# Les tableaux en langage d'assemblage

***Introduction***

•Nous avons vu comment sont gérés les variables possédant des types élémentaires : entier signés ou non, caractères

•Un tableau en langage de haut niveau:

-Une liste d'éléments d'un type donné

-Un tableau a une taille fixée à sa déclaration

-Chaque élément est numéroté, on parle d'indice du tableau (les numéros "se suivent")

-Ces indices peuvent varier dans différents intervalles:

• Toujours 0 à NB_ELEMENT –1 en C

•i à i+NB_ELEMENT-1 dans d'autres langages (comme ADA)

•Exemple :

N l'entier 12

T : un tableau sur [0.. N-1] d'entiers

•Stockage et accès aux éléments en assembleur ?



# Tableau à une dimension

•Stockage d'un tableau en mémoire.

-Pour faciliter l'accès aux éléments, ils seront stockés de façon contigue en mémoire.

-Soit *TAILLE_ELEMENT* la taille d'un élément (en octet) et *NB_ELEMENT* le nombre d'éléments du tableau

-La quantité de mémoire nécessaire est *TAILLE_ELEMENT* NB_ELEMENT* (pas toujours vrai , voir plus bas)

- Il faut réserver en mémoire *TAILLE_ELEMENT* NB_ELEMENT* octets (en l'initialisant éventuellement)

-Si le tableau n'est pas initialisé à la déclaration, on peut se servir de la zone ***bss***



# Exemple

*NB_ELEMENT* l'entier 12
*Tab* un tableau de [0.. *NB_ELEMENT*-1] d'entiers
Chaque entier est codé sur 4 octets

•**En langage C :**

```
#define NB_ELEMENT 12
 int Tab [NB_ELEMENT] ;
```

• **En ADA :**

```
Tab: array(0..NB_ELEMENT-1) of INTEGER;
```

•**En assembleur ARM:**

```
.bss
Ad_Tab : .skip 12*4
```

•Pour rendre le programme plus lisible on peut utiliser des constantes en assembleur, on peut ensuite se resservir de ces constantes dans les instructions



# Utilisation de constantes en assembleur

•**Définition d'une constante :**

-Avant la zone bss et data

-La portée : toute la suite du programme (zone data bss et text)

- Syntaxe **.set NOM_CONST, Valeur** ou **NOM_CONST=Valeur**

- Norme usuelle : constante en majuscule

•**Exemple :**

```
.set NB_ELEMENT, 12
.set TAILLE_ELEMENT, 4
.bss
Ad_Tab : .skip TAILLE_ELEMENT* NB_ELEMENT
```

# Exemple 2

*NB_ELEMENT* l'entier 4

*Tabc* un tableau de [7.. 7+NB_ELEMENT-1] de caractères initialisé à {'r', 'i', 'c', 'm'}

(Chaque caractère est codé sur 1 octet)

- **En langage C :**

```
#define NB_ELEMENT 4
char Tabc [NB_ELEMENT]= {'r', 'i', 'c', 'm'} ; /* indice forcément de 0 à 3 */
```

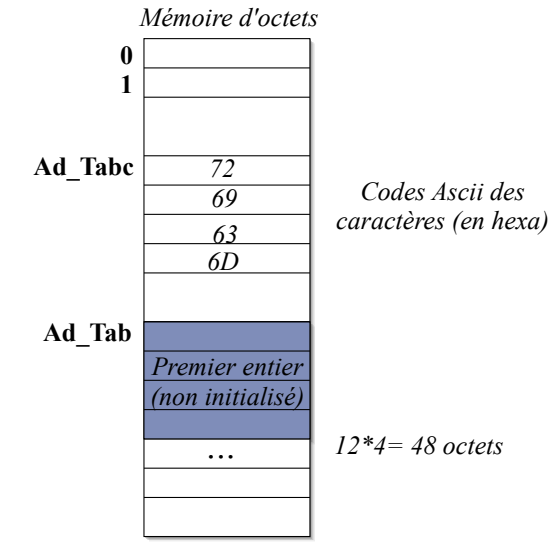
- **En ADA :**

```
Tabc: array(7.. 7+ NB_ELEMENT-1) of CHARACTER= ('r', 'i', 'c', 'm')
```

- **En assembleur ARM:**

```
.set NB_ELEMENT, 4
.set TAILLE_ELEMENT, 1 @ peut servir dans le programme
.data
Ad_Tabc : .byte 'r'
          .byte 'i'
          .byte 'c'
          .byte 'm'
```



