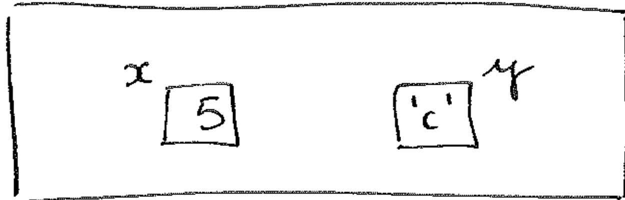
Chapitre $3^{1/2}$

Exécution d'un programme C

Un programme est un texte qui est compris par la machine comme une série d'ordres à exécuter. Pour l'essentiel, ces ordres sont destinés à modifier la mémoire de la machine. L'objet de ce chapitre est d'apprendre à représenter de manière simplifiée (mais instructive) l'état de la mémoire et les modifications induites par les exécutions.

Une mémoire est un espace dans lequel on peut créer et supprimer des boîtes ; ces boîtes ont un nom, permettant de les désigner et peuvent contenir une valeur.

exemple :



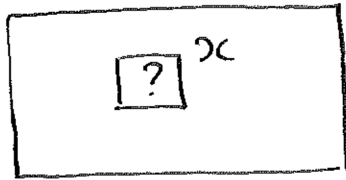
cette mémoire contient 2 boîtes appelées x et y et contenant respectivement un entier 5 et un caractère 'c'. Nous allons à présent relier cette représentation aux instructions exprimables en langage C.

1. Instructions simples

Les boîtes vues plus haut correspondent exactement aux variables du C. Une instruction

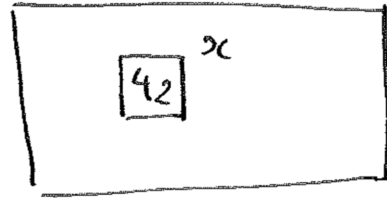
```
int x;
```

crée une boîte en mémoire, nommée x, ne pouvant contenir que des valeurs entières, et non initialisée :

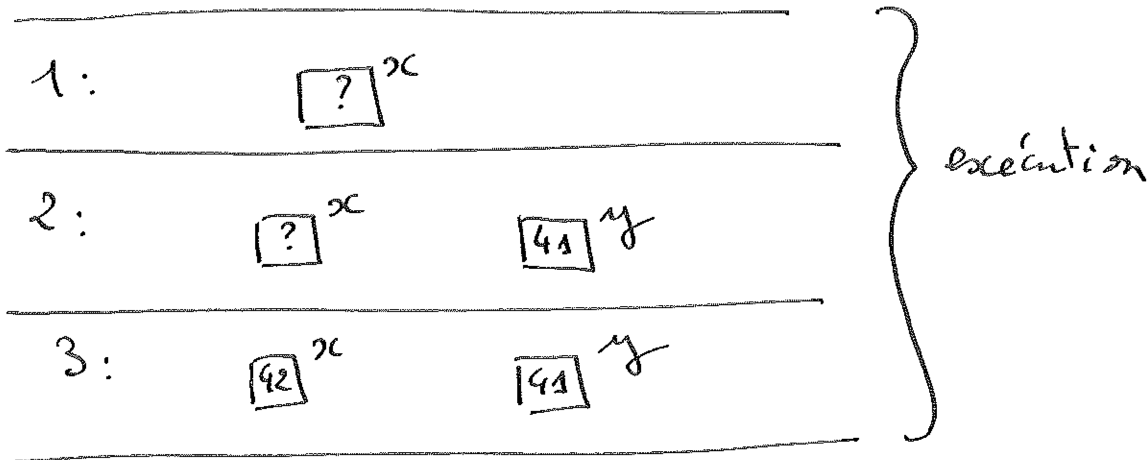


L'affectation permet de mettre une valeur dans la boîte :

x = 42;



Si on exécute une suite d'instructions, on peut définir l'état mémoire après chaque ';'



Rappel : on s'interdit absolument de lire le contenu d'une variable non initialisée : à chaque fois que l'on écrit le nom d'une variable il FAUT savoir où elle a été initialisée.

