

Langage C
ENSTA - TC 1ère année

François Pessaux

U2IS

2017-2018

`francois.pessaux@ensta-paristech.fr`

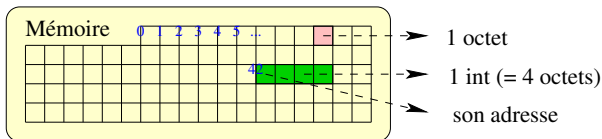
Organisation mémoire et pointeurs

- Pour le processeur : mémoire = grand tableau fini d'octets :
 - ▶ de taille 2^{32} sur machine 32 bits,
 - ▶ de taille 2^{64} sur machine 64 bits.
- Tout le « *tableau* » n'est pas utilisable :
 - ▶ Une partie de la mémoire est utilisée par les autres processus.
 - ▶ Certaines zones sont réservées au système.
 - ▶ Certaines zones peuvent être restreintes en accès (lecture/écriture/exécution).
 - ▶ Certaines zones peuvent être « *non connectées* » (pas de mémoire physique présente).

- Ressource en quantité **finie** ⇒ gestion de ressource :
 - En **demander** au besoin : **allocation**.
 - La **restituer** : **libération** (« *désallocation* »).
- Arbitrage et autorisation d'accès gérés par le système d'exploitation (« *OS* »).
- Allocation et libération via des **appels systèmes**.
- Accès à une zone non attribuée au processus → « *segmentation fault* ».
- ⇒ **Toujours avoir demandé** la mémoire que l'on veut utiliser.

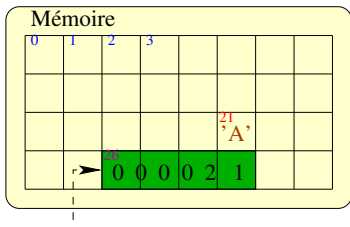
Adresse mémoire

- Les informations, octets, `int`'s, chaînes sont stockées en mémoire.
- \Rightarrow Sont logées à un « *certain endroit* » de la mémoire : **adresse**.
- **Adresse** \approx index de l'info dans le grand « *tableau* » de la mémoire.



Adresses et pointeurs (1)

- Une adresse est une information comme une autre.
- ⇒ Besoin de la mémoriser, manipuler et d'accéder à son contenu.
- Une valeur « *adresse* » est un *pointeur*.
- Une *variable* contenant une adresse est de *type pointeur*.



char c = 'A'

Adresse c = 21

- Pointeur sur c
contient la valeur 21

Adresse du pointeur = 26

- À une adresse se trouve une donnée d'un certain *type*.
▶ ⇒ Un pointeur est une *adresse* avec un *type* associé : le type des données stockées à cette adresse.

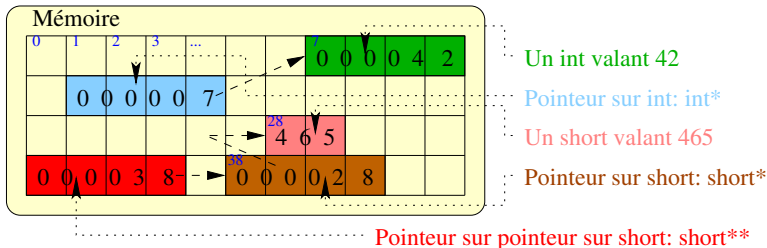
Adresses et pointeurs (2)

- Un pointeur désigne un **point** dans la mémoire :
 - emplacement d'une donnée,
 - ou emplacement du début d'une zone contiguë de données.



- Déclaration de pointeurs en C :

```
int *a ;           // a est un pointeur vers un/des int.
short *b ;        // b est un pointeur vers un/des short (int).
short **c ;       // c est un pointeur vers un/des short*.
```



Adresses et pointeurs (3)

