



UNIVERSITÉ
CÔTE D'AZUR

Programmation impérative en C

Cours 3. Gestion de la mémoire et pointeurs

Olivier Baldellon

Courriel : `prénom.nom@univ-cotedazur.fr`

Page professionnelle : <https://upinfo.univ-cotedazur.fr/~obaldellon/>

LICENCE 2 — FACULTÉ DES SCIENCES ET INGÉNIERIE DE NICE — UNIVERSITÉ CÔTE D'AZUR

 Partie I. Généralité sur la mémoire

 Partie II. Codage des données

 Partie III. Pointeurs

 Partie IV. Allocation dynamique

 Partie V. Divers

 Partie VI. Table des matières

Chaque exécutable se voit attribuer une mémoire virtuelle de 2^{64} octets.

- ▶ 16 exioctet adressables (\approx 16 exaoctets).
- ▶ En pratique, seules certaines de ces adresses sont valides

▶ La mémoire est partagée en trois.

- **DATA** : les données

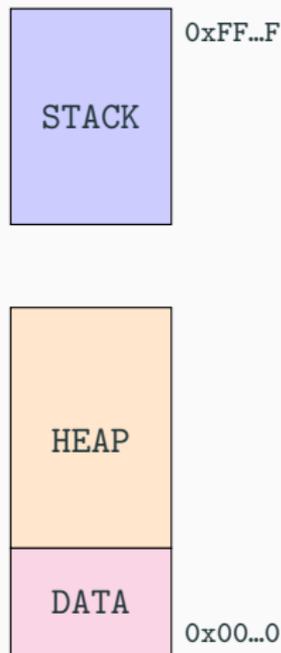
- ▶ contient les instructions du programme
- ▶ contient les variables statiques
- ▶ son organisation est statique

- **STACK** : la pile

- ▶ contient les variables locales...
- ▶ ... dont la taille est connue à la compilation.

- **HEAP** : le tas

- ▶ contient les variables créés dynamiquement
- ▶ géré par le développeur.

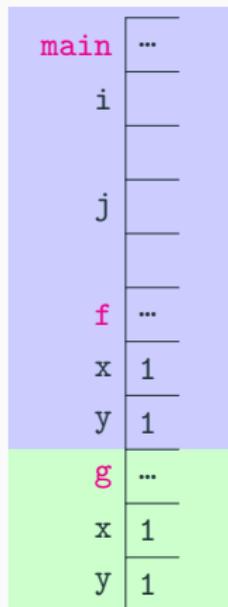


- ▶ En pratique, sur 64 bits, les adresses ne vont pas de 0 à $2^{64} - 1$.
 - ▶ C'est le système d'exploitation qui décide (Linux, Windows, etc.)
- ▶ La mémoire du programme est stockée sur la RAM, mais parfois peut-être stockée sur le disque dur s'il n'y a plus de place.
 - ▶ Partition `swap` sous Linux : le programme rame alors méchamment!
- ▶ Pour déclarer une variable statique, il suffit d'utiliser le mot-clé `static`
 - ▶ La variable ne sera pas supprimée à la fin de l'appel de la fonction.
 - ▶ Elle est déclarée à 0.
 - ▶ À chaque appel de la fonction, la variable `c` est incrémentée.

```
void compteur(void) {  
    static int c = 0;  
    printf("C = %d\n", c);  
    c++;  
}
```

`*.c`

- ▶ La pile est remplie du haut vers le bas.
 - ▶ chaque fonction a accès à un morceau de pile
 - ▶ dans lequel sont stockées ses variables
 - ▶ on parle de *stack frame*
- ▶ Lors de l'appel d'une fonction
 - ▶ on empile une nouvelle *frame*
 - ▶ pour y stocker les variables de cette fonction
- ▶ On recommence ainsi à chaque appel.
 - ▶ si le nombre d'appel est trop grand : plantage
 - ▶ la taille de la pile est fixée par l'OS (Linux)
 - ▶ débordement de pile (*stack overflow*)
- ▶ Lorsqu'une fonction retourne
 - ▶ on dépile la *frame*
 - ▶ la fonction appelante peut continuer son exécution.