Voyons maintenant un exemple avec une boucle while

```
1:    int i = 1;
2:    int x = 2;
3:    while (x < 8) {
4:        i++;
5:        x *= 2;
      }
```

Ligne	i	x
1 :	1	
2 :	1	2
3 :	1	2
4 :	2	2
5 :	2	4
3 :	2	4
4 :	3	4
5 :	3	8
3 :	3	8

FIN

2. Les types de variable

Nous précisons ici comment représenter par des boîtes les différents types de variable :

- les types primitifs : int, float, char etc... sont représentés par une boîte simple

```c
int x = 3;
```

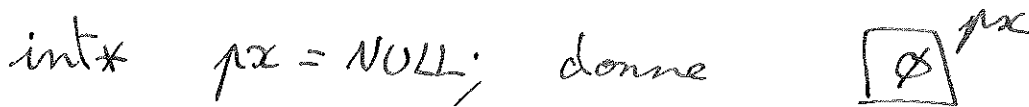
$\boxed{3}^{x}$

- les pointeurs sont des boîtes dont le contenu désigne une autre boîte

```c
1: int x = 3;
2: int* px = &x;
```

} donne 2 :

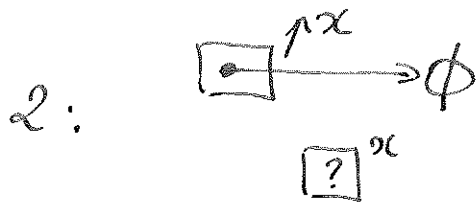
Précisons deux cas particuliers, à savoir le pointeur nul



```c
int* px = NULL;
```

donne

et un pointeur désignant une zone non allouée (et donc INTERDITE)



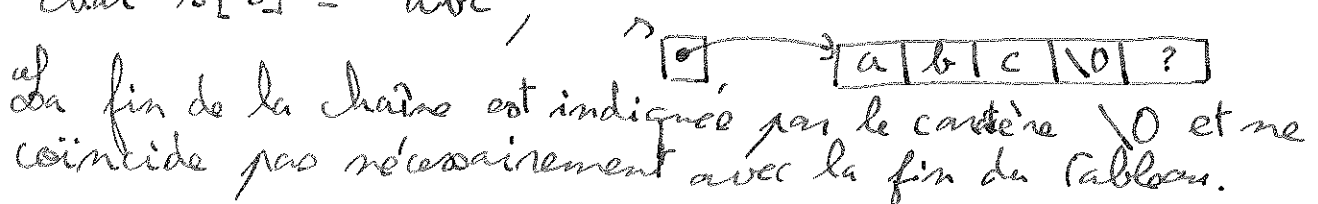

```c
1: int x;
2: int* px = &x + 42;
```

- les tableaux sont des pointeurs désignant la première d'une série de boîtes contiguës en mémoire

```c
int x[3] = {0, 1, 2};
```


Rappel : les chaînes de caractères (qui sont des tableaux de char) ont un codage particulier par convention

```c
char s[5] = "abc";
```

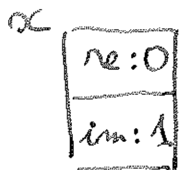


La fin de la chaîne est indiquée par le caractère \0 et ne coïncide pas nécessairement avec la fin du tableau.