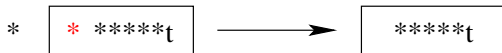
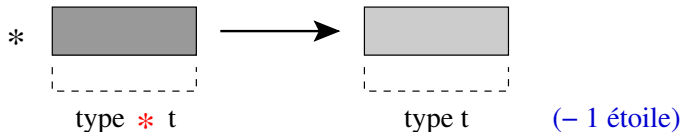
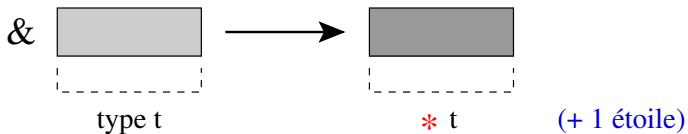
- Chaque **variable** possède **une** adresse.
- Obtenir l'adresse d'une variable : opérateur préfixe **&**.
- Accéder au **contenu** d'une adresse : opérateur préfixe *****.

```
int a ;           // a est un int.
int *b ;         // b est un pointeur vers un int.
b = &a ;        // b contient l'adresse de a -> Pointeur vers a.
*b = 5 ;        // On stocke 5 dans *b. -> Dans a.
a = 1 + *b ;    // On stocke dans a 1 + la valeur contenue à
                // l'adresse b. -> a = a + 1.
```

- **&** et ***** ont des rôles symétriques : ***(&a)** est la même chose que **a**.

ATTENTION : avoir un pointeur sous la main **ne** signifie **pas** avoir le droit de manipuler la mémoire à l'adresse contenue par ce pointeur !

Adresses et pointeurs (4)



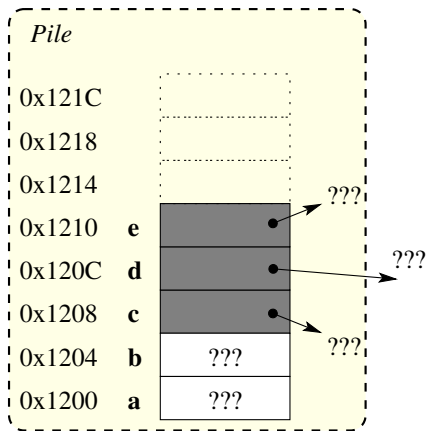
Pourquoi un pointeur doit avoir un type (1)

- Une adresse est juste un entier positif.
- En mémoire on stocke des infos d'un certain type :
 - ▶ ⇒ donc d'une certaine forme.
- L'accès à une info dépend de la forme de cette info.
 - ▶ ⇒ L'accès à une info par un pointeur dépend de la forme de l'info contenue à cette adresse.
- ⇒ Un pointeur doit spécifier l'adresse de « *quoi* » il est.
- Les pointeurs sont donc typés
 - ▶ `char*` : pointeur sur `char`.
 - ▶ `int**` : pointeur sur `int*`.

Pourquoi un pointeur doit avoir un type (2)

- Quand on lit la valeur de `*v` :
 - 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.
- Les tableaux sont désignés par l'adresse de leur 1^{ère} case.
 - `t[i]` n'est pas le *i*^{ème} octet, mais le *i*^{ème} élément.
 - ⇒ Il faut savoir de combien d'octets avancer ($i \times 4$, $i \times 8$...)
 - ⇒ Nécessaire de connaître la taille d'un élément → donc son type.
- Rappel (cours 4) : accès au champ d'un pointeur de type `struct` :
 - `struct foo_t { int ch ; };`
 - `struct foo_t my_foo ;`
 - Accès au champ `ch` par notation pointée : `my_foo.ch`
 - `struct foo_t *my_pfoo ;`
 - Accès au champ `ch` par notation fléchée : `my_pfoo->ch`
 - Équivalence : `my_pfoo->ch = (*my_pfoo).ch`

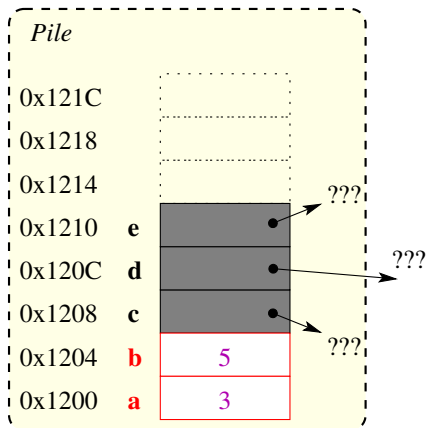
Exemple de manipulations de pointeurs (1)



```
int main ()  
{  
▶ int a, b, *c, *d, **e ;  
  a = 3 ;    b = 5 ;  
  c = &b ;   d = &a ;  
  *c = 4 ;  
  *d = (*c) + 3 ;  
  e = &c ;  
  **e = *d ;  
  *e = &a ;  
  ...  
}
```

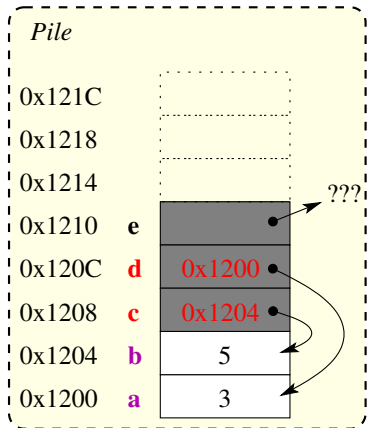
- Chaque case mémoire a une adresse.
- Par habitude et simplicité, adresses écrites en hexadécimal.

Exemple de manipulations de pointeurs (2)



```
int main ()  
{  
    int a, b, *c, *d, **e ;  
    ▶ a = 3 ; b = 5 ;  
      c = &b ; d = &a ;  
      *c = 4 ;  
      *d = (*c) + 3 ;  
      e = &c ;  
      **e = *d ;  
      *e = &a ;  
      ...  
}
```

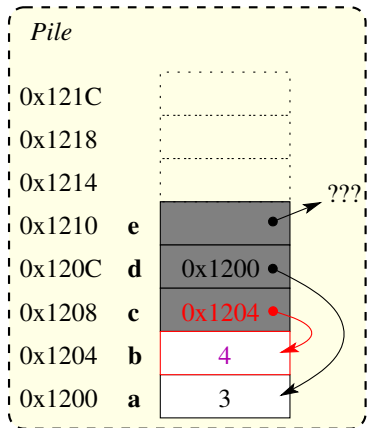
Exemple de manipulations de pointeurs (3)



```
int main ()  
{  
    int a, b, *c, *d, **e ;  
    a = 3 ;    b = 5 ;  
    ▶ c = &b ; d = &a ;  
    *c = 4 ;  
    *d = (*c) + 3 ;  
    e = &c ;  
    **e = *d ;  
    *e = &a ;  
    ...  
}
```

- c et d contiennent les adresses de b et a.
 - ▶ Ils « pointent » vers b et a.

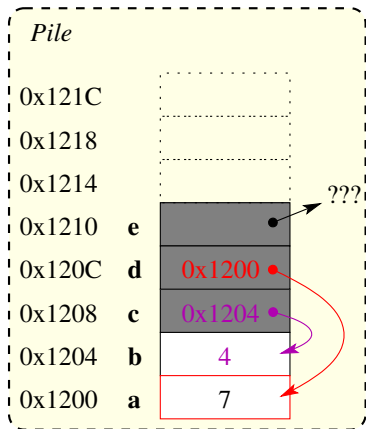
Exemple de manipulations de pointeurs (4)



```
int main ()  
{  
    int a, b, *c, *d, **e ;  
    a = 3 ;    b = 5 ;  
    c = &b ;   d = &a ;  
    ▶ *c = 4 ;  
    *d = (*c) + 3 ;  
    e = &c ;  
    **e = *d ;  
    *e = &a ;  
    ...  
}
```

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

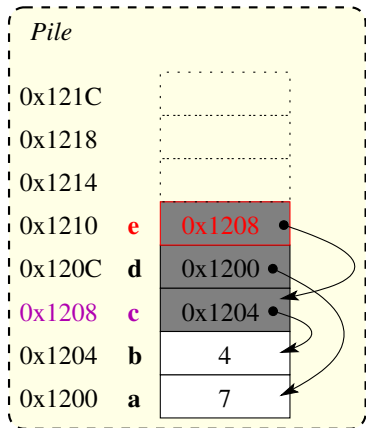
Exemple de manipulations de pointeurs (5)



```
int main ()  
{  
    int a, b, *c, *d, **e ;  
    a = 3 ;    b = 5 ;  
    c = &b ;   d = &a ;  
    *c = 4 ;  
    ▶ *d = (*c) + 3 ;  
    e = &c ;  
    **e = *d ;  
    *e = &a ;  
    ...  
}
```

- À gauche du "=" : case mémoire où ira le résultat.
- À droite du "=" : on évalue, i.e. on suit le pointeur `c` et on regarde ce qui est dans la case.

