

# Analyse et programmation langage ADA



## Informatique 1<sup>ère</sup> année

**Utilisation en mode diaporama**

**La page suivante donne le sommaire.**

**Pour accéder à un chapitre cliquer sur le lien correspondant**

**De n'importe quel transparent, on revient au sommaire en appuyant sur le logo EISTI**

# Tableaux

- Introduction au type tableau
- Attributs First, Last, Length et Range
- Contrainte d'indice
- Présentation d'un type tableau
- Initialisation
- Accès à un élément particulier du tableau
- Présentation de types spécifiques
- Agrégats
- Opérations sur les tableaux
- Fixation de la taille du tableau à l'exécution
- Passage et retour de paramètres de type tableau

# Introduction au type tableau.

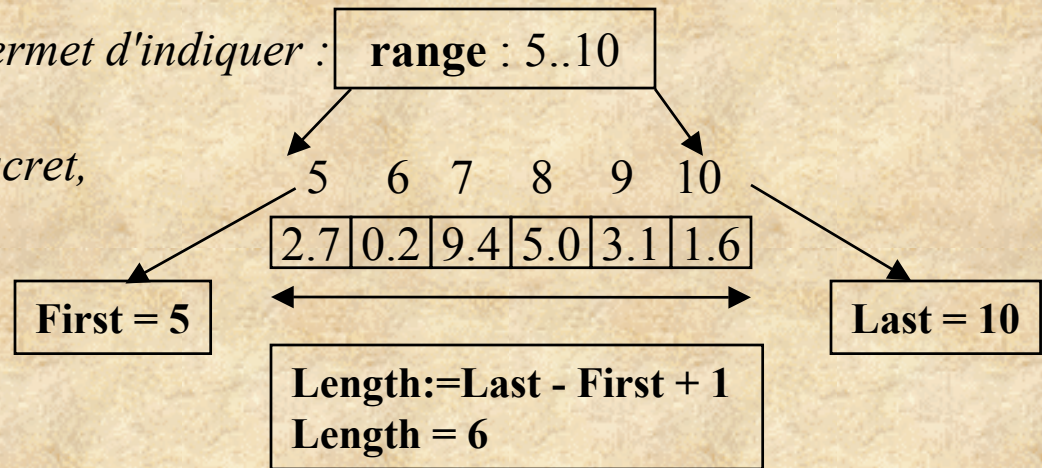
## Attributs First, Last, Length et Range

- *Un tableau est un ensemble de composantes de même type dont chacune est repérée par un ou plusieurs indices.*

La spécification d'un type tableau permet d'indiquer :

- le nombre d'indices  $N$ ,
- le type des indices qui doit être Discret,
- le type des éléments.

dimension  $N:=1$   
 indice entier de 5 à 10  
 contenant 6 réels



- Les attributs *First*, *Last*, *Length*, et *Range* sont applicables aux indices d'un tableau T

