

# IN 101 - Cours 07

21 octobre 2011



présenté par  
**Matthieu Finiasz**

## Pointeurs et adresses mémoires

### Qu'est-ce qu'une adresse mémoire ?

- ✗ Pour le processeur, la mémoire est un grand tableau d'octets :
  - de  $2^{32}$  sur une machine 32 bits ou  $2^{64}$  sur une 64 bits.
- ✗ Tout le tableau n'est pas utilisable :
  - ✗ la mémoire a une taille limitée (capacité physique des barrettes)
  - ✗ une partie de la mémoire est utilisée par les autres processeurs.
- ✗ C'est le système d'exploitation qui arbitre et décide qui peut accéder à quelle partie de la mémoire :
  - ✗ il faut un appel système pour réserver de la mémoire
    - l'OS note à qui elle appartient,
  - ✗ si on essaye d'accéder à une partie interdite, l'OS bloque l'accès
    - erreur de segmentation.

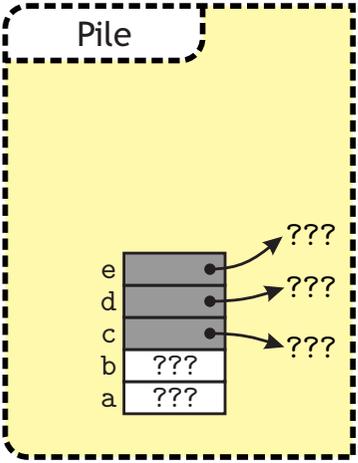
### Pointeurs et adresses mémoires

- ✗ En C, une adresse mémoire est en général appelée **pointeur**
    - ✗ c'est une adresse mémoire avec un type associé
      - le type des éléments qui se trouvent à cette adresse.
- 
- ```

1 int* a;      // a est un pointeur vers un/des int
2 double* b;  // b est un pointeur vers un/des double
3 int** c;    // c est un pointeur vers un/des int*
```
- 
- ✗ Chaque variable possède une adresse et on peut y accéder avec le préfixe **&**. On peut accéder au contenu d'une case avec le préfixe **\***.
- 
- ```

1 int a;      // a est un int
2 int* b;    // b est un pointeur vers un int
3 b = &a;    // b contient l'adresse de a -> pointeur vers a
4 *b = 5;    // on stocke 5 dans *b -> dans a
```
- 
- **&** et **\*** ont des rôles symétriques : **\*(&a)** est la même chose que **a**.

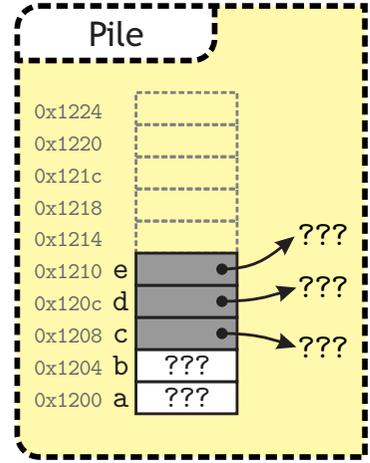
### Exemple de manipulations de pointeurs



```

1 int main() {
2   int a, b, *c, *d, **e;
3   a = 3;   b = 5;
4   c = &b;  d = &a;
5   *c = 4;
6   *d = (*c) + 3;
7   e = &c;
8   **e = *d;
9   *e = &a;
10 }
    
```

### Exemple de manipulations de pointeurs

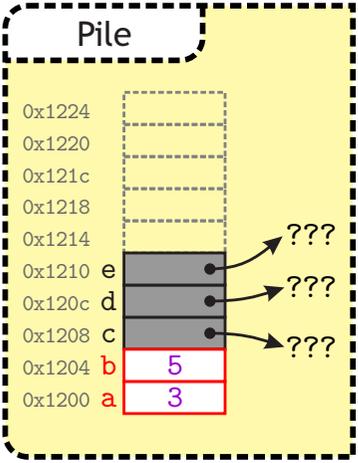


```

1 int main() {
2   int a, b, *c, *d, **e;
3   a = 3;   b = 5;
4   c = &b;  d = &a;
5   *c = 4;
6   *d = (*c) + 3;
7   e = &c;
8   **e = *d;
9   *e = &a;
10 }
    
```

✗ Chaque case mémoire a une adresse : souvent représentée en hexadécimal.

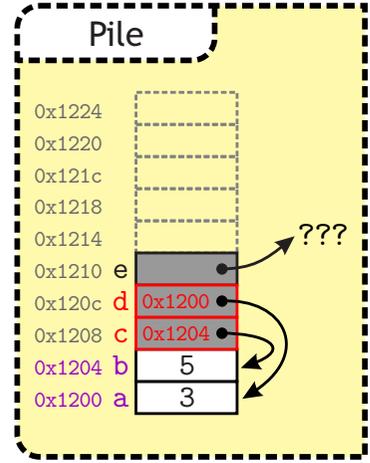
### Exemple de manipulations de pointeurs



```

1 int main() {
2   int a, b, *c, *d, **e;
3   a = 3;   b = 5;
4   c = &b;  d = &a;
5   *c = 4;
6   *d = (*c) + 3;
7   e = &c;
8   **e = *d;
9   *e = &a;
10 }
    
```