- les structures permettent de créer des "sous-boites".
Le programme suivant

```
struct complex_t {
    float re ;
    float im
}
struct complex_t x;
x.re = 0;
x.im = 1;
```

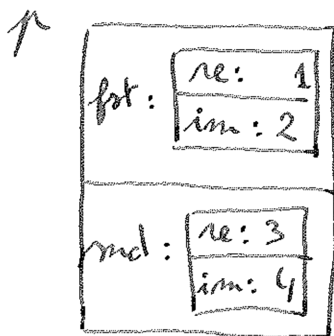
produit à la fin l'état :

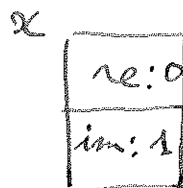


Intuitivement, on subdivise la boîte en autant de compartiments que la structure contient de composantes.
Si on poursuit le programme précédent avec

```
struct pair_t {
    struct complex_t fst, snd;
}
struct pair_t p = { {1, 2}, {3, 4} };
```

on obtient

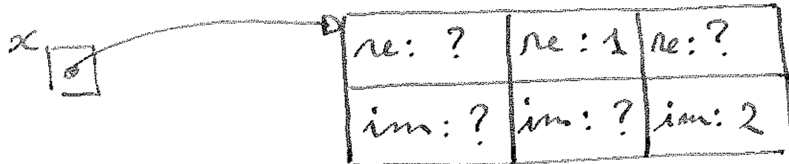




Un autre exemple, cette fois avec des tableaux de structures :

```
struct complex_t * x = (struct complex_t *) malloc (
                              3 * sizeof(struct complex_t));
x[1].re = 1;
x[2].im = 2;
```

produit en fin d'exécution



Rappel : la création / destruction des variables est soit automatique quand il s'agit de variables locales, soit manuelle dès qu'on utilise malloc/free.

3. Appel de fonction

Un appel de fonction revient à mettre temporairement en attente l'exécution d'un bloc pour aller exécuter un autre bloc et en tirer un résultat. Ce saut vers une autre partie du programme s'accompagne d'une transmission d'arguments (c'est à dire de valeurs) que le bloc de la fonction peut utiliser. Le bloc appelé (le "corps" de la fonction) rend la main au bloc appelant par l'instruction `return`, qui permet au passage de rendre le résultat.

Le mécanisme utilisé pour communiquer entre l'appelant et l'appelé repose sur un tableau spécial en mémoire appelé <u>pile d'appel</u>. Avant de sauter vers le corps de la fonction, la machine stocke la ligne de code où l'appel de fonction a eu lieu. La fonction commence par créer une variable locale par argument et initialise ces variables avec les arguments stockés dans la pile. La fonction rend ensuite la main avec l'instruction return, qui récupère la ligne de code de l'appelant dans la pile, y place (s'il y a lieu) la valeur calculée par la fonction et revient à l'appelant. Ce dernier récupère la valeur de retour dans la pile et poursuit l'exécution. En ce qui nous concerne, deux choses à retenir :

- au début d'un bloc de fonction, on a toujours les variables locales, correspondant aux arguments, et initialisées avec les valeurs données lors de l'appel. Par exemple pour la fonction:

```
4:      int f(int x) {
5:          int j = 42;
```

on observera:

x [28]

---

x [28]   j [42]

- quand le programme arrive à la fin du corps d'une fonction, il exécute une instruction return (même si elle n'est pas explicitement écrite). Si la fonction est censée rendre un résultat, l'appelant va le chercher dans la pile. Or, si on a oublié d'en placer un grâce à une instruction return (explicite), l'appelant regardera au mauvais endroit et utilisera un résultat erroné, pouvant causer toutes sortes de problèmes. Moralité : ne pas oublier l'instruction return pour rendre le résultat d'une fonction.