T'First(N)      donne la première valeur d'indice de la dimension N du tableau T

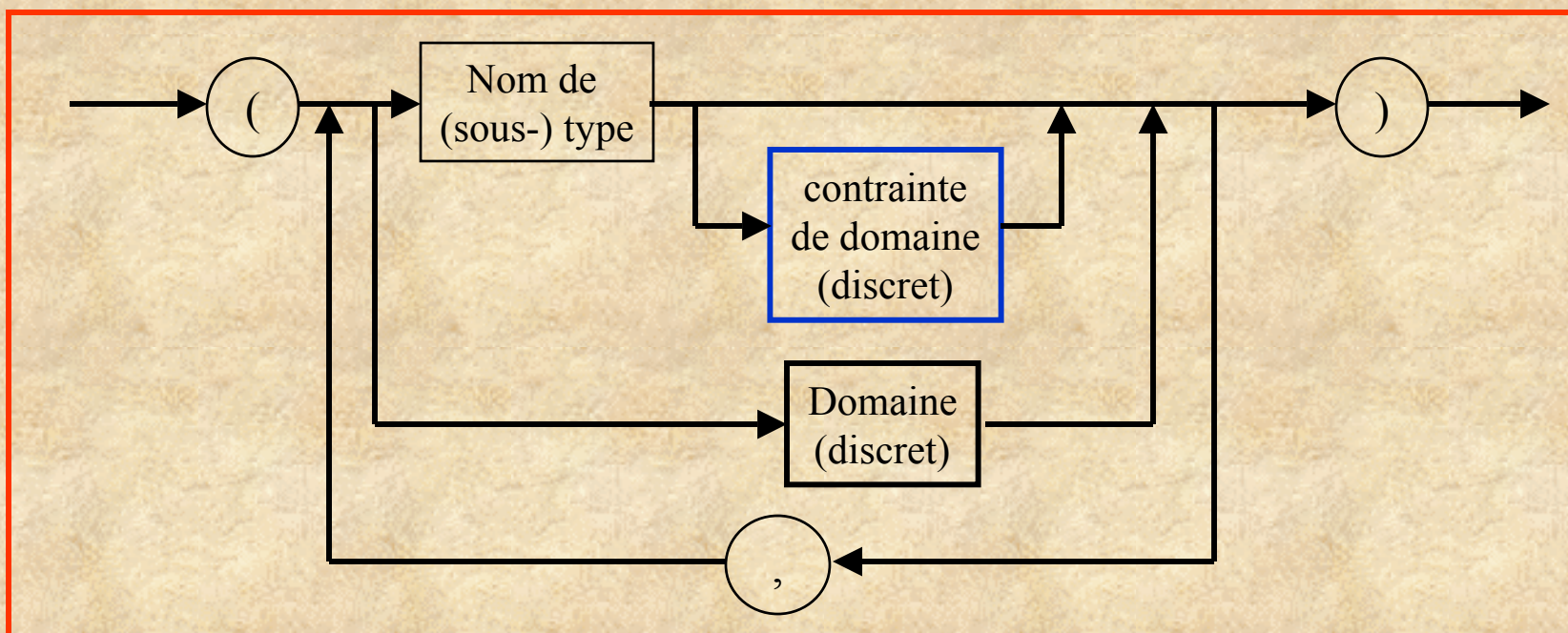
T'Last(N)      donne la dernière valeur d'indice de la dimension N du tableau T

T'Length(N)    donne la longueur de la dimension N du tableau T

T'Range(N)    représente l'intervalle de la dimension N du tableau T

## Contrainte d'indice

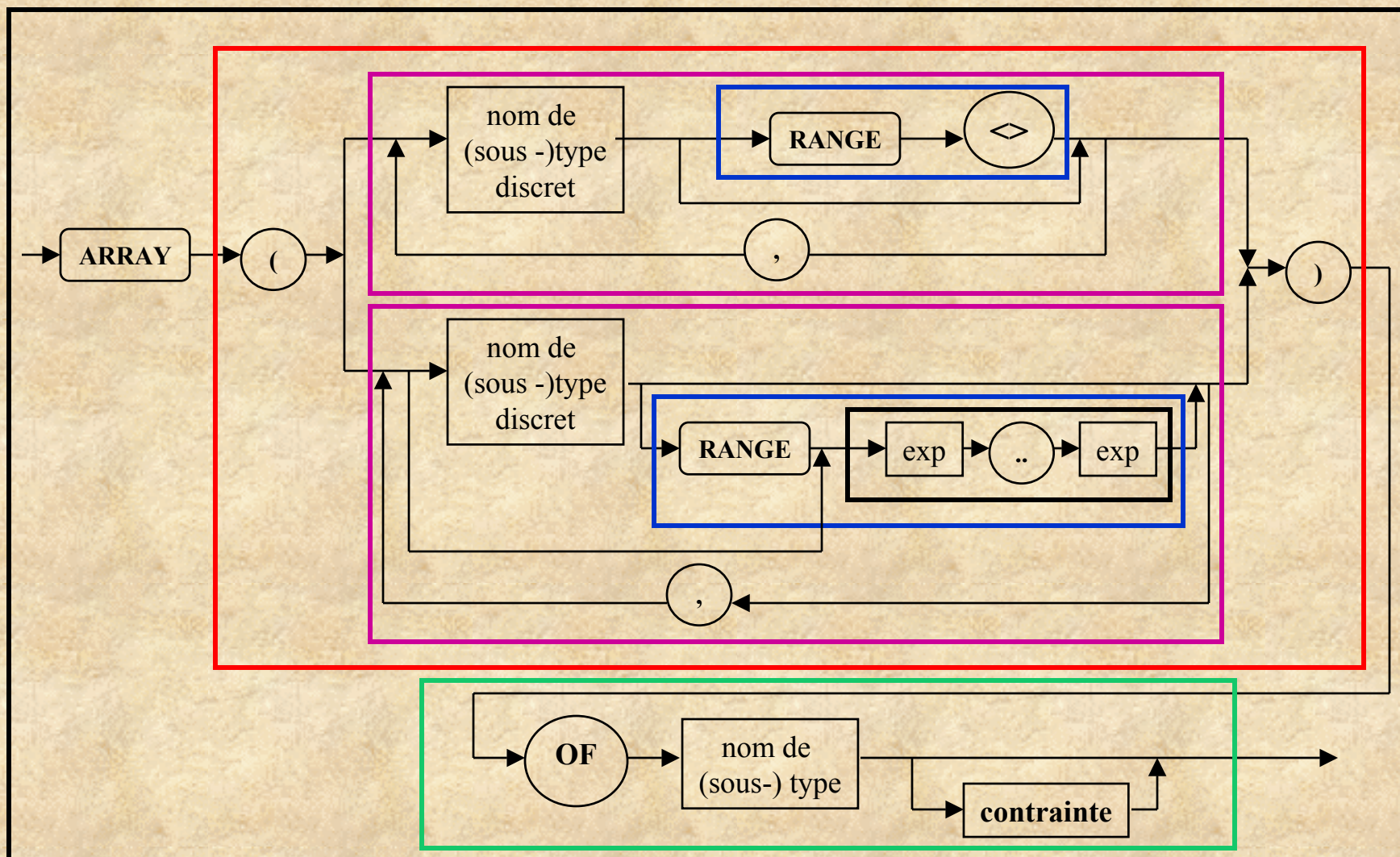
- Une *contrainte d'indice* est définie par le diagramme syntaxique suivant :



