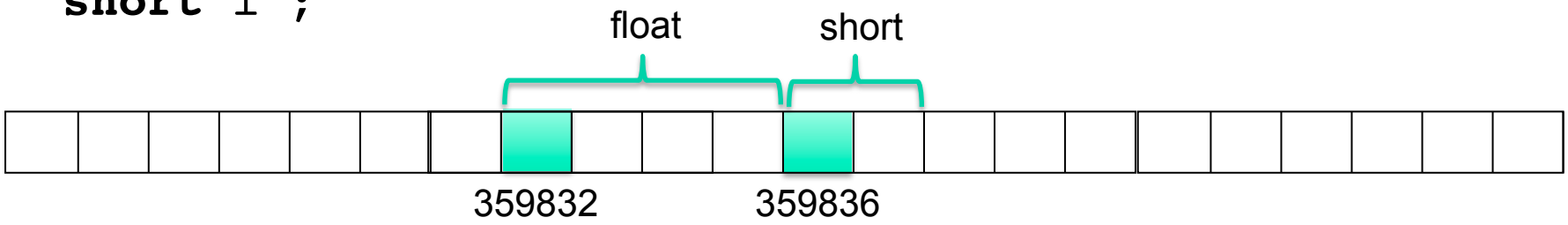
IX. Adresse et pointeur

1. Adresse d'une variable (et notion de pointeur)

Mémoire centrale : grand tableau d'octets numérotés

```
float x ;
short i ;
```



Adresse d'une variable : n° du 1er octet dans la mémoire centrale

- on peut accéder à une variable par :
  - son nom (identificateur)
  - ce n° (l'adresse)
- certaines variables n'ont pas de nom …

Qu'est-ce qu'une adresse (n° dans la mémoire) ?

- un entier positif
- pourra être stocké dans ~~un long non signé ?~~ un **pointeur**
- Pour déclarer un pointeur, on précise le type de la variable qui sera pointée
  - pour connaître le nbre d'octets nécessaires à cette variable
  - pour disposer d'une arithmétique sur les pointeurs :
    adresse+1 : adresse de la variable qui suit

Définition d'un pointeur :

type *****_identificateur_ [,*****_identificateur_] ;

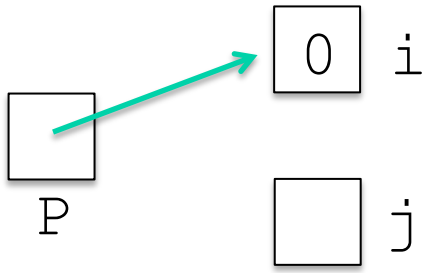
```
long *a, *b;
float *p=NULL; /*affectation conseillée*/
```

Utilisation des pointeurs

Les opérateurs * et & (*esperluette*)

- adresse d'une variable : **&***variable*
- emplacement pointé : ***pointeur**

```
int *p=NULL, i, j;
p = &i ;          p reçoit l'adr de i
*p = 0 ;          i reçoit 0
p = &j ;          p reçoit l'adr de j
*p = 1 ;          j reçoit 1
```

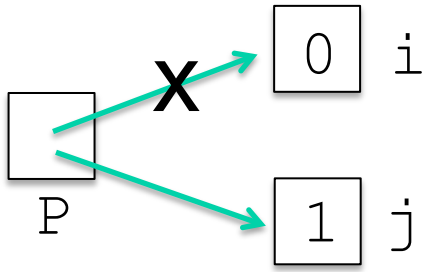


Utilisation des pointeurs

Les opérateurs * et & (*esperluette*)

- adresse d'une variable : **&***variable*
- emplacement pointé : ***pointeur**

```
int *p=NULL, i, j;
p = &i ;          p reçoit l'adr de i
*p = 0 ;          i reçoit 0
p = &j ;          p reçoit l'adr de j
*p = 1 ;          j reçoit 1
```



Remarques :

- `p = NULL ;` *p pointe sur rien*
- `scanf("%d",&i);` *la fct scan affecte le résultat de la lecture à l'adresse indiquée (adresse de i)*

2. Notion de lvalue et de rvalue

Deux attributs pour une expression :

1. sa *valeur*
2. son *type*

➔ c'est le cas des ***rvalue*** (*peuvent figurer à droite de* =)

exemple : `j = i/3+1 ;`

les ***lvalue*** (*peuvent figurer à gauche de* =) et ont trois attributs :

1. une *valeur*
2. un *type*
3. un emplacement dans la mémoire (une adresse)

➔ désigne le contenu ou le contenant suivant sa position dans l'affectation

Exemples

```
float x=2, *p;
p = &x;
```

`x`	*lvalue ?*
`x+1`	*lvalue ?*
`*p`	*lvalue ?*
`*(p+1)`	*lvalue ?*
`*p+1`	*lvalue ?*

Exemples

```
float x=2, *p;
p = &x;
```

`x`	est une *lvalue*
`x+1`	n'est pas une *lvalue*
`*p`	est une *lvalue*
`*(p+1)`	est une *lvalue*
`*p+1`	n'est pas une *lvalue*

Remarques :

- il y a une arithmétique sur les pointeurs
- les variables indexées et les pointeurs seront d'autres occasions de manipuler les *lvalue*

3. Allocation dynamique de mémoire

a) Opérateur `sizeof`

Donne le nombre d'octets occupés

- **`sizeof`**`(`*`variable`*`)`
- **`sizeof`**`(`*`type`*`)`
- **`sizeof`**`(`*`tableau`*`)`

b) Adresse et tableau

- l'adresse d'un tableau est l'adresse de son premier élément (1er octet).
- l'identificateur d'un tableau est un **pointeur** vers son premier élément.

```
int i, t[10];  // t est un pointeur constant
```

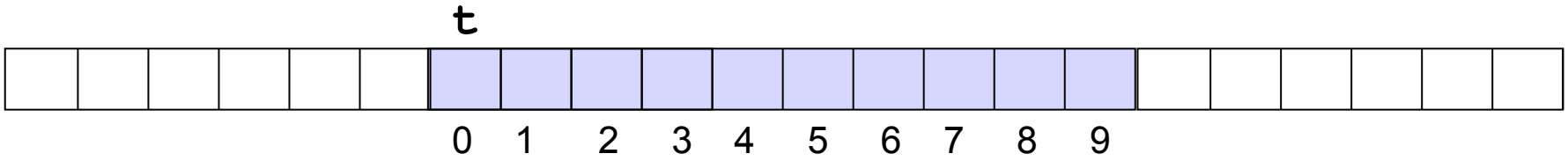
`t` est équivalent à `&(t[0])`

`*t` est équivalent à `t[0]`

`*(t+i)` est équivalent à `t[i]`, *c'est une lvalue*

```
*(t+i) = 0 ;
```

(arithmétique sur les pointeurs)



c) Allocation simple

La fonction **`malloc`**(*`entier positif`*)

- alloue le nbre d'octets précisé en argument
- retourne l'adresse du 1er octet alloué

Exemple :

```
#include <stdlib.h> /* contient la fct malloc */
 int *p=NULL;
 p = malloc(sizeof(int));
 *p = 0 ;
```

Attention :

- une nouvelle affectation de `p` entraine la perte de l'accès à l'emplacement alloué
- pour libérer (restituer) un emplacement : **`free`**(*`pointeur`*)

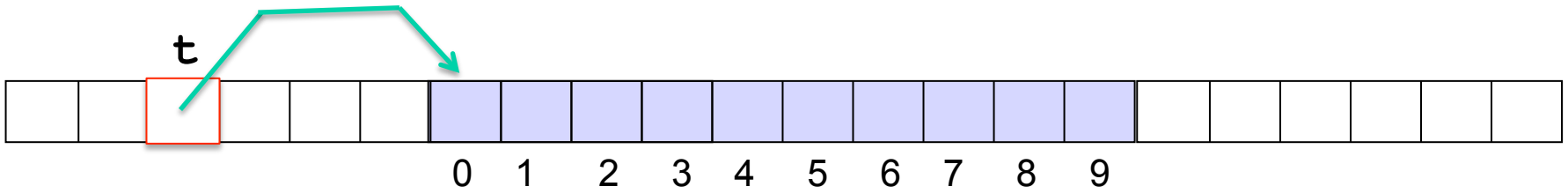
  ```
  free(p) ;
  ```

d) Allocation dynamique d'un tableau

- allocation à l'exécution (et non à la compilation)
- en fonction de la quantité et de la taille des données

Exemple : *allocation d'un tableau de n* ***double***

```
int n ;  double *t=NULL;
scanf("%d", &n);
t = malloc(n*sizeof(double)) ;
```



- accès à une case n° i :
  - `*(t+i)`
  - `t[i] // écriture la plus courante`
- restitution de tout l'espace occupé par `t` :

```
free(t) ;
```

Résumé sur l'allocation dynamique

La fonction **`malloc`**(*`entier positif`*)

- Alloue le nbre d'octets précisé en argument
- Retourne l'adresse du 1er octet alloué

```
float *p=NULL;
p = malloc(10*sizeof(float));
```

- Il faut libérer l'espace alloué une fois son utilisation terminée :

```
free(p);
```

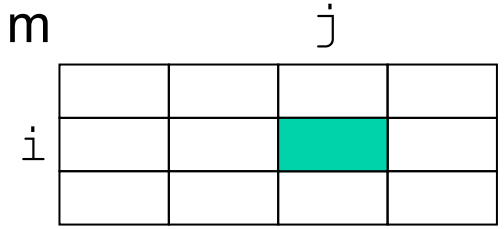
- Sinon, une nouvelle affectation de `p` entraine la perte de l'accès à l'emplacement alloué
- Définie dans la librairie **`stdlib.h`**

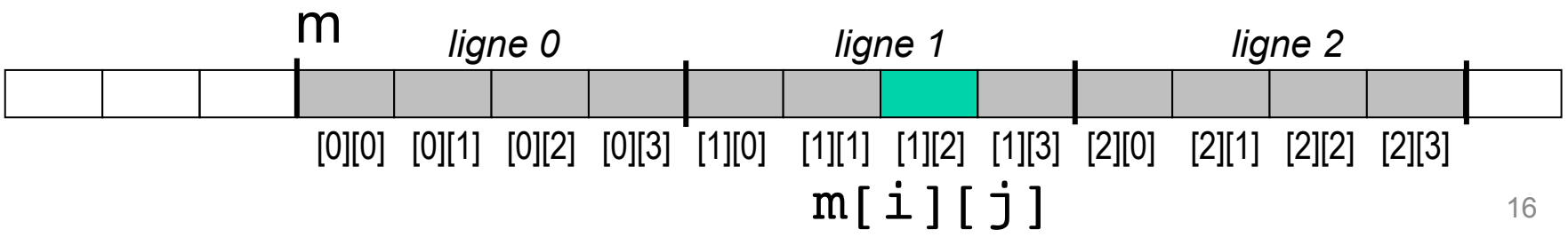
```
#include <stdlib.h>
```

Exemples :

- *Allouer un tableau de notes, saisir les notes, faire la moyenne*
- *Allouer une matrice **m** avec une constante **k** de colonnes*
  Le calcul de l'adresse d'une cellule m[i][j] sera mis en place par le compilateur

```
#define K 10   // nbre de colonnes
typedef float Ligne[K];  // type à adapter aux données
int n, i, j;  // nbre de lignes et 2 indices
Ligne *m ;     // m[i] sera l'adresse de la ligne i après allocation
scanf("%d", &n);  // nbre de lignes
m = malloc(n*sizeof(Ligne));
// Accès à une cellule (i, j) : m[i][j]
```



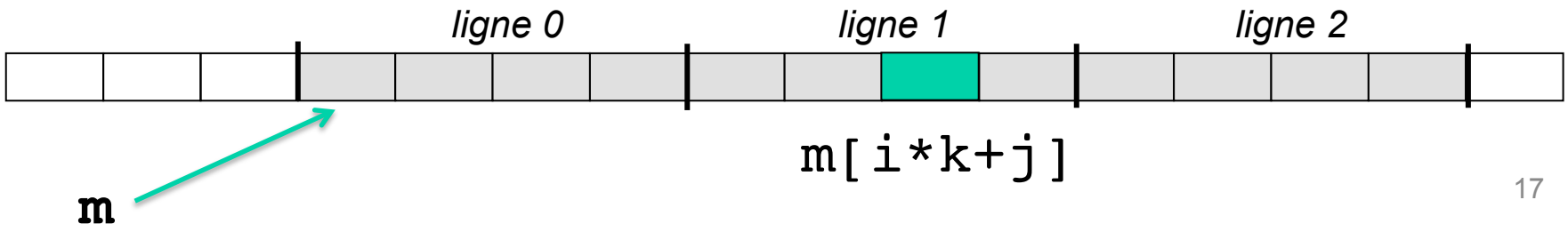


- *Allouer une matrice **m** avec un nombre **k** de colonnes défini à l'exécution*

  k n'étant connu qu'à l'exécution, le compilateur n'a pas pu mettre en place le calcul de la position d'une cellule en fonction de ses indices. C'est le programmeur qui doit gérer l'accès à une cellule.

  Solution 1 : calculer l'accès dans un tableau de dimension 1

```
float *m ;  // type à redéfinir en fct des données
int n, k, i, j;  // nbre de lignes, de colonnes et 2 indices
scanf("%d %d", &n, &k);
m = malloc(n*k*sizeof(float))
// Accès à une cellule (i, j) :   m[i*k+j]
```



Solution 2 : mémoriser les adr de lignes pour garder l'accès double indice

```
int n, k, i, j; // nbre de lignes, de colonnes et 2 indices
float *bloc, *p, **m; // type à redéfinir en fct des données
scanf("%d %d", &n, &k);
bloc = malloc(n*k*sizeof(float)); //allouer la matrice
// allouer un tableau d'adresses de ligne :
m = malloc(n*sizeof(float*));
// stocker les adresses de ligne dans m :
p = bloc ; // adr de la ligne 0
for (i=0 ; i<n ; i++)
{ m[i] = p;
  p = p+k ; // adr de la ligne suivante
}
```

// Accès à une cellule (i, j) : `m[i][j]`