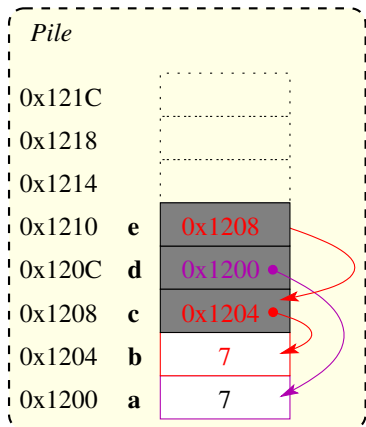
Exemple de manipulations de pointeurs (6)



● e pointe vers c.

```
int main ()  
{  
    int a, b, *c, *d, **e ;  
    a = 3 ;    b = 5 ;  
    c = &b ;   d = &a ;  
    *c = 4 ;  
    *d = (*c) + 3 ;  
    ▶ e = &c ;  
    **e = *d ;  
    *e = &a ;  
    ...  
}
```

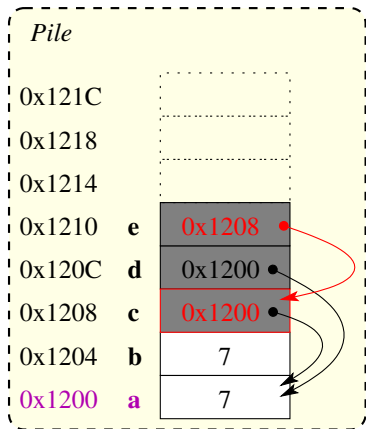
Exemple de manipulations de pointeurs (7)



```
int main ()  
{  
    int a, b, *c, *d, **e ;  
    a = 3 ;    b = 5 ;  
    c = &b ;   d = &a ;  
    *c = 4 ;  
    *d = (*c) + 3 ;  
    e = &c ;  
    **e = *d ;  
    *e = &a ;  
    ...  
}
```

- Double étoile : on suit deux pointeurs à la suite
- I.e. pointeur sur un pointeur.
- I.e. variable qui contient (l'adresse d'une variable qui contient (l'adresse d'une variable)).

Exemple de manipulations de pointeurs (8)



```
int main ()  
{  
    int a, b, *c, *d, **e ;  
    a = 3 ;    b = 5 ;  
    c = &b ;   d = &a ;  
    *c = 4 ;  
    *d = (*c) + 3 ;  
    e = &c ;  
    **e = *d ;  
    ▶ *e = &a ;  
    ...  
}
```

- Deux pointeurs peuvent contenir la même adresse (« *alias* »).
 - ▶ On peut faire un **test d'égalité** dessus (`c == d`).
 - ▶ On peut aussi comparer les **contenus** (`*c == *d`).

Allocation et libération de mémoire

Statique versus dynamique (1)

- Allocation statique : effectuée à la **compilation**.
- Ex : variables entières déclarées, tableaux de taille fixe :
 - `int v = 42 ;`
 - `char s[10] ;`
 - Dans la pile (variable locale) ou dans le tas/section-data (variable globale).
- Allocation **statique locale** : automatiquement libérée en fin de fonction.
 - ▶ Comment faire si on veut « *faire durer* » la donnée plus longtemps ?

Statique versus dynamique (2)

- Parfois, mémoire nécessaire connue seulement lors de l'exécution :
 - ▶ Lecture du contenu d'un fichier : taille des fichiers variable.
 - ▶ Résolution d'un système d'équations : nombre d'inconnues à la demande de l'utilisateur.
- Changer le programme manuellement \Rightarrow recompiler ?
- \Rightarrow Inefficace, non-maintenable, inacceptable.
- On demande de la mémoire selon les besoins à l'exécution.
 - ▶ Zone allouée retournée désignée par son adresse de début.

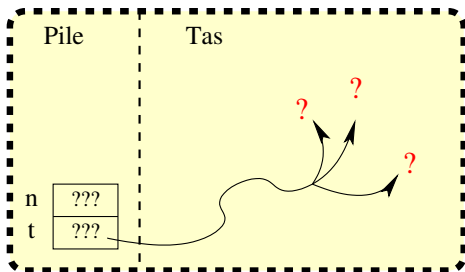
- Allocation de n'importe quoi : entier, entiers (tableau), **struct**, etc.
- Tableau dynamique C \equiv type **pointeur** dont la **valeur** est l'**adresse** de début du tableau.
 - ▶ $t[0] \equiv *t$.
 - ▶ $\&(t[0]) \equiv t$.
- On **déclare** un tableau comme un pointeur.
 - ▶ Le pointeur est alors **non initialisé** (pointe « *n'importe où* »).
 - ▶ On **alloue** de la mémoire (fonction **malloc**) \leadsto une adresse ...
 - ▶ ... et on enregistre cette adresse dans le pointeur.