- ▶ La gestion de la pile est automatique
 - ▶ pas de bug !
 - ▶ pas de fuite mémoire
 - ▶ pas de gestion mémoire
 - ▶ que du bonheur ❤
- ▶ Mais la taille de toutes les données doivent être connues à la compilation.
 - ▶ On ne peut pas modifier la taille des tableaux
 - ▶ On ne peut pas créer de nouvelles données à la volée...
 - ▶ ...mais uniquement modifier celles déclarées à la compilation.
- ▶ Pour dépasser ces limitations on utilise le tas :
 - ▶ partie 3

```
void f(void) {
    int tab[] = {0};
    int i = 0;;
    while (tab[i] != 777) {
        printf("t[%.2d] = %i\n",
              i, tab[i]);
        i++;
    }
    printf("t[%.2d] = %i\n",
          i, tab[i]);
    tab[i]=666;
}

void fonction(void) {
    f();
}

int main() {
    int variable = 777;
    fonction();
    printf("%d\n", variable);
    return 0;
}
```

*.c

- ▶ En C, chaque fonction peut accéder à-peu-près toute la mémoire allouée.

```
t[00] = 0
t[01] = 1
t[02] = 671837296
t[03] = 32764
t[04] = -1972874112
t[05] = 22038
t[06] = 671837328
t[07] = 32764
t[08] = -1972874089
t[09] = 22038
t[10] = 671837568
t[11] = 32764
t[12] = 0
t[13] = 777
666
```

stdout

- ▶ *Undefined Behavior*
- ▶ On peut même la modifier !
(dangereux!)

 Partie I. Généralité sur la mémoire

 Partie II. Codage des données

 Partie III. Pointeurs

 Partie IV. Allocation dynamique

 Partie V. Divers

 Partie VI. Table des matières

- ▶ Comment savoir ce qu'écrit C lorsqu'on stocke une donnée en mémoire ?

```
typedef struct { char r, g, b;} rgb;
typedef struct { char c;
                short s; } alignement1;
typedef struct { unsigned char petit1;
                unsigned short grand;
                unsigned char petit2; } alignement2;

union mon_type {
    long long_s; unsigned long long_u;
    int  int_s;  unsigned int  int_u;
    float float_t;

    alignement1 alignement1_t;
    alignement2 alignement2_t;
    unsigned char tableau[8];
};
```

- ▶ Méthode
 - ▶ on stocke un type en mémoire : par exemple `x.long_s = 17`
 - ▶ on affiche le tableau `x.tableau` pour voir ce qui est codé.

- ▶ Comment est stocké un entier ?

```
x.long_s = 655957 *.C
+-----+-----+-----+-----+-----+-----+-----+-----+
| 85  | 2  | 10 | 0  | 0  | 0  | 0  | 0  |
| 55  | 02 | 0A | 00 | 00 | 00 | 00 | 00 |
|01010101|00000010|00001010|00000000|00000000|00000000|00000000|00000000|
+-----+-----+-----+-----+-----+-----+-----+-----+

```

- ▶ Pas facile de comprendre ce qu'il se passe...
- ▶ Pour comprendre, choisissons un entier en héra.

```
x.long_s = 0xCAFE0230 *.C
+-----+-----+-----+-----+-----+-----+-----+-----+
| 48  | 2  | 254 | 202 | 0  | 0  | 0  | 0  |
| 30  | 02 | FE  | CA  | 00 | 00 | 00 | 00 |
|00110000|00000010|11111110|11001010|00000000|00000000|00000000|00000000|
+-----+-----+-----+-----+-----+-----+-----+-----+

