§4. Pointeurs

Challenge simple + complexe i) ii) → vendredi
(iii) fait en TD).

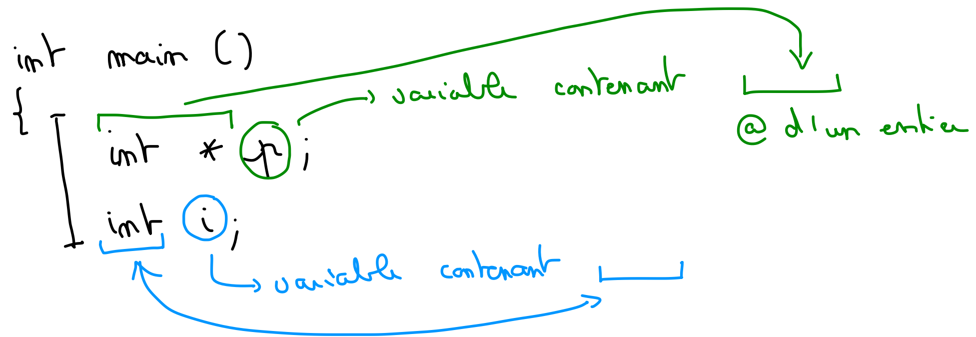
I. Pointeurs et adresses

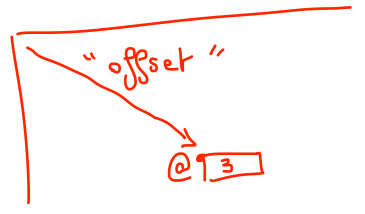
Pointeur : | variable (standard) d'un
| nouveau type qui désigne une adresse

type → T * — type : adresse d'une donnée de type T

adresse → d'une donnée du type T

ex : variables de type int * (nom type ← adresse de ...)
↳ variables contenant l'@ d'un entier.





// p est un pointeur ⇒ contient des adresses

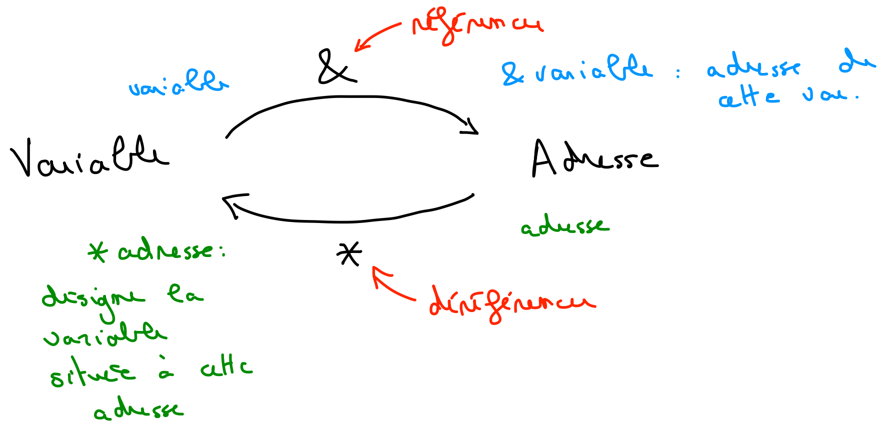
// ici : non initialisée

p = ...
adresse ?

Comment récupérer / "fabriquer" des adresses
↳ adresse de quelque chose (donnée en mémoire)

variables → tableaux

Récupérer l'adresse d'une variable ...

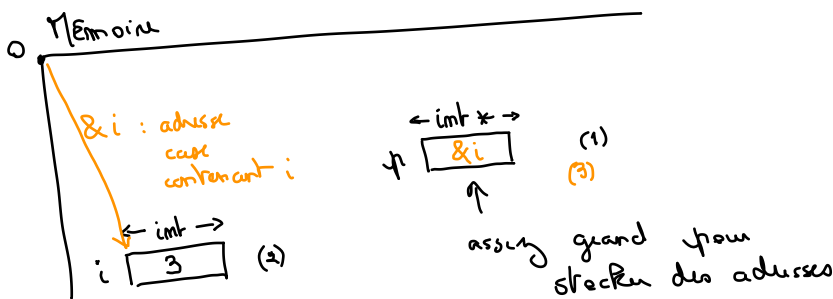


ex: (1) int * p; // p : variable de type pointeur ↔ contient une @

(2) int i = 3; // i : ——— int initialisée à 3

(3) p = &i;

// déclarer des var ⟶ réserver de la place en mémoire
mémoire ~ gros tableau (4Go)
cases ← indices → adresses.