- Exemple :
- (Integer, Integer) permet de définir un tableau à 2 indices entiers
  - (Integer **range** -1..1, Integer **range** 1..3) permet de définir une matrice à 9 composantes
  - (1..5, 1..5) permet de définir une matrice carré à 25 composantes

# Présentation d'un objet tableau

- Soit le diagramme syntaxique suivant, définissant un objet tableau à n dimensions de type quelconque.





## Présentation d'un objet tableau

- Le type d'indice doit être discret alors que le type des éléments peut être n'importe lequel sauf un type (ou sous type) : un tableau non contraint ou un article à discriminants sans valeurs par défaut

➤ **array (Integer range <>) of Float** ⇔ **array (Integer) of Float**

matrice d'entiers, bornes des indices indéfinies

➤ **array (Integer range <>, Integer range <>) of Integer**

➤ **array (Natural range 1..100) of Float** *vecteur de cent composantes réelles*

➤ **array (1..100) of Float** *idem*

Idem, mais contrainte sur les composantes

➤ **array (1..100) of Float delta 0.1 range -10.0..10.0**

➤ **array (Boolean) of Character** (*vecteur à indice, False ou True*) de caractères

➤ **array (Boolean, Integer range 1..3) of Float** *matrice avec six composantes réelles*

➤ **array (Natural range <>) of Character range 'A'..'Z'** *un tableau caractères*

# Initialisation d'un tableau

## Notation algorithmique

**T : tableau sur [bi..bs] de typeéléments**

**Exemples :**

**T : tableau sur [1..10] d'entiers = [0,-1, 0, 1, 2, 3, 4, 4, 4, 4]**

**P : tableau sur [1..2,1..3] d'entiers = [ [1,2,3],  
[4,5,6] ]**

## ADA

**T : array(bi..bs) of typeelements;**

**Exemples :**

**T : array(1..10) of Integer := (0,-1,0,1,2,3,others => 4);**

**P : array (1..2,1..3) of Integer := ((1,2,3),(4,5,6));**

## Accès à un élément particulier du tableau

### Notation algorithmique

Désignation d'un élément de tableau :

$T[i]$  désigne l'élément d'indice  $i$  du tableau  $T$ .

### ADA

**$T(i)$  désigne l'élément d'indice  $i$  du tableau  $T$ .**

**Remarque :**

```
T : array(1..10) of Integer;
```

```
begin
```

```
    T(12) := 3;    -- génère une erreur, dépacement d'indice  
                  -- un code plus sûr
```





## Présentation d'un type tableau

- Les types composés : les tableaux

```
type T_Fruits is (Orange, Banane, Mandarine, Kiwi);
```

### Objet tableau

```
Mon_Tableau : array (1 .. 5) of T_Fruits;
```

```
-- L'objet Mon_Tableau est de type anonyme
```

### Type tableau

```
type T_Tab_Fruits is array (1 .. 5) of T_Fruits;
```

```
Mon_Tableau : T_Tab_Fruits ;
```

```
Un_Tableau   : T_Tab_Fruits ;
```

### Initialisation

```
Mon_Tableau := (1|3|5 => Orange, others => Banane);
```

```
Mon_Tableau := (1..3 => Orange, others => Banane);
```



## Présentation d'un type tableau

- Les types composés : les tableaux

```
type T_Fruits is (Orange, Banane, Mandarine, Kiwi);
```

### Objet tableau

```
Mon_Tableau : array (in T_Fruit) of Natural;
```

### Type tableau

```
type T_Tab_Fruits is array (in T_Fruit) of Natural;
```

```
Mon_Tableau : T_Tab_Fruits ;
```

```
Un_Tableau   : T_Tab_Fruits ;
```

### Initialisation

```
Mon_Tableau := (Orange|Kiwi => 2, others => 3);
```

```
Mon_Tableau := (Orange..Mandarine => 1, others => 2);
```



## Tableau non contraint

### Présentation de types spécifiques

Pour déclarer un nouveau *type spécifique* en utilisant *le type* de base *array*  
Un seul type non contraint, les variables déclarées sont obligatoirement contraintes

```
type T_Vecteur is array(Integer range <>) of Float ; -- tableau non contraint  
-- <> se dit "boite"
```

Il est possible de déclarer un *sous-type* d'un type (de base) tableau non contraint. La forme générale d'une telle déclaration est :

```
subtype identificateur is id_type_tableau (Intervalle_1.. Intervalle_N) ;
```

- identificateur est le nom du sous-type
- id\_type\_tableau est le nom d'un type tableau non contraint
- intervalle\_i est un intervalle fixant les bornes

```
subtype Vecteur_5D is T_Vecteur(1..5); -- tableau de réels à 5 composantes.
```

# Agrégats

Un *agrégat* est la forme littérale d'une valeur de type tableau (un élément de l'ensemble de valeurs du type tableau). Il existe deux formes d'agrégats qui ne doivent pas être mélangées :

- **l'agrégat positionnel** où les valeurs sont donnés dans l'ordre où elles seront stockées dans la variable de type tableau. Les valeurs sont séparées par des virgules.

```
v : Vecteur_5D := (1.0, 2.0, 3.0, 4.0, 5.0);
```

- **l'agrégat par nom** où chaque valeur est précédée de la valeur de l'indice et du symbole =>, ainsi l'ordre des indices n'a plus à être respecté.

```
v : Vecteur_5D := (1 => 1.0, 2 => 2.0, 4 => 4.0, 5 => 5.0, 3 => 3.0);
```

Remarque : Pas d'agrégat mixte, seule la clause "others" est autorisée

## Agrégats

Les règles de formation des agrégats ressemblent à celles utilisées pour l'instruction case. Chaque valeur peut être donnée pour un seul indice ou un intervalle ou plusieurs indices donnés explicitement.

**La clause others** permet de donner une valeur à tous les éléments non spécifiés mais doit toujours être placée à la fin de l'agrégat :

```
w : Vecteur(1..1000) := Vecteur'(1 => 1.0,  
                                2 | 4 =>2.0,  
                                3 => 3.0,  
                                7..15 => 5.0,  
                                others => 0.5);
```

```
m3 : Mat_3_3 := (1=> (1=>1.0, others=>0.0),  
                2=> (2=>1.0, others=>0.0),  
                3=> (3=>1.0, others=>0.0)); -- c'est une matrice unitaire
```

```
v : Vecteur(1 .. 15) := (1.0, 2.0, 3.0, 4.0, 5.0, others => 0.0);
```

La déclaration multiple suivie d'une affectation provoque l'initialisation des deux tableaux :  
v1, v2 : Vecteur\_5D := (others => 1.0); initialise toutes les valeurs de v1 et de v2 à 1.0. 13

# Opérations sur les tableaux

- Affectation
- Tranches de tableaux
- Concaténation
- Comparaison (égalité)
- Comparaison
- Opérations booléennes
- Conversion