```

- ▶ Comment écrire 0x1234CAFE (en hexa) dans la mémoire ?
 - ▶ Gros-Boutisme : dans le bon ordre (naturel pour nous autres humains)
 - ▶ Gros-Boutisme : on commence par les bits de poids fort
 - ▶ Petit-Boutisme : à priori ce que votre machine utilise (ARM, x86, x86-64)
 - ▶ Petit-Boutisme : on commence par les bits de poids faibles

Le Gros-Boutisme *big-endian*

12	34	CA	FE
----	----	----	----

Le Petit-Boutisme *little-endian*

FE	CA	34	12
----	----	----	----

- ▶ Intérêt de petit boutisme ?
 - ▶ L'entier 17 est codé pareil quelque soit son type : `int`, `short`, etc.
 - ▶ Plus rapide pour les calculs (qui commencent souvent avec les bits de poids faibles)
- ▶ Ne pas tenter de lire un `int` dans la mémoire octet par octet
 - ▶ sauf pour la curiosité : **risqué et non portable**
 - ▶ les opérations bits à bits marchent indépendamment du boutisme.
 - ▶ utilisez-les si besoin de décomposer un nombre (tp2).

*.C

```
x.int_s = 1
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
|  1  |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
| 01  | 00  | 00  | 00  | 00  | 00  | 00  | 00  |
|00000001|00000000|00000000|00000000|00000000|00000000|00000000|00000000|
+-----+-----+-----+-----+-----+-----+-----+-----+
```

```
x.long_s = 1
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
|  1  |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
| 01  | 00  | 00  | 00  | 00  | 00  | 00  | 00  |
|00000001|00000000|00000000|00000000|00000000|00000000|00000000|00000000|
+-----+-----+-----+-----+-----+-----+-----+-----+
```

```
x.int_s = -1 ou x.int_u = 4294967295
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| 255 | 255 | 255 | 255 |  0  |  0  |  0  |  0  |
| FF  | FF  | FF  | FF  | 00  | 00  | 00  | 00  |
|11111111|11111111|11111111|11111111|00000000|00000000|00000000|00000000|
+-----+-----+-----+-----+-----+-----+-----+-----+
```

```
x.long_u = -1
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| 255 | 255 | 255 | 255 | 255 | 255 | 255 | 255 |
| FF  |
|11111111|11111111|11111111|11111111|11111111|11111111|11111111|11111111|
+-----+-----+-----+-----+-----+-----+-----+-----+
```

- ▶ Comment sont codé les flottants ? Prenons l'exemple du nombre 5.

x.float_t = 5.0;								*.c
0	0	160	64	0	0	0	0	
00	00	A0	40	00	00	00	00	
00000000	00000000	10100000	01000000	00000000	00000000	00000000	00000000	

- ▶ Quel est le rapport avec le nombre 5 ? (car oui, il y en a un !)

 - Conservons seulement 32 bits et remettons les dans l'ordre.
 - **01000000 10100000 00000000 00000000**
 - ▶ 1 bit pour le **signe** : (0 pour +1 et 1 pour -1)
 - ▶ 8 bits pour l'**exposant biaisé** : ici **1000000 1** donne 129.
 - ▶ 24 bits pour la **mantisse** : ici **0100000 00000000 00000000** ce qui donne 2097152.

- ▶ Bref, rien de bien compliqué...
 - ▶ des questions?

- ▶ Comment obtenir 5 à partir de ces trois valeurs.

$$\text{signe} \times \left(1 + \frac{\text{mantisse}}{2^{23}} \right) \times 2^{\text{exposant}-127}$$

- ▶ De même que :

