# LC_Structures

## I/ Principes fondamentaux des structures

En langage C, il est **impossible** *(ou à ses risques périls plutôt)* de déclarer des **tableaux hétérogènes**, c'est-à-dire comportant des types différents, Ceci est normal puisqu'un pointeur (typé) ne pourrait pas le parcourir. Le résultat est aléatoire si l'on tente de ne pas respecter cette règle !

Rappelons que cela est possible en Python via les listes.

Un exemple très parlant : le programme compile malgré des avertissements et le résultat n'est pas celui attendu.

```c
#include "stdio.h"


int main() {

    // Création d'un tableau de 4 entiers naturels
    int tab[5];
    int i; // Compteur

    tab[0] = 3;    // Correct
    tab[1] = 6.5;  // Attention, il y aura une conversion en 6
    tab[2] = 'c';  // Renvoie le code ASCII de la lettre c
    tab[3] = "coucou";   // Renvoie ... on ne sait pas trop quoi !
    tab[4] = "youpi";    // Renvoie la valeur précédente + 7 ce qui correspond à la longueur de la chaîne + \0.

    // Un entier prend bien 4 octets
    printf("Taille d'une entier : '%d \n", sizeof(int));

    // Adresse mémoire de tab
    printf("Adresse m\x82moire de tab : %d \n", &tab[0]);

    // Affichage des valeurs du tableau
    for (i=0; i<5; i++)
        printf("tab[%d] = %d \t", i, tab[i]);

    return 0;
}
```

```
C:\Users\Laurent\Desktop\test.exe
Taille d'une entier : '4
Adresse mémoire de tab : 6487552
tab[0] = 3      tab[1] = 6      tab[2] = 99     tab[3] = 4214784        tab[4] = 4214791
--------------------------------
Process exited after 0.03011 seconds with return value 0
Appuyez sur une touche pour continuer...
```

Voici un extrait de la norme ASCII qui illustre la valeur de *tab[2]* :

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 95 | _ | 96 | ` | 97 | a | 98 | b | 99 | c | 100 | d | 101 | e | 102 | f | 103 | g |
| 104 | h | 105 | i | 106 | j | 107 | k | 108 | l | 109 | m | 110 | n | 111 | o | 112 | p |
| 113 | q | 114 | r | 115 | s | 116 | t | 117 | u | 118 | v | 119 | w | 120 | x | 121 | y |

Remarque : pour tab[3], il semblerait que le programme donne l'emplacement par défaut en mémoire des chaînes de caractères constantes, à la discrétion du système d'exploitation qui les attribue, à vérifier.

Le principe des structures en langage C est un moyen de développer des tableaux hétérogènes, on peut aussi les associer aux attributs des classes en Python.

A noter le mot clé *typedef* qui permet de simplifier la déclaration des structures dans un programme.

Voici un premier exemple de programme avec une structure basique :

```c
#include "stdio.h"


typedef struct {

    float x; // Abscisse du point
    float y; // Ordonnée du point

} Point;



int main() {

    Point p1, p2;

    // L'opérateur "." permet d'accéder aux attributs de la structure
    // comme en Python avec les classes
    p1.x = -1.5;
    p1.y = 4;
    p2.x = 3;
    p2.y = -1.5;

    // Toujours ce souci pour afficher les "é" avant un "e" avec la console ...
    printf("Coordonn%ces de p1 : abs = %f, ord = %f \n", 130, p1.x, p1.y);
    printf("Coordonn%ces de p2 : abs = %f, ord = %f \n", 130, p2.x, p2.y);


    return 0;
}
```

Il n'y a pas de méthodes associées en langage C comme en Python mais on peut les simuler (c'est d'ailleurs le rôle du mot clé *self* (Python) ou *this* (C++, Java) qui désigne l'objet courant).

Voici un exemple avec l'initialisation :

```c
#include "stdio.h"


typedef struct {

    float x; // Abscisse du point
    float y; // Ordonnée du point

} Point;

// "Constructeur" version langage C :)
void init(Point *, float, float);



int main() {

    Point p1, p2;

    // Initialisation
    init(&p1, -1.5, 3.2);
    init(&p2, -1.3, -2.6);

    // Toujours ce souci pour afficher les "é" avant un "e" avec la console ...
    printf("Coordonn%ces de p1 : abs = %f, ord = %f \n", 130, p1.x, p1.y);
    printf("Coordonn%ces de p2 : abs = %f, ord = %f \n", 130, p2.x, p2.y);

    return 0;
}
```

```c
// Envoi de l'adresse mémoire des points pour leur modification
void init(Point *pt, float abs, float ord) {
    pt->x = abs;
    pt->y = ord;
}
```

```
C:\Program Files (x86)\Dev-Cpp\Projets\Term_NSI.exe
Coordonnées de p1 : abs = -1.500000, ord = 3.200000
Coordonnées de p2 : abs = -1.300000, ord = -2.600000
```

**<u>Important</u> : on notera que l'instruction *p1->x* qui est l'équivalent de l'association (*p1).x qui permet d'accéder aux valeurs de l'objet p1.**

Voici une version avec un allocation dynamique :

```c
typedef struct {

    float x; // Abscisse du point
    float y; // Ordonnée du point

} Point;

// "Constructeur" version langage C :)
void init(Point *, float, float);


int main() {

    Point *p1;  // Déclaration d'un pointeur de type Point
    p1 = (Point *)malloc(sizeof(Point)); // Allocation de la mémoire

    // Initialisation
    init(p1, -1.5, 3.2);

    // Toujours ce souci pour afficher les "é" avant un "e" avec la console ...
    printf("Coordonn%ces de p1 : abs = %f, ord = %f \n", 130, p1->x, p1->y);

    // Libération de la mémoire allouée
    free(p1);
    p1 = NULL; // Annulation du pointeur (facultatif)

    return 0;
}


// Envoi de l'adresse mémoire des points pour leur modification
void init(Point *pt, float abs, float ord) {
    pt->x = abs;
    pt->y = ord;
}
```

```
C:\Program Files (x86)\Dev-Cpp\Projets\Term_NSI.exe
Coordonnées de p1 : abs = -1.500000, ord = 3.200000
```

On  notera l'association *malloc() / free()* pour une gestion efficace de la mémoire.

