

COURS 6

GESTION DE LA MÉMOIRE EN C

PARTIE ①

LES POINTEURS

ESPACE MÉMOIRE

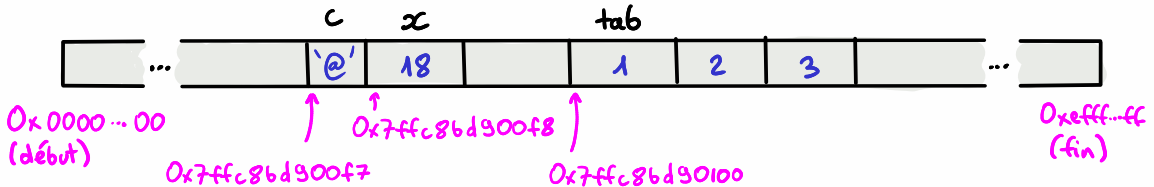
Le langage C permet une gestion fine de la mémoire...
Mais c'est quoi la mémoire ?

L'espace mémoire est assimilable à un long bloc linéaire fait d'octets où se trouvent entre autres le code machine, la pile d'exécution, et ce qui nous intéresse le plus ici : les variables

Par exemple,

```
int x = '18';  
char c = '@';  
int tab[] = {1,2,3};
```

peut engendrer l'espace mémoire :



Chaque variable dans la mémoire a une adresse en hexadécimal (cela correspond à l'adresse de son premier octet)

Ex : Ici `c` a pour adresse `0x7ffc86d900f7` (soit en décimal `140722654740727`)

LES POINTEURS

Définition	Un pointeur c'est juste une adresse mémoire (associée à un type de variable)
------------	--

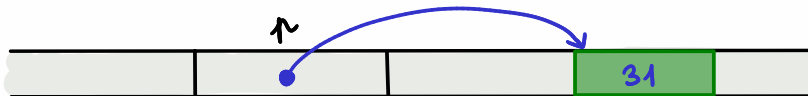
Ex: \uparrow est une variable "pointeur vers un entier"
de valeur $0x7ffc8bd900fc$

Représentation en mémoire:



↑ ici c'est au $0x7ffc8bd900fc$

Représentation équivalente avec une flèche



Il existe un pointeur particulier qui ne pointe nulle part dans la mémoire
c'est le pointeur nul (qui s'écrit NULL en C)

Représentation :



← en vrai c'est l'adresse 0

TYPE D'UN POINTEUR

En C, les pointeurs sont **typés** : ils ne "pointent" que vers un unique **type** de variable (par ex. entier, flottant, caractère)

Question légitime

Quelle est la différence entre un pointeur "vers un entier" et un pointeur "vers un caractère" vu que leur type de valeur c'est la même chose (à savoir une **adresse mémoire**) ?

POINTEUR VERS ENTIER

p_int

0x7ffc8bd900fc



(vs) ⚡

POINTEUR VERS CARACTÈRE

p_char

0x7ffc8bd900fc



Une raison (pas la seule) La fenêtre considérée dépend du type
Un caractère occupe moins d'octets qu'un entier

DÉCLARATION D'UN POINTEUR

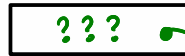
Le type d'un pointeur s'écrit type_base^*

Ex Type d'un pointeur vers un entier int^*
Type d'un pointeur vers un flottant float^*
Type d'un pointeur vers un caractère char^*
Type d'un pointeur vers un pointeur vers un entier int^{**}

Exemples de déclaration :

$\text{int}^* \text{p_int};$

p - int



??

on ne sait pas
où ça pointe
(sûrement
nulle part)

$\text{char}^* \text{p_char} = \text{NULL};$

p - char



! Ne déclarez pas 2 pointeurs sur une même ligne: $\text{int}^* \text{p}, \text{q};$ marche pas!

c'est un
pointeur

ce n'est
pas un pointeur

RÉCUPÉRER L'ADRESSE / LA VALEUR

RÉCUPÉRER L'ADRESSE D'UNE VARIABLE

L'opérateur & (opérateur référencement) permet de récupérer l'adresse d'une variable :

SYNTAXE : adresse de var = &var

Ex

```
float y = 12.5;  
float* adresse_y = &y;
```

y

12.5

adresse_y

LIRE / MODIFIER LA VALEUR POINTÉE

L'opérateur * (opérateur déréférencement) permet de récupérer la valeur de la case à l'adresse d'un pointeur.

SYNTAXE: *pointeur

On peut même l'utiliser pour modifier la valeur :

SYNTAXE: *pointeur = ...