## Affectation

- Pour pouvoir affecter un tableau à un autre il est nécessaire qu'ils soient de même type, et de même taille pour chaque dimension, sans avoir forcément les mêmes indices, faute de quoi l'exception **Constraint\_Error** sera levée .
- Quand les premiers indices ne sont pas identiques, on parle de glissement

### *Exemples :*

T : **array**(1..5) **of** Integer := (0,-1,0,1,2);

P : **array**(6..10) **of** Integer;

L : **array**(6..11) **of** Integer;

P := T; est correct;

L := T; incorrect; --levée d'une exception, **Constraint\_Error**

**type** Vecteur **is** **array**(Integer **range** <>) **of** Integer;

V : Vecteur (1..5) := (0,-1,0,1,2);

W : Vecteur (0..4);

W := V;

## Tranches de tableaux

Une *tranche* (slice) de tableau à une dimension est un *tableau partiel* défini par un intervalle compris dans l'intervalle de définition des indices du tableau d'origine.

### *Exemple :*

```
T : array(1..8) of Integer := (2,-1,0,1,2,5,3,2);  
P := T(3..5);  -- contenu de P est de (0,1,2)  
P'First = 3;   -- on garde les indices du tableau initial  
P'Last = 5;
```

### *Remarque :*

Il ne faut pas confondre  $P(0)$  et  $P(0..0)$

$P(0)$  : est l'élément de la tranche  $P$  à la position d'indice 0

$P(0..0)$  : est une tranche, donc un tableau à un seul élément dont l'indice est 0





## Concaténation

- L'opération de *concaténation* consiste à mettre bout à bout des tableaux ou tranches de tableaux de même type. Cet opérateur est noté `&`, et possède la priorité des opérateurs binaire (`+` et `-`).
- Le tableau résultat doit avoir autant d'éléments que la somme des élément des deux tableaux à concaténer.

*Exemple :*

```
type Vecteur is array( Integer) of Float; -- type non contraint
```

```
W : Vecteur (-10..10) := (others => 0.0);
```

```
X : Vecteur (1..10) := (1.5, 2.5, 3.5, others => 4.0);
```

```
V1 : Vecteur(1..10) := W (1..5) & X (1..5); -- utilise les tranches de tableaux
```

```
V2 : Vecteur := W & X; -- V2 aura la borne inférieure de W et 31 éléments
```

## Comparaison (Égalité)

- *Égalité, inégalité* =, *≠* : il est possible de comparer des tableaux de même type et de même longueur pour chaque dimension, sans avoir forcément les mêmes indices.
- Les tests d'égalité suivent les mêmes règles de glissement que l'affectation sauf que si les tableaux n'ont pas la même longueur, la valeur retournée sera *false*.
- Les deux tableaux seront égaux s'ils possèdent le même nombre d'éléments et que ceux-ci soient égaux (en respectant l'ordre des éléments).

### *Exemple :*

$(1, 2) = (1, 3, 5, 7)$	expression fausse; -- <i>pas la même longueur</i>
$(2, 1) \neq (1, 3, 5, 7)$	expression fausse; -- <i>pas la même longueur</i>
$(2, 1) = (2, 1)$	expression vraie;
$(2, 0) = (2, 1)$	expression fausse;

### *Remarque :*

U,V **is array**(1..5) **of** Float := ( **others** => 0.0); -- *type anonyme*  
**if** U = V **then** .....(provoque une erreur car U et V sont de type différent)



## Comparaison

- Opérateurs de *comparaison*  $<$ ,  $\leq$ ,  $>$ ,  $\geq$  : s'appliquent uniquement à des tableaux à une dimension et contenant des éléments de type *discret*.
- Durant l'opération de *comparaison*, les éléments sont comparés un à un jusqu'à épuisement de l'un des tableaux ou jusqu'à ce que l'une des comparaisons permette de répondre à la question.

### **Exemple :**

$(1, 2) < (1, 2, 3, 4)$

expression vraie; -- *longueur 1 < longueur 2*

$(1, 2) > (1, 2, 3, 4)$

expression fausse;

$(1, 2) < (1, 3, 5, 7)$

expression vraie; --  $1 < 2$  et  $2 < 3 + L1 < L2$

$(2, 1) < (1, 3, 5, 7)$

expression fausse; --  $2 > 1$

$(2, 1) > (1, 3, 5, 7)$

expression vraie; --  $2 > 1$

$(2, 1) > (2, 1)$

expression fausse;

$(2, 1) \geq (2, 1)$

expression vraie; --  $2 = 2$

$(2, 0) \geq (2, 1)$

expression vraie; --  $2 = 2$



## Opérations booléennes

Les opérateurs logiques *and*, *or*, *xor* et *not* s'appliquent aux tableaux unidimensionnels de même longueur dont le type des éléments est le type *Boolean*.

