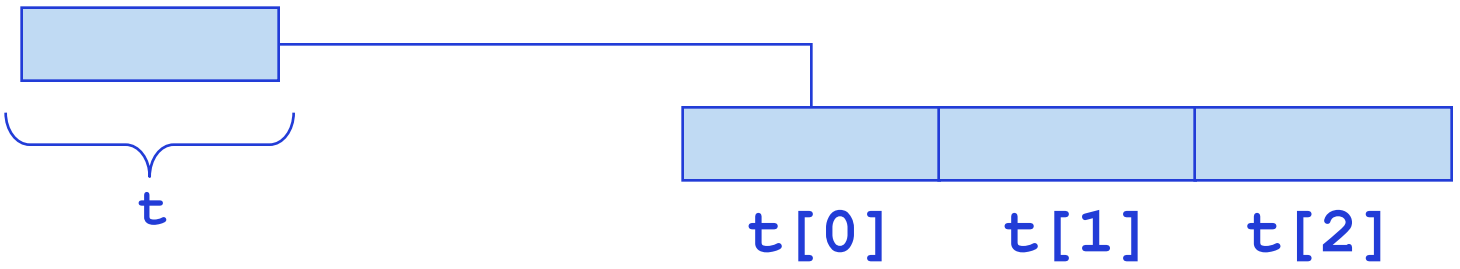
⇒ tableau de 3 entiers



Tableaux dynamiques

```
int t[3];
```

⇒ tableau de 3 entiers

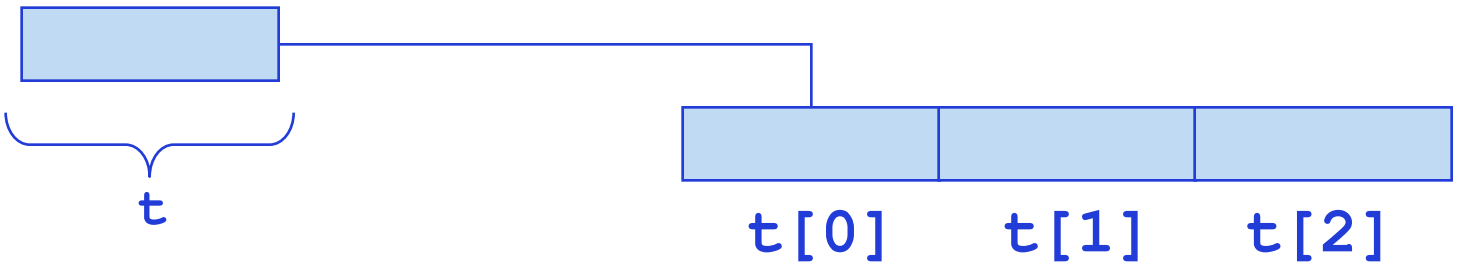


équivalent à

Tableaux dynamiques

```
int t[3];
```

⇒ tableau de 3 entiers



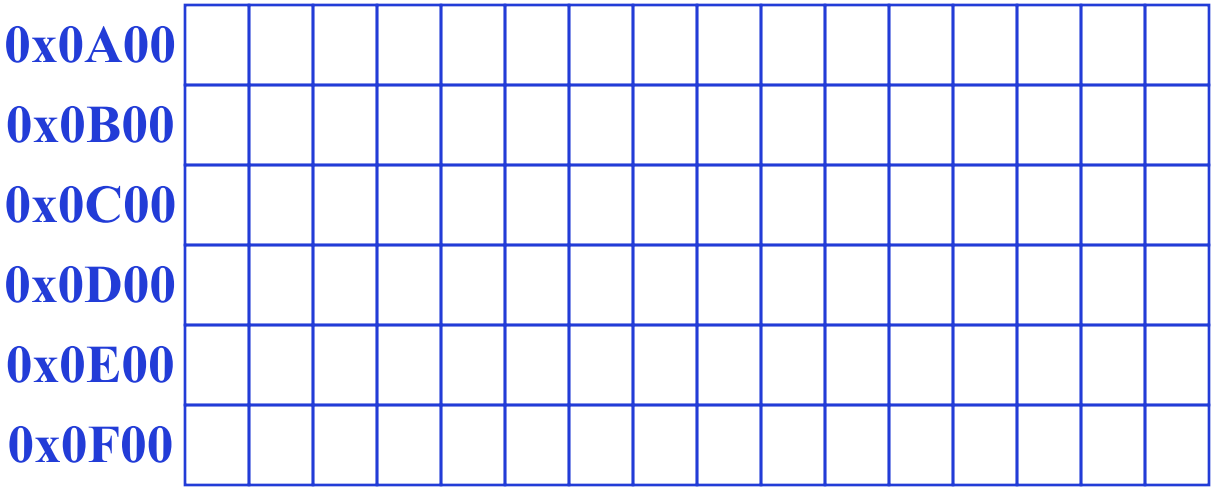
équivalent à

```
int *t = malloc(3*(sizeof(int)));
```

Tableaux dynamiques (suite)

```
float *Tf;
```

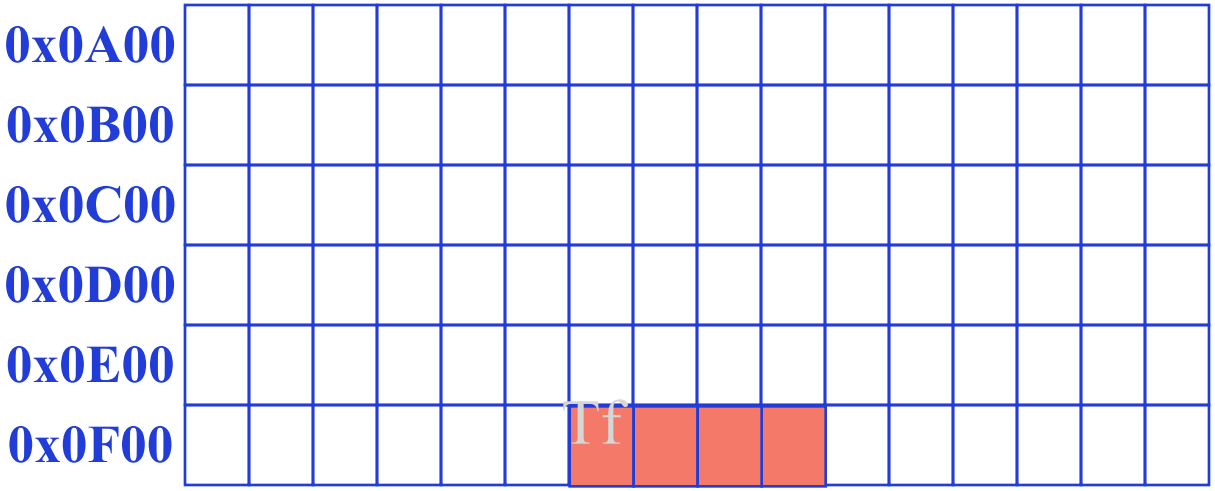
⇒ déclaration d'un pointeur `Tf` sur un `float`



Tableaux dynamiques (suite)

```
float *Tf;
```

⇒ déclaration d'un pointeur `Tf` sur un `float`

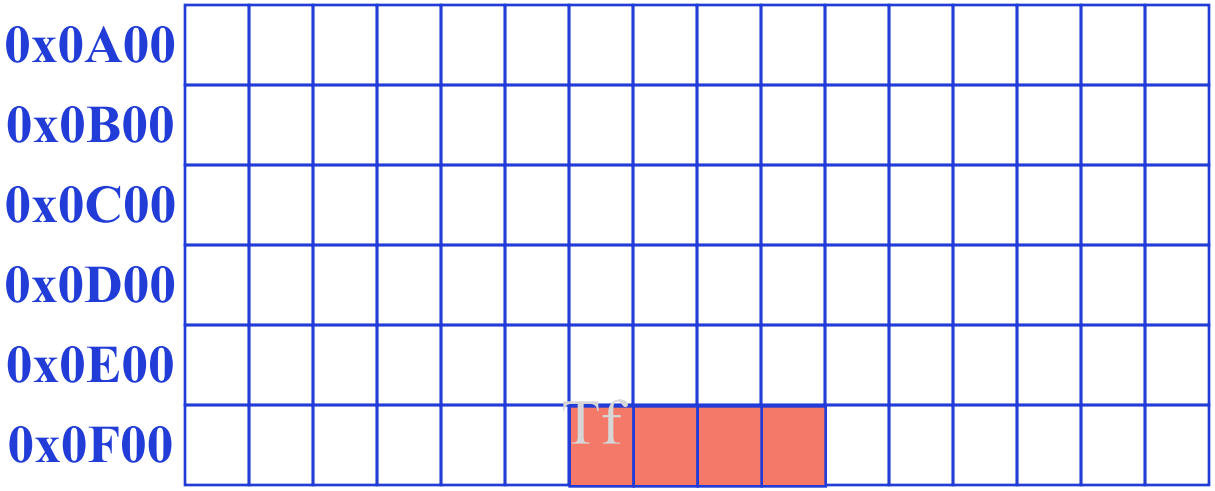


Tableaux dynamiques (suite)

```
float *Tf;
```

⇒ déclaration d'un pointeur `Tf` sur un `float`

```
Tf = malloc(6*(sizeof(float)));
```

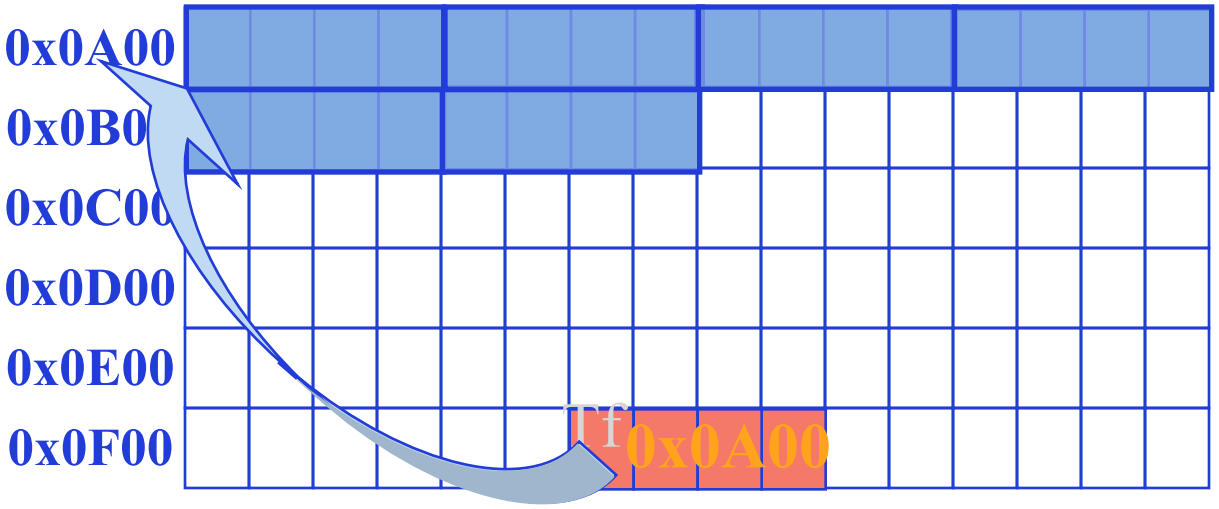


Tableaux dynamiques (suite)

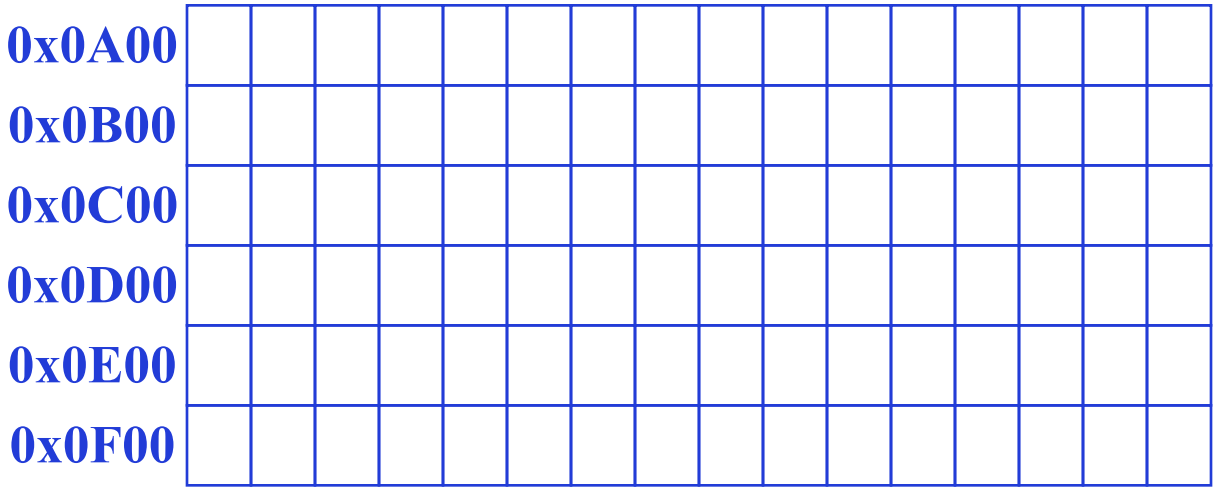
```
float *Tf;
```

⇒ déclaration d'un pointeur `Tf` sur un `float`

```
Tf = malloc(6*(sizeof(float)));
```