### Exemple de manipulations de pointeurs

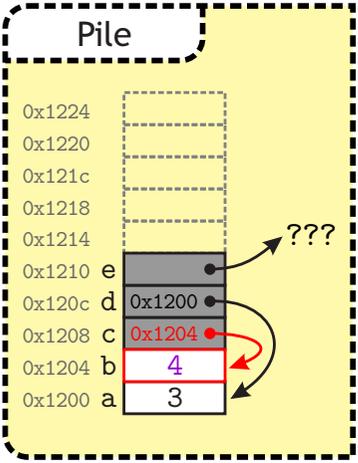


```

1 int main() {
2   int a, b, *c, *d, **e;
3   a = 3;   b = 5;
4   c = &b;  d = &a;
5   *c = 4;
6   *d = (*c) + 3;
7   e = &c;
8   **e = *d;
9   *e = &a;
10 }
    
```

✗ c et d contiennent les adresses de b et a  
 → ils pointent vers b et a.

### Exemple de manipulations de pointeurs

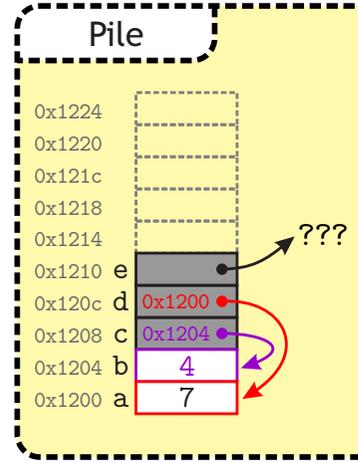


```

1 int main() {
2   int a, b, *c, *d, **e;
3   a = 3;   b = 5;
4   c = &b;  d = &a;
5   *c = 4;
6   *d = (*c) + 3;
7   e = &c;
8   **e = *d;
9   *e = &a;
10 }
    
```

✘ On suit le pointeur dans c et on stocke 4 dans la case mémoire correspondante → dans b.

### Exemple de manipulations de pointeurs

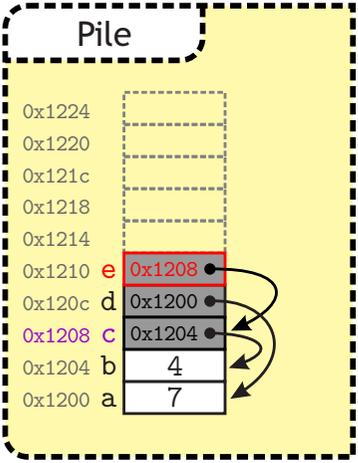


```

1 int main() {
2   int a, b, *c, *d, **e;
3   a = 3;   b = 5;
4   c = &b;  d = &a;
5   *c = 4;
6   *d = (*c) + 3;
7   e = &c;
8   **e = *d;
9   *e = &a;
10 }
    
```

✘ À gauche du "=" il y a la case mémoire où ira le résultat  
 ✘ À droite du "=" on évalue  
 → on suit le pointeur c et on regarde ce qui est dans la case.

### Exemple de manipulations de pointeurs

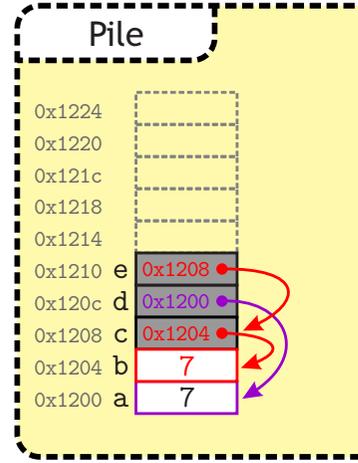


```

1 int main() {
2   int a, b, *c, *d, **e;
3   a = 3;   b = 5;
4   c = &b;  d = &a;
5   *c = 4;
6   *d = (*c) + 3;
7   e = &c;
8   **e = *d;
9   *e = &a;
10 }
    
```

✘ e pointe vers c.

### Exemple de manipulations de pointeurs

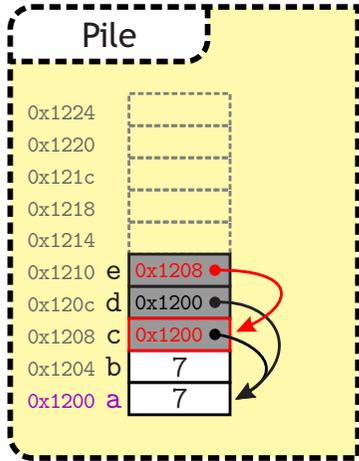


```

1 int main() {
2   int a, b, *c, *d, **e;
3   a = 3;   b = 5;
4   c = &b;  d = &a;
5   *c = 4;
6   *d = (*c) + 3;
7   e = &c;
8   **e = *d;
9   *e = &a;
10 }
    
```

✘ Avec une double étoile, on suit deux pointeurs à la suite.