4. Quelques exemples

4.1 Matrices

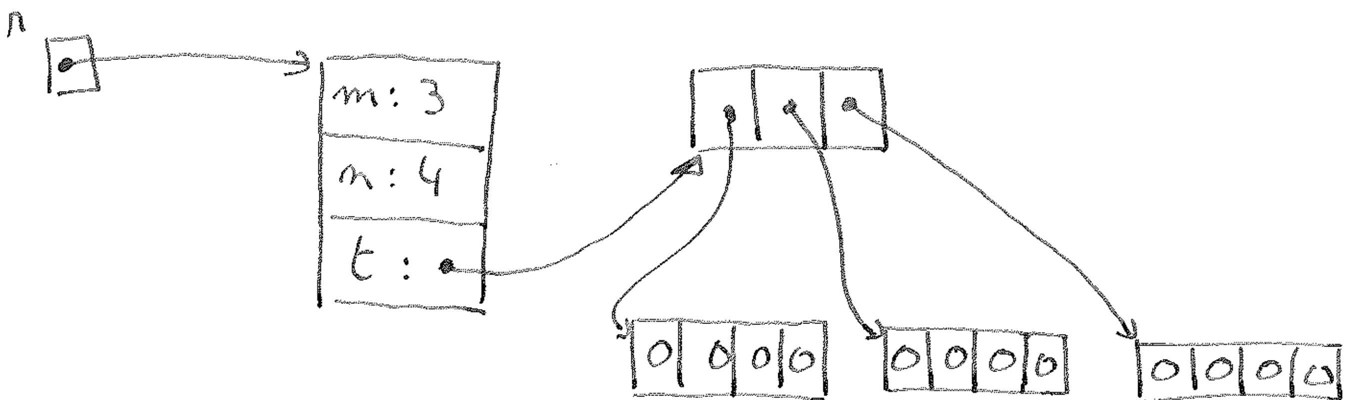
Les matrices ont été définies en TD par

```
struct matrix_t {
    int m;  // nb de lignes
    int n;  // nb de colonnes
    float** t;
};
typedef struct matrix_t* matrix;
```

Le constructeur de matrice nulle s'écrit :

```
matrix   null_matrix (int m, int n) {
    int i, j;
    matrix r = (matrix) malloc(sizeof(struct matrix_t));
    r->m = m;
    r->n = n;
    r->t = (float**) malloc(m * sizeof(float*));
    for (i=0; i < m ; i++) {
        r->t [i]= (float*) malloc (n* sizeof(float));
        for (j=0; j<n; j++) {
            r->t [i][j] = 0;
        }
    }
    return r;
}
```

L'appel matrix m = null_matrix (3, 4); produit ce résultat en mémoire



4.2 Listes

Rappel : il faut savoir ce qu'est une liste (suivez mon regard).

La représentation vue en cours, pour les listes d'entier

```
struct list_t {
    int head;
    struct list_t* tail;
};
typedef struct list_t* list;
```

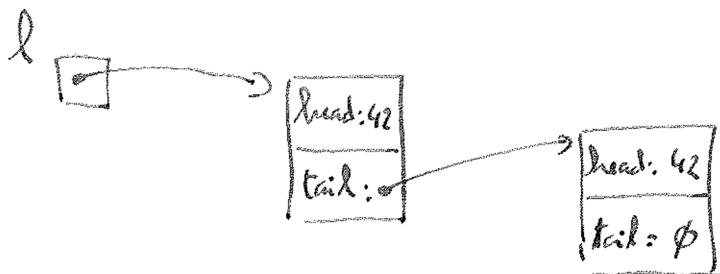
et la valeur NULL représente par convention la liste vide.
Le constructeur s'écrit :

```
list cons(int h, list t) {
    list r = (list) malloc (sizeof(struct list_t));
    r->head = h;
    r->tail = t
    return r
}
```