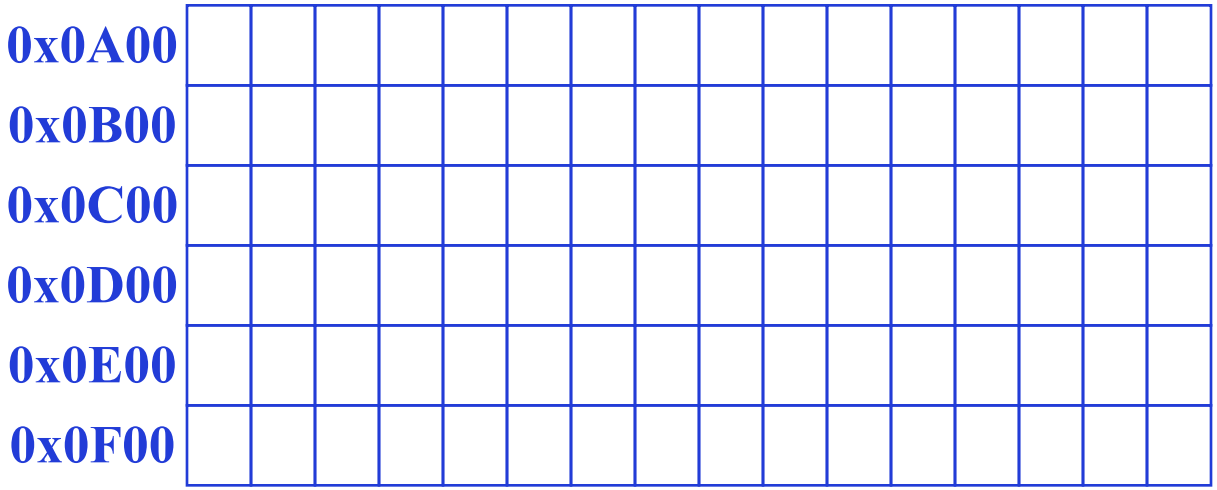


Tableaux dynamiques (suite)



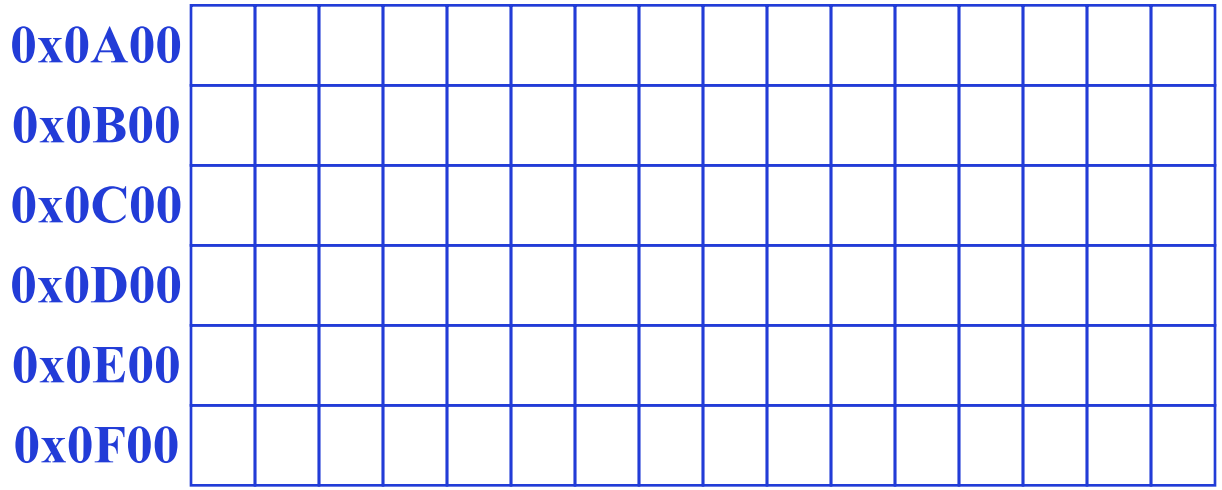
Tableaux dynamiques (suite)

```
float *Tf;
```



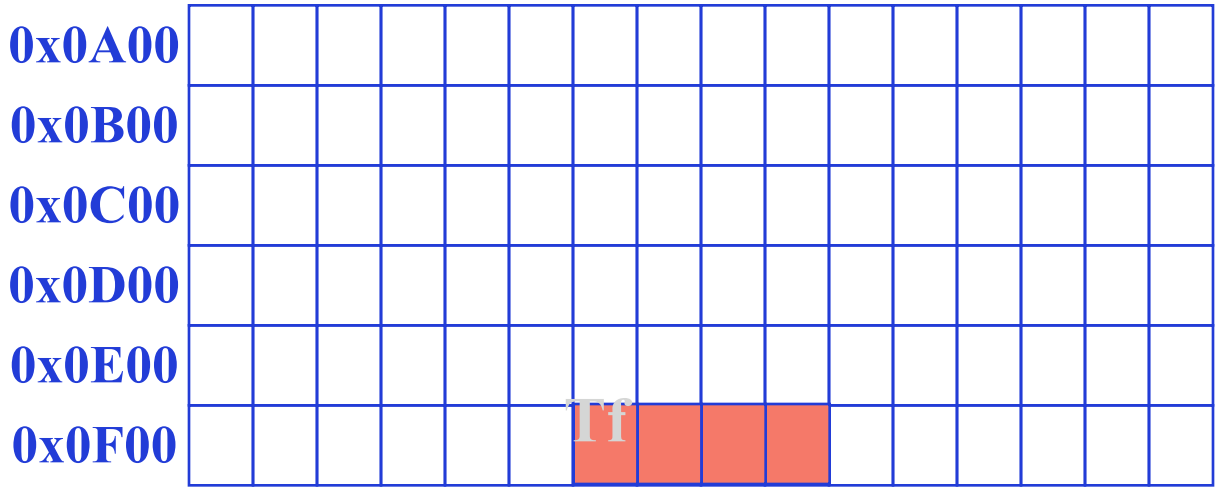
Tableaux dynamiques (suite)

```
float *Tf;
Tf = malloc(6*(sizeof(float)));
```



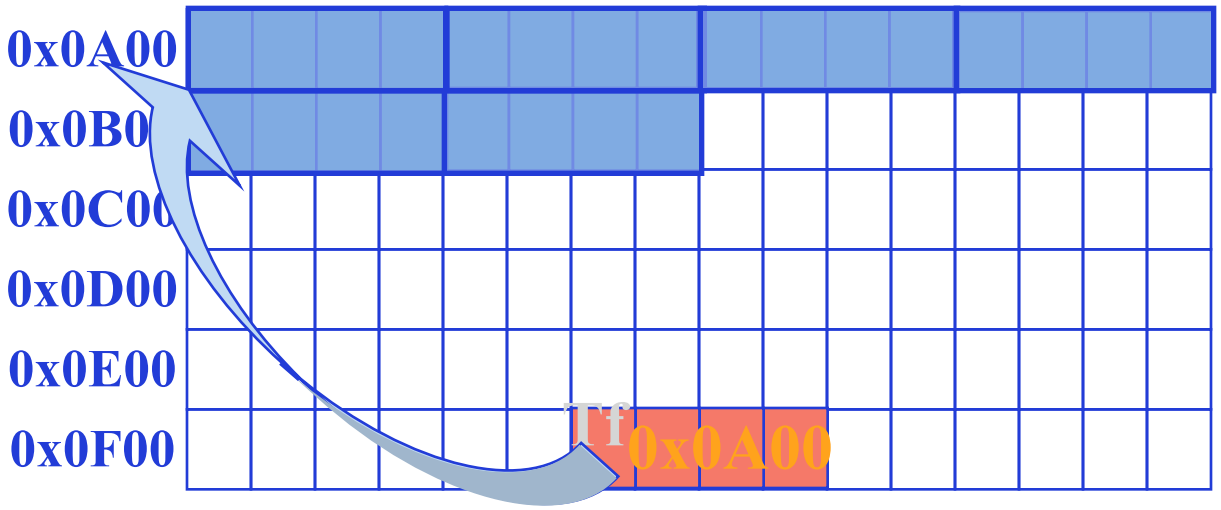
Tableaux dynamiques (suite)

```
float *Tf;
Tf = malloc(6*(sizeof(float)));
```



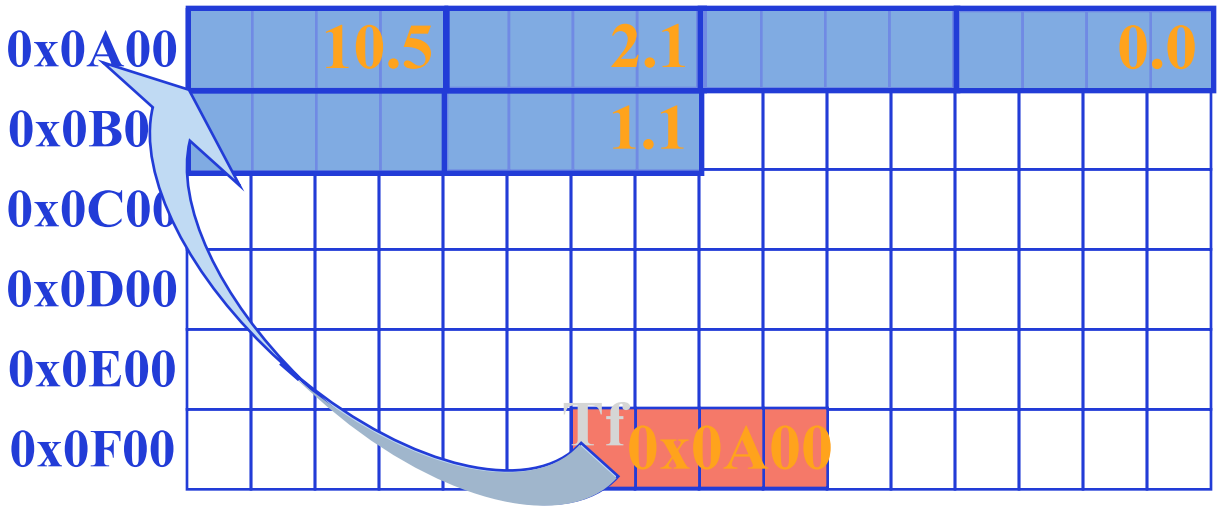
Tableaux dynamiques (suite)

```
float *Tf;
Tf = malloc(6*(sizeof(float)));
```



Tableaux dynamiques (suite)

```
float *Tf;
Tf = malloc(6*(sizeof(float)));
```




```
Tf[0] = 10.5; Tf[1] = 2.1;
 Tf[3] = 0;  Tf[5] = 1.1;
```

Tableaux dynamiques (suite)

Tableaux dynamiques (suite)

Ainsi,

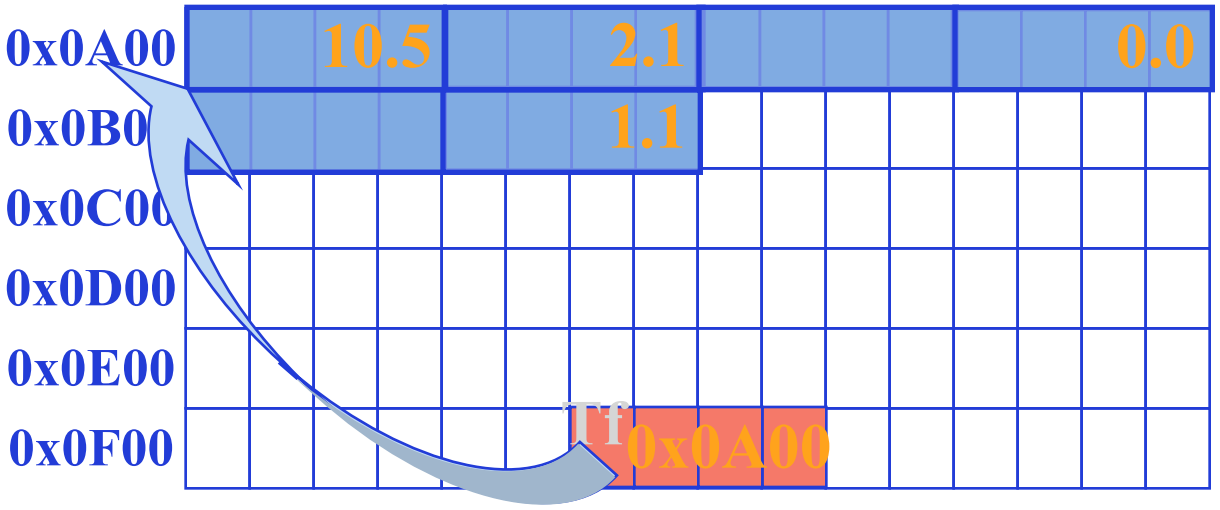
`<type> *t = malloc(k*(sizeof(<type>)));`

définit un tableau de **`k`** objets de type **`<type>`**, où

- **`k`** peut être une variable :
  taille non définie à la compilation
- **`<type>`** peut être un tableau (pointeur) :
  tableau à plusieurs dimensions

Tableaux et pointeurs

```
float *Tf;
Tf = malloc(6*(sizeof(float)));
```




```
*Tf     = 10.5; *(Tf+1) = 2.1;
 *(Tf+3)= 0;   *(Tf+5) = 1.1;
```

Tableaux et pointeurs

```
float *Tf;
Tf = malloc(6*(sizeof(float)));
```

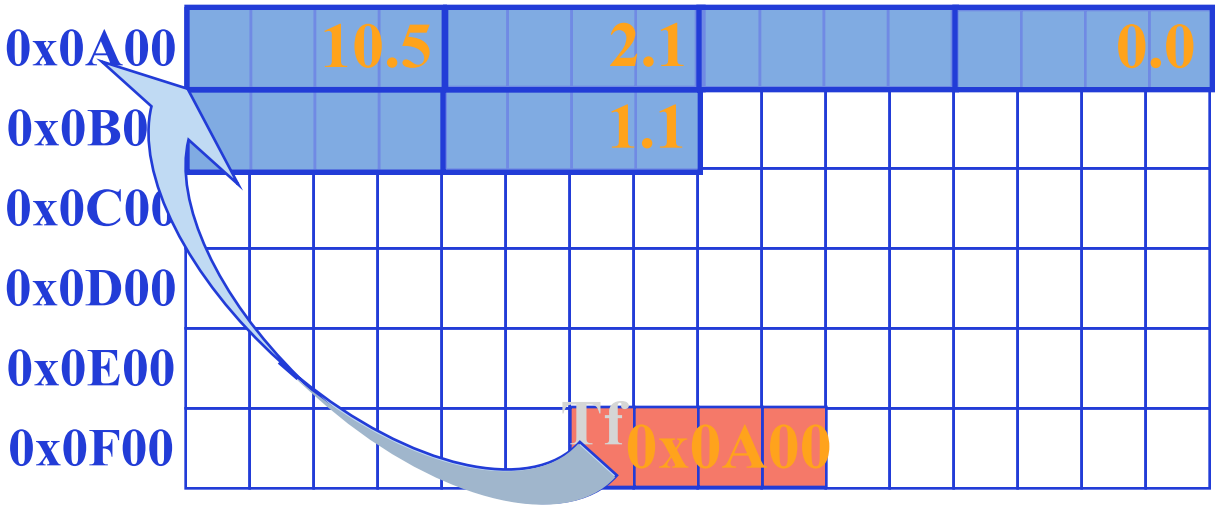


Tableau de `float`

```
Tf   : 0x0A00
Tf+1 : 0x0A04
Tf+3 : 0x0A0C
Tf+5 : 0x0B04
```

```
*Tf     = 10.5; *(Tf+1) = 2.1;
 *(Tf+3)= 0;   *(Tf+5) = 1.1;
```

Tableaux à plusieurs dimensions

```
int i, m = 10, n = 15;
int **T;
T = malloc(m * sizeof(int *));
for (i=0; i<m; i++)
T[i] = malloc(n * sizeof(int));
```

Tableaux à plusieurs dimensions

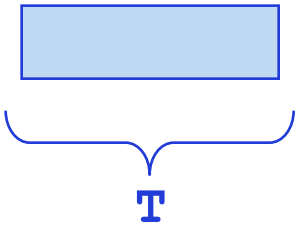
```
int i, m = 10, n = 15;
int **T;
T = malloc(m * sizeof(int *));
for (i=0; i<m; i++)
T[i] = malloc(n * sizeof(int));
```

`T` est un pointeur de pointeur d'entier
= tableau de tableau d'entiers

Tableaux à plusieurs dimensions

```
int i, m = 10, n = 15;
int **T;
T = malloc(m * sizeof(int *));
for (i=0; i<m; i++)
T[i] = malloc(n * sizeof(int));
```

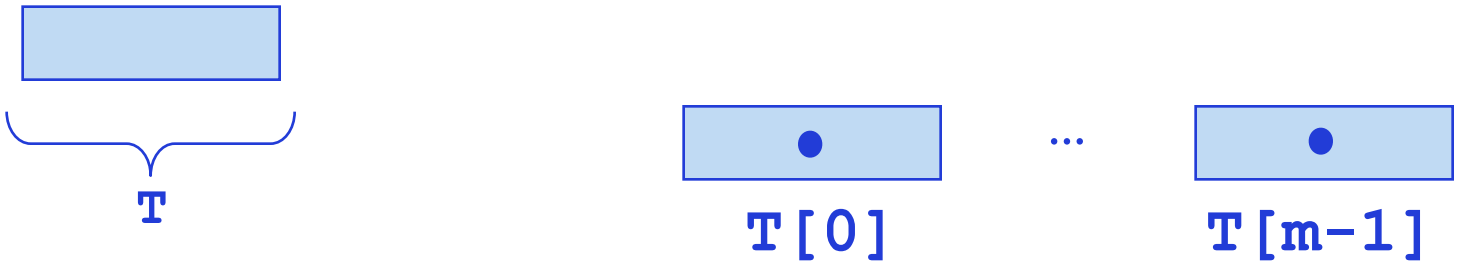
`T` est un pointeur de pointeur d'entier
= tableau de tableau d'entiers



Tableaux à plusieurs dimensions

```
int i, m = 10, n = 15;
int **T;
T = malloc(m * sizeof(int *));
for (i=0; i<m; i++)
T[i] = malloc(n * sizeof(int));
```

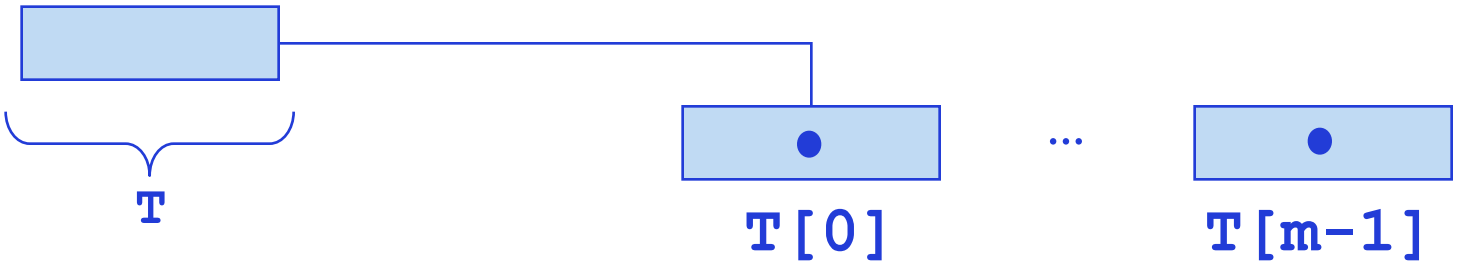
`T` est un pointeur de pointeur d'entier
= tableau de tableau d'entiers



Tableaux à plusieurs dimensions

```
int i, m = 10, n = 15;
int **T;
T = malloc(m * sizeof(int *));
for (i=0; i<m; i++)
T[i] = malloc(n * sizeof(int));
```

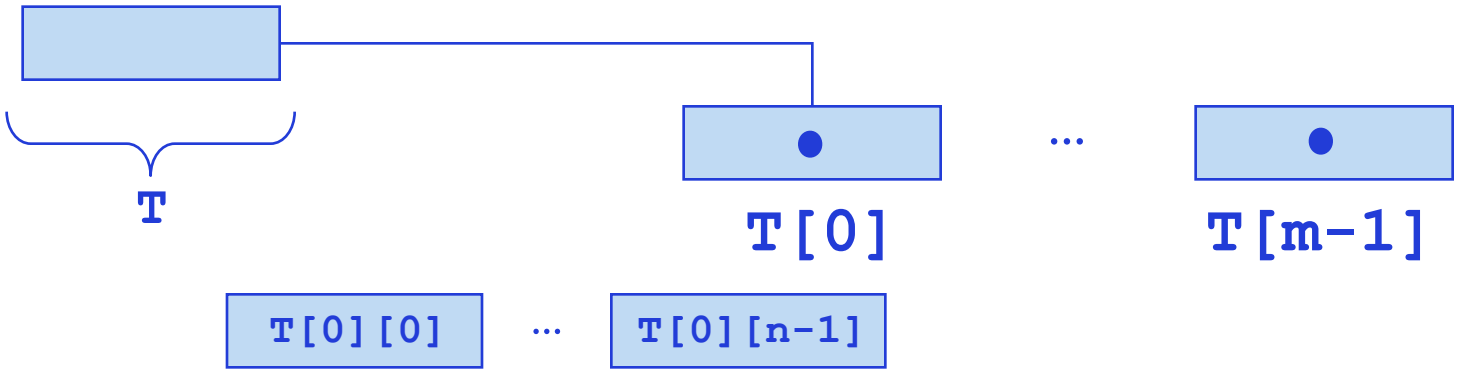
`T` est un pointeur de pointeur d'entier
= tableau de tableau d'entiers



Tableaux à plusieurs dimensions

```
int i, m = 10, n = 15;
int **T;
T = malloc(m * sizeof(int *));
for (i=0; i<m; i++)
T[i] = malloc(n * sizeof(int));
```

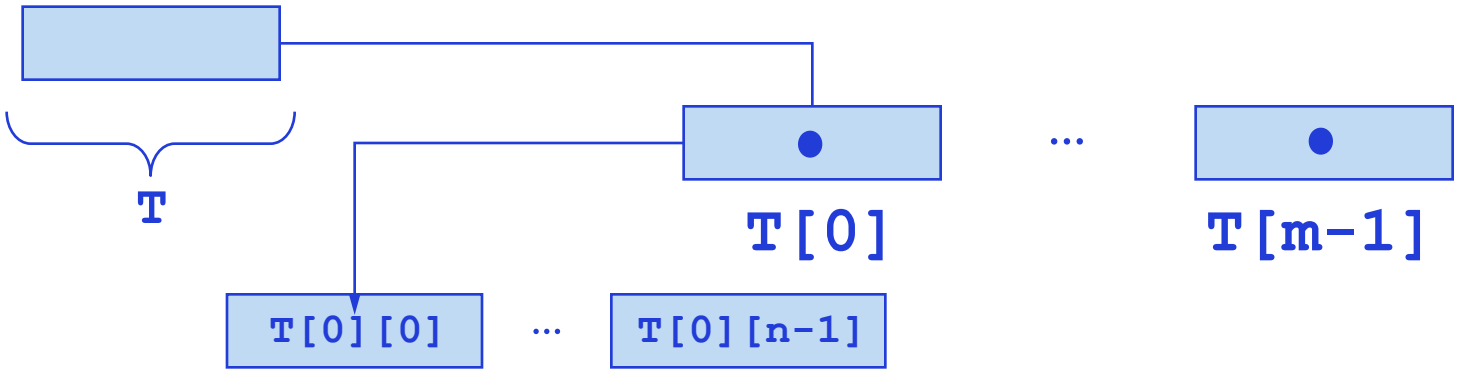
`T` est un pointeur de pointeur d'entier
= tableau de tableau d'entiers



Tableaux à plusieurs dimensions

```
int i, m = 10, n = 15;
int **T;
T = malloc(m * sizeof(int *));
for (i=0; i<m; i++)
T[i] = malloc(n * sizeof(int));
```

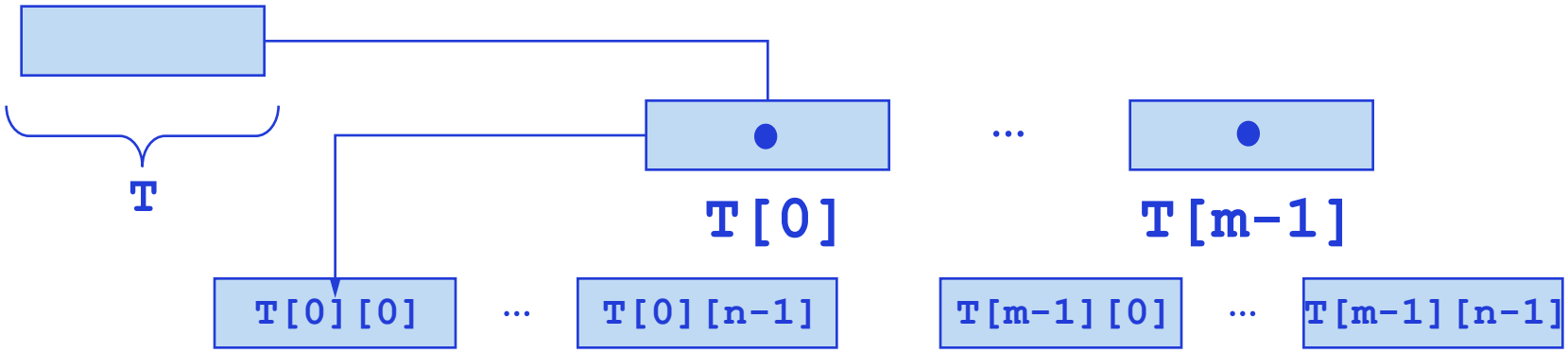
`T` est un pointeur de pointeur d'entier
= tableau de tableau d'entiers



Tableaux à plusieurs dimensions

```
int i, m = 10, n = 15;
int **T;
T = malloc(m * sizeof(int *));
for (i=0; i<m; i++)
T[i] = malloc(n * sizeof(int));
```

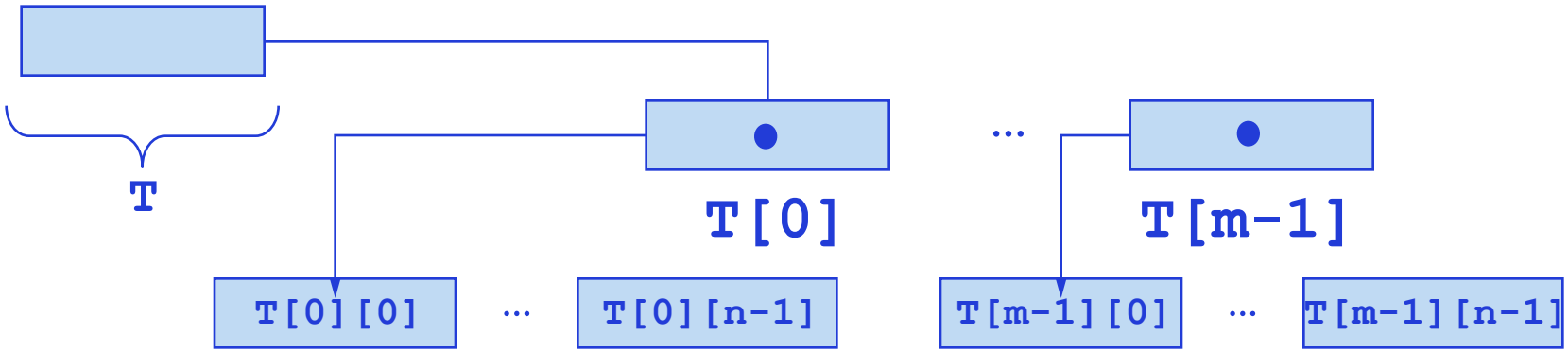
`T` est un pointeur de pointeur d'entier
= tableau de tableau d'entiers



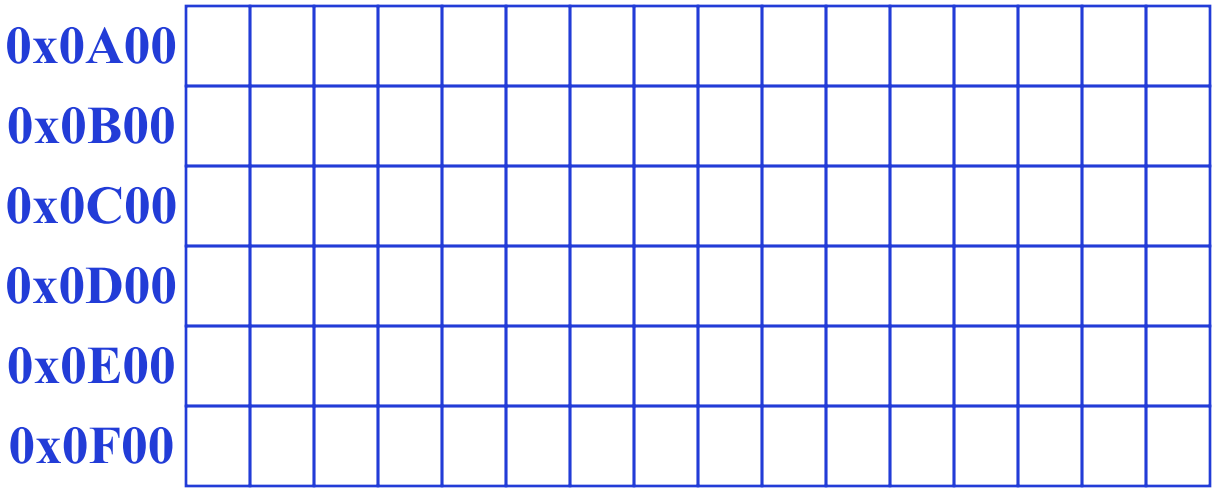
Tableaux à plusieurs dimensions

```
int i, m = 10, n = 15;
int **T;
T = malloc(m * sizeof(int *));
for (i=0; i<m; i++)
T[i] = malloc(n * sizeof(int));
```

`T` est un pointeur de pointeur d'entier
= tableau de tableau d'entiers

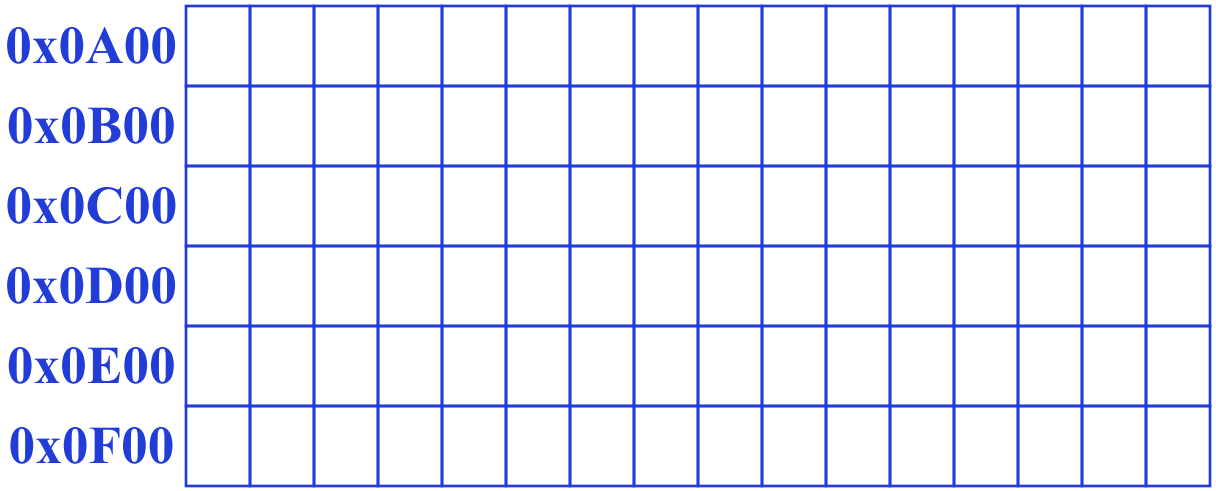


Plusieurs dimensions (suite)



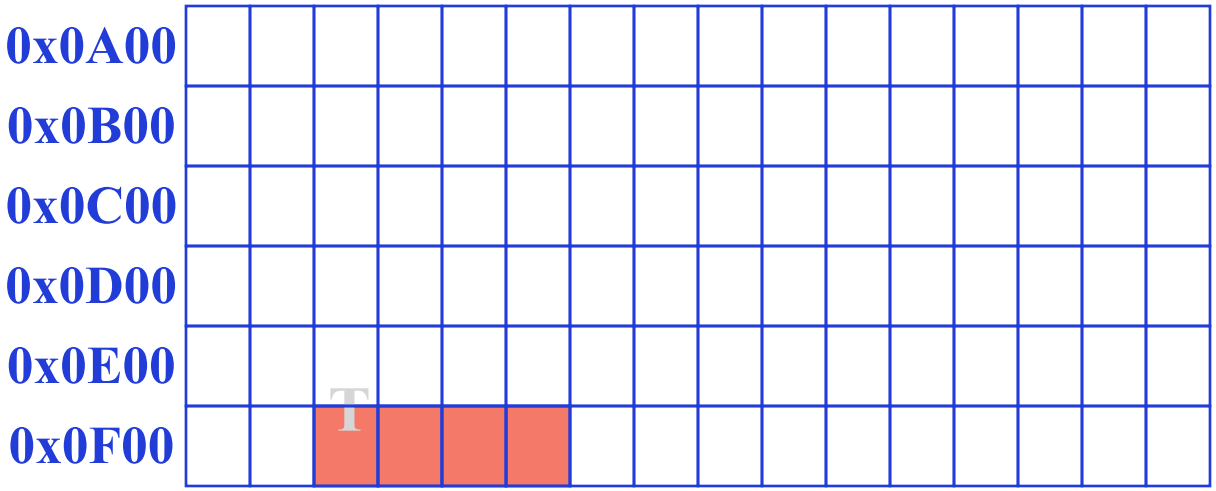
Plusieurs dimensions (suite)

```
int i, m = 3, n = 5;
char **T;
T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)
T[i] = malloc(n * sizeof(char));
```



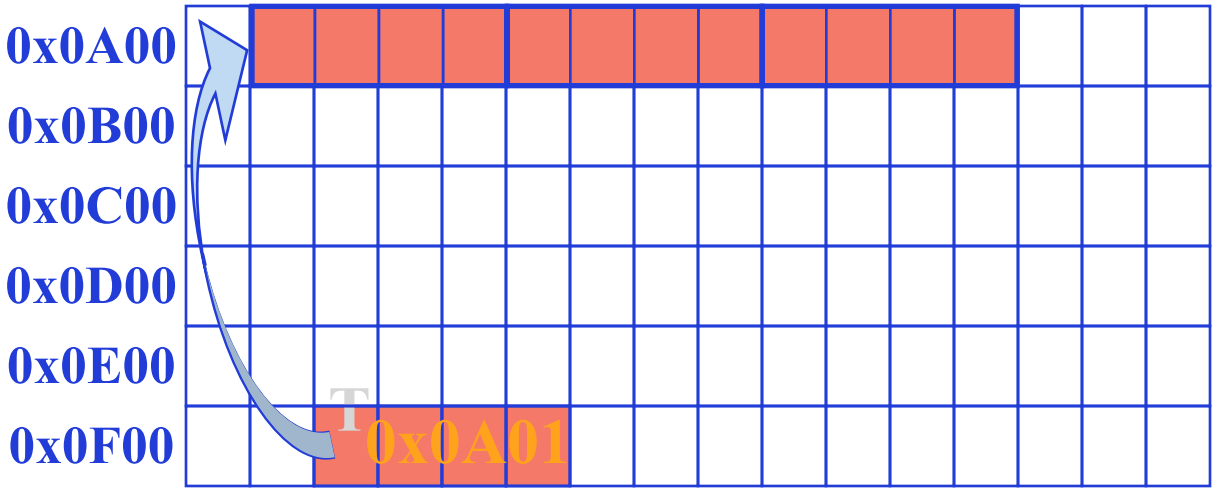
Plusieurs dimensions (suite)

```
int i, m = 3, n = 5;
char **T;
T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)
T[i] = malloc(n * sizeof(char));
```



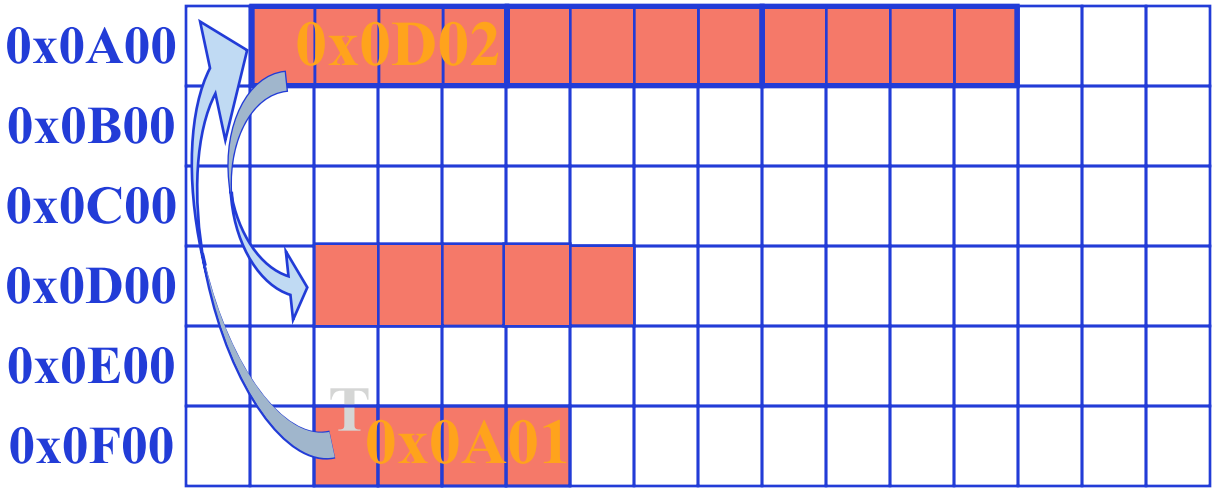
Plusieurs dimensions (suite)

```
int i, m = 3, n = 5;
char **T;
T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)
T[i] = malloc(n * sizeof(char));
```



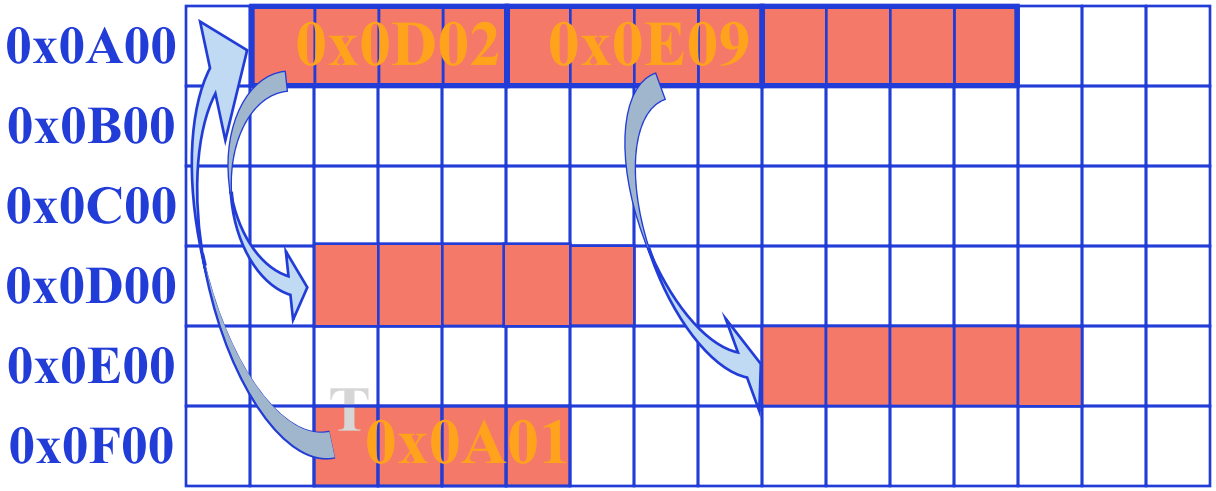
Plusieurs dimensions (suite)

```
int i, m = 3, n = 5;
char **T;
T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)
T[i] = malloc(n * sizeof(char));
```



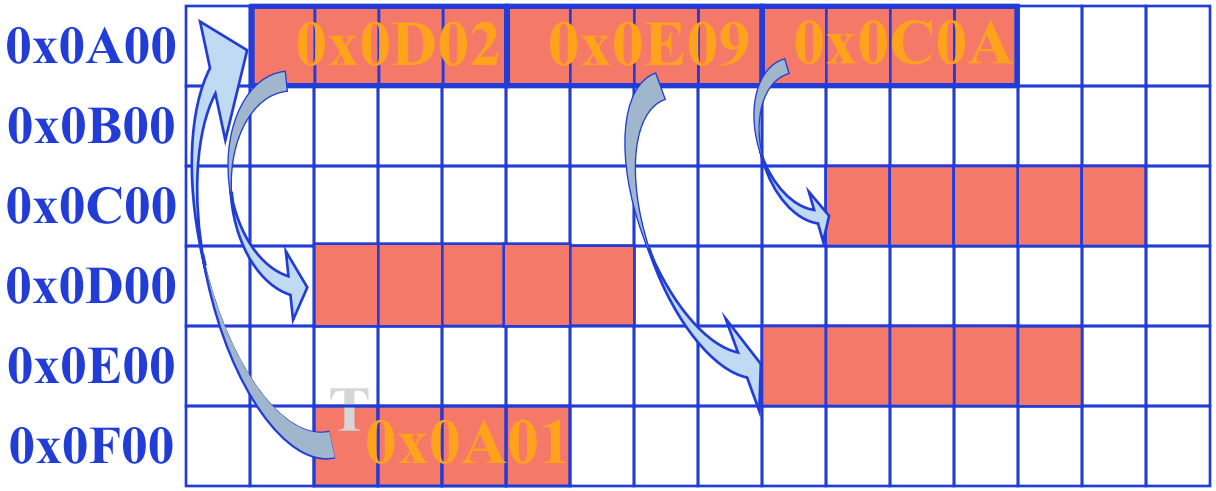
Plusieurs dimensions (suite)

```
int i, m = 3, n = 5;
char **T;
T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)
T[i] = malloc(n * sizeof(char));
```



Plusieurs dimensions (suite)

```
int i, m = 3, n = 5;
char **T;
T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)
T[i] = malloc(n * sizeof(char));
```

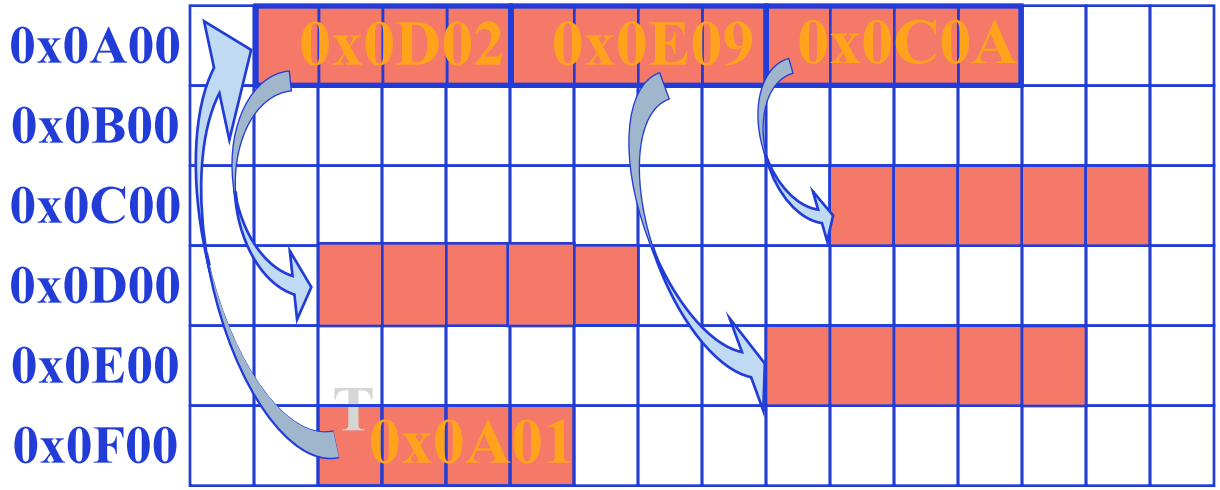


Plusieurs dimensions (suite)

```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));
```

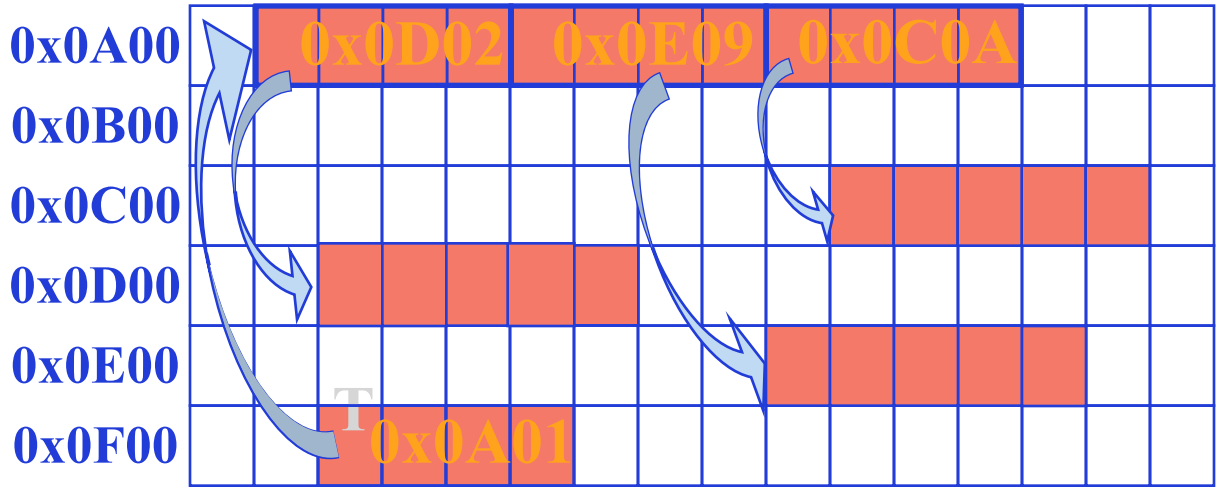
Plusieurs dimensions (suite)

```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));
```



Plusieurs dimensions (suite)

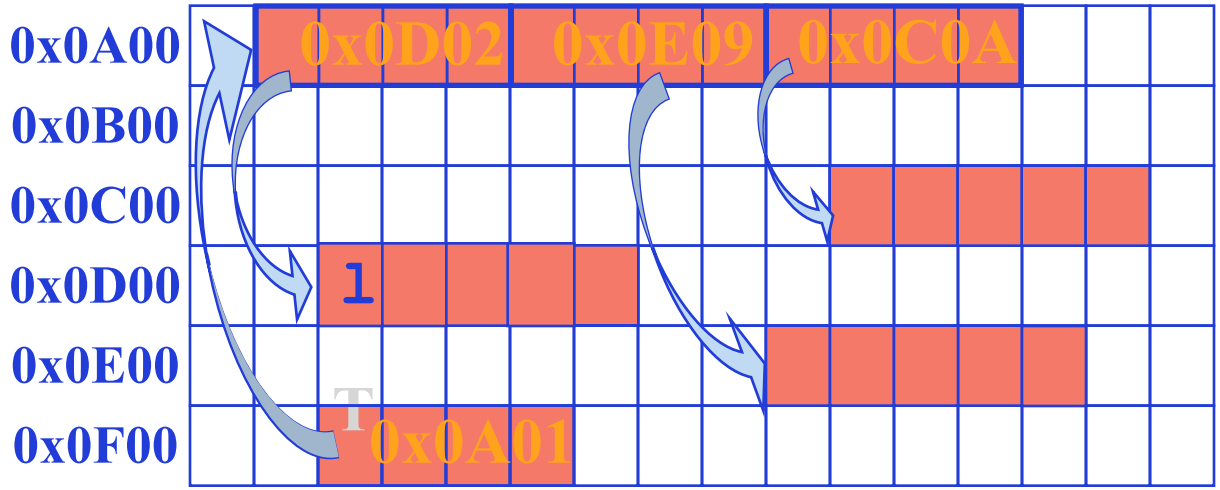
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));
```




```
T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
```

Plusieurs dimensions (suite)

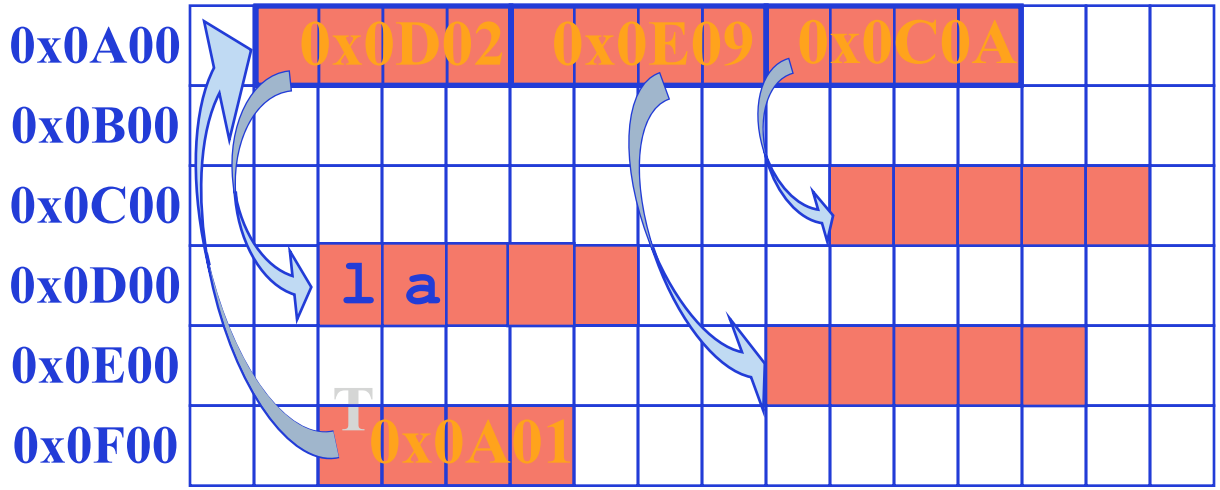
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));
```




```
T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
```

Plusieurs dimensions (suite)

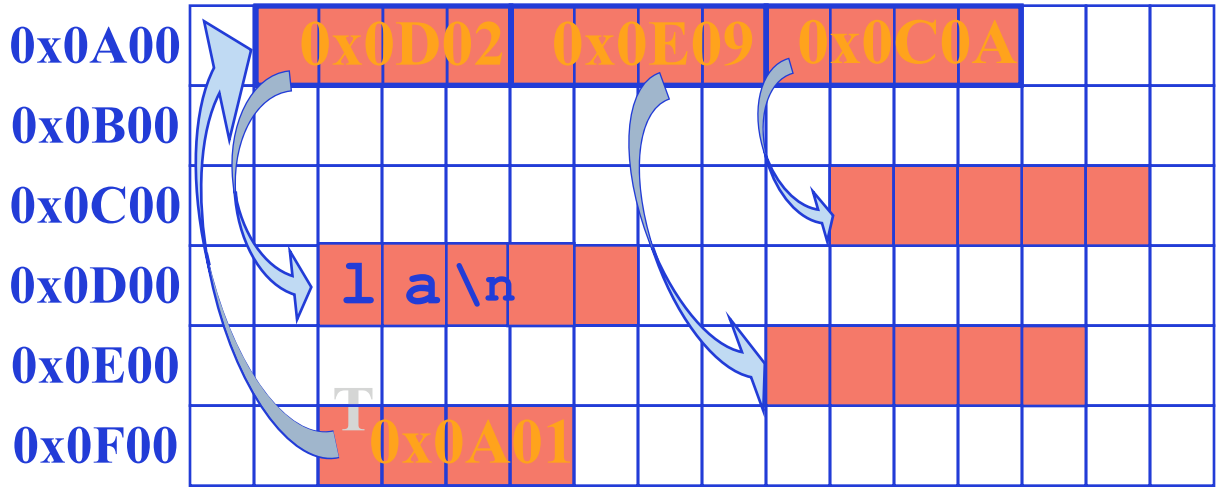
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));
```




```
T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
```

Plusieurs dimensions (suite)

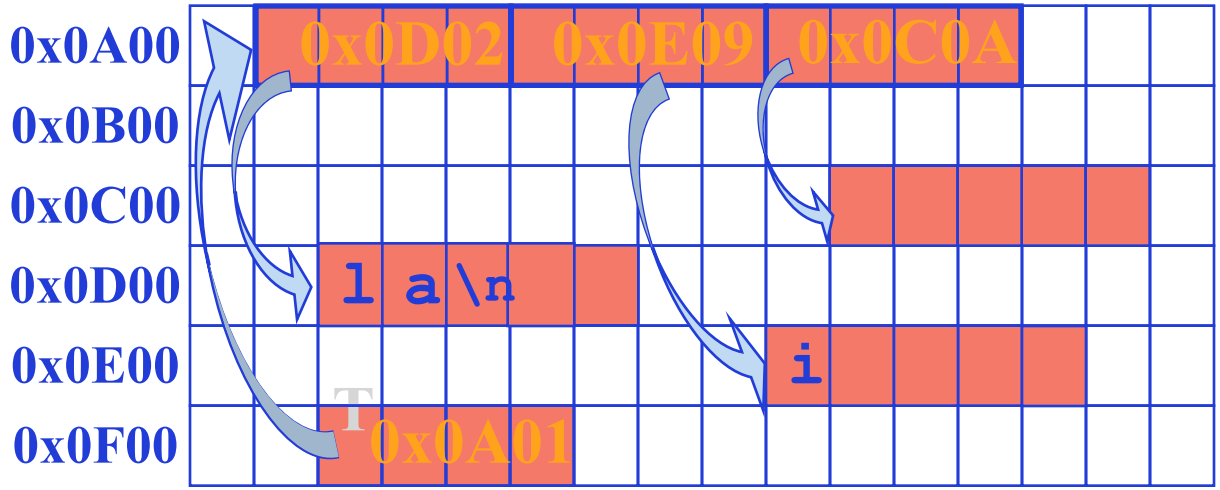
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));
```




```
T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
```

Plusieurs dimensions (suite)

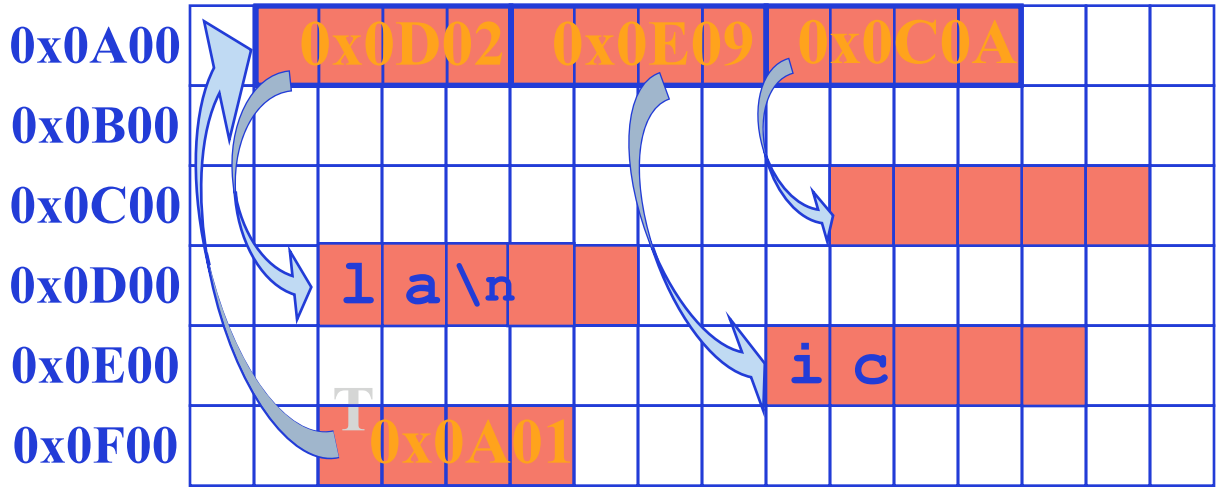
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));
```




```
T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
```

Plusieurs dimensions (suite)

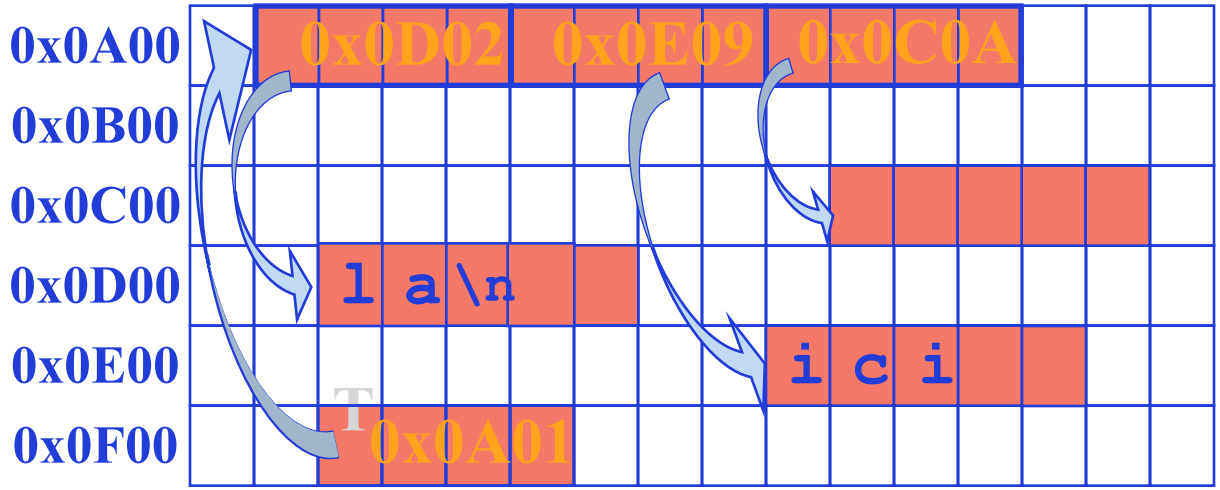
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));
```




```
T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
```

Plusieurs dimensions (suite)

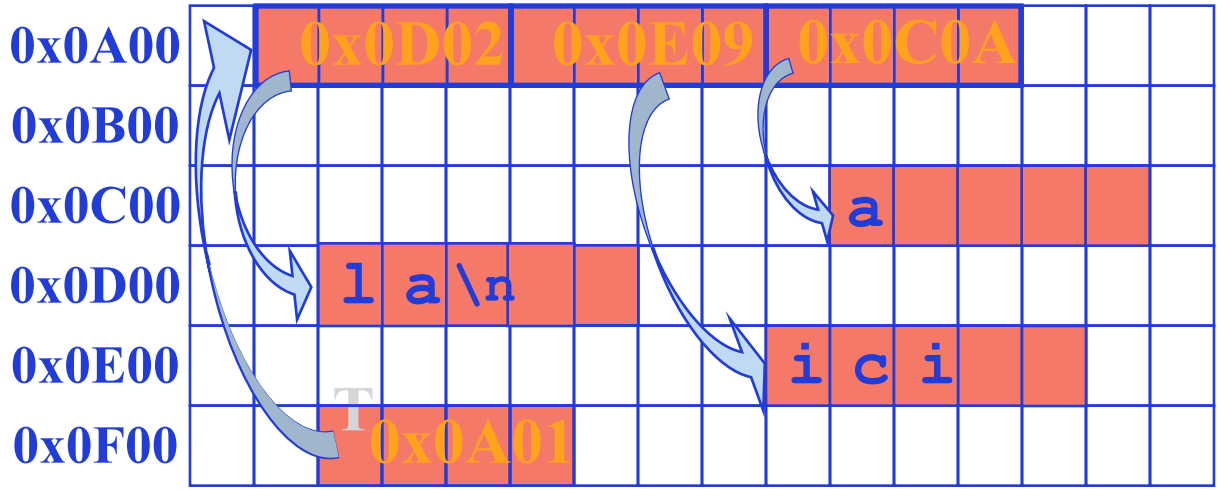
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));
```




```
T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
```

Plusieurs dimensions (suite)

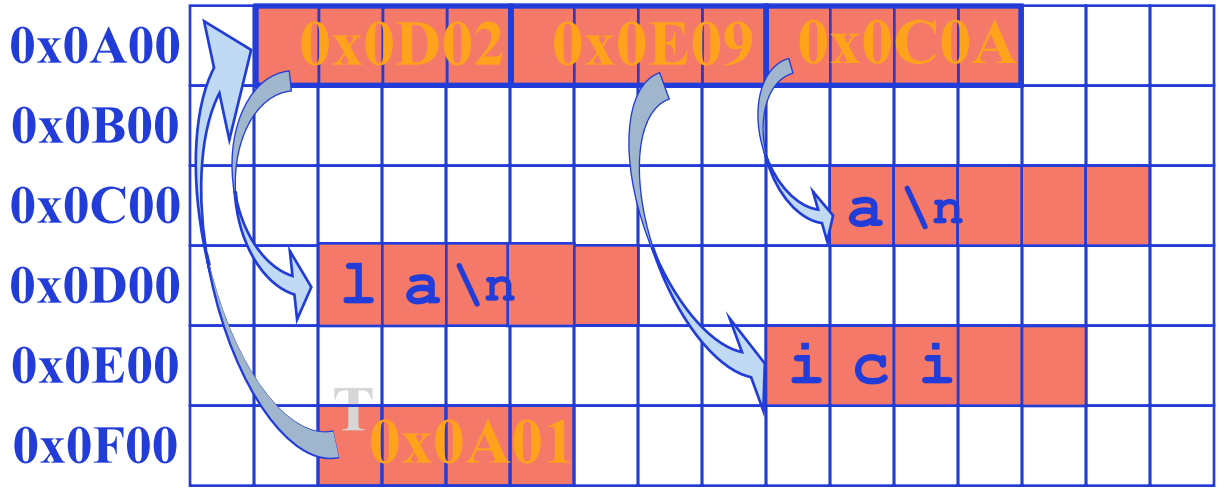
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));
```




```
T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
```

Plusieurs dimensions (suite)

```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));
```




```
T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
```

Libération de mémoire : `free`

De la mémoire allouée avec **`malloc`**
ne sera libérée qu'à la fin du programme

On peut cependant vouloir libérer cette mémoire pour un autre usage (la mémoire de l'ordinateur n'est pas illimitée)

⇒ commande **`free`**

Libération de mémoire : `free`

De la mémoire allouée avec **`malloc`**
ne sera libérée qu'à la fin du programme

On peut cependant vouloir libérer cette mémoire pour un autre usage (la mémoire de l'ordinateur n'est pas illimitée)

⇒ commande **`free`**

```
int *pa = malloc((sizeof(int)); /*allocation*/
```

Libération de mémoire : `free`

De la mémoire allouée avec **`malloc`**
ne sera libérée qu'à la fin du programme

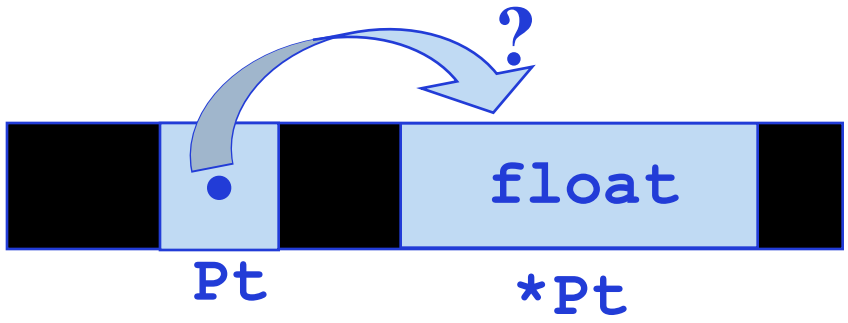
On peut cependant vouloir libérer cette mémoire pour un autre usage (la mémoire de l'ordinateur n'est pas illimitée)

⇒ commande **`free`**

```
int *pa = malloc((sizeof(int)); /*allocation*/
      free(pa);                 /*libération*/
```

Fonctions et pointeurs

```
float *Pt;
```



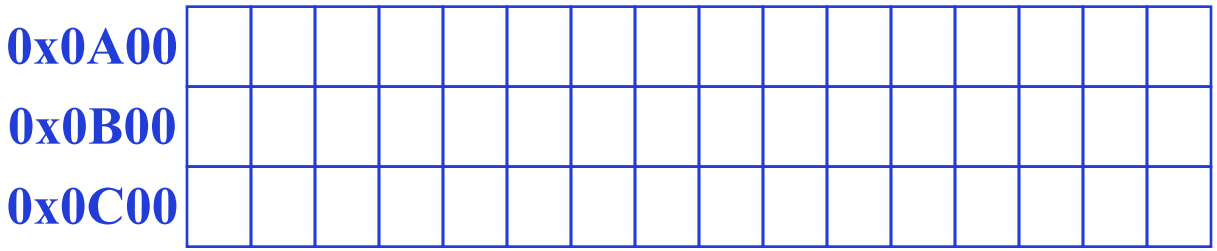
Une fonction manipule des **copies**

⇒ ne peut modifier ses arguments

```
void fonction(int A, double *B)
```

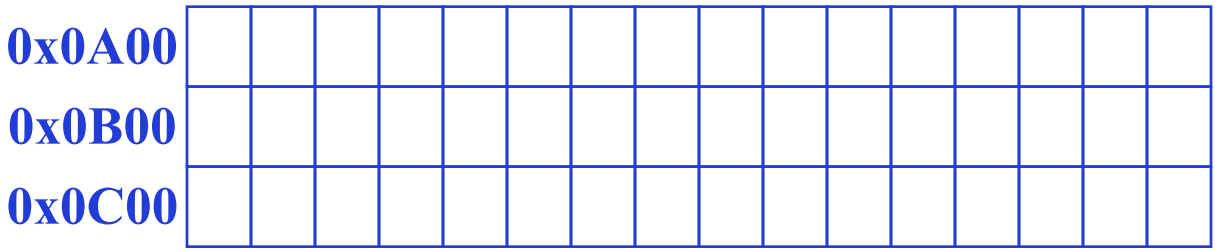
- `int A,` **A** reçoit une copie du 1er arg.
- `double *B,` **B** reçoit copie du 2nd arg., un pointeur sur un `double`

Passage par adresse



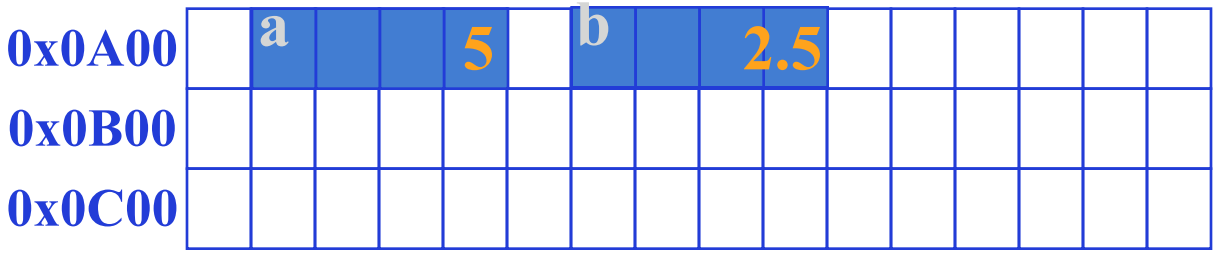
Passage par adresse

```
void fonction(int A, double *B) {…}
fonction(a, &b);
```



Passage par adresse

```
void fonction(int A, double *B) {…}
fonction(a, &b);
```

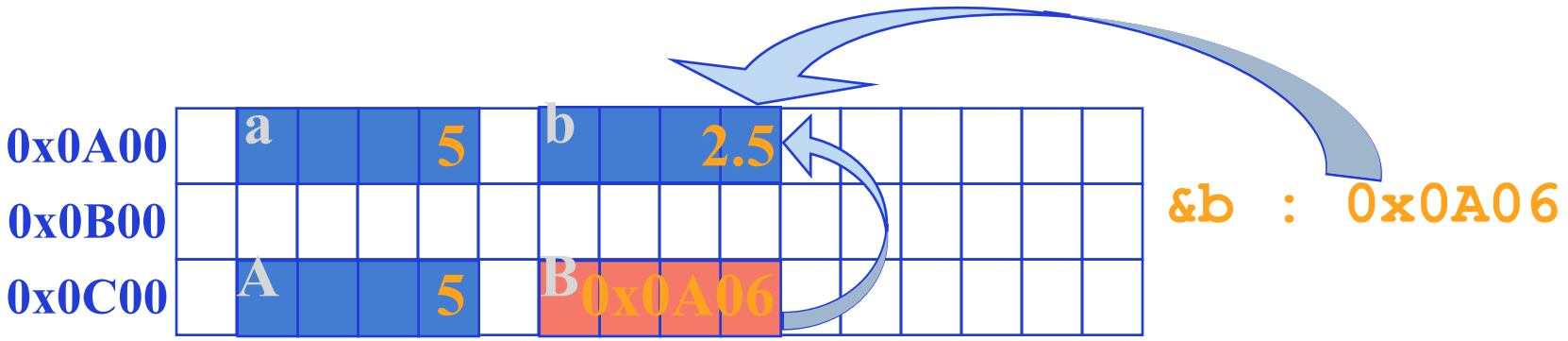


Passage par adresse

```
void fonction(int A, double *B) {…}
fonction(a, &b);
```

A est une copie de **a**

B est une copie de **&b**, un pointeur sur **b**



Passage par adresse

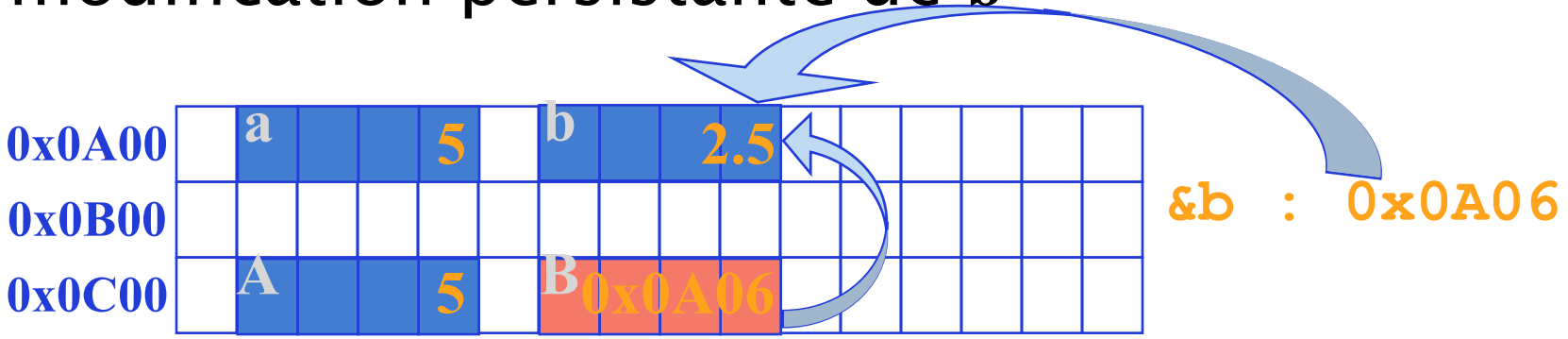
```
void fonction(int A, double *B) {…}
fonction(a, &b);
```

A est une copie de **a**

B est une copie de **&b**, un pointeur sur **b**

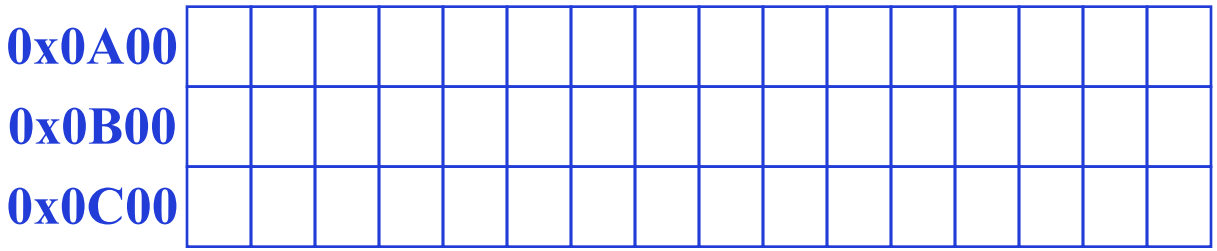
⇒ une modification de ***B** modifiera l'objet pointé par **B**, soit **b**

⇒ modification persistante de **b**



Exemple II

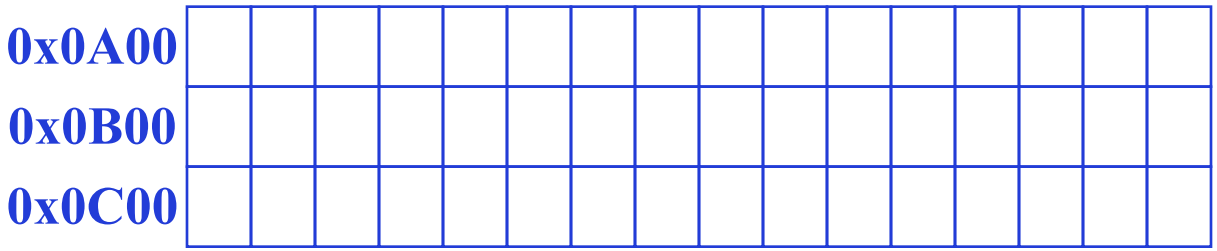
```
void incrementation(int *pa) {
        (*pa) = (*pa)+1;
              }
```



Exemple II

```
void incrementation(int *pa) {
        (*pa) = (*pa)+1;
              }
```

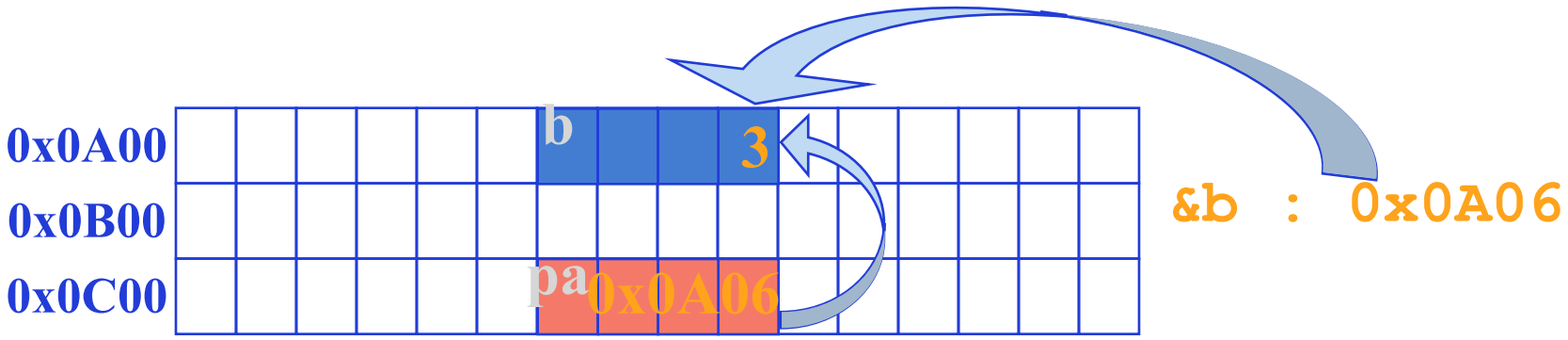
```
        b = 3;
incrementation(&b);
```



Exemple II

```
void incrementation(int *pa) {
        (*pa) = (*pa)+1;
              }
```

```
        b = 3;
incrementation(&b);
```



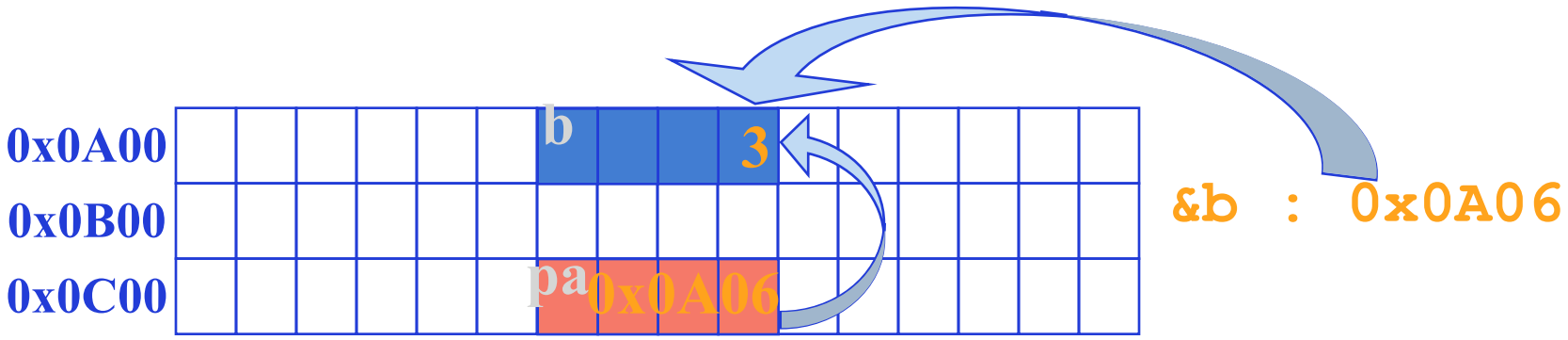
Exemple II

```
void incrementation(int *pa) {
        (*pa) = (*pa)+1;
              }
```

```
b = 3;
incrementation(&b);
```

Une modification de `*pa` modifie `b` :

`(*pa) = (*pa) + 1`



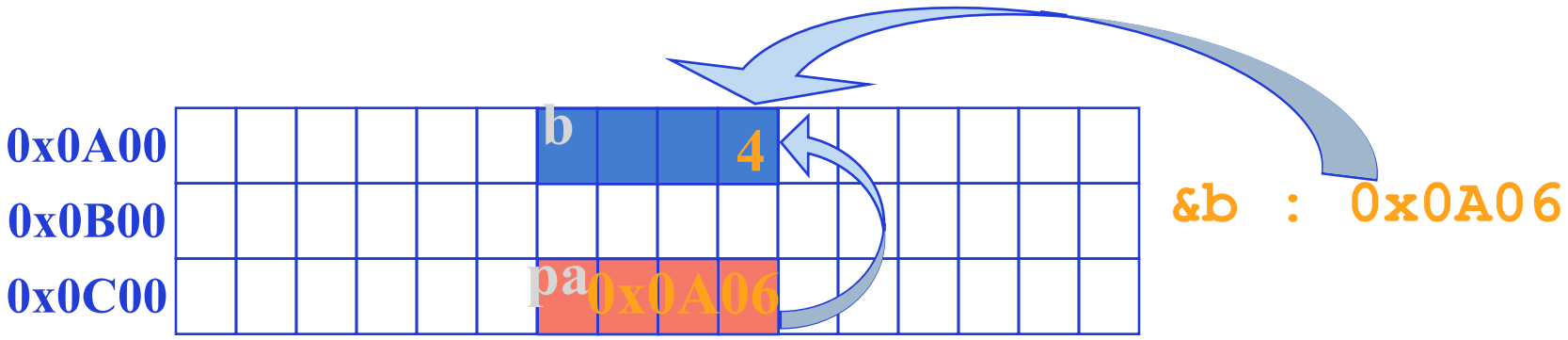
Exemple II

```
void incrementation(int *pa) {
        (*pa) = (*pa)+1;
              }
```

```
b = 3;
incrementation(&b);
```

Une modification de `*pa` modifie `b` :

`(*pa) = (*pa) + 1`



Exemple II

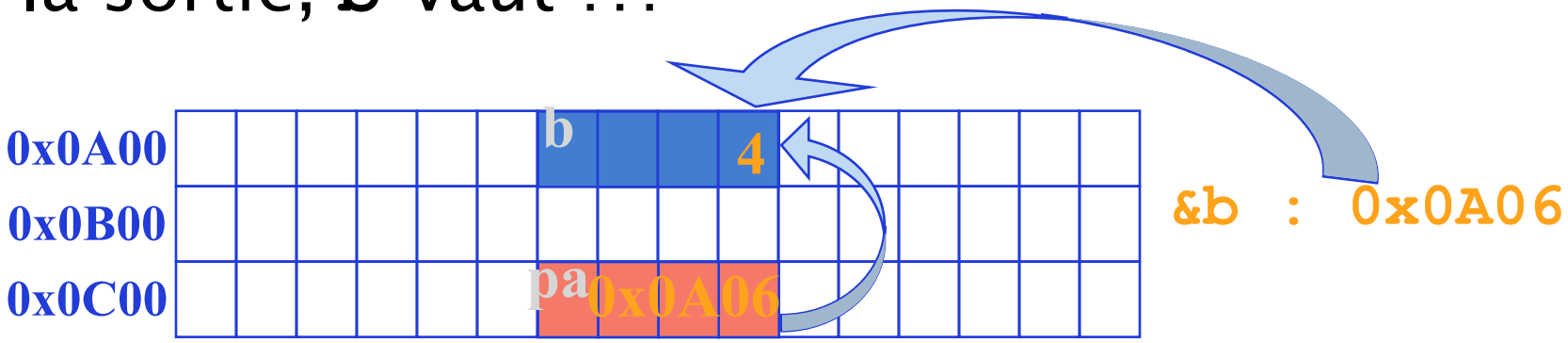
```
void incrementation(int *pa) {
        (*pa) = (*pa)+1;
              }
```

```
b = 3;
incrementation(&b);
```

à la sortie, `b` vaut ???

Une modification de `*pa` modifie `b` :

`(*pa) = (*pa) + 1`



Exemple II

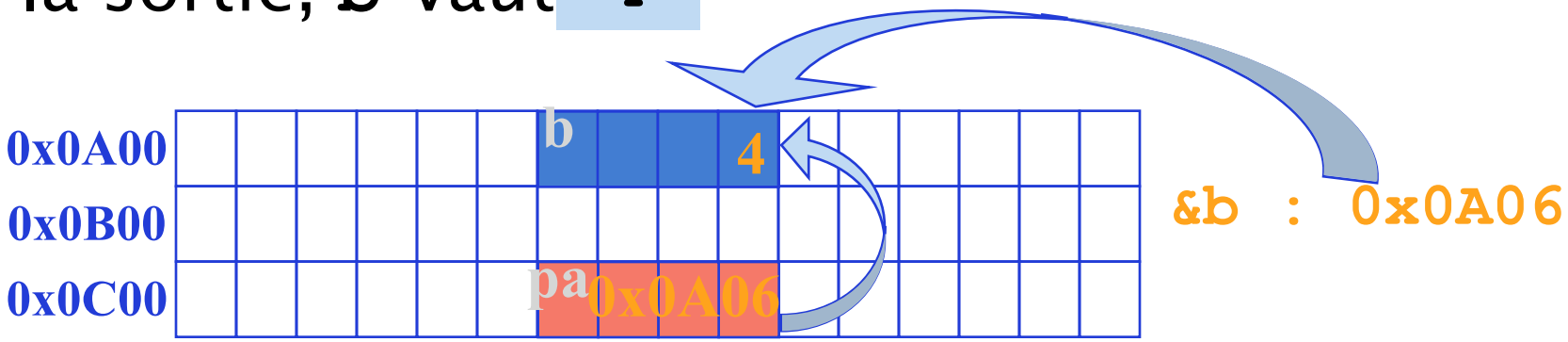
```
void incrementation(int *pa) {
        (*pa) = (*pa)+1;
              }
```

```
b = 3;
incrementation(&b);
```

Une modification de `*pa` modifie `b` :

`(*pa) = (*pa) + 1`

à la sortie, `b` vaut **4**



Cas des tableaux

La fonction ne manipule que des copies des variables passées en arguments

```
void initialisation(int *tab, int n){…}
initialisation(T,l);
```

⇒ le passage du tableau **`T`** ne communique que le « pointeur » **`T`** sur le début du tableau

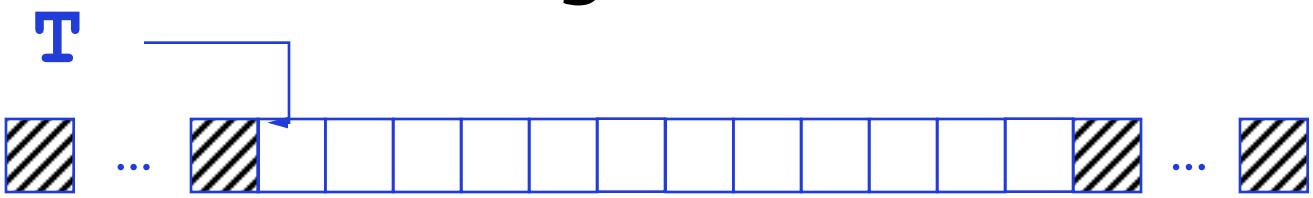
Cas des tableaux

La fonction ne manipule que des copies des variables passées en arguments

```
void initialisation(int *tab, int n){…}
initialisation(T,l);
```

⇒ le passage du tableau `T` ne communique que le « pointeur » `T` sur le début du tableau

`T` désigne la zone mémoire



Allocation permanente

```
#define N 10
int *initialisation() {
   int T[N];
   int i;
   for (i=0; i<N; i++) T[i] = 0;
   return T;
}
```

Allocation permanente

```
#define N 10
int *initialisation() {
   int T[N];
   int i;
   for (i=0; i<N; i++) T[i] = 0;
   return T;
}
```

Tableau `T` local à la fonction, libéré à la fin

Allocation permanente

```
#define N 10
int *initialisation() {
   int T[N];
   int i;
   for (i=0; i<N; i++) T[i] = 0;
   return T;
}
```

Tableau `T` local à la fonction, libéré à la fin

```
int *initialisation(int n) {
   int i;
   int *T = malloc(n * sizeof(int));
   for (i=0; i<n; i++) T[i] = 0;
   return T;
}
```

Conversion par valeur → par adresse

Soit une fonction d'addition qui ajoute le deuxième argument au premier

```
void addition(int a, int b)
{
   a = a + b;
}
```

Conversion par valeur → par adresse

Soit une fonction d'addition qui ajoute le deuxième argument au premier

```
void addition(int a, int b)
{
   a = a + b;
}
```

Passage par valeur : 1er argument inchangé

⇒ passage par adresse du 1er argument

Conversion par valeur → par adresse

Soit une fonction d'addition qui ajoute le deuxième argument au premier

```
void addition(int a, int b)
{
   a = a + b;
}
```

Passage par valeur : 1er argument inchangé
⇒ passage par adresse du 1er argument

```
void addition(int *pa, int b)
{
   (*pa) = (*pa) + b;
}
```

Conversion par valeur → par adresse

Soit une fonction d'addition qui ajoute le deuxième argument au premier

```
void addition(int a, int b)
{
   a = a + b;
}
```

Passage par valeur : 1er argument inchangé
⇒ passage par adresse du 1er argument

```
void addition(int *pa, int b)
{
   (*pa) = (*pa) + b;
}
```

Conversion par valeur → par adresse

Soit une fonction d'addition qui ajoute le deuxième argument au premier

```
void addition(int a, int b)
{
  a = a + b;
}
```

Passage par valeur : 1er argument inchangé
⇒ passage par adresse du 1er argument

```
void addition(int *pa, int b)
{
   (*pa) = (*pa) + b;
}
```

Conversion par valeur → par adresse

Soit une fonction d'addition qui ajoute le deuxième argument au premier

```
void addition(int a, int b)
{
  a = a + b;
}
```

Passage par valeur : 1er argument inchangé
⇒ passage par adresse du 1er argument

```
void addition(int *pa, int b)
{
   (*pa) = (*pa) + b;
}
```

Conclusion

Pointeur

Allocation dynamique

- allocation de mémoire pour des pointeurs
- tableaux de taille variable
- tableaux à plusieurs dimensions
- libération dynamique