## Exemple de manipulations de pointeurs



```

1 int main() {
2   int a, b, *c, *d, **e;
3   a = 3;   b = 5;
4   c = &b;  d = &a;
5   *c = 4;
6   *d = (*c) + 3;
7   e = &c;
8   **e = *d;
9   ► *e = &a;
10 }
    
```

- × Deux pointeurs peuvent contenir la même adresse
  - × on peut faire un test d'égalité dessus (`c == d`),
  - × on peut aussi comparer les contenus (`(*c) == (*d)`).

## Pointeurs et tableaux

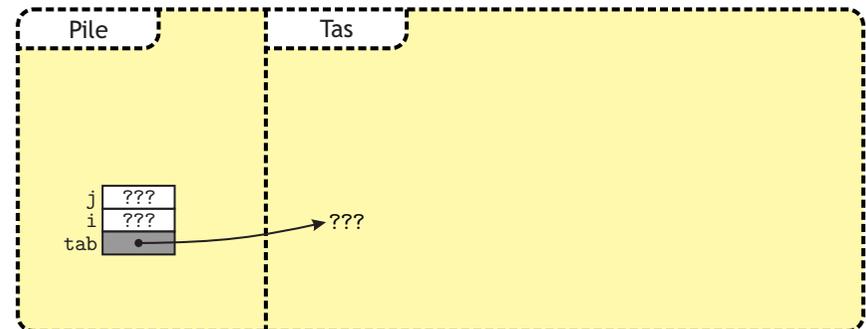
## Tableaux et allocation dynamique

- × Un tableau (dynamique) en C est simplement un pointeur :
  - × c'est l'adresse mémoire du premier élément du tableau,
    - `tab[0]` et `*tab` sont équivalents.
- × On déclare un tableau comme on déclare un pointeur,
  - × il faut ensuite allouer de la mémoire avec `malloc`,
  - × `malloc` prend en argument le **nombre d'octets à réserver**
  - × `sizeof` permet de connaître la taille (en octets) d'un type
    - la taille peut changer d'une architecture à une autre.
- × Le type de retour de `malloc` est `void*`
  - × c'est un pointeur sans type
    - on utilise un cast pour le convertir dans le bon type.
- × `void*` n'a rien à voir avec le type de retour `void` d'une fonction...

## Tableaux et allocation dynamique

```

1 int main() {
2   ► int* tab,i,j; // déclaration
3   tab = (int*) malloc(5*sizeof(int)); // allocation
4   tab[3] = 5;
5   printf("%d\n",tab[5]);
6 }
    
```

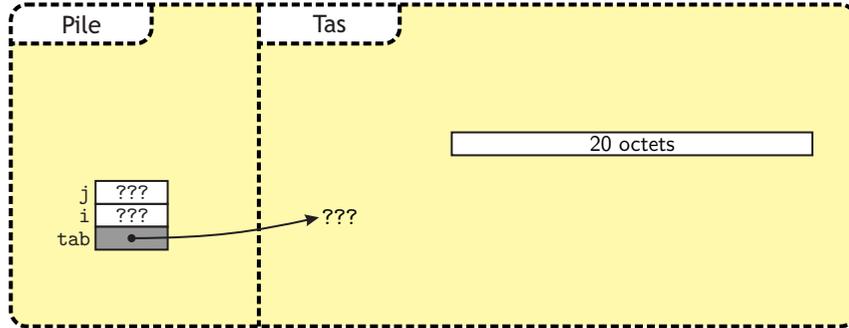


## Tableaux et allocation dynamique

```

1 int main() {
2   int* tab,i,j;           // déclaration
3   ▶ tab = (int*) malloc(5*sizeof(int)); // allocation
4   tab[3] = 5;
5   printf("%d\n",tab[5]);
6 }

```

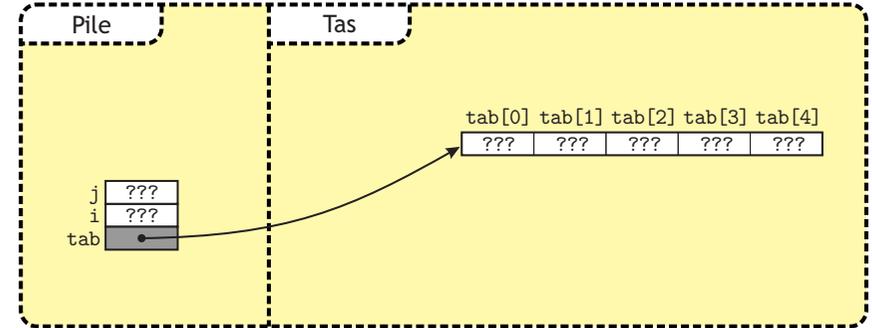


## Tableaux et allocation dynamique

```

1 int main() {
2   int* tab,i,j;           // déclaration
3   ▶ tab = (int*) malloc(5*sizeof(int)); // allocation
4   tab[3] = 5;
5   printf("%d\n",tab[5]);
6 }