→ on stocke ds p l'adresse de i

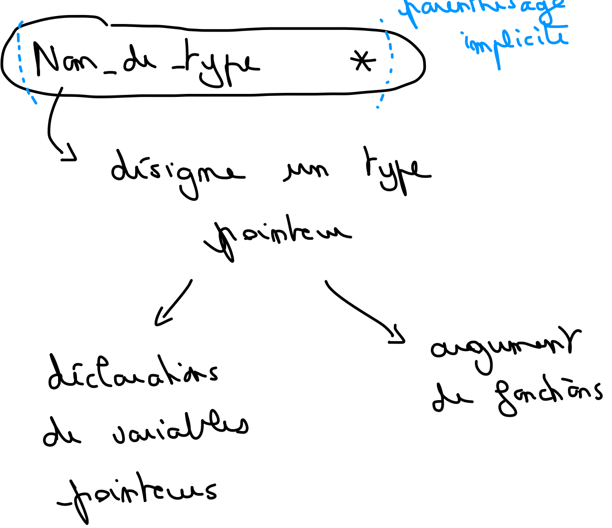
```
#include <stdio.h>

int main ()
{
  int i = 3 ;
  int * p ;

  p = &i ;

  (*p) = (*p) + (*p);

  printf("Valeur de p (entier) : %u\n", p) ;
  printf("Valeur de *p (entier) : %d\n", *p) ;

  return 0 ;
}
```

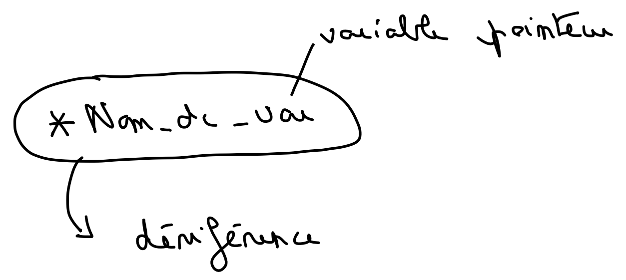
p = &i ; → p contient @i

(*p) = (*p) + (*p); → *p ⤳ variable située à l'adresse contenue ds p
↓
i

i ← i + i

⚠ A la notation *



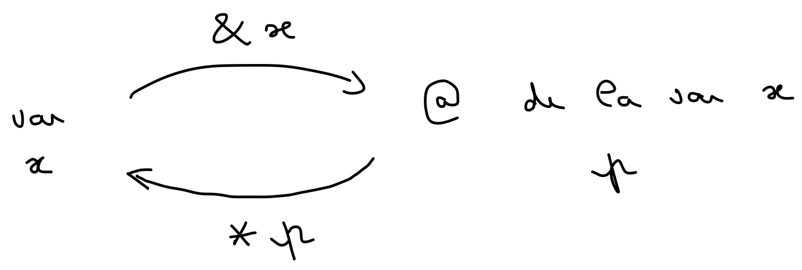


ex: p : pointeur

*p : ce qui est à l'adresse p.

ex: float * p ;

type float *

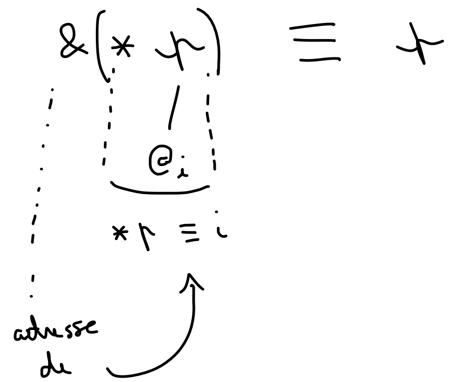


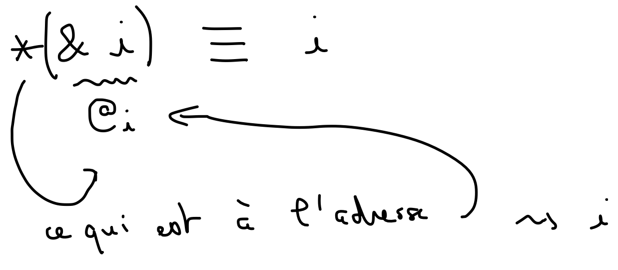
ex:
```
int i = 3 ;
int * p = & i ;
int * * q ;
```
&p → $@_p$

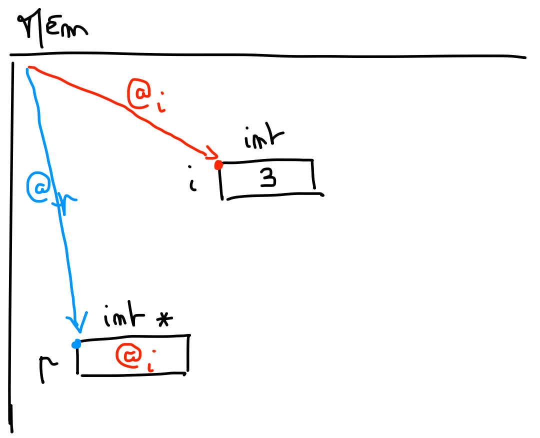
adresse ...

