Vettori e matrici (array)

Introduction to modern Fortran

Tommaso Gorni
t.gorni@cineca.it

COSA VIENE PRESENTATO

- Classificazione degli array
- Dichiarare un array
- Inizializzare un array
- La notazione vettoriale
- Qualche avvertimento

ARRAY - CLASSIFICAZIONE

Gli **array** sono serie di **variabili** dello **stesso tipo**, salvate in maniera contigua in memoria, ciascuna accessibile mediante indici numerici.

Il Fortran consente di gestire array multidimensionali (fino a 7 dimensioni). Gli array in Fortran90 possono essere di 3 tipi:

- **Array statici:** hanno dimensioni fissate al momento della dichiarazione e il numero di elementi nell'array non può essere modificato durante l'esecuzione del programma.

- **Array automatici:** all'interno di una procedura non hanno dimensioni fissate, ma esse vengono definite con il passaggio degli argomenti alla procedura.

- **Array dinamici:** la dimensione di questi array è calcolata durante l'esecuzione del programma. Sono anche detti array **allocabili**.

ARRAY - TERMINOLOGIA

Rango	numero di dimensioni della matrice
Estensione	numero di elementi in una singola dimensione
Forma	vettore delle estensioni
Dimensione	prodotto delle estensioni
Conformità	due matrici con la stessa forma

Per dichiarare un array è necessario fornire 3 informazioni:

- *Tipo*
- *Rango*
- *Forma*

```
INTEGER, DIMENSION(40,60) :: A
REAL, DIMENSION(10000) :: B
```

ARRAY - DICHIARAZIONE

La dichiarazione di un array è identificata dall'attributo DIMENSION secondo la sintassi generale:

```
<TYPE>, DIMENSION([x1:]x2,[y1:]y2,...) :: array
```

Ogni dimensione è separata da virgole e i limiti degli indici per ogni dimensione sono separati da ":"

Se come nell'esempio che segue non è specificato l'indice inferiore, **si assume che l'array parta dall'indice 1.**

```
REAL, DIMENSION(40,60) :: A
INTEGER, DIMENSION(0:2,0:40) :: B
```

ARRAY - DICHIARAZIONE

Dichiarazione compatta: per dichiarare le dimensioni dell'array si può anche non utilizzare l'attributo `DIMENSION`, bensì la forma:

```
<TYPE> :: A([x1:]x2,[y1:]y2,...)
```

Sono pertanto **equivalenti** le forme:

```
INTEGER, DIMENSION(4,2) :: A, B, C
INTEGER :: A(4,2), B(4,2), C(4,2)
```

ARRAY STATICI E ARRAY AUTOMATICI

Uso di PARAMETER: in generale possono essere utili uno o più PARAMETER per dichiarare le dimensioni degli array statici.

```
INTEGER, PARAMETER :: n=10
REAL, DIMENSION(n,n+1) :: A,B,C
```

Nelle **procedure** è possibile utilizzare argomenti di tipo INTEGER per dichiarare le dimensioni di array passati ad una subroutine e array automatici.

```
SUBROUTINE compute(n, a, b, c)
   INTEGER :: n
   REAL, DIMENSION(n,2*n) :: a,b,c     ! Arrays passed by reference
   REAL, DIMENSION(2*n) :: temp        ! Automatic array
```

ARRAY A DIMENSIONI PRESUNTE

Dimensioni presunte (*assumed-shape*): nel caso di array passