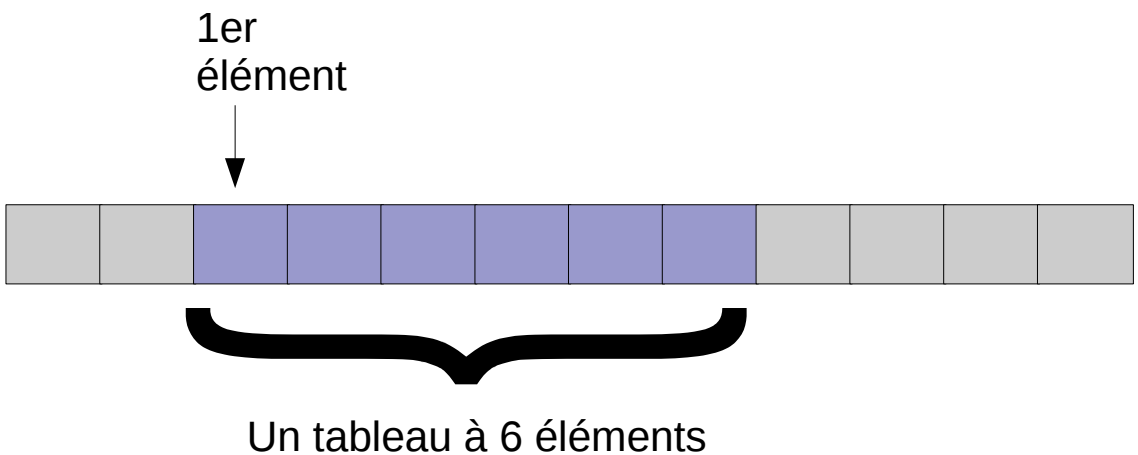
Plan

La structure de donnée tableau

Définition

Un tableau est une suite finie de données de **même type** stockée de manière **contiguë** en mémoire



- un tableau est identifiable par l'adresse de son 1^{er} élément et sa taille (son nombre d'éléments)
- les éléments sont accessibles en parcourant la mémoire à partir du 1^{er} élément

Les différents types de tableaux C

On distingue en C trois types de tableaux :

- les **tableaux statiques** dont la taille doit être connue à la compilation : elle sera donc spécifiée par un **littéral entier**. Selon l'emplacement de la déclaration de la variable (ou mot clé `static`), ils seront stockés dans le segment de code ou la pile.
- les **tableaux de longueur variable** (depuis C99) dont la taille doit être connue lors de leur déclaration et spécifiée par une **expression de type entier**. Ils ne peuvent pas apparaître qu'en variable locale et sont donc stockés dans la pile.
- les **tableaux dynamiques** qui n'ont pas de taille connue à la déclaration et devront faire l'objet d'une **demande explicite de mémoire**. Ils seront étudiés plus tard.

Plan

Déclaration de tableaux statiques

Définition

Un tableau permet de définir avec une seul nom de variable une suite de variables d'un même type.