Pour plus de contrôle, on pourrait tester si la mémoire allouée associée à *p1* à bien été allouée avec cette instruction mais cela n'est plus vraiment une obligation avec la quantité de RAM actuellement disponible. Le programme s'interrompt avec un message d'erreur.

```c
#include "stdio.h"
#include "stdlib.h"


typedef struct {

    float x; // Abscisse du point
    float y; // Ordonnée du point

} Point;

// "Constructeur" version langage C :)
void init(Point *, float, float);


int main() {

    Point *p1;  // Déclaration d'un pointeur de type Point
    p1 = (Point *)malloc(sizeof(Point)); // Allocation de la mémoire

    if (p1 == NULL) {
        printf("Erreur d'allocation de mémoire de p1");
        return EXIT_FAILURE;
    }
```

On notera la bibliothèque *stdlib.h* qu'il faut importer. **D'une manière générale, il faut inclure *stdio.h* et *stdlib.h* dans tout programme en langage C.**

Remarque : on pourrait penser que la taille d'une structure est celle de la somme des tailles de ses éléments. Ce n'est pas forcément le cas, il peut y avoir des « trous » non référencés dans la structure.

Voici un exemple :

```c
typedef struct {
    char c;
    int val;
} My_struct;


int main()
{
    printf("Taille de la structure = %i \n", sizeof(My_struct));
    printf("Taille de la somme des membres = %i \n", sizeof(char) + sizeof(int));

    return 0;
}
```

```
C:\Users\Laurent\Documents\C Programs\First_Prog.exe
Taille de la structure = 8
Taille de la somme des membres = 5
```

Exercice : Ecrire un programme permettant d'additionner deux vecteurs (en 2D). On rappelle que les abscisses et ordonnées d'additionnent, voir le lien suivant si besoin : https://www.youtube.com/watch?v=jjrhzhbHOIQ (à partir de la 3ème minute).

# II/ Combinaisons de structures

Une **structure** peut tout à fait contenir **une ou plusieurs autres structures**. Il faut cependant être vigilant sur la gestion de mémoire si celle-ci est dynamique.

Voici un exemple de liste chaînée simple :

```c
struct Node {
    int val;            // Valeur associée
    struct Node *next;  // Pointe sur le noeud suivant
};


struct Node * init(int);        // Initialise la chaîne
void add(struct Node *, int);   // Ajout d'un élément
void del(struct Node *);        // Supprime la chaîne
void print(struct Node *);      // Affiche la liste chaînée


int main()
{
    // Déclaration d'une liste chaînée
    struct Node *lst = NULL;

    lst = init(1); add(lst, 2); add(lst, 3);

    // Attendu : 1,2,3
    print(lst);

    // Suppression de la liste
    del(lst);
    lst = NULL;

    return 0;
}
struct Node *  init(int n_val) {
    // Attention à indiquer la taille de la structure
    struct Node *c_node = (struct Node *)malloc(sizeof(struct Node));

    // et non de son pointeur
    c_node->next = NULL; // Pas de suivant
    c_node->val = n_val;

    return c_node;
}


void add(struct Node *c_node, int n_val) {
    // Si la chaîne n'est pas valide
    if (c_node == NULL) {
        printf("Chaîne vide ou incorrecte");
        return; // On pourrait quitter le programme
    }

    // Constrution du noeud suivant
    struct Node *n_node = NULL;
    n_node = (struct Node *)malloc(sizeof(struct Node));
    n_node->next = NULL; // Il est le dernier maillon
    n_node->val = n_val;

    // Accès au dernier élément de la liste
    // pour ajouter le nouvel élément en fin de liste
    while (c_node->next != NULL)
        // Récupération du noeud suivant
        c_node = c_node->next;

    // Le noeud précédent a désormais un suivant
    c_node->next = n_node;
}
```

```c
void del(struct Node *c_node) {

    struct Node *n_node = NULL;

    // Tant que la chaîne n'est pas vide
    while (c_node != NULL) {
        // Récupération du noeud suivant
        n_node = c_node->next;
        // Désallouement de c_node
        free(c_node);
        // c_node devient le noeud suivant
        c_node = n_node;
    }
}


void print(struct Node *c_node) {
    if (c_node == NULL) {
        printf("Chaîne vide");
        return;
        }

    // Tant que la chaîne n'est pas vide
    while (c_node != NULL) {
        // On affiche la valeur du noeud
        printf("%i \t", c_node->val);
        // c_node devient le noeud suivant
        c_node = c_node->next;
    }

    printf("\n");
}
```

Résultat

```
1       2       3

--------------------------------
Process exited after 0.2441 seconds with return value 0
Appuyez sur une touche pour continuer...
```

Exercice : Modifier le programme suivant pour que l'on puisse également accéder au maillon précédent s'il existe (liste doublement chaînée).

Voici la structure :

```
struct Node {
    int val;            // Valeur associée
    struct Node *next;  // Pointe sur le noeud suivant
    struct Node *before;// Pointe sur le noeud précédent
};
```

Attention à la gestion de la mémoire !!