Ex:

```
int x = 12;  
int* adr_x = &x;  
printf("%d", *adr_x + 3);
```

x

12

adr_x

Affiche:

15

```
*adr_x = *adr_x / 2;  
printf("%d", *adr_x);
```

x

6

adr_x

Affiche:

6

AFFICHAGE DE L'ADRESSE MÉMOIRE

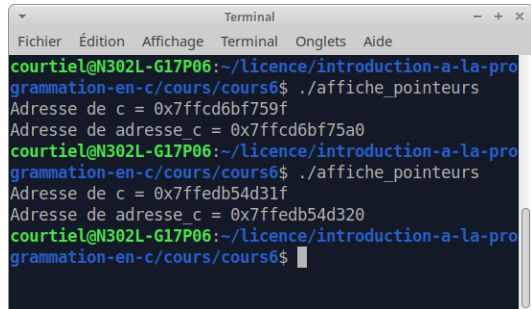
On peut simplement utiliser `printf`

Ici le bon format pour le type `pointeur` est `%p`

(Ex)

```
char c = '$';  
char* adresse_c = &c;  
printf("Adresse de c = %p\n",adresse_c);  
printf("Adresse de adresse_c = %p\n",&adresse_c);
```

Remarquez que pour des raisons de sécurité, les adresses des variables changent à chaque exécution.



```
Terminal  
Fichier  Édition  Affichage  Terminal  Onglets  Aide  
courtiel@N302L-G17P06:~/licence/introduction-a-la-pro  
grammation-en-c/cours/cours6$ ./affiche_pointeurs  
Adresse de c = 0x7ffcd6bf759f  
Adresse de adresse_c = 0x7ffcd6bf75a0  
courtiel@N302L-G17P06:~/licence/introduction-a-la-pro  
grammation-en-c/cours/cours6$ ./affiche_pointeurs  
Adresse de c = 0x7ffedb54d31f  
Adresse de adresse_c = 0x7ffedb54d320  
courtiel@N302L-G17P06:~/licence/introduction-a-la-pro  
grammation-en-c/cours/cours6$
```


ARITHMÉTIQUE DES POINTEURS

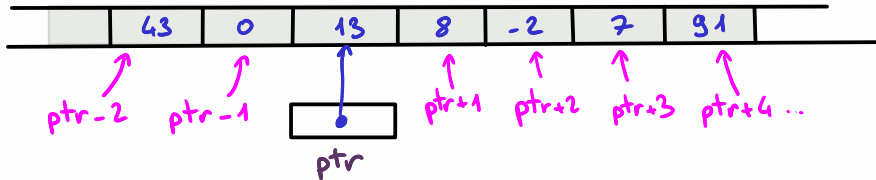
ADDITION POINTEUR + ENTIER

On peut additionner un pointeur ptr et un entier :

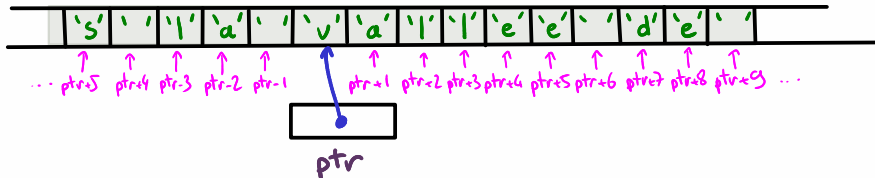
$\text{ptr} + 1$ = adresse de la case mémoire juste à droite

$\text{ptr} - 1$ = adresse de la case mémoire juste à gauche

$\text{ptr} + X$ = adresse de la case mémoire X cases à droite



Remarque: la valeur de $\text{ptr} + n$ dépend du type de base de la variable :



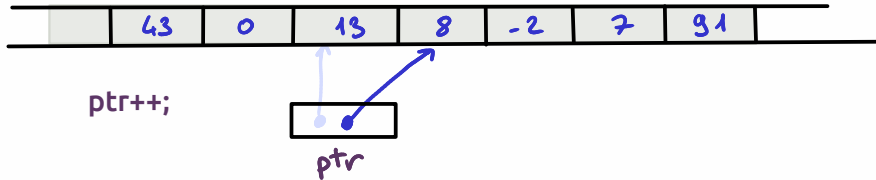
ARITHMÉTIQUE DES POINTEURS

INCRÉMENTATION / DÉCRÉMENTATION

Si ptr pointe sur une case, alors :

- $ptr++$; fait décaler le pointeur vers la case de droite.
- $ptr--$; fait décaler le pointeur vers la case de gauche.