Allocation dynamique : malloc

- `malloc` : `int` → `void*`
- Argument : nombre d'octets demandés.
- Résultat : adresse du début de la zone octroyée.
- `void*` : pointeur sans type
 - Implicitement compatible avec autres types de pointeurs.
 - ⇒ On ne le « cast » pas!
- Si mémoire non disponible : résultat = pointeur `NULL`.
- ⇒ Toujours tester la réussite de l'allocation!
 - ▶ `if (ptr == NULL)` gérer l'erreur
 - ▶ Sinon écriture dans une zone non allouée et « *segmentation fault* » à la clef.

Pas à pas (1)

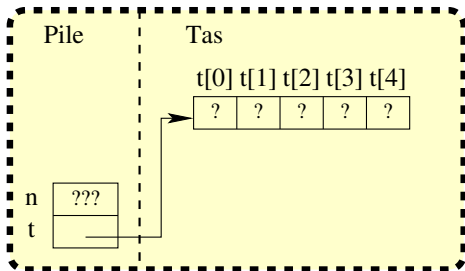
```
#include <stdio.h> // printf
#include <stdlib.h> // malloc, free
int main ()
{
    float *t ;
    int n ;
    t = malloc (5 * sizeof (float)) ;
    if (t == NULL) exit (-1) ;
    t[2] = 3.14159 ;
    printf ("%f\n", t[2]) ;
    free (t) ;
    return (0) ;
}
```



- Variables locales **t** et **n** allouées sur la pile...
- ... mais pas initialisées.
- ⇒ Pointeur **t** désigne une adresse quelconque...
 - ▶ ... et vraisemblablement invalide.

Pas à pas (2)

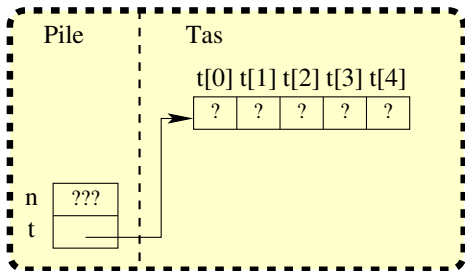
```
#include <stdio.h> // printf
#include <stdlib.h> // malloc, free
int main ()
{
    float *t ;
    int n ;
    ▶ t = malloc (5 * sizeof (float)) ;
    if (t == NULL) exit (-1) ;
    t[2] = 3.14159 ;
    printf ("%f\n", t[2]) ;
    free (t) ;
    return (0) ;
}
```



- **sizeof** : construction du compilateur, → taille d'une donnée en octets.
 - ▶ N'est pas une fonction.
 - ▶ Taille élémentaire : dépend de plein de choses (architecture, modèle de compilation, etc.).
- Appel à la fonction d'allocation de mémoire `malloc`.
- Assignation du résultat au pointeur `t`.
- ⇒ `t` désormais **initialisé**...
- ... Mais à quoi ?

Pas à pas (3)

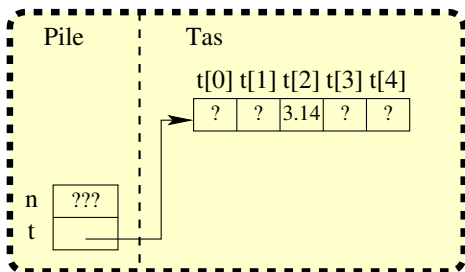
```
#include <stdio.h> // printf
#include <stdlib.h> // malloc, free
int main ()
{
    float *t ;
    int n ;
    t = malloc (5 * sizeof (float)) ;
    ► if (t == NULL) exit (-1) ;
    t[2] = 3.14159 ;
    printf ("%f\n", t[2]) ;
    free (t) ;
    return (0) ;
}
```



- Si pas de mémoire octroyée, pointeur retourné alors **NULL**.
- Alors pas possible de continuer (ou gérer l'erreur « *comme il se doit* »).
- **NULL** désigne une adresse **interdite en lecture/écriture**
 - ⇒ « *Segmentation Fault* ».