Syntaxe

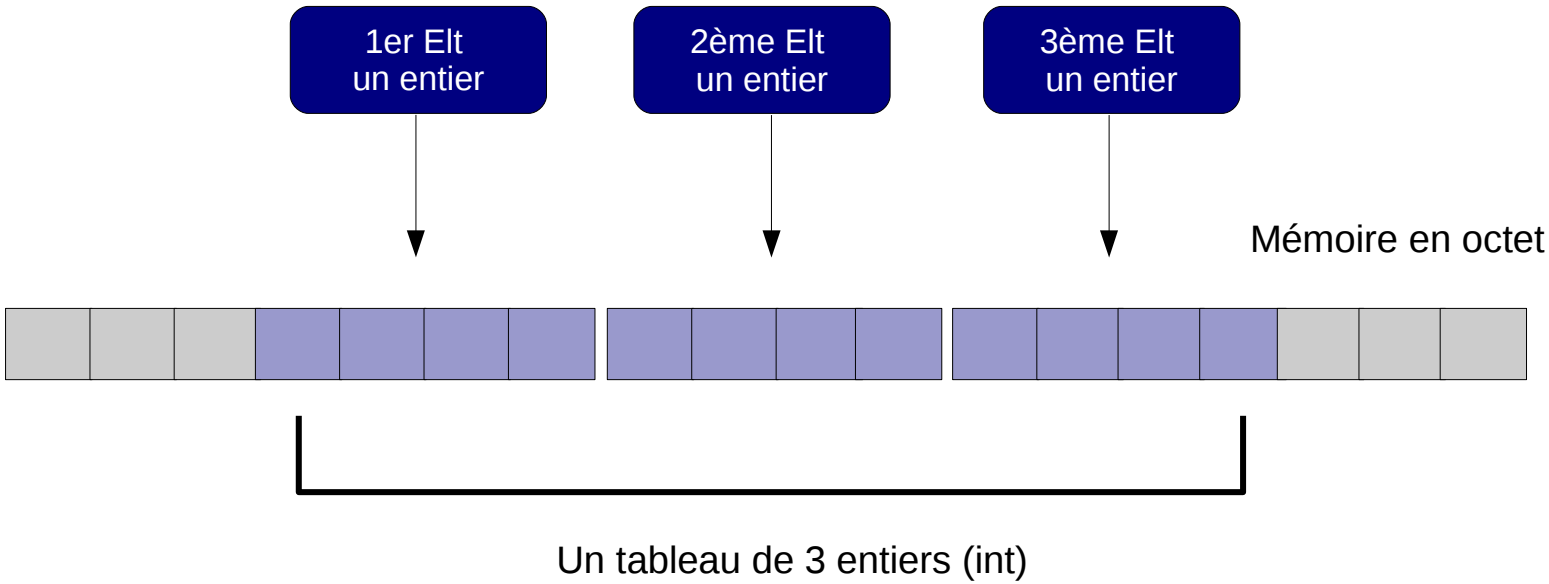
```
type nom[n];
```

- `nom` est un identificateur spécifiant le nom du tableau ;
- `type` est le type des éléments du tableau ;
- *n* est la taille du tableau (son nombre d'éléments) qui doit être une constante positive entière.

Exemple

`int t[3];` définit la variable t comme un tableau de trois entiers.

Vue mémoire



Un tableau de `n` éléments occupe `n×sizeof(type)` octets en mémoire où `type` est le type de donnée des éléments du tableau.

Ici, un tableau de 3 `int` occupe 12 octets en mémoire.

L'accès aux éléments

L'accès aux éléments d'un tableau se fait par un calcul d'adresse mémoire à partir de l'adresse du 1er élément : l'opérateur [ ] facilite le calcul.