L'appel

```
list l = cons (42, cons(42, NULL));
```

produit



Étude de la fonction copy_list

prototype : list copy_list(list l);

description : prend en argument une liste et produit une copie en mémoire

précondition : aucune

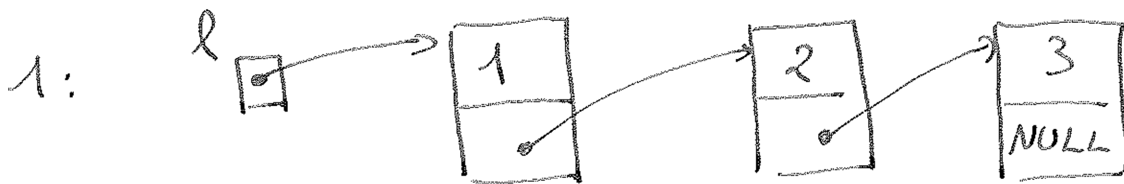
implémentation récursive :

```
1:  list copy_list (list l) {
2:       if(l==NULL) return l;
         else {
3:           list tail = copy_list(l->tail);
4:           list r = cons(l->head, tail);
             return r;
         }
    }
```

test sur un exemple

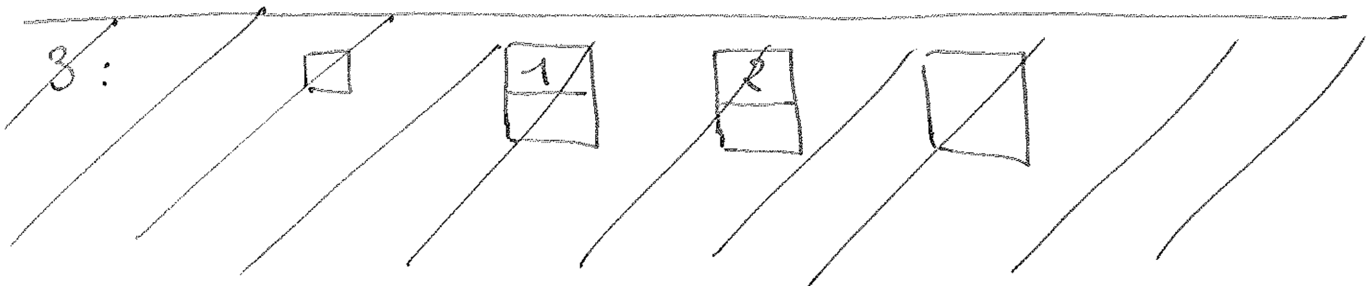
On suppose que la liste donnée en entrée représente 1 :: 2 :: 3 :: ϕ

1:

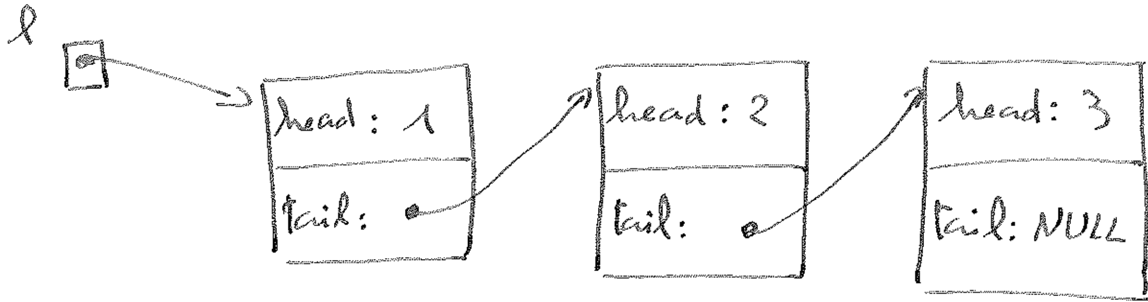


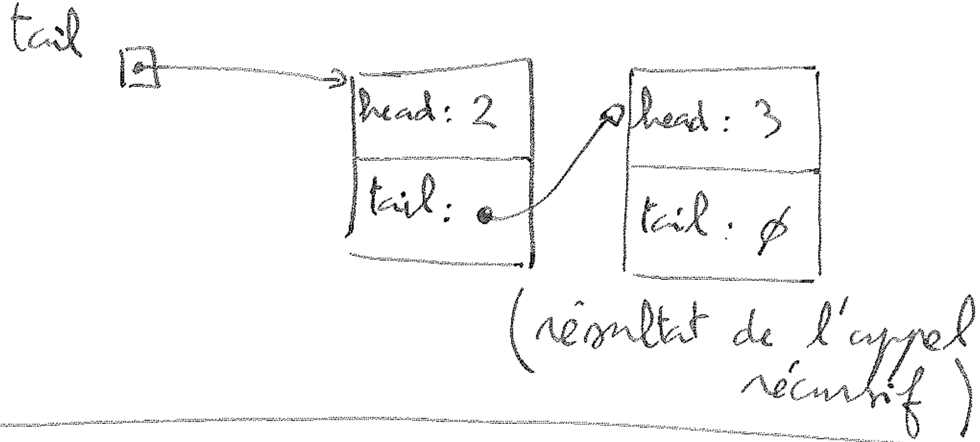
2: idem

3:

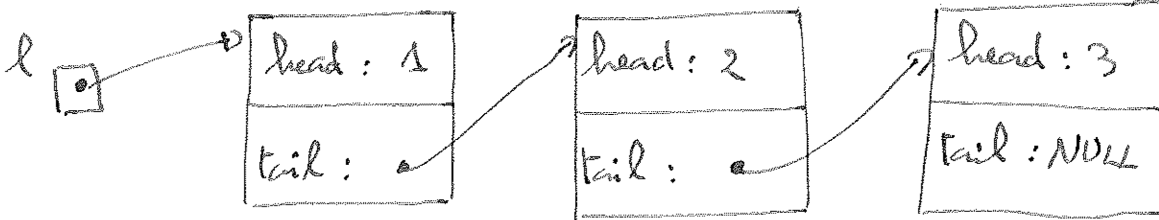


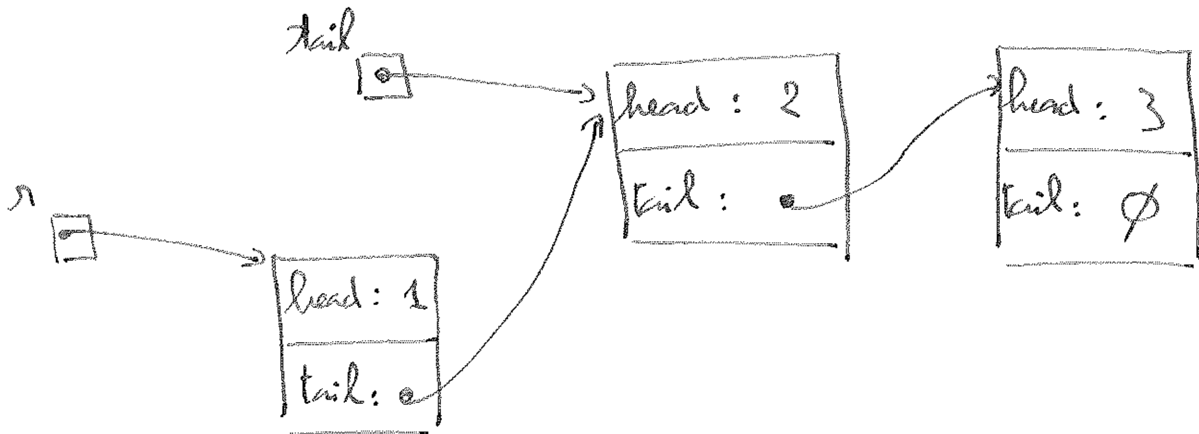
3:





4:





5/ La fonction prend fin et retourne la valeur de r.

complexité : on choisit de compter le nombre d'appel T à cons. On a la récurrence sur la taille de la liste en entrée

$$\left.\begin{array}{l} T(0) = 0 \\ T(n) = T(n-1) + 1 \end{array}\right\} \Rightarrow T(n) = n \Rightarrow \text{complexité linéaire}$$