q = & p ;

↳ @ de p type int *







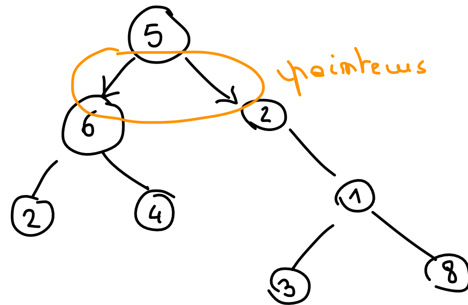
Utilité des pointeurs :

1) Pointeurs et fonction ⟶ régler le pb du passage par val
↳ une fct doit pouvoir modifier une variable

ex : scanf

2) Pointeurs et tableaux ⟶ créer des tableaux dynamiques
| long. variable

3) Pointeurs et structures de données
↓
ex : arbres



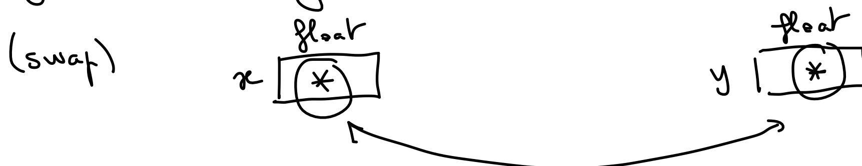
II . Pointeurs et fonctions .

Règle : si une fonction doit modifier la valeur d'une variable ⟶ passer l'adresse de cette var. en argument (pointeur)

(***)

Sauf s'il s'agit d'un tableau. (c.f. III).

ex : fonction échangeant le contenu de deux variables (float)
(swap)



⚠ Version fausse ... (naïve)

```
void swap(float a, float b)
{
    float tmp;
    (1) tmp = a;
    (2) a = b;
    (3) b = tmp;
} (4)
```

init. avec les val. reçues

variables locales

× détruites à la fin

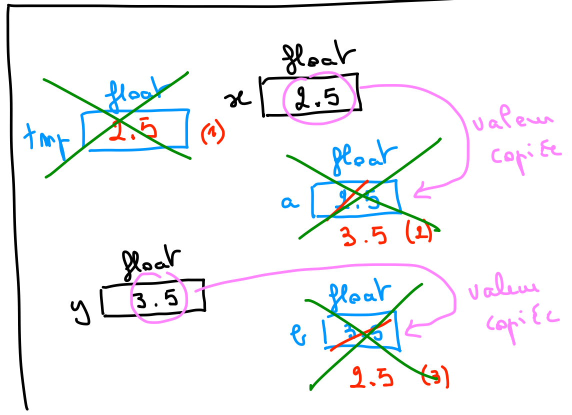
```
int main()
{
    float x = 2.5, y = 3.5;
    swap(x, y);   (1)
    // Rien ne se passe
```

valeur

lance swap sur les valeurs de x, y sont utilisées pour init (a, b) variables locales à swap.

passage par valeurs en C/C++ Java

Mém.



↳ x, y inchangées

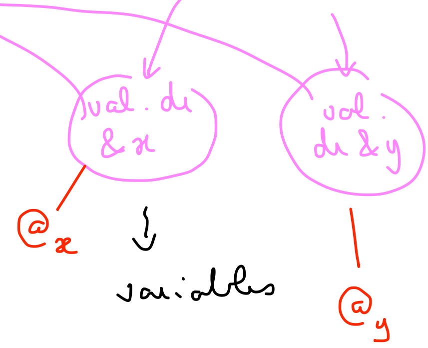
Version correcte   cf : ***

x : float ⟶ @x : float *
y ⟶ @y : float *

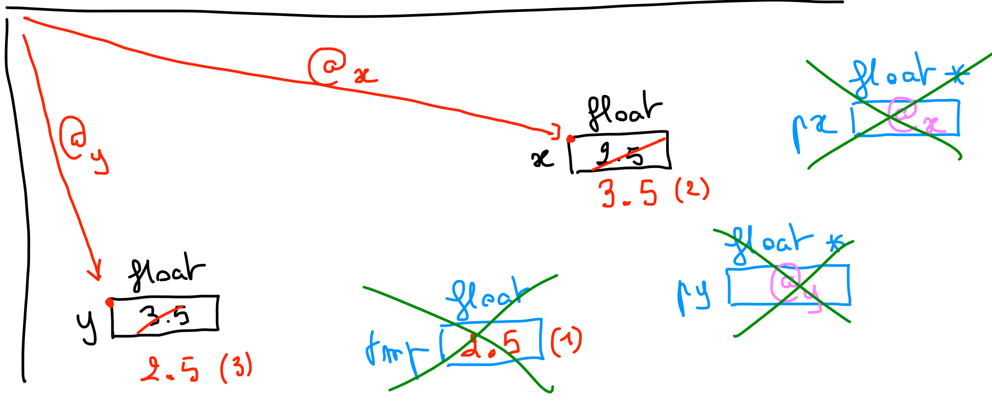
```
void swap(float* px, float* py)
{
    float tmp;
    (1) tmp = *px;      // val. ds x
    (2) *px = *py;      // variable x ← variable y
    (3) *py = tmp;      // variable y ← tmp
} (4)
```

variables locales

```
int main()
{
    int x = 2.5, y = 3.5;
    swap(&x, &y);
```



Mém.



```c
#include <stdio.h>
#include <math.h> //pour utiliser la fonction sqrt
#define N 10

//(i) Fonction calculant dans res les racines des éléments de tab (err est un code d'erreur: 0 s'il n'y a pas d'erreurs,1 sinon -
et s'il y a une erreur, err_pos indique toutes les positions d'erreur: 0 si pas d'erreur, 1 s'il y a une erreur)

int racines (float tab[], float res[], int *err, int err_pos[], int dimension)
{
   int i; //variable pour parcourir le tableau tab
   for (i=0; i<dimension; i=i+1) //on connait la taille du tableau: boucle for pour parcourir toutes les cases
   {
      if (tab[i]>=0) //si on peut prendre la racine du nombre contenue dans la case tab[i]
      {
         *err=0; //il n'y a pas d'erreur car le calcul est possible
         err_pos[i]=0; //il n'y a donc pas d'erreur à la case err_pos[i]
         res[i]=sqrt(tab[i]); //et on stocke, dans res[i], la valeur absolue de tab[i]
      }
      else //si on ne peut pas prendre la racine du nombre contenue dans la case tab[i]
      {
         *err=1; //il y a une erreur car le calcul n'est pas possible
         err_pos[i]=1; //cette erreur se trouve à la case err_pos[i]
         res[i]=-1; //critère pour montrer à l'utilisateur qu'il y a un problème car on ne peut pas avoir une racine négative
      }
   }
   return *err; //pas d'erreur: 0 sinon 1
}

// (ii) Main testant la fonction

int main()
{
   float T[N]; //tableau initial contenant les valeurs (réelles)
   float Tracine[N]; //tableau final contenant les racines des valeurs initiales (réelles)
   int* erreur; //la variable entière "erreur" stocke une adresse
   int erreur_position[N]; //tableau qui indique les positions qui présentent des erreurs
   int j; //entier pour parcourir les tableaux
   int dim; //entier définisant la dimension des tableaux
   int res;

   //Saisie de la dimension des tableaux
   printf("Saisir la dimension des tableaux (max 10)");
   scanf("%d", &dim);

   //Saisie des valeurs du tableau initial
   for (j=0; j<dim; j=j+1)
   {
      printf("Saisir la valeur de la case %d du tableau:", j);
      scanf("%f", &T[j]);
   }

   //Utilisation de la fonction
   res=racines(T, Tracine, erreur, erreur_position, dim);

   //Affichage du résultat
   for (j=0; j<dim; j=j+1)
   {
      printf("La valeur absolue de la case %d vaut %f d'erreur de position valant %d\n",j, Tracine[j], erreur_position[j]);
   }
   return 0;
}
```

fct modifie la var → pointeur s/ variable

type * | int * → type pointeur s/ entier

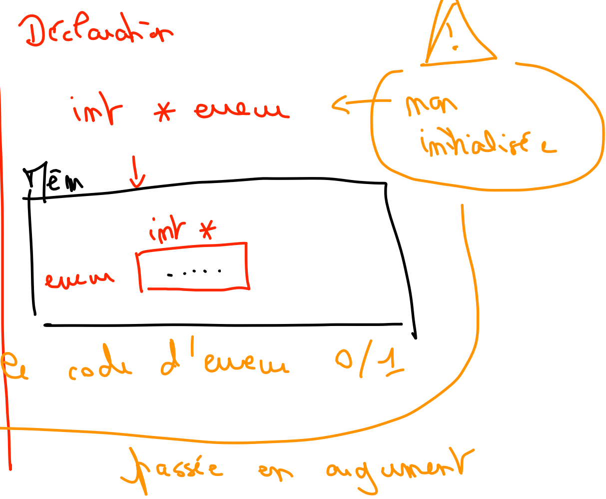
déréférencement : @ en * variable à cette adresse

va à une @ ..... ↓ n'importe quoi

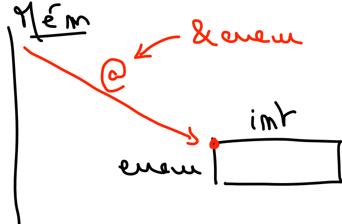
segmentation fault.

type ok

adresse d'un entier dans lequel on met le code d'erreur 0/1



passée en argument



alloue

Solution main

int erreur;

res = racines(T, Tracine, ..., ...)

&erreur

@ d'un entier où stocker le code d'erreur.

int racines (float tab[], float res[], int *~~err~~ perr, int err_pos[], int dimension)

III. Pointeurs et tableaux

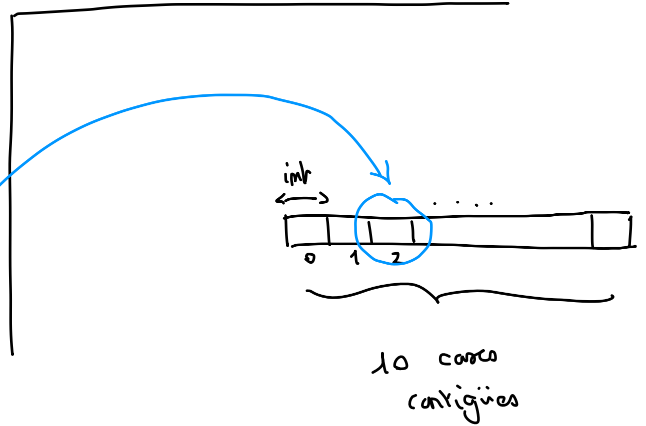
But : comprendre la relation pointeurs – tableaux
⟹ tableaux dynamiques.

① Tableaux et pointeurs en C

En C :

```
int main()
{
    int T[10];
    ⋮
```

→ tableau de 10 entiers (case contiguës)



T[2] = 3;   T[i] → i+1 ème case

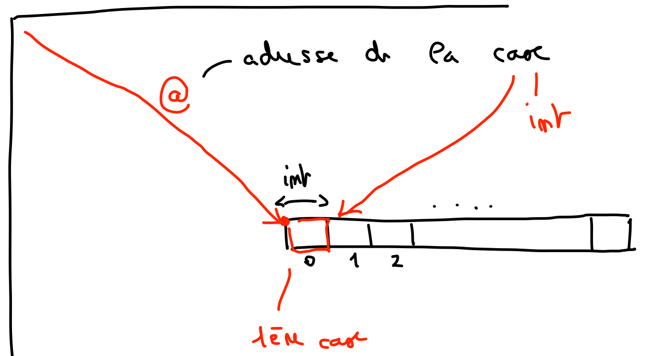
Variable T ?
→ tableau ... ?
qu'est-ce que ça désigne en C

> En C un tableau est identifié / désigné à l'adresse de sa 1ère case.

T ~ adresse de sa 1ère case

T = @ ← case : int

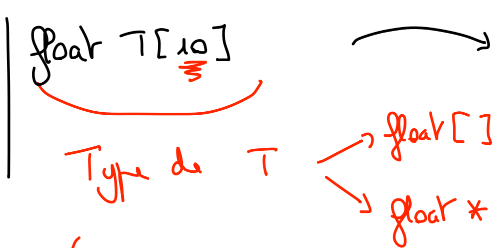
type : int *



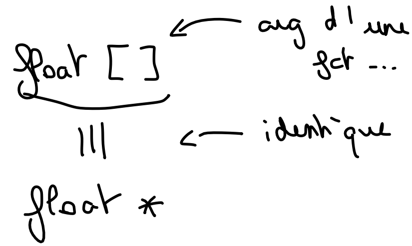
Conséquence

> tableau de ... ≡ ... *

ex :



→ T : type



↓ passer en arg à une fct :

ex 2 :

int racines (float T[ ] , ...)

| identique à

int racines (float * T , ...)

> tableau
> |||
> adresse de
> la 1ère case

```c
#include <stdio.h>
#define N 10

int main()
{
  float T[N] ;

  printf("Affichage de la var T : %p\n", T) ;
  printf("Affichage de l'adresse de la 1ere case de T : %p\n", &T[0]) ;

  return 0 ;
}
```

T ≡ @ 1ère case (float *)

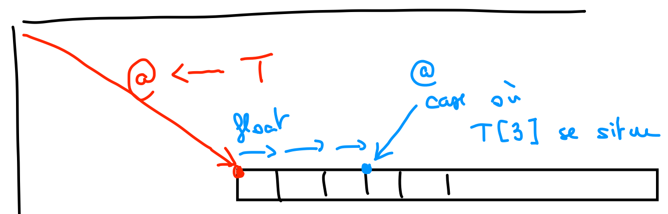
A partir de l'adresse 1ère case :