- ▶ en décimal $524 = 5,24 \times 10^2$
- ▶ en binaire cinq vaut $\overline{101} = \overline{1,01} \times 2^2$

$$5 = +1 \times \overline{1,01} \times 2^{129-127}$$

- ▶ Pourquoi $1 + \text{mantisse} \times 2^{-23}$?
 - ▶ En binaire le premier chiffre vaut toujours 1 : inutile de le stocker.
 - ▶ On conserve les 23 premiers bits après la virgule : c'est la mantisse.
 - ▶ $1 + \text{mantisse} \times 2^{-23} = 1.\text{0100000 0000000 0000000}$
- ▶ C'est la norme IEEE 754 utilisée dans presque tous les langages.


```
typedef struct { unsigned char petit1;
                unsigned short grand;
                unsigned char petit2; } alignement2;
```

*.C

- ▶ Un champs de longueur n doit être sur une adresse multiple de n

- ▶ `grand` est un `short` sur deux octets : l'adresse doit être paire.

0	1	2	3	4	5	Adresses 2 et 4 valides	0	1	2	3	4	5
0	1	2	3	4	5	Adresses 1 et 3 invalides	0	1	2	3	4	5

- ▶ La taille de la structure doit être un multiple de la taille de ces champs.
 - ▶ La structure ne peut pas être de taille 5
 - ▶ Le compilateur rajoute une case après (et y met n'importe quoi : 77)

```
x.alignement2_t petit1=0xFF grand=0xCAFE petit2=0xAA
```

*.C

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| 255  | 0    | 254  | 202  | 170  | 77   | 0    | 0    |
| FF   | 00   | FE   | CA   | AA   | 4D   | 00   | 00   |
|11111111|00000000|11111110|11001010|10101010|01001101|00000000|00000000|
+-----+-----+-----+-----+-----+-----+-----+-----+
```

- ▶ Le compilateur fait ce travail pour vous, mais comment connaître la taille ?

- ▶ Pour connaître la taille des données en mémoire on utilise `sizeof`
 - ▶ l'unité est le *byte*, donc concrètement l'octet.
- ▶ On peut l'utiliser avec un type : `sizeof(int)`
- ▶ On peut l'utiliser avec une variable : `sizeof(variable)`

```
typedef struct{
    unsigned char petit1;
    unsigned short grand;
    unsigned char petit2;
} alignement2;

int main() {
    ma_struct s;
    /* Affiche 1 2 1 : logique */
    printf("%ld %ld %ld\n", sizeof(s.petit1)
           , sizeof(s.grand)
           , sizeof(s.petit2));

    /* Affiche 6 : Pourquoi ? */
    printf("%ld\n", sizeof(s));
    printf("%ld\n", sizeof(alignement2));
    return 0;
}
```

*.C

 Partie I. Généralité sur la mémoire

 Partie II. Codage des données

 **Partie III. Pointeurs**

 Partie IV. Allocation dynamique

 Partie V. Divers

 Partie VI. Table des matières

- ▶ Un pointeur est un type de donnée représentant une adresse mémoire.

```

int a = 2;
int *p;

/* On met l'adresse de a dans p */
p = &a;

printf("a = %d\n",a); /* affiche « a = 2 » */

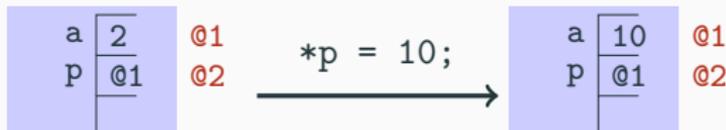
/* À l'adresse p on écrit 10*/
*p = 10;