```

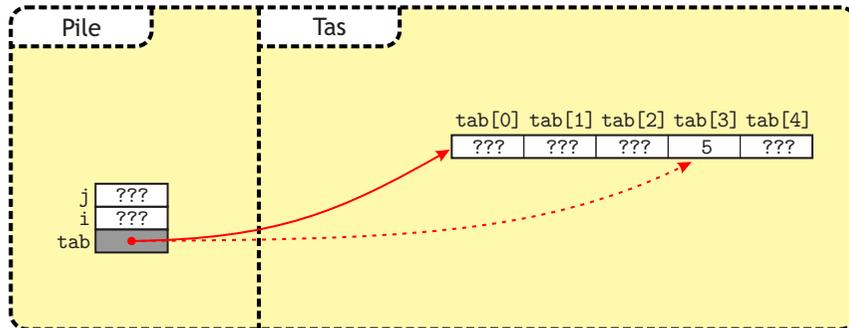


## Tableaux et allocation dynamique

```

1 int main() {
2   int* tab,i,j;           // déclaration
3   tab = (int*) malloc(5*sizeof(int)); // allocation
4   ▶ tab[3] = 5;
5   printf("%d\n",tab[5]);
6 }

```

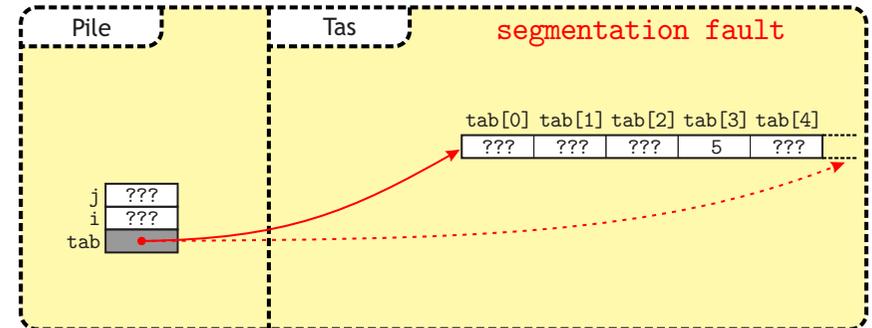


## Tableaux et allocation dynamique

```

1 int main() {
2   int* tab,i,j;           // déclaration
3   tab = (int*) malloc(5*sizeof(int)); // allocation
4   tab[3] = 5;
5   ▶ printf("%d\n",tab[5]);
6 }

```



## Différences entre tableaux dynamiques et statiques

- ✘ Les tableaux statiques et dynamiques s'utilisent de façon similaire, mais sont assez différents en pratique.
- ✘ Les tableaux statiques :
  - ✘ sont alloués directement dans la pile
    - automatiquement libérés en fin de fonction,
  - ✘ n'ont pas de "case" contenant un pointeur
    - avec `int d[4]`; on ne peut pas affecter de valeur à d.
- ✘ Les tableaux dynamiques :
  - ✘ sont alloués dans le tas avec `malloc`,
  - ✘ persistent à la fin d'une fonction
    - on peut renvoyer leur adresse avec `return`,
  - ✘ doivent être libérés spécifiquement
    - la commande `free(tab)`; libère le bloc mémoire à l'adresse contenue dans la variable `tab`.

## La fonction free

- ✘ À chaque `malloc` doit correspondre un `free` :
  - ✘ chaque bloc de mémoire alloué doit être libéré explicitement,
    - il peut alors être réalloué (à un autre programme),
  - ✘ la mémoire est toujours libérée (par l'OS) à la fin d'un programme
    - la mémoire n'est pas définitivement perdue!
- ✘ Attention en particulier aux allocations dans une boucle/fonction :
  - ✘ il faut libérer la mémoire dès qu'on n'en a plus besoin,
  - ✘ sinon, on peut avoir des fuites de mémoire :
    - utilise plus de mémoire que nécessaire.
  - ✘ éviter aussi les blocs perdus,
    - blocs alloués dont l'adresse n'est plus stockée nul part...

## Le pointeur NULL

- ✘ Le pointeur `NULL` est l'équivalent d'un 0 pour les pointeurs.
  - ✘ l'adresse mémoire 0 n'est jamais utilisée
  - ✘ un pointeur égal à `NULL` n'est donc pas une vraie adresse
    - permet de faire des tests (cf. listes chaînées),
  - ✘ suivre un pointeur `NULL` fait toujours une erreur de segmentation
    - pas forcément le cas pour un pointeur non initialisé.
- ✘ La fonction `malloc` peut ne pas réussir à allouer la mémoire (mémoire pleine, trop gros bloc...)
  - ✘ elle renvoie alors un pointeur `NULL`,
  - ✘ il faudrait toujours tester la valeur de retour de `malloc`.

```

1 int* tab = (int*) malloc(1000000*sizeof(int));
2 if (tab == NULL) {
3     printf("Out of memory...\n");
4     exit(1);
5 }

```

## Autres fonctions pour manipuler des zones mémoires

- ✘ Il existe d'autres fonctions de `stdlib.h` pour manipuler la mémoire :
  - ✘ `void* calloc(int nb, int size);`
    - alloue un tableau de `nb` cases de taille `size` initialisées à 0.
  - ✘ `void* memcpy(void* dest, void* src, int size);`
    - recopie un bloc mémoire de taille `size` de `src` vers `dest`.
  - ✘ `void* realloc(void* src, int size);`
    - alloue un bloc mémoire de taille `size` et recopie le contenu de `src` dedans.
- ✘ Elles peuvent être pratiques, mais sont remplaçables par des `malloc`, `free` et une boucle `for`.