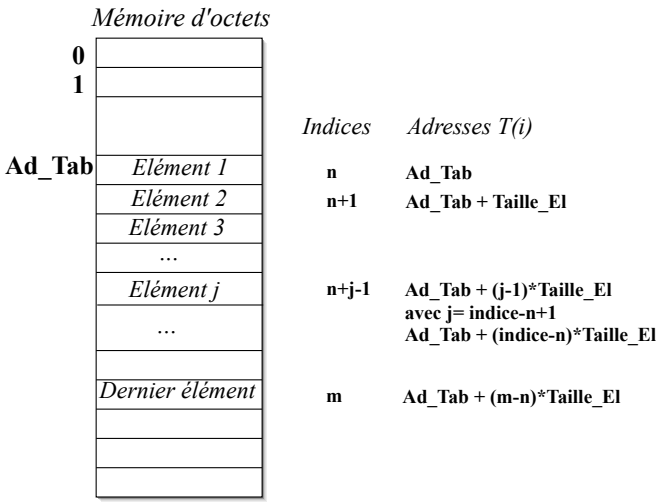





# Accès aux éléments d'un tableau

- Nom du tableau et valeur (ou variable) servant d'indice ;
- En C: T[i], en ADA: T(i)
- Il faut calculer l'adresse de cet élément en mémoire
- **Indice de 0 à NB_ELEMENT -1 :**
  - Adresse de T[i]= Ad_T + i*TAILLE_ELEMENT
- **Indice de 1 à TAILLE :**
  - Adresse de T[i]= ad_T + (i-1)*TAILLE_ELEMENT
- **Indice de n à m avec n > 1 :**
  - NB_ELEMENT = (m-n) +1
  - Adresse de T[i]= ad_T + (i-n)*TAILLE_ELEMENT





# En ARM

•Soit N l'entier 10, Soit un T un tableau sur [0..N-1] d'entiers sur 4 octets

```
.bss
.set N, 10
.set TAILLE_ELT, 4
Ad_T : .skip N*TAILLE_ELT
```

•Comment traduire T[3]= 5 en ARM ?

```
    .text
        ldr r1, Relais_T        @ r1 contient l'adresse
        correspondant à l'étiquette T (début du tableau)
        add r1, r1,#3*TAILLE_ELT @r1 contient l'adresse de T[3],
        attention valeur immédiate max 255
        mov r2, #5
        str r2, [r1]
  Relais_T :  .word Ad_T
```

•**Remarque :** on peut aussi écrire : ldr r1,=Ad_T et remplacer le relais



# OPTIMISATION

**•Particularité du jeu d'instruction du processeur ARM**

**•Très répandue pour optimiser les accès aux tableaux**

- **Mode adressage pre-indéxé** pour str et ldr :

  str r2, [r1, #3*TAILLE_ELEMENT]

  @Mem[r1, #3*TAILLE_ELEMENT ]= r2; T(3)=r2
- Attention r1 n'est pas modifié
- La valeur immédiate est sur 12 bits max
- On peut en cas de besoin aussi faire str r2, [r1, r3]
- On gagne une instruction



# Parcours d'un tableau

•Soit l'algorithme :

N : l'entier 12
T :un tableau sur [0..N-1] d'entiers sur 4 octets
i parcourant 0..N-1 :T [i]←i

On traduit le parcourant en tant que :

```
i←0 ;
Tant que i < N faire
    début
    T[i]←i ; i←i+1 ;
    fin
```



# Parcours en ARM

```
 .set N, 12
.bss
    Ad_T :.skip N*4
.text
                mov r1,#0        @i supposé dans r1, initialisé à 0
    tantque :   cmp r1,#N
                beq fintq
                ldr r0 , =Ad_T
                add r2, r0, r1, LSL#2 @r2= Ad_T+4*i= adresse de T[i]
                str r1, [r2]          @ T[i]=i
                add r1, r1, #1        @i=i+1
                bal tantque
     fintq :
     .ltorg
```

Remarque : On oublie ici les chargement/stockage de i.