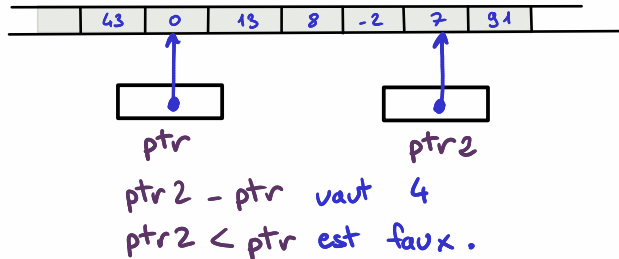
(Ex)



Moins fréquent, mais on peut aussi :

- soustraire 2 pointeurs
renvoie la différence en
termes de cases mémoire
- comparer 2 pointeurs
si un pointeur pointe à
gauche ou à droite d'un
autre pointeur

(Ex)

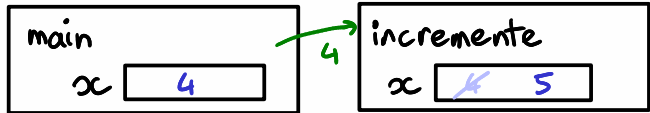


PREMIÈRE APPLICATION DES POINTEURS : LE PASSAGE PAR ADRESSE

Rappelez-vous ...

```
void incremente(int x){  
    x=x+1;  
}  
int main(){  
    int x = 4;  
    incremente(x);  
    printf("%d\n",x);  
}
```

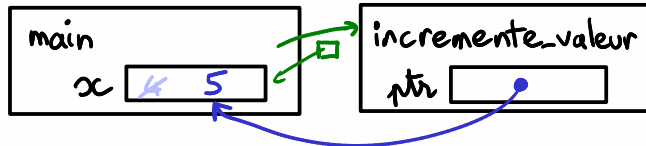
n'affiche pas 5 mais 4 car la variable x dans `incremente` est locale



Pour qu'une fonction modifie une variable non locale, on lui passse son adresse:

```
void incremente_valeur(int* ptr){  
    *ptr = *ptr+1;  
}  
int main(){  
    int x = 4;  
    incremente_valeur(&x);  
    printf("%d\n",x);  
}
```

Ga affiche 5 : la variable x est bien modifiée car on a transmis son adresse à la fonction `incremente_valeur`



On comprend maintenant pourquoi `scanf("%d",&x)` nécessite un `&` !

PARTIE ~~II~~

ALLOCATION
DYNAMIQUE

MANIPULER LA MÉMOIRE SANS CONTRAINTE?

Les pointeurs nous donnent la possibilité de lire et modifier la mémoire à partir de l'adresse

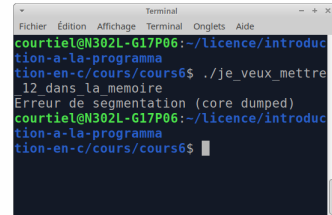
Mais est-ce toujours possible?

Non, le système d'exploitation nous autorise seulement à lire et/ou modifier des cases mémoire qu'il nous a allouées

Si on enfreint cette règle, alors le programme plante et c'est la fameuse "erreur de segmentation". 

(Ex)

```
int* pointeur = (int*) 0x7ffc8bd900f8;  
*pointeur = 12;  
printf("%d\n", *pointeur);
```



```
Terminal  
Fichier Édition Affichage Terminal Onglets Aide  
courtiel@N302L-G17P06:~/licence/introduction-a-la-programmation-en-c/cours/cours$ ./je_veux_mettre_12_dans_la_memoire  
Erreur de segmentation (core dumped)  
courtiel@N302L-G17P06:~/licence/introduction-a-la-programmation-en-c/cours/cours$
```



On va voir comment demander (gentiment) à la machine de nous allouer de la mémoire au fil de l'exécution du programme
C'est l'allocation dynamique

MALLOC

Dans la bibliothèque `stdlib.h`, la fonction `malloc` permet d'allouer un bloc de mémoire :

SYNTAXE	<code>ptr = malloc (<nombre d'octets>);</code>
---------	--

- Effets :
- ① Alloue un bloc mémoire occupant le nombre d'octets mis en paramètre
 - ② Renvoie l'adresse mémoire de ce bloc (on le stocke ici dans la variable `ptr`)

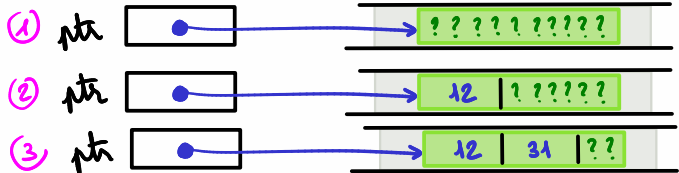
Exemple:

CODE

```
① int* ptr = malloc(10);  
② *ptr = 12;  
③ *(ptr+1) = 31;
```

à ne pas faire

EN MÉMOIRE (dans l'ordre d'exécution)



Comme quand on déclare une variable, rien ne garantit que les octets de l'espace mémoire soient initialisés à 0

SIZEOF

⚠ VOUS NE DEVEZ JAMAIS UTILISER malloc (<un nombre>)

Étudiant
qui remet
en doute la
parole du
divin prof



Mais je sais qu'un
entier occupe 4 octets
donc je peux utiliser
malloc(4) pour stocker
un entier?

Réponse: Non.

La nombre d'octets nécessaire pour encoder un entier (ou un caractère, un flottant...) dépend de la machine.

Pour que le programme soit portable, ce n'est pas une bonne chose d'écrire en dur malloc(<un nombre en dur>)

Heureusement qu'il existe sizeof !

SYNTAXE	sizeof(un-type) ou sizeof(une-variable)
---------	---

→ donne la taille en terme du nombre d'octets

Un malloc va
tout le temps
de pair avec un
sizeof !

Ex d'utilisation:

```
int* ptr_int = malloc(sizeof(int));  
*ptr_int = 12;  
char c = 'Q';  
char* ptr_char = malloc(sizeof(c));  
*ptr_char = c;
```

DEUXIÈME APPLICATION DES POINTEURS : TABLEAUX ALLOUÉS DYNAMIQUEMENT

Avec malloc, on peut déclarer des tableaux à taille variable !

SYNTAXE

Déclarer un tableau de n entiers :

```
int* tab = malloc(n * sizeof(int));
```

Déclarer une chaîne de caractères avec l caractères

```
char* phrase = malloc((l+1) * sizeof(char));
```

(caractère nul exclus)

Vous pouvez utiliser ces pointeurs comme des tableaux :

Ex:

```
int nb = 10;
int* tableau_aleatoires = malloc(nb * sizeof(int));
for (int i = 0; i < nb; i++){
    tableau_aleatoires[i] = rand() % 100;
}
```


DEUXIÈME APPLICATION DES POINTEURS : TABLEAUX ALLOUÉS DYNAMIQUEMENT

Avec malloc, on peut déclarer des tableaux à taille variable !

SYNTAXE

Déclarer un tableau de n entiers :

```
int* tab = malloc(n * sizeof(int));
```

Déclarer une chaîne de caractères avec l caractères

```
char* phrase = malloc((l+1) * sizeof(char));
```

(caractère nul exclus)

On peut même renvoyer dans une fonction un tableau issu de malloc :

Ex:

```
int* tableau_aleatoire(int taille){  
    int* tab = malloc(taille * sizeof(int));  
    for (int i = 0; i < taille; i++){  
        tab[i] = rand() % 100;  
    }  
    return tab;  
}
```



FUITE DE MÉMOIRE



Gros inconvénient de malloc :

La mémoire allouée reste allouée, même si on n'utilise plus la variable .

On risque la fuite de mémoire :

Les données vont occuper de \oplus en \oplus de place jusqu'à saturation de la mémoire

Ex idiot

```
int* pointeur;
for(int i = 1; i <= 300000000; i++){
    pointeur = malloc(sizeof(int));
}
scanf("%d", pointeur);
printf("%d\n", *pointeur);
```

main de fuite_memoire.c

Occupation en mémoire :

```

Fichier  Edition  Affichage  Terminal  Onglets  Aide
courtiel@N302L-G17P06:/tmp$ free
              total        used        libre      partagé    tamp/cache    disponible
Mem:      16062664      3318248      11960188      482680      784228      11972680
Partition d'échange:         0         0         0
courtiel@N302L-G17P06:/tmp$ ./fuite_memoire &
[1] 1736
courtiel@N302L-G17P06:/tmp$ free
              total        used        libre      partagé    tamp/cache    disponible
Mem:      16062664      12713252      2562140      482628      787272      2577716
Partition d'échange:         0         0         0
[1]+  Arrêté                  ./fuite_memoire
courtiel@N302L-G17P06:/tmp$
```

FREE

La solution à la fuite de mémoire : free

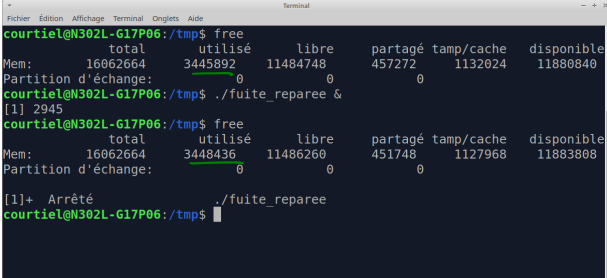
SYNTAXE	<code>free(ptr);</code> (où <code>ptr</code> provient d'un <code>malloc</code>)
---------	--

Important Un `malloc` doit toujours s'accompagner d'un `free`.
Dès qu'on finit d'utiliser un tableau issu d'un `malloc`, on le libère avec `free`.

Ex idiot corrigé

```
int* pointeur;  
for(int i = 1; i <= 300000000; i++){  
    pointeur = malloc(sizeof(int));  
    free(pointeur);  
}  
scanf("%d", pointeur);  
printf("%d\n", *pointeur);
```

Occupation en mémoire :



```
courtiel@N302L-G17P06:/tmp$ free  
total          utilisé      libre      partagé  tamp/cache  disponible  
Mem:    16062664    3445892    11484748    457272    1132024    11880840  
Partition d'échange:    0          0          0  
courtiel@N302L-G17P06:/tmp$ ./fuite_reparee &  
[1] 2945  
courtiel@N302L-G17P06:/tmp$ free  
total          utilisé      libre      partagé  tamp/cache  disponible  
Mem:    16062664    3448436    11486260    451748    1127968    11883808  
Partition d'échange:    0          0          0  
[1]+  Arrêté                ./fuite_reparee  
courtiel@N302L-G17P06:/tmp$
```

main de `fuite_reparee.c`

PARTIE ~~III~~

POINTEURS ET TABLEAUX

Petit épilogue pour celles et ceux qui sont à l'aise

POINTEUR = TABLEAU ?

Est-ce qu'un pointeur c'est la même chose qu'un tableau ?

Bien qu'ils s'utilisent de manière similaire ,
ce sont deux objets différents

Ex en mémoire :

TABLEAU



POINTEUR



Autres différences notables :

TABLEAU STATIQUE	POINTEUR VERS UN TABLEAU ALLOCÉ DYNAMIQUEMENT
local à la fonction est désalloué automatiquement après fin de la fonction	déclaré hors fonction nécessite free pour être désalloué
ne peut pas être NULL	peut être NULL
sizeof renvoie le nombre d'octets dans le tableau	sizeof renvoie le nombre d'octets dans une adresse mémoire

FONCTIONS ET TABLEAUX

Rappelez-vous : Quand on passe un tableau en paramètre, on passe en réalité un pointeur ! D'où le `int*` en paramètre

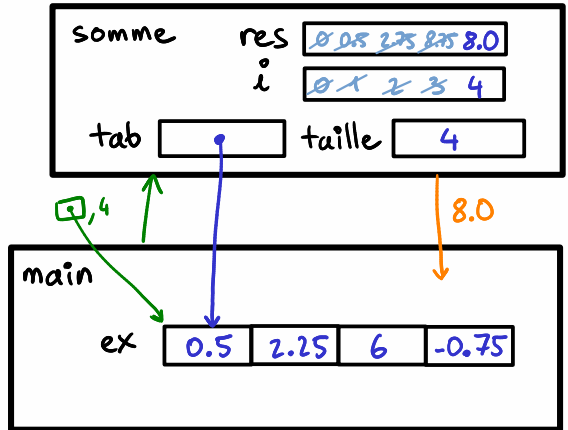
PROGRAMME

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

float somme(float* tab, int taille) {
    float res = 0;
    for (int i = 0; i < taille; i++) {
        res = res + tab[i];
    }
    return res;
}

int main() {
    float ex[] = {0.5, 2.25, 6, -0.75};
    printf("La somme vaut %d.\n", somme(ex, 4));
    return EXIT_SUCCESS;
}
```

EN MÉMOIRE



Preuve que c'est un pointeur:

```
float tab[] = {1.5, 2, 3};
tab = NULL;
```

ne compile pas alors que

```
void fait_rien(int* tab) {
    tab = NULL;
}
```

compile!

même si on change `int* tab` par `int tab[]`