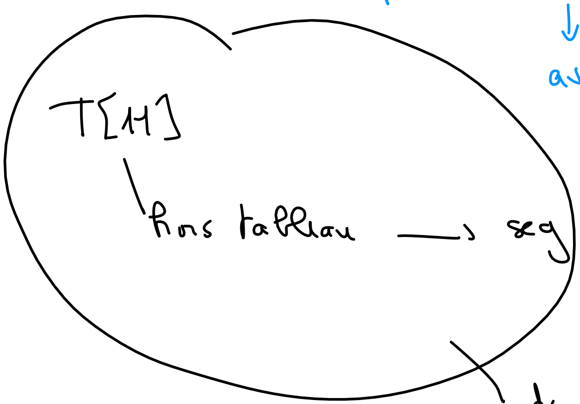
T[3]

↳ à partir de la 1ère case (T)

↓

avancer de 3 cases float (car T : tableau de float)





→ seg fault si cet emplacement est réservé

ds une zone non réservée — non déclarée.

2 Allocation dynamique

(≠ allocation statique jusqu'à présent
↓
taille fixe / début main)

Alloc. dynamique :

fonction **malloc**

entrée → taille à réserver (octets)

sortie → renvoie l'@ de l'espace alloué

@ du début de l'espace alloué
ex : tableau.

Taille à réserver pour un tableau de n cases :

↓ dépend du type des cases

n × taille d'une case
≡
taille du type contenu ds la case

→ fonction sizeof

type → taille en octets

ex : tableau de n float

↳ taille : n * sizeof(float) (multiplié)

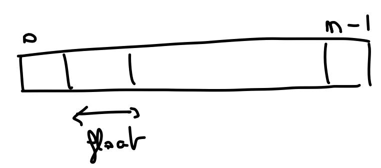
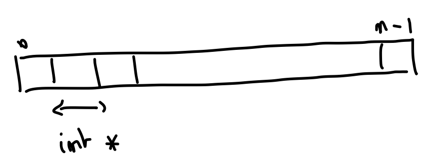


tableau de n (int *) (n pointeurs)

↳ taille : n * sizeof(int *)



Allocation dynamique

ex : créer un tableau dynamique de n char

```
int main()
{
    int n;
(1) char * T;
    ⋮
```

← T : reçoit le résultat de malloc
↳ @ 1ère case

↓ pointeur s/ char

Déclaration avec ~~char T[]~~

[..] ⟺ tableau statique → long fixe et spécifiée.

// Saisie n par l'utilisateur
printf("n ? \n");
scanf("%d", &n);

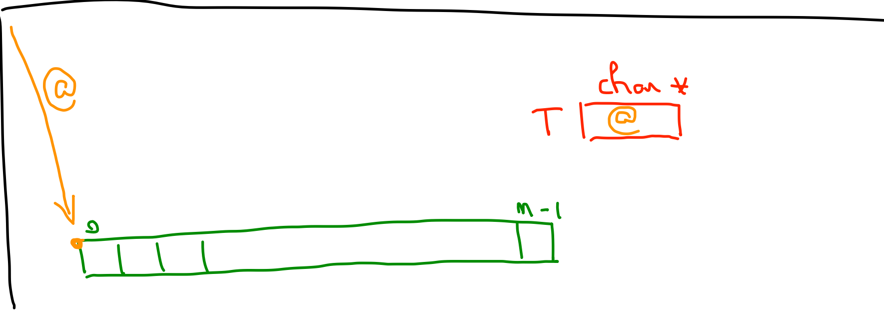
// Allocation dynamique ~> n char

(2) T.... = malloc(n * sizeof(char));

↳ renvoie @ 1ère case du tableau) ≡ tableau ↗ pointeur s/ char ≡ Tableau

~> accès aux cases normal

T[0], T[1], T[2] .... T[n-1]



(1) T : char *
↓
variable contenant 1 @ de char

(2) → alloue un espace de n char

→ renvoie l'@ 1ère case

⚠ malloc est ds stdlib.h → malloc, free, ... } alloc mém.

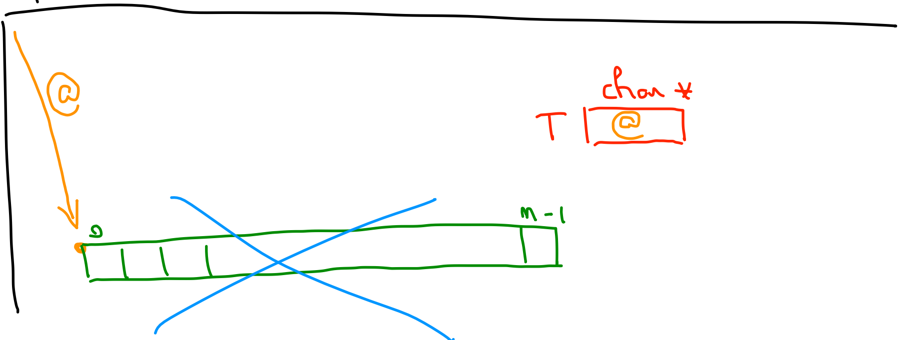
Désallocation dynamique

~> fonction free

free(pointeur) → libère ce qui est à cette adresse (ex: tableau).

ex : suite au code précédent

free(T);

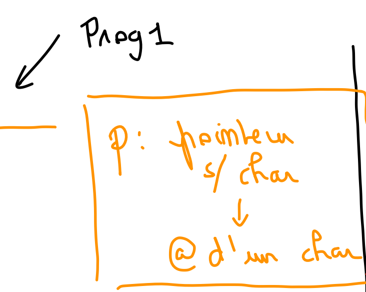


Règle : on peut faire un free sur ce qui créé par malloc .....

Attention : danger en C ....

Ds le main : Prog 1

```
char *p ;
char c = 'a' ;
p = &c ;
```



p ? → adresse d'un char

Prog 2

```
char *p ;
p = malloc(15 * sizeof(char)) ;
```

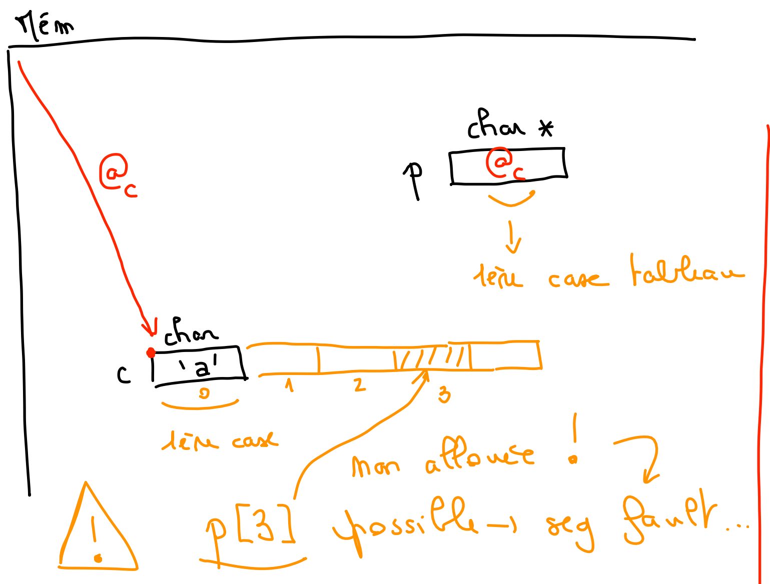
→ retourne @ 1ère case

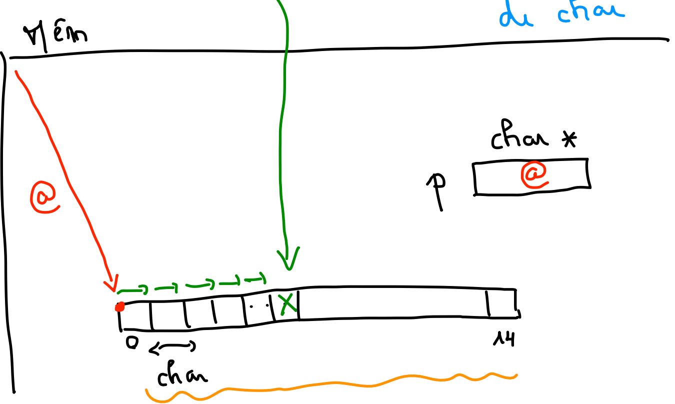
tableau ≡ @ 1ère case
char *

p ? → adresse de la 1ère case d'un tableau de char

p ≡ tableau de char

p[5]





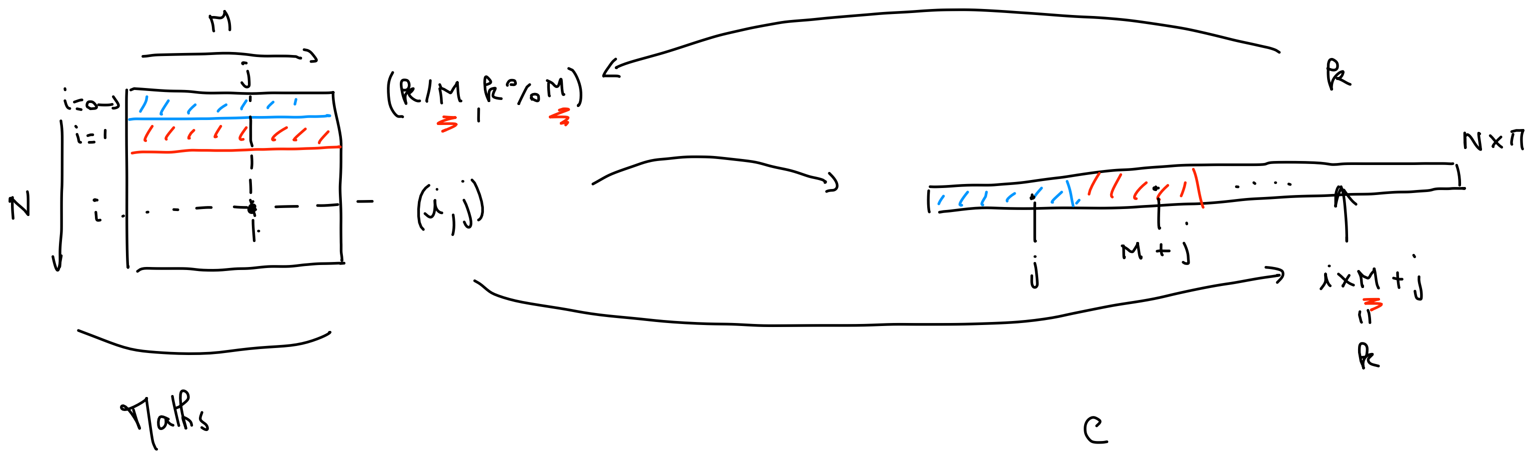
seul le programmeur sait ce qu'il met ds p et peut distinguer les 2 .....