## Tableaux à 2 dimensions

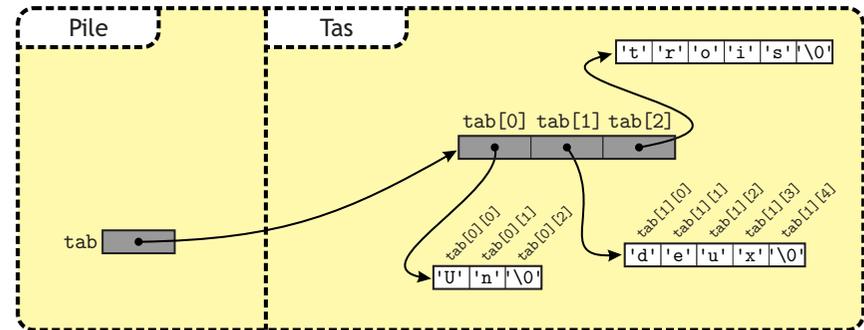
- × Un tableau à 2 dimensions est un tableau de pointeurs :
  - × le type `int** tab` peut aussi se lire `int* *tab`
  - × il faut donc allouer le tableau de pointeurs, puis allouer chaque tableau d'entiers (ou autre).
  - × il faut aussi libérer tous les tableaux...

```

1 int i;
2 int** tab = (int**) malloc(20*sizeof(int*));
3 for (i=0; i<20; i++) {
4     tab[i] = (int*) malloc(30*sizeof(int));
5 }
6 ...
7 for (i=0; i<20; i++) {
8     free(tab[i]);
9 }
10 free(tab);
    
```

## Tableaux à 2 dimensions

- × Un tableau à 2 dimensions n'est pas forcément rectangulaire :
  - × c'est simplement un tableau de pointeurs
    - chaque pointeur pointe vers un tableau qui a une taille propre.
  - × par exemple l'argument `char* argv[]` du `main`
    - les chaînes de caractères ont des longueurs différentes.

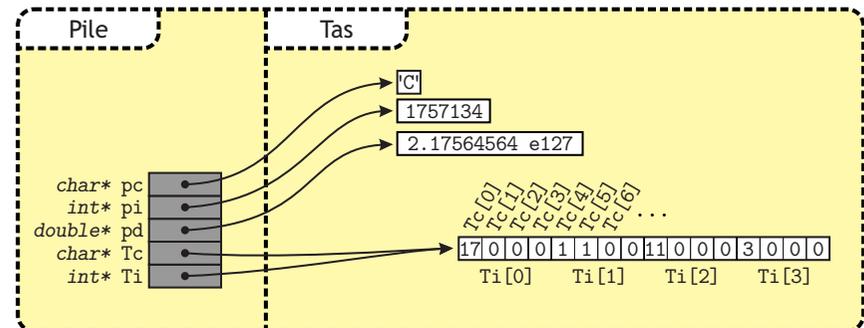


## Pourquoi un pointeur doit il avoir un type ?

- × Si un pointeur n'est que le numéro d'une case mémoire, pourquoi a-t-il un type associé ?
  - × cela permet de savoir la taille de ce qu'il faut lire au bout du pointeur.
- × Quand on lit la valeur de `*a` :
  - × si `a` est un `char*`, on ne lit qu'un octet,
  - × si `a` est un `int*`, on lit 4 octets,
  - × si `a` est un `double*`, on lit 8 octets, et on les lit comme un `double`, pas un entier.
- × De même pour un tableau, `tab[i]` n'est pas le  $i$ -ème octet, mais le  $i$ -ème élément :
  - × il faut savoir de combien d'octets avancer ( $i \times 4$ ,  $i \times 8$ ...),
  - × il faut savoir la taille de ce qu'on lit.

## Pourquoi un pointeur doit il avoir un type ?

- × Selon le type du pointeur, le programme ne va pas lire la même taille de bloc en le suivant.



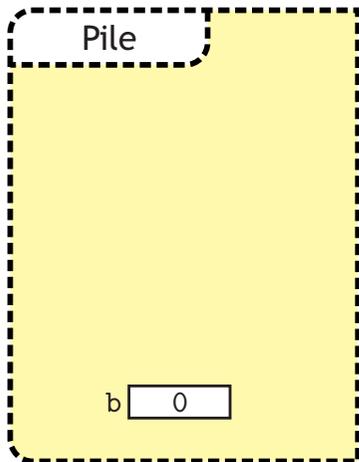
# Pointeurs et fonctions

## Passage par adresse

- × Lors d'un appel de fonction les paramètres sont évalués et copiés
  - × impossible de modifier une variable dans une fonction appelée,
  - × sauf si au lieu de passer sa valeur on passe son adresse
    - c'est alors l'adresse qui est copiée,
    - × la fonction peut alors écrire à l'adresse qu'on lui a passée.
- × C'est ce que fait la fonction scanf.

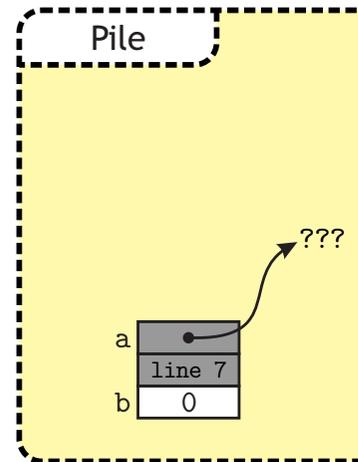
```
1 void incr(int* a) {  
2     (*a) = (*a) + 1; // on écrit à l'adresse vers  
3 } // laquelle pointe a  
4  
5 int main() {  
6     int b = 0;  
7     incr(&b); // on passe l'adresse de b  
8     printf("%d\n",b); // cela affiche 1  
9 }
```

## Passage par adresse Ce qui se passe en mémoire



```
1 void incr(int* a) {  
2     (*a) = (*a) + 1;  
3 }  
4  
5 int main() {  
6     int b = 0;  
7     incr(&b);  
8     printf("%d\n",b);  
9 }
```

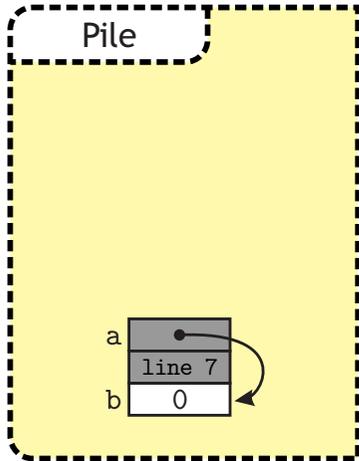
## Passage par adresse Ce qui se passe en mémoire



```
1 void incr(int* a) {  
2     (*a) = (*a) + 1;  
3 }  
4  
5 int main() {  
6     int b = 0;  
7     incr(&b);  
8     printf("%d\n",b);  
9 }
```

## Passage par adresse

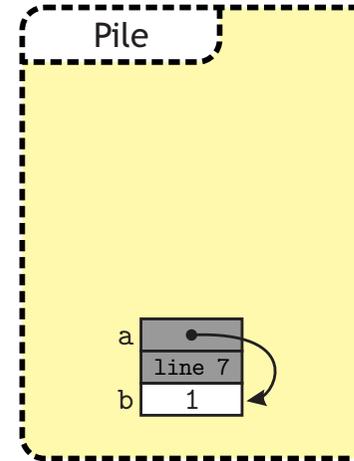
Ce qui se passe en mémoire



```
1 void incr(int* a) {  
2   (*a) = (*a) + 1;  
3 }  
4  
5 int main() {  
6   int b = 0;  
7   ▶ incr(&b);  
8   printf("%d\n",b);  
9 }
```

## Passage par adresse

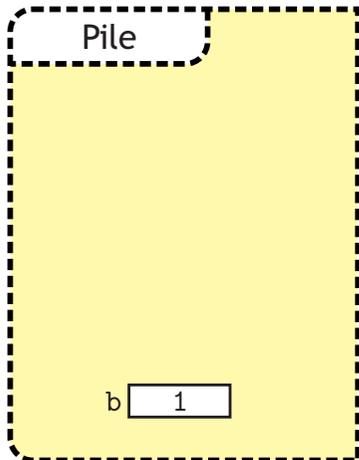
Ce qui se passe en mémoire



```
1 void incr(int* a) {  
2   ▶ (*a) = (*a) + 1;  
3 }  
4  
5 int main() {  
6   int b = 0;  
7   incr(&b);  
8   printf("%d\n",b);  
9 }
```

## Passage par adresse

Ce qui se passe en mémoire



```
1 void incr(int* a) {  
2   (*a) = (*a) + 1;  
3 }  
4  
5 int main() {  
6   int b = 0;  
7   incr(&b);  
8   ▶ printf("%d\n",b);  
9 }
```

## Fonction retournant un tableau

- ✗ Une fonction ne peut renvoyer qu'une seule valeur
  - ✗ mais elle peut renvoyer une adresse mémoire!
- ✗ Une fonction qui doit **modifier** le contenu d'un tableau utilisera du passage par adresse → c'est la façon naturelle avec un tableau.
- ✗ Une fonction qui doit **allouer** un tableau peut renvoyer son adresse :

```
1 int* init(int n) {  
2   int i;  
3   int* tab = (int*) malloc(n*sizeof(int));  
4   for (i=0; i<n; i++) {  
5     tab[i] = i;  
6   }  
7   return tab;  
8 }  
9 int main() {  
10  int* T = init(5);  
11  printf("%d\n", T[3]);  
12  free(T);  
13 }
```

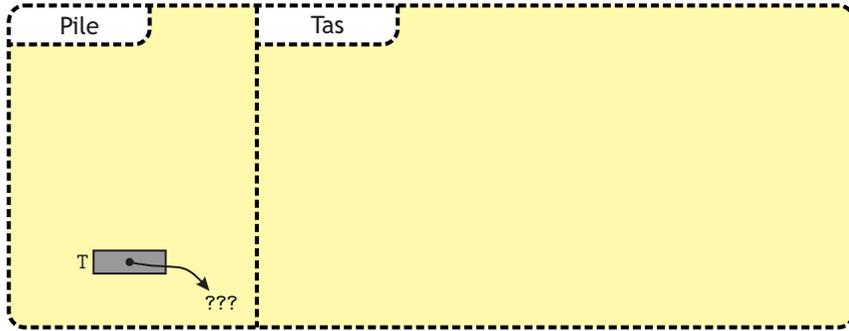
## Fonction retournant un tableau

Ce qui se passe en mémoire

```

1 int* init(int n) {
2   int i;
3   int* tab = (int*) malloc(
4     n*sizeof(int));
5   for (i=0; i<n; i++) {
6     tab[i] = i;
7   }
8   return tab;
9 }
10 int main() {
11   int* T = init(5);
12   printf("%d\n", T[3]);
13   free(T);
14 }