# Optimisation 1 d'un parcours de tableau

- **On évite le chargement de l'adresse à chaque tour de boucle**
- Algorithme : T vu comme un tableau d'octet MEM

```
I←0 ;Ad←Ad_T ;
Tant que i < N faire
    MEM[Ad]←i ; i←i+1 ;Ad←Ad+4 ;
```

- **En ARM:**

```
    .set TAILLE_ELT
            mov r1,#0          @i dans r1 initialisé à 0
            ldr r0 , =Ad_T     @r0 (Ad) contient l'adresse de T
tantque : cmp r1,#N
            beq fintq
            str r1, [r0]       @ T[i]=i
            add r0,r0,#TAILLE_ELT      @Ad=Ad+4
            add r1, r1, #1     @i=i+1
            bal tantque
.ltorg
```



# Optimisation 2

- Mode d'adressage de l'instruction str particulier :
- **Adressage post-incrémenté:**
  - **str regj, [regi], #valeur-immediate**
- **Exemple:**

  str r1, [r0], #TAILLE_ELT @ additionne TAILLE_ELT à r0 après le stockage en mémoire
- équivalent à

  str r1,[r0]

  add r0, r0, #TAILLE_ELT



# Structure

**Exemple:**
nom_struct : le type < champ1 : un entier, champ2 : un caractère, champ3 : un entier>
Nomvar : un nom_struct

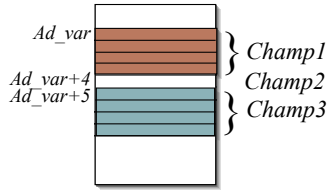
**En C** :

```
struct nom_struct
    {
    int champ1 ;
    char champ2 ;
    int champ3 ;
        }
    struct nom_struct nom_var ;
```



# Structure en assembleur

- Ad_var: .skip 9
- Accès aux champ1 (par des ldrb/strb) à l'adresse de la variable *nom_var: Ad_var*
- Accès aux champ2 à l'adresse *Ad_var* + 4
- Accès aux champ3 à l'adresse *Ad_var* + 5
- **Problème de l'alignement :**
- *Ad_var* et *Ad_var* +5 ne peuvent pas être multiples de 4 si les champs sont stockés de façon contiguë en mémoire

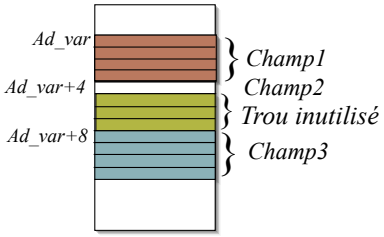


## Problème d'alignement dans une structure

•Solutions au problème d'alignement

-Changer l'ordre de stockage en mémoire des différents champs

-Laisser des "trous" pour garder le bon alignement du champ suivant






## Directive d'alignement d'adresse

•**.balign x** l'étiquette (ou la réservation) suivante a une adresse mulitple de x. (**.align x** adresse suivante multiple de $2^x$)

•Exemple:

```
Ad_var :  .skip 5
          .balign 4
          .skip 4
```

•Il y a un trou de 3 octets

•On peut aussi faire:

Ad_var: .skip 12

•Il faudra se rappeler dans le programme de l'organisation de la structure en mémoire



## Accès aux champs d'une structure

- On définit les déplacements (par rapport à l'adresse de la variable) pour accéder à chaque champ (Sans oublier les trous éventuels)
- On additionne le bon déplacement à l'adresse avant le chargement/stockage de la variable
- Exemple : nom_var.champ2='a' ;

```
.set DELTA_CHAMP1, 0
.set DELTA_CHAMP2, 4          @déplacement pour accéder au champ2
.set DELTA_CHAMP3, 8

ldr r1, =nom_var
mov r2, 'a'                   @ou #0x65
add r1,r1,# DELTA_CHAMP2      @r1 contient l'adresse du champ1
str r2,[r1]
```

- On peut aussi utiliser le mode d'adressage pré-indexé du ARM :

```
str r2, [r1,#DELTA_CHAMP2]
```



## Tableau de structure

•Pour les tableaux, entre les éléments structures le problème de l'alignement peut se poser aussi

•Exemple

```
struct nom_struct
      {
      int champ1 ;
      char champ2 ;
       }
      struct nom_struct nom_var ;
struct nom_struct tab[TAILLE] ;
```

•Il faudra laisser un trou de 3 octets entre chaque élément du tableau

```
.set TAILLE, 4
.set TAILLE_ELT, 8   @ 5 +3 pour faire un multiple de 4
Ad_Tab :.skip TAILLE*TAILLE_ELT
```