printf("a = %d\n",a); /* affiche « a = 10 » */

```

*.c

- ▶ Dans le code précédant la ligne `*p = 10;`
 - ▶ ne modifie pas le pointeur `p` (l'adresse ne change pas)
 - ▶ mais modifie la variable `a`



- ▶ La notation `&a` représente un pointeur mais `*p` représente une donnée.

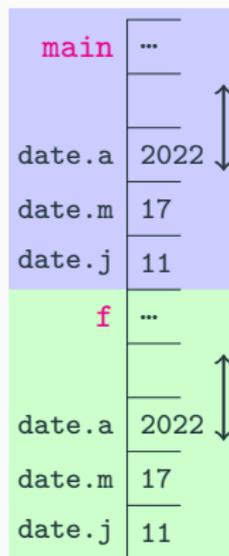
Adresse	Donnée
<code>&a</code>	<code>a</code>
<code>p</code>	<code>*p</code>

- ▶ Déclaration d'un pointeur vers un entier : `int *p`
 - ▶ se lit « p est un pointeur vers `int` »
- ▶ Évitez la notation `int* p`
 - ▶ syntaxiquement `int*` n'est pas un type.
 - ▶ « `int* p1, p2` » est lu par le compilateur « `int *p1, p2` »
 - ▶ `p1` est un pointeur mais `p2` un entier !

- ▶ Lorsqu'une valeur est passée en argument,
 - ▶ elle est **copiée** dans la nouvelle pile.
 - ▶ on parle de **passage par valeur**.
 - ▶ Pour les grosses structures, l'opération peut être coûteuse.

```
typedef struct {  
    unsigned char j, m;  
    unsigned short a;  
} date_type;  
  
void f(date_type date) {  
    ...  
}  
  
int main(void) {  
    date_type date = {17,11,2022};  
    f(date);  
    return 0;  
}
```

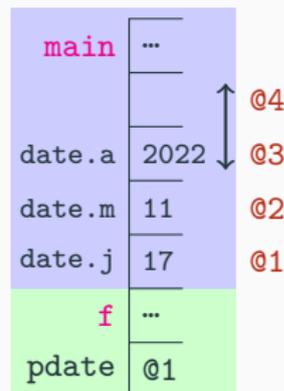
*.c



- ▶ Ici, on se contente de donner le pointeur
 - ▶ on parle de passage par paramètre
 - ▶ généralement bien plus efficace.
 - ▶ Attention, la fonction `f` peut modifier `date` à sa guise

```
typedef struct {  
    unsigned char j, m;  
    unsigned short a;  
} date_type;  
  
void f(date_type *pdate) {  
    ...  
}  
  
int main(void) {  
    date_type date = {17,11,2022};  
    f(&date);  
    return 0;  
}
```

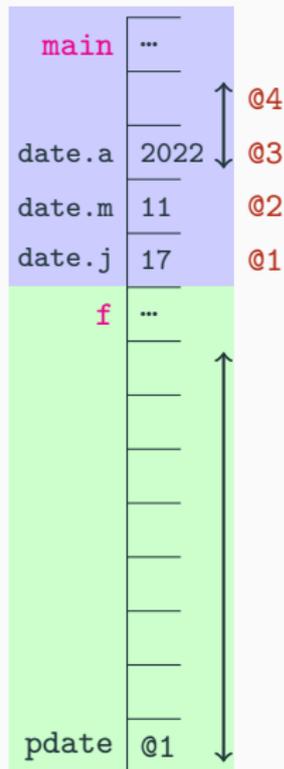
*.c



- ▶ Dans l'exemple précédent
 - ▶ le pointeur faisait 64 bits (8 octets)
 - ▶ le gain est ici limité :)
- ▶ Mais souvent, la taille du pointeur est très inférieure à la taille des données.
 - ▶ sauf les types de base
 - ▶ (entiers, caractères et flottants)
- ▶ Autre intérêt : pouvoir modifier la variable directement.

```
int i;
code_erreur = scanf("%d", &i);
```

*.c



- ▶ On peut utiliser les pointeurs pour modifier une unique valeur.
- ▶ On utilise aussi les pointeurs pour travailler sur les tableaux :
 - ▶ L'adresse correspond à celle de la première case.
 - ▶ La longueur du tableau doit être connue par le développeur.
- ▶ Il n'y a (presque) pas de différences en C entre tableaux et pointeurs.