L'application d'un opérateur sur un ou des tableaux se fait élément par élément de telle manière à générer un tableau résultat de même longueur que le ou les tableaux initiaux.

### *Exemple :*

<b>not</b> ( False , True, True)	résultat (True, False, False)
( True , True) <b>and</b> ( False, True)	résultat (False, True)
( True , True) <b>or</b> ( False, True)	résultat (True, True)
( True , True) <b>xor</b> ( False, True)	résultat (True, False)

**Remarque :** Les bornes du tableau résultat sont celles de l'unique opérande ou de l'opérande de gauche



## Conversion entre types tableaux

- Il faut que :
- les types aient le même nombre de dimensions;
  - les types d'indices sont identiques ou convertibles entre eux;
  - les types des éléments sont identiques.

```
type T_Vecteur is array (Integer range <>) of Float;
```

```
subtype T_Vecteur_10 is T_Vecteur (1..10);
```

```
type T_Autre_Vecteur is array (Natural range <>) of Float;
```

```
Vecteur : T_Autre_Vecteur (0..100);
```

```
Vecteur_Test : T_Vecteur (-100..100);
```

T_Vecteur ( Vecteur )	bornes de Vecteur, 0 et 100;
T_Vecteur ( Vecteur(20..30) )	bornes de la tranche, 20 et 30;
T_Vecteur_10 ( Vecteur(20..29) )	bornes de T_Vecteur_10, 1 et 10;

```
T_Autre_Vecteur ( Vecteur_Test )  
provoque Constraint_Error à l'exécution (-100..0 hors de Natural);
```

```
T_Vecteur_10 ( Vecteur(20..30) )  
provoque Constraint_Error à l'exécution (longueurs différentes).
```



## Fixation de la taille du tableau à l'exécution

```
with ada.Text_IO; with ada.Integer_Text_IO; use ada.Text_IO; use ada.Integer_Text_IO;
procedure VecteurSomme is
    N : Integer ;                                -- dimension du tableau
    type Vecteur is array (Integer range<>) of Float; -- nouveau type vecteur
begin --VecteurSomme
    Put(" Entrer la dimension des vecteurs : "); -- Saisie de la dimension des vecteurs
    Get(N);
    Skip_Line;
    Somme : declare
        Vect_Somme, Vecteur_1, Vecteur_2 : Vecteur(1..N); -- variables de type vecteur
    begin
        Put_Line(" Saisie des composantes de vecteur_1 et vecteur_2"); -- Saisie des vecteurs
        for I in 1..N loop
            Get(Vecteur_1(I)); Get(Vecteur_2(I));
            Skip_line;
        end loop;
        for I in Vect_Somme'First .. Vect_Somme'Last loop -- Vect_Somme'Range
            Vect_Somme(I):=Vecteur_1(I)+ Vecteur_2(I);
        end loop;
    end Somme ;
end VecteurSomme;
```

# Passage et retour de paramètres de type tableau

**procedure** Transformation **is**

```

type T_Vect is array (1..3) of Float ;
type T_Mat is array (1..3;1..3) of Float ;
  
```

```

Mrot, Mtrans, Mhom : T_Mat := ((1.0,0.0,0.0),
                               (0.0,1.0,0.0),
                               (0.0,0.0,1.0));
  
```

```

V, Vr : T_Vect := (others => 0.0);
  
```

**begin**

```

Vr := "*" (Mrot, V);
AfficherVect(Vr);
  
```

```

Vr := Mhom * V;
AfficherVect(Vr);
  
```

**end** Transformation ;

Paramètres formels

Partie déclarative ou spécification

```

function "*" (M : in T_Mat ;
              Ve : in T_Vect) return T_Vect ;
  
```

```

function "*" (M : in T_Mat ;
              Ve : in T_Vect) return T_Vect is
Vs : T_Vect := (others => 0.0);
begin
  for i in M'range(1) loop
    for j in M'range(2) loop
      Vs(i) := Vs(i)+M(i,j)*Ve(j) ;
    end loop;
  end loop;
  return Vs;
end "*" ;
  
```

Paramètres effectifs

Partie non visible