```



## Fonction retournant un tableau

Ce qui se passe en mémoire

```

1 int* init(int n) {
2   int i;
3   int* tab = (int*) malloc(
4     n*sizeof(int));
5   for (i=0; i<n; i++) {
6     tab[i] = i;
7   }
8   return tab;
9 }
10 int main() {
11   int* T = init(5);
12   printf("%d\n", T[3]);
13   free(T);
14 }

```



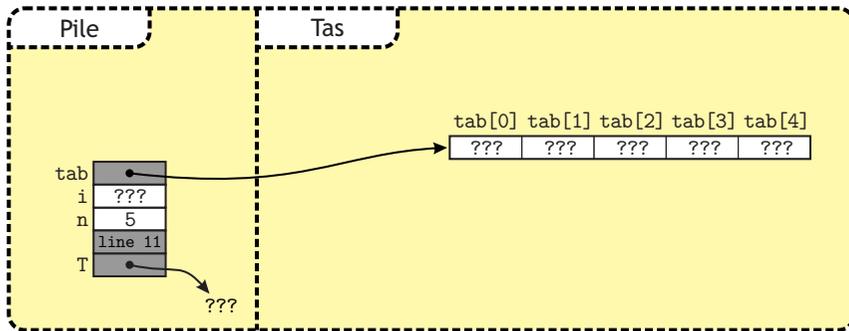
## Fonction retournant un tableau

Ce qui se passe en mémoire

```

1 int* init(int n) {
2   int i;
3   int* tab = (int*) malloc(
4     n*sizeof(int));
5   for (i=0; i<n; i++) {
6     tab[i] = i;
7   }
8   return tab;
9 }
10 int main() {
11   int* T = init(5);
12   printf("%d\n", T[3]);
13   free(T);
14 }

```



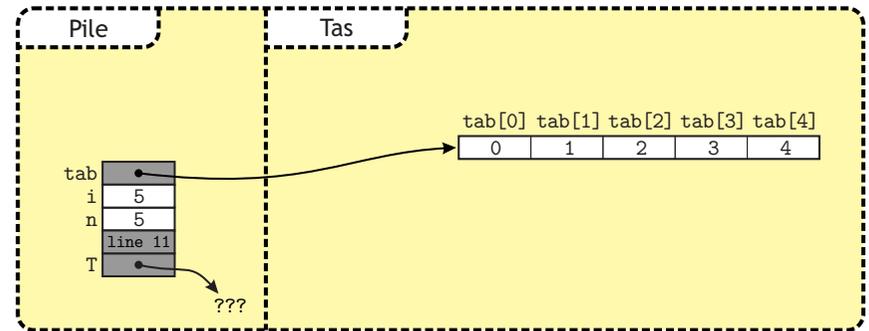
## Fonction retournant un tableau

Ce qui se passe en mémoire

```

1 int* init(int n) {
2   int i;
3   int* tab = (int*) malloc(
4     n*sizeof(int));
5   for (i=0; i<n; i++) {
6     tab[i] = i;
7   }
8   return tab;
9 }
10 int main() {
11   int* T = init(5);
12   printf("%d\n", T[3]);
13   free(T);
14 }

```



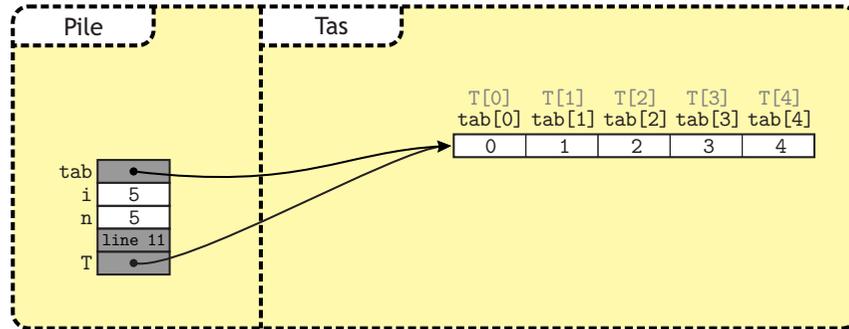
## Fonction retournant un tableau

Ce qui se passe en mémoire

```

1 int* init(int n) {
2   int i;
3   int* tab = (int*) malloc(
4     n*sizeof(int));
5   for (i=0; i<n; i++) {
6     tab[i] = i;
7   }
8   return tab;
9 }
10 int main() {
11   int* T = init(5);
12   printf("%d\n", T[3]);
13   free(T);
14 }