```
int i = 7;
int *p = &i;
printf("%d == %d\n", *p, p[0]); /* 7 == 7 */
```

*.c

- ▶ Ici T: [11,22,33] et p: [22,33]

```
int *p = &i;
int T[3] = {11,22,33};
p = T+1; /* p correspond à l'adresse après T */
for (i=0; i<3; i++) printf("%d ",T[i]); /* 11 22 33 */
for (i=0; i<2; i++) printf("%d ",p[i]); /* 22 33 */
```

*.c

- ▶ Les quatre notations suivantes sont équivalentes :
 - ▶ $\text{tab}[3] = *(\text{tab} + 3) = *(3 + \text{tab}) = 3[\text{tab}]$

- ▶ Dans une expression, tableau et pointeur correspondent à des adresses.
 - La valeur d'une variable est le contenu de sa case.
 - ▶ A vaut 44 (type entier) et p vaut @11 (type adresse)
 - SAUF pour les tableaux : la valeur de T est son adresse.
 - ▶ T vaut @1. Mais cette valeur ne se trouve dans aucune case mémoire.
 - ▶ T et &T ont la même valeur.
 - Une variable est associée à une case mémoire
 - ▶ cette association ne change jamais.
 - ▶ Pour cette raison, l'association entre T et @1 est immuable.
 - Par contre on peut modifier le contenu d'une case mémoire.
 - ▶ $p = p+1$; est valide et remplace dans la case p la valeur @11 par @12.
 - ▶ $T = \dots$ n'a par contre aucun sens. On ne peut pas modifier l'adresse !

	33	@3
	22	@2
T	11	@1
A	44	@0

	3	@13
	2	@12
	1	@11
P	@11	@10

- ▶ Déclaration et allocation
 - Tableau : alloué sur la pile. (Les pointeurs pointent souvent vers le tas)
 - ▶ `int tab[]` ; le contenu du tableau est alloué sur la pile ;
 - ▶ `char tab[]` ; idem ;
 - ▶ `char *s = ":"` ; est allouée dans la partie statique de la mémoire.
 - ▶ `int *s` ; pas d'allocation (sauf la case pour stocker l'adresse).
 - On peut initialiser le contenu d'un tableau à la déclaration
 - ▶ et non celui d'un pointeur (sauf chaîne statique)
 - ▶ `int tab[] = {1, 2, 3}` ; est valide
- ▶ On peut modifier l'adresse d'un pointeur (mais pas un tableau)
 - ▶ `p = ...` ; est valide
 - ▶ On peut modifier le contenu du tableau `T[i]`, mais pas la valeur de `T`.
- ▶ La fonction `sizeof`
 - ▶ Pour un tableau, renvoie la taille complète de tous les éléments.
 - ▶ Pour un pointeur, affiche la taille d'un pointeur

- ▶ Dans une expression, tableau et pointeurs sont tous deux des adresses.
- ▶ Les deux signatures suivantes sont strictement équivalentes.
 - ▶ `void ma_fonction(int tab[])`
 - ▶ `void ma_fonction(int *tab)`
 - ▶ dans les deux cas `tab` est un pointeur (et non un tableau)

```
void ma_fonction1(int tab[]) {  
    printf("%ld", sizeof(tab)); /* tab est un pointeur */  
}  
  
void ma_fonction2(int *tab) {  
    printf("%ld", sizeof(tab)); /* tab est un pointeur */  
}  
  
int main() {  
    int tab[] = {1, 2, 3}; /* sizeof(int) = 4*/  
    int * p;  
    printf("%d", sizeof(tab)); /* affiche 12 (= 3*4) */  
    printf("%d", sizeof(p)); /* affiche 8 | 8 octets */  
    ma_fonction1(tab); /* affiche 8 | valent */  
    ma_fonction2(tab); /* affiche 8 | 64 bits */  
}
```

*.c

- Pour un pointeur de structure la notation `(*pointeur).champs` peut être remplacée par la notation plus agréable `pointeur->champs`.