Pas à pas (4)

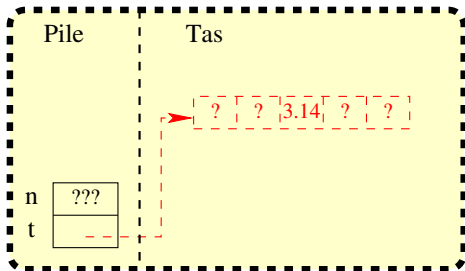
```
#include <stdio.h> // printf
#include <stdlib.h> // malloc, free
int main ()
{
    float *t ;
    int n ;
    t = malloc (5 * sizeof (float)) ;
    if (t == NULL) exit (-1) ;
    ▶ t[2] = 3.14159 ;
    printf ("%f\n", t[2]) ;
    free (t) ;
    return (0) ;
}
```



- Écriture dans une partie de la mémoire allouée.
- Autorisé puisque la mémoire « *nous* » appartient.

Pas à pas (5)

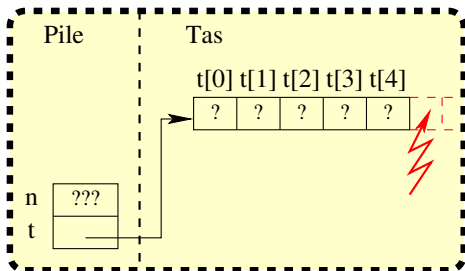
```
#include <stdio.h> // printf
#include <stdlib.h> // malloc, free
int main ()
{
    float *t ;
    int n ;
    t = malloc (5 * sizeof (float)) ;
    if (t == NULL) exit (-1) ;
    t[2] = 3.14159 ;
    printf ("%f\n", t[2]) ;
    free (t) ;
    return (0) ;
}
```



- Rendre la mémoire plus utilisée au système → éviter la pénurie.
- N'écrite pas la mémoire.
- N'écrite pas le pointeur.
- Révoque simplement l'autorisation d'accès.
- ⇒ On a encore les moyens « techniques » d'y accéder...
- ... mais « Segmentation Fault » guette.

Pas à pas (5^{1/2})

```
#include <stdio.h> // printf
#include <stdlib.h> // malloc, free
int main ()
{
    float *t ;
    int n ;
    t = malloc (5 * sizeof (float)) ;
    if (t == NULL) exit (-1) ;
    ▶ t[5] = 3.14159 ;
    printf ("%f\n", t[2]) ;
    free (t) ;
    return (0) ;
}
```



- Tentative d'accès à la case 5 (6^{ème} « case »)...
- Accès en **dehors** de la zone allouée.
- Peut « *appartenir* » à un autre (système, processus ...)
- ⇒ « *Segmentation Fault* » à la clef !
- **Mais** ... c'est pas forcément immédiat ! ☹
 - ▶ Bugs diaboliques à trouver et rendre reproductibles.

- Fonction **free** : **void*** → **void**
 - ▶ Argument : adresse de **début** de la zone à libérer.
 - ▶ Résultat : néant.
- À **chaque malloc** doit correspondre **1 et 1 seul free**.
- ⇒ La libération de mémoire est **explicite**.
- Taille de la zone connue par le système : pas besoin de la spécifier.
- Libérer au plus tôt : inutile de consommer de la mémoire inutilement.
 - ▶ → Pourra être ré-attribuée à quelqu'un d'autre.
- Ne pas libérer « **trop** » tôt.
 - ▶ Tant qu'une zone mémoire **accessible** y fait référence, la zone allouée **ne doit pas** être libérée.

- Ne pas initialiser un pointeur (i.e. ne pas allouer via `malloc`).
- Accéder en dehors d'une zone allouée (débordement).
- Faire `free` dans une boucle (désallocations multiples de la zone).
- Faire `free` avec une adresse non allouée dynamiquement.
- Faire `free` avec une adresse qui n'est pas le **début** de la zone.
- « *Perdre* » l'adresse d'une zone allouée :

```
int *p = malloc (...);  
...    // On ne mémorise pas la valeur de p ailleurs.  
return (...);
```

- ▶ → Impossible de désallouer.
- ▶ ⇒ Fuites mémoire et dégradation de performances avec le temps.