Question : tableaux 2D / matrices ⇝ matrices statiques

```
int main()
{
    float M[N][M] ;
    M[i][j]
```

mémoire → tableau de taille N×M → stockage / ligne

(i,j) → indice ds le tableau



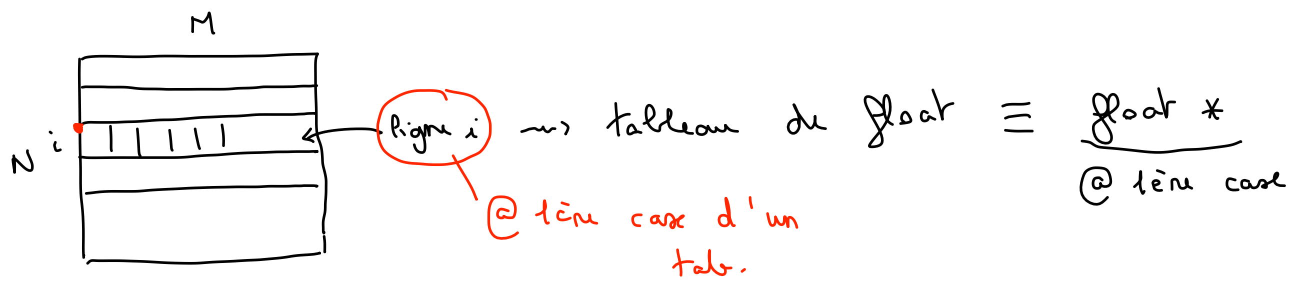
Pour passer une matrice 2D à 1 fct

... ma_fct(float M[][M], ...)

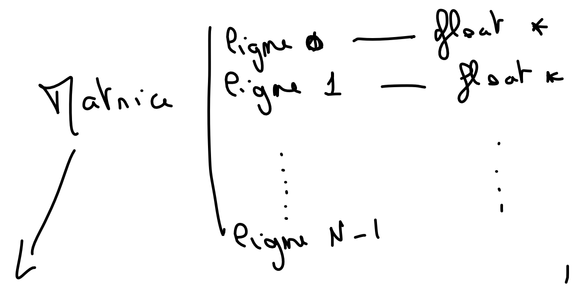
(nbre col)

→ matrice 2D statiques ... lourd

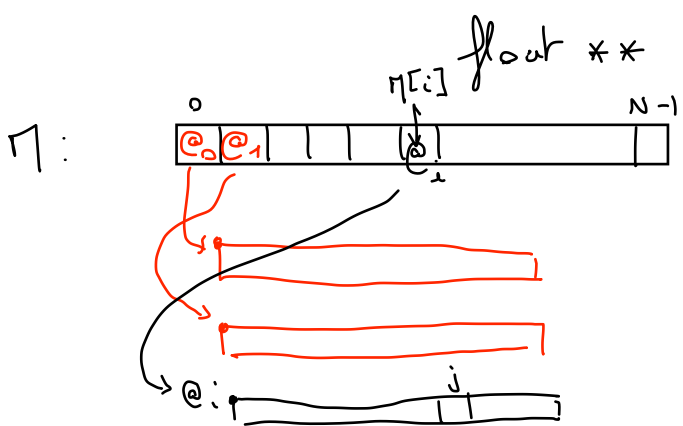
Matrices 2D dynamiques (optimales)



matrice de float



float **



→ arg : float **

M[i][j]

↳ $@_i$

IV. Chaînes de caractères

chaîne de caractères $\xrightarrow{\text{codé}}$ tableau de caractères | fini par '\0' (caract. spécial) +

"abc" ⇝