# Accès au champ d'une structure élément d'un tableau

- Il faut calculer l'adresse de l'élément puis ajouter le déplacement du champ voulu
- Exemple:
  tab[3].champ2='a'

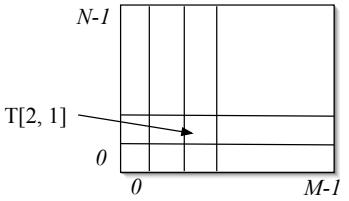
```
.set TAILLE, 4
.set TAILLE_ELT, 8                @ 5 +3 pour faire un multiple de 4
.set DELTA_CHAMP1, 0
.set DELTA_CHAMP2, 4          @ déplacements pour l'accès au champs
Ad_Tab : .skip TAILLE*TAILLE_ELT

      mov r1,#3          @indice 3 dans r1
      ldr r0 , =Ad_Tab                  @r0 contient l'adresse de T
      mov r2, 'a'
      add r0, r0, r1, LSL#3          @r0 = Ad_Tab+ 8*3 = adresse de T[3]
      str r2, [r0,#DELTA_CHAMP2]
```



# Tableaux à 2 dimensions

- Exemple un tableau T sur [0..M-1]*[0..N-1] d'éléments de taille TAILLE_ELT
- On peut le voir comme une matrice
- Il faut ranger les éléments de ce tableau en mémoire (tableau à une dimension)






# Tableaux à 2 dimensions

- Deux façons « simples » :
  - Soit par « lignes » (on fait d'abord évoluer i)
  - Soit par colonnes (on fait d'abord évoluer j) (voir figure)
- Calcul de l'indice en mémoire (déplacement par rapport à l'adresse de début du tableau) à partir des deux indices dans la matrice
- Taille nécessaire en mémoire identique : TAILLE_ELT*M*N

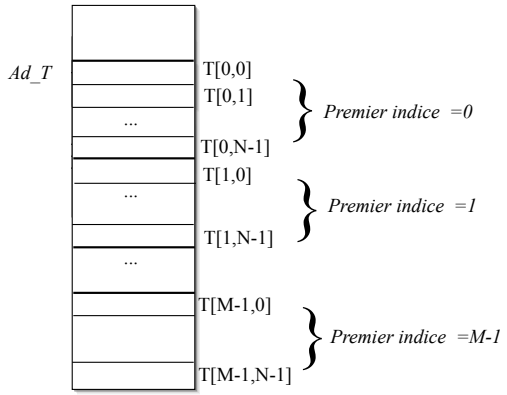
```
.set N, 12
.set M, 8
.set TAILLE_ELT, 4

.bss
Ad_T :.skip TAILLE_ELT*M*N
```



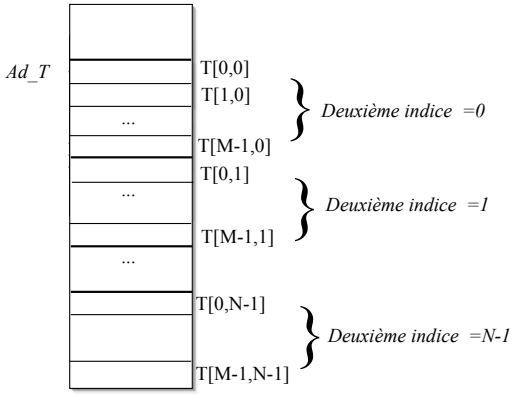
# Rangement en mémoire

- Rangement en faisant évoluer le deuxième indice (le plus à droite) d'abord



# Rangement en mémoire

•Rangement en faisant évoluer le premier indice (le plus à gauche) d'abord






# Accès aux éléments d'un tableau à 2 dimensions

•Calculs différents suivant la convention de stockage (à définir au départ):

•Evolution du premier indice d'abord:

- -Adresse de T[i,j] :
- -Ad_T + i * TAILLE_ELT + M* j * TAILLE_ELT = Ad_T + TAILLE_ELT * (i+ M*j)
- -M*TAILLE_ELT: taille d'une "ligne"

•Evolution du deuxième indice d'abord:

- -adresse de T[i,j] :
- -Ad_T + i * N* TAILLE_ELT + j* TAILLE_ELT = Ad_T + TAILLE_ELT * (i*N+j)
- -N*TAILLE_ELT : taille d'une "colonne"