```
typedef struct { unsigned char r, g, b; } couleur;
```

`*.C`

```
void afficher(couleur *coul) {  
    printf("(%d,%d,%d)\n", (*coul).r, (*coul).g, (*coul).b);  
}
```

```
void assombrir(couleur *coul) {  
    coul->r = coul->r / 2;  
    coul->g = coul->g / 2; /* équivalent à (*coul).g = (*coul).g/2 */  
    coul->b = coul->b / 2;  
}
```

```
int main() {  
    couleur rougeatre = {255, 100, 100};  
    afficher(&rougeatre); /* (255,100,100) */  
    assombrir(&rougeatre);  
    afficher(&rougeatre); /* (127,50,50) */  
    return 0;  
}
```

- ▶ Que se passe-t-il si on ajoute un entier à un pointeur ?
 - ▶ pour afficher on utilise `%p` dans le `printf`
 - ▶ il faut avant le convertir en pointeur non typé : `(void *)`

```
int *p0, *q0;
char *p1, *q1;
q0 = p0+1;
q1 = p1+1;
printf("%p %p\n", (void *) p0, (void *) q0);
printf("%p %p\n", (void *) p1, (void *) q1);
```

*.c

- ▶ Lorsqu'on incrémente un pointeur de type `truc *`, les calculs se font selon un pas de `sizeof(truc)`.
 - pour `p0` et `q0` : `0x7ffd4586e1b0` `0x7ffd4586e1b4`
 - ▶ La différence observée est de 4 octets (taille d'un `int`)
 - pour `p1` et `q1` : `0x7ffd4586e1af` `0x7ffd4586e1b0`
 - ▶ La différence observée est de 1 octet (taille d'un `char`)
- ▶ si `truc *p` pointe sur une case d'un tableau de `truc`, `p++` fera pointer `p` sur la case suivante (en dehors d'un tableau : *UB*).

- 🍃 Partie I. Généralité sur la mémoire
- 🍃 Partie II. Codage des données
- 🍃 Partie III. Pointeurs
- 🍃 **Partie IV. Allocation dynamique**
- 🍃 Partie V. Divers
- 🍃 Partie VI. Table des matières

- ▶ Travailler sur la pile est très efficace, mais ne permet pas de créer :
 - des objets dont la taille n'est connue qu'à l'exécution
 - ▶ Si un utilisateur renseigne son nom quel, en sera la longueur ?
 - des objets dont l'existence n'est connue qu'à l'exécution
 - ▶ Si plusieurs joueurs se connectent, combien seront-ils ?
 - des objets dont la taille change.
 - ▶ La liste des joueurs a une taille qui peut changer durant le jeu.
- ▶ On peut toujours réserver beaucoup de mémoire sur la pile...
 - ▶ en espérant que cela suffise.
 - ▶ en gâchant beaucoup de mémoire pour rien.
 - ▶ bref, la pile n'est pas la réponse à tout.

*.c

```
#include <stdio.h>
#include <stdlib.h> /* Pour utiliser malloc et free*/

int * intervalle(int a, int b) { /* b exclue */
    int i;
    int n = b-a; /* Nombre d'éléments dans le tableau */
    int *tab = malloc( n * sizeof(int));

    for (i=0; i<n; i++) {
        tab[i] = a+i;
    }
    return tab;
}

int main() {
    int i, *tableau;
    /* On crée tableau dynamiquement*/
    tableau = intervalle(12,15);

    for (i=0; i<3; i++) {
        printf("%d ",tableau[i]); /* affiche 12 13 14 */
    }
    printf("\n");
    /* On n'a plus besoin de tableau : on libère l'espace associé */
    free(tableau);
    return 0;
}
```

▶ Création

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

*.C

- On réserve `n` cases de taille `sizeof(int)`
- `tab` est un pointeur vers le premier entier.
 - ▶ mais en C, un tableau est un pointeur.
- Les cases de `tab[0]` à `tab[n-1]` sont bien réservées.
 - ▶ Ce sera à nous de bien faire attention de ne pas travailler avec des indices invalides.

▶ Destruction