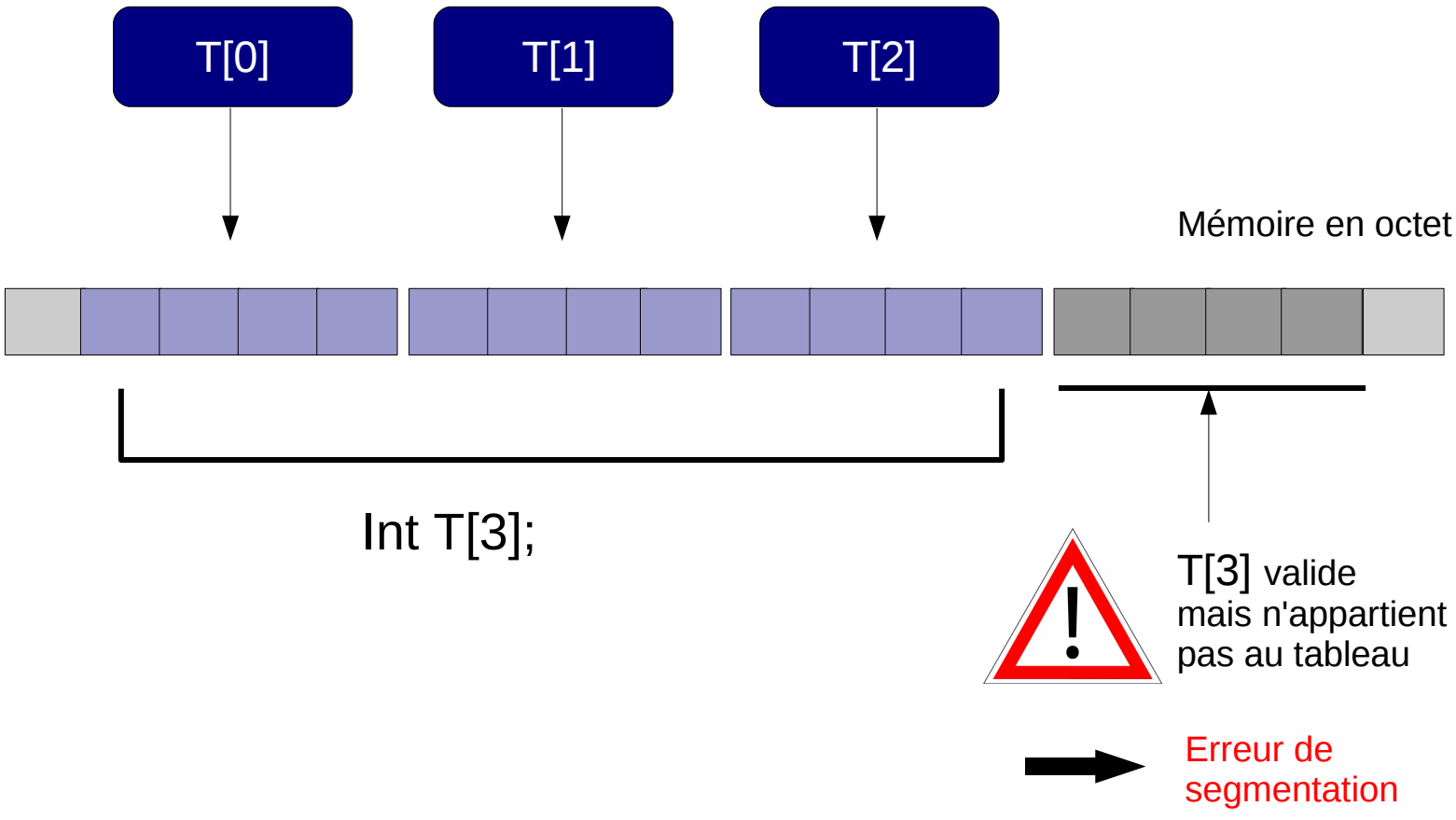
Définition

Soit la déclaration : `type t[n];`
L'instruction `t[i]` accède à la i+1-ième case du tableau `t`.
i est appelé l'indice du i+1 élément du tableau.

Attention :

- Les indices commencent à 0 et se terminent donc à n-1
- Aucune vérification numérique n'est faite pour la valeur de `i` qui doit appartenir à l'intervalle [0,n-1] (sinon débordement).
- `t[i]` peut être manipulé comme une variable accessible en lecture/écriture : t[i] = t[i] + ...

Vue mémoire


T[0]
T[1]
T[2]
Mémoire en octet
Int T[3];
T[3] valide
mais n'appartient
pas au tableau
Erreur de
segmentation

Initialisation des tableaux

Une déclaration de tableau n'initialise pas ses éléments (comme pour toute variable). On peut attribuer une valeur aux éléments par affectations successives : `t[0]=...; t[1]=...; ...`

On peut donc initialiser les éléments d'un tableau dans une boucle :

Exemple

```
int t[10];
for(int i=0;i<10;i++)
   t[i] = 0;
```

Ici on initialise chaque élément du tableau à 0.

Initialisation des tableaux

Une déclaration de tableau n'initialise pas ses éléments (comme pour toute variable). On peut attribuer une valeur aux éléments par affectations successives : `t[0]=...; t[1]=...; ...`

On peut donc initialiser les éléments d'un tableau dans une boucle :

Exemple

```
int t[10];
for(int i=0;i<10;i++)
   t[i] = 0;
```

Ici on initialise chaque élément du tableau à 0.

Remarque : on peut remplacer l'affectation par une saisie

```
scanf("%d",&t[i]);
```

Déclaration avec initialisation

Alternativement, on peut faire une déclaration avec initialisation :

Déclaration

`type t[n] = {`exp_0`,` exp_1`, ...,` exp_{n-1}`};`

Cette déclaration initialise chaque élément du tableau avec son expression correspondante : $t[i] = exp_i$.

Exemple

```
int t[3]={4, 17, 3};
```

Remarque : *Si le nombre d'expressions est inférieur à la taille du tableau, les derniers éléments sont initialisés à 0 (quelque soit le type). S'il est plus grand, les expressions en trop sont ignorées.*

Déclaration avec initialisation

Il n'est pas obligatoire de préciser la taille d'un tableau déclaré avec initialisation.

↪ Le compilateur la détermine automatiquement à partir du nombre d'expressions

Exemple

`int t[] = {4, 17, 3};` déclare un tableau de 3 éléments.

Remarque

`int t[5]={4, 17, 3};` déclare un tableau de 5 éléments, les deux derniers étant initialisés à 0.

Parcours des éléments

Dans de nombreux traitements, il est nécessaire de parcourir tous les éléments d'un tableau. Ceci ce fait grâce à une boucle **pour** sur les indices :

Exemple

```
#define TAILLE 10
...
int t[TAILLE];
...
for (int i=0;i<TAILLE;i++) {
    instructions avec t[i]
}
```

Attention avec `t[i]`

Il faut toujours garantir que $0 \leq i <$ taille du tableau
le compilateur ne le fait pas pour vous !

Parcours des éléments

Dans d'autres cas, on souhaite s'arrêter dès qu'une condition est remplie sur `t[i]`. Ceci ce fait grâce à une boucle **tant que** :

Exemple

```
#define TAILLE 10
...
int t[TAILLE];
...
int i = 0;
while (i<TAILLE && !(condition sur t[i])) {
   instructions avec t[i]
   i++; }
```

Exemple : Affichage d'un tableau

```c
#include <stdio.h>
#define TAILLE 5

int main() {
   int t[TAILLE]={4,3,1,8,6};
   int i;
   printf("[");
   for (i=0;i<TAILLE;i++){
      if(i==0) printf("%d", t[i]);
      else printf(",%d", t[i]);
   }
   printf("]\n");
   return 0;
}
```

Exécution : [4,3,1,8,6]

Exemple : Existence d'un impair dans un tableau

```c
#include <stdio.h>
#define TAILLE 5

int main() {
   int t[TAILLE]={4,3,1,8,6};
   int i = 0;
   while(i<TAILLE && t[i]%2==0)
      i++;
   }
   if(i==TAILLE) printf("Pas d'impair dans le tableau\n");
   else printf("%d est un impair du tableau\n",i);
   return 0;
}
```

Exécution : `3 est un impair du tableau`

Copie de tableau

Contrairement aux structures, il n'y a **pas d'opérations globales** sur les tableaux : pas d'affectation, pas de transmission par copie lors d'un appel de fonction.

Exemple

```
int t1[TAILLE] = {1,2,3,4,5};
int t2[TAILLE];
t2 = t1;
```

provoque une erreur de compilation !

Pour recopier un tableau dans un autre, il est donc **obligatoire de passer par des affectations successives** :

```
1 int t1[TAILLE] = {1,2,3,4,5};
2 int t2[TAILLE];
3 for (int i=0; i<TAILLE; i++)
4     t2[i] = t1[i];
```

Tableau et fonction

- On ne peut **pas passer une copie** d'un tableau à une fonction :
  - Les tableaux statiques ne peuvent pas être paramètres d'entrée d'une fonction.
  - Les tableaux statiques ne peuvent pas être paramètres de sortie d'une fonction, car comme toute variable locale si le tableau est déclaré dans la fonction, ils sont détruits à la fin de la fonction.
  - Les tableaux statiques ne peuvent pas être résultat d'une fonction, car, là encore, comme précédemment, ils seraient détruits à la fin de la fonction.
- Les tableaux statiques ne peuvent être que des paramètres d'entrée-sorties : on passe leur adresse.

Retour sur les variables d'un tableau

Soit `type t[TAILLE]` une déclaration de tableau statique :

- La variable **`t` n'est pas manipulable comme une variable classique** : pas de valeur associée, pas d'adresse de la variable.
- Seuls ses éléments `t[i]` sont des variables classiques :
  - Ils ont une adresse : `&(t[i])`
  - Ils ont une valeur : `t[i]`
  - On peut les affecter : `t[i] = ....`

Retour sur les variables d'un tableau

Soit `type t[TAILLE]` une déclaration de tableau statique :

- La variable **`t` n'est pas manipulable comme une variable classique** : pas de valeur associée, pas d'adresse de la variable.
- Seuls ses éléments `t[i]` sont des variables classiques :
  - Ils ont une adresse : `&(t[i])`
  - Ils ont une valeur : `t[i]`
  - On peut les affecter : `t[i] = ....`

Cependant, un tableau est une **suite contiguë de variables classiques** de même type. Ainsi on a les propriétés suivantes :
`&(t[i+1])` est `sizeof(type)` octets après celle de `&(t[i])`

Retour sur les variables d'un tableau

Soit `type t[TAILLE]` une déclaration de tableau statique :

- La variable **`t` n'est pas manipulable comme une variable classique** : pas de valeur associée, pas d'adresse de la variable.
- Seuls ses éléments `t[i]` sont des variables classiques :
  - Ils ont une adresse : `&(t[i])`
  - Ils ont une valeur : `t[i]`
  - On peut les affecter : `t[i] = ....`

Cependant, un tableau est une **suite contiguë de variables classiques** de même type. Ainsi on a les propriétés suivantes :

`&(t[i+1])` est `sizeof(type)` octets après celle de `&(t[i])`

`&(t[i])` est `i*sizeof(type)` octets après celle de `&(t[0])`

Tableau vu comme un pointeur

Puisqu'on ne peut pas copier globalement un tableau **t**, la solution pour transmettre un tableau à une fonction est de **transmettre l'adresse de son premier élément**.

↪ Cette adresse est un pointeur du type de données du tableau.

Le langage C propose une **arithmétique sur les pointeurs**.
Soit `type *p` un pointeur et `i` un entier naturel, `(p+i)` désigne l'adresse située `i*sizeof(type)` octets après celle de `p`.

Ainsi un pointeur `type *p` peut-être manipulé comme un tableau (sans réservation de mémoire) avec les équivalences suivantes :

`p+i` **est égal à** `&(p[i])`
`*(p+i)` **est égal à** `p[i]`

Autour du nom du tableau

Afin d'alléger l'écriture des appels de fonction, le C donne la possibilité d'utiliser **le nom du tableau** comme un **alias pour l'adresse de sa première case** :

`t` **est donc égal à** `&(t[0])`

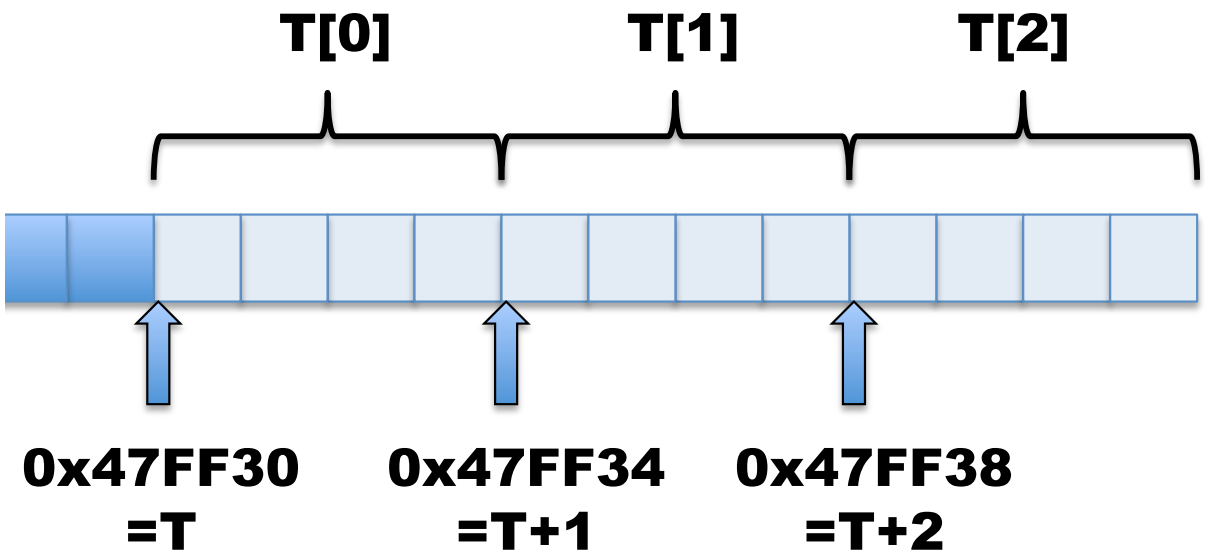
`*t` **est donc égal à** `t[0]`

`*(t+i)` **est donc égal à** `t[i]`

Ainsi un nom de variable tableau statique peut être manipulé comme un pointeur du type de donnée du tableau. Cependant :

- `t` n'est pas modifiable
- `&t` **est égal** à `t` (et donc aussi à `&(t[0])`)
- `sizeof(t)` n'est pas égal à 8 mais à `sizeof(type)*TAILLE`
  (en supposant `t` déclaré par `type t[TAILLE];`)

Vue mémoire



- `T = &T[0] = &T`
- `T+i = &(T[i])`
- `*(T+i) = T[i]`

Tableau comme paramètre d'entrée-sortie

Pour passer un tableau à une fonction :

$=>$ On transmet l'adresse de son premier élément.

Comme on doit éviter les débordements, il faut donner à la fonction appelée un moyen de contrôler que l'accès aux cases du tableau ne dépasse pas sa taille :

$=>$ On transmet en plus la taille du tableau.

Syntaxe d'un appel

```
#define TAILLE n
type t[TAILLE];
f(&(t[0]), TAILLE, ...)
```
ou plus simplement `f(t,TAILLE,...)`

Syntaxe d'une déclaration de fonction

```
type_res f(type tab[], int taille, ...)
```
ou plus simplement
```
type_res f(type *tab, int taille, ...)
```

Exemples

```c
void afficheTab(int *tab, int taille)
{
  for (int i=0;i<taille;i++)
     printf(" %d ",tab[i]);
  printf("\n");
}
```

```c
void afficheTab(int tab[], int taille)
{
  for (int *p=tab;p<tab+taille;p++)
     printf(" %d ",*p);
  printf("\n");
}
```

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

int main() {
    int t[TAILLE] = {1,2,3,4,5,6,7,8,9,10};
    afficheTab(t, TAILLE);
    return 0;
}
```

Exemple : inversion des éléments d'un tableau

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

void inverser(int *t, int taille)
{
   int m = taille / 2;
   int temp;
   for (int i=0;i<m; i++) {
      temp = t[i];
      t[i] = t[taille-1-i];
      t[taille-1-i] = temp;
   }
}

int main() {
   int t[TAILLE] = {1,2,3,4,5,6,7,8,9,10};
   inverser(t,TAILLE);
   afficheTab(t,TAILLE);
   return 0;
}
```

Les tableaux multidimensionnels

Il est possible de déclarer des tableaux de tableaux. On les appelle des tableaux multidimensionnels.

Exemple

```
int t[3][2];
```

t est un tableau à 3 éléments où chaque élément est un tableau de 2 entiers.

On peut voir t comme une matrice : $t = \begin{bmatrix} * & * \\ * & * \\ * & * \end{bmatrix}$

Les tableaux multidimensionnels

L'enchaînement des opérateurs `[ ]` permet de récupérer les éléments d'un tableau multidimensionnel.

Déclaration

Soit la déclaration `int t[m][n];`
`t[i][j]` donne accès au `j-ème` élément du `i-ème` tableau de `t`

Dans une vue matricielle, cela donne :

$$\texttt{int t[3][2]} = \begin{bmatrix} \texttt{t[0][0]} & \texttt{t[0][1]} \\ \texttt{t[1][0]} & \texttt{t[1][1]} \\ \texttt{t[2][0]} & \texttt{t[2][1]} \end{bmatrix}$$

Les tableaux multidimensionnels

Comme pour les tableaux unidimensionnels, il est possible d'initialiser les éléments du tableau lors de la déclaration.

Déclaration

```
int t[3][2] = {{1,2},{3,4},{5,6}};
```

$$\texttt{int t} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Remarque

La déclaration d'un paramètre tableau à 2 dimensions nécessite de préciser la taille de la seconde dimension pour que le compilateur puisse faire correctement ses calculs d'adresse :

```
type_res f(int (*t)[NBCOL], int nbLignes,...)
```

La notion de tableau multidimensionnel s'étend à n dimensions.

Plan

Déclaration de tableaux de longueur variable (VLA)

Définition

Contrairement aux statiques qui ont une taille fixée à la compilation, la taille des VLA (Variable Length Array) n'est connue qu'à l'exécution.

Syntaxe

```
type nom[exp];
```

- `exp` est une **expression entière contenant au moins une variable ou un paramètre** spécifiant la taille du tableau.
- Sa valeur calculée lors de l'exécution devra être positive
- Attention : `exp` évaluable à la compilation $\Rightarrow$ tableau statique

Exemple

`int n = 3; int t[n];` définit t comme un VLA de 3 entiers.
`int t[2+1];` définit t comme un tableau statique de 3 entiers

Limitations des VLA

Ils souffrent des mêmes limitations que les tableaux statiques :

- pas d'affectation globale
- pas de copie possible de leur valeur :
  - ils ne peuvent pas être paramètre d'entrée d'une fonction
  - ils ne peuvent pas être résultat d'une fonction

Il souffrent en plus des limitations suivantes :

- ils ne peuvent pas être initialisés à la déclaration
- ils ne peuvent pas être membre de structure
- la syntaxe d'une fonction prenant en paramètre entrée-sortie un tel tableau multidimensionnel est encore plus complexe :
  - il faut ajouter comme paramètre, avant le tableau, la taille de la seconde dimension :

```
type_res f(int nbCol, int (*t)[nbCol], int nbLignes,...)
```

Plan

Tableau dans une structure

Les membres d'une structure peuvent être des **tableaux statiques**

Syntaxe

```
struct nomStruct {
    type1 champ1[10];
    type2 champ2;
};
```

- `champ1` contient un tableau de 10 `type1`

Attention

La structure ne contient pas que l'adresse du 1er élément du tableau mais bien l'ensemble des éléments du tableau.

Exemple

```
1 struct etudiant {
2   char nom[32];
3   int numero;
4   int naissance[3];
5 };
```

- tous les éléments du tableau sont intégrés dans la structure
  taille de la structure `etudiant` : 48 octets = 32 char + 4 int
- ils sont accessibles par l'accès au champ puis au tableau
  `struct etudiant e; e.naissance[2]=1984;`

On peut imbriquer les initialisations lors de la déclaration :

```
1 struct etudiant e={"Beri",2010003111,{12,4,1987}};
```

Bénéfices du tableau dans une structure

Rappel : une structure peut être globalement copiée (cela correspond à une copie récursive de chacun de ces membres ...)
↪ un tableau dans une structure sera copié élément par élément lors

- de l'affectation de la structure
- du passage en argument d'une fonction
- du retour de la structure par une fonction

Remarque

Intégrer un tableau dans une structure permet donc la copie du tableau !

- On peut alors transmettre à une fonction une copie de sa valeur (ses éléments) ou le récupérer comme résultat d'une fonction !
- Si la taille du tableau est grande, cela a un coût en espace et temps !

Tableau dans les structures : Exemple

```
1  #include <stdio.h>
2  struct tab {
3      int t[3];
4  };
5
6  struct tab reinit(struct tab st) {
7      for(int i=0;i<3;i++) st.t[i] = 0;
8      return st;
9  }
10
11 int main() {
12     struct tab st1 = {{1,2,3}};
13     struct tab st2 = reinit(st1);
14     printf("st1: %d %d %d\n",st1.t[0],st1.t[1],st1.t[2]);
15     printf("st2: %d %d %d\n",st2.t[0],st2.t[1],st2.t[2]);
16     return 0;
17 }
```

Exécution : st1: 1 2 3
st2: 0 0 0

Et un tableau de structures ?

Les structures étant un type de données, on peut les mettre dans un tableau (statique ou VLA) :

Syntaxe

```
struct nomStruct var[exp];
```

déclare un tableau de structures.

Attention

`var` reste un tableau de `struct nomStruct` et contient l'adresse mémoire où se trouvent les `exp` éléments structurés.
↪ Pas d'affectation ni transmission de valeur à une fonction.

Exemple de tableau de structures

```
1 typedef struct date {
2   int    jj, mm, aaaa;
3 } date;
4 ...
5 date agenda[] = {{2,1,1923},{23,12,2004},{13,01,2011}};
```

Questions :

- comment accéder au mois de la 2ième date ?
- agenda[2] peut-il être copié ?
- agenda peut-il être copié ?

Exemple de tableau de structures

```
1 typedef struct date {
2   int    jj, mm, aaaa;
3 } date;
4 ...
5 date agenda[] = {{2,1,1923},{23,12,2004},{13,01,2011}};
```

Questions :

- comment accéder au mois de la 2ième date ? `agenda[1].mm`
- agenda[2] peut-il être copié ?
- agenda peut-il être copié ?

Exemple de tableau de structures

```
1 typedef struct date {
2   int    jj, mm, aaaa;
3 } date;
4 ...
5 date agenda[] = {{2,1,1923},{23,12,2004},{13,01,2011}};
```

Questions :

- comment accéder au mois de la 2ième date ? `agenda[1].mm`
- agenda[2] peut-il être copié ? oui
- agenda peut-il être copié ?

Exemple de tableau de structures

```
1 typedef struct date {
2   int    jj, mm, aaaa;
3 } date;
4 ...
5 date agenda[] = {{2,1,1923},{23,12,2004},{13,01,2011}};
```

Questions :

- comment accéder au mois de la 2ième date ? `agenda[1].mm`
- agenda[2] peut-il être copié ? oui
- agenda peut-il être copié ? non