```



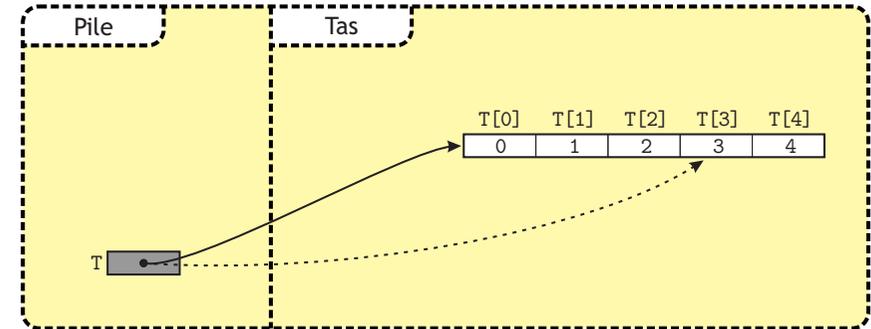
## Fonction retournant un tableau

Ce qui se passe en mémoire

```

1 int* init(int n) {
2   int i;
3   int* tab = (int*) malloc(
4     n*sizeof(int));
5   for (i=0; i<n; i++) {
6     tab[i] = i;
7   }
8   return tab;
9 }
10 int main() {
11   int* T = init(5);
12   printf("%d\n", T[3]);
13   free(T);
14 }

```



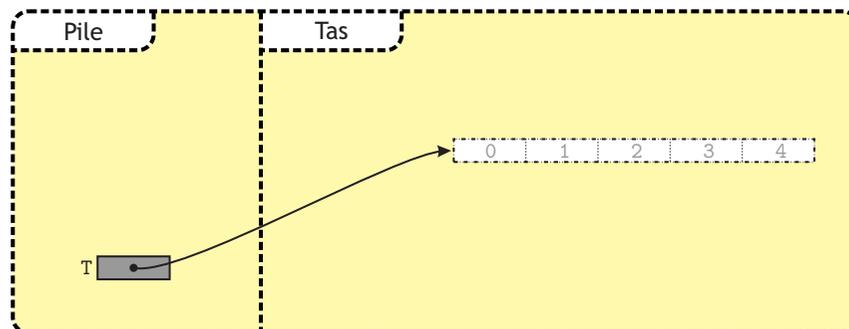
## Fonction retournant un tableau

Ce qui se passe en mémoire

```

1 int* init(int n) {
2   int i;
3   int* tab = (int*) malloc(
4     n*sizeof(int));
5   for (i=0; i<n; i++) {
6     tab[i] = i;
7   }
8   return tab;
9 }
10 int main() {
11   int* T = init(5);
12   printf("%d\n", T[3]);
13   free(T);
14 }

```



## Pointeurs de fonctions

- ✗ En C il est possible de passer une fonction en argument à une autre, ou de faire un tableau de fonctions
  - ✗ on utilise alors un **pointeur de fonction**.
- ✗ Le pointeur de fonction est un pointeur comme un autre, mais qui pointe vers une adresse mémoire où se trouve du code exécutable,
  - ✗ son type dépend de la **signature de la fonction**.

```

1 int add(int a, int b) { return a+b; }
2 int mul(int a, int b) { return a*b; }
3 int main() {
4   int (*func) (int, int) = &add; // déclaration
5   printf("%d\n", func(3,5));      // affiche 8
6   func = &mul;
7   printf("%d\n", func(3,5));      // affiche 15
8   return 0;
9 }

```

## Pointeurs de fonctions

### Exemple de fonction passée en argument

```

1 int max(void* tab, int n, int elem_size, int (*comp) (void*, void*)) {
2     int i;
3     int max_pos = 0;
4     for (i=1; i<n; i++) {
5         if (comp(tab+(max_pos*elem_size), tab+(i*elem_size)) < 0) {
6             max_pos = i;
7         }
8     }
9     return max_pos;
10 }
11 int double_comp(double* a, double* b) {
12     if ((*a) > (*b)) { return 1; }
13     if ((*a) < (*b)) { return -1; }
14     return 0;
15 }
16 int main() {
17     double tab[5] = {1.5, 3.2, 7.9, 12.3, -5.2};
18     printf("%d\n", max(tab, 5, sizeof(double),
19                       (int (*)(void*, void*)) &double_comp));
20     return 0;
21 }

```

## Ce qu'il faut retenir de ce cours

- ✘ Les pointeurs ne sont rien d'autres que des adresses mémoires
  - ✘ ce n'est pas compliqué à manipuler, mais il faut bien garder en tête ce qu'il se passe dans la mémoire,
  - ✘ l'étoile \* a plusieurs significations en C,
    - pensez à mettre des parenthèses pour éviter les ambiguïtés.
- ✘ Les pointeurs ont de très nombreuses utilisations :
  - ✘ tableaux et tableaux de dimension supérieure,
  - ✘ passage d'arguments de fonctions par adresse
    - permet de modifier un argument de la fonction
  - ✘ allocation de zones mémoires
    - vous faites ce que vous voulez dans une zone allouée.
- ⚠ Pensez à toujours initialiser vos pointeurs à NULL et à utiliser des free pour libérer la mémoire.