```
free(tableau);
```

*.C

- L'espace sera libérée
- Utilisable pour une future variable

- ▶ En C, pour gérer la mémoire, on passe par les fonctions suivantes
 - `void *malloc (size_t taille);`
 - ▶ permet d'allouer `taille` octets dans le tas
 - `void *calloc (size_t nb, size_t taille);`
 - ▶ permet d'allouer `taille×nb` octets dans le tas, en les initialisant à 0
 - `void *realloc (void *ptr, size_t taille);`
 - ▶ permet d'agrandir la zone mémoire
 - ▶ renvoie un pointeur vers la nouvelle zone allouée.
 - ▶ si ré-allocation : libère l'ancien pointeur après avoir copié les données.
 - `void free (void *ptr);`
 - ▶ permet de libérer la mémoire attribuée à `ptr`.
- ▶ Ces fonctions se trouvent dans le fichier de déclarations `stdlib.h`

- 🍃 Partie I. Généralité sur la mémoire
- 🍃 Partie II. Codage des données
- 🍃 Partie III. Pointeurs
- 🍃 Partie IV. Allocation dynamique
- 🍃 **Partie V. Divers**
- 🍃 Partie VI. Table des matières

- ▶ C'est tout simple en fait ! Mais il y a de nombreux risques.
 - Fuites mémoires.
 - ▶ On oublie de libérer la mémoire
 - ▶ Le programme consomme de plus en plus de mémoire RAM
 - Lecture et écriture sur une zone mémoire non valide.
 - ▶ Écriture dans un tableau avec des indices trop grands.
 - ▶ *Use after free.*
- ▶ En cas d'écriture sur une zone non allouée ou non permise :
 - ▶ Erreur de segmentation ou segmentation fault
 - ▶ Aucune information supplémentaire (ligne concernée ?)
 - ▶ Bon courage pour corriger le problème !

- ▶ Attention, ne jamais renvoyer de pointeur appartenant à sa pile!

```
#include <stdlib.h>

int * f1(void) {
    int a = 3;
    return &a;
    /* warning: function returns address of local variable */
}

int * f2(void) {
    int *a = malloc(sizeof(int));
    return a;
}

int main(void) {
    int *p1 = f1(); /* p1 invalide */
    int *p2 = f2();
    return 0;
}
```

*.c

- ▶ On peut ajouter des variables à un Makefile.
 - ▶ Comme en *shell*, elle s'utilisent avec un dollar.

```
CC = gcc
OPTIONS = -Wall -ansi -pedantic

all: exercice1 exercice2

exercice1: exercice1.c
    $(CC) $(OPTIONS) exercice1.c -o exercice1

exercice2: exercice2.c
    $(CC) $(OPTIONS) exercice2.c -o exercice3
```

Makefile

Merci pour votre attention

Questions



Hello world !



Cours 3 — Gestion de la mémoire et pointeurs

Partie I. Généralité sur la mémoire

Les trois parties de la mémoire

Remarques

Fonctionnement de la pile

La pile et le tas

La mémoire est en accès libre

Partie II. Codage des données

Union et inspection mémoire

Union : exemple

Gros-boutisme et petit-boutisme

Nombres négatifs

Les flottants

Les flottants : suites

Structures

Structure et alignement

Taille d'un type

Partie III. Pointeurs

Définition

Syntaxe : remarques

Passage par valeur

Passage par référence

Passage par référence (honnête)

Les pointeurs sont des tableaux!

Les pointeurs ne sont pas des tableaux!

Liste des différences entre tableaux et pointeurs

Pointeurs et tableaux comme paramètres

Pointeurs et structures

Arithmétique de pointeur

Partie IV. Allocation dynamique

Pourquoi?

Comment?

Explications

Allouer et désallouer de la mémoire

Partie V. Divers

C'est tout simple

Références fantômes

Améliorer un makefile

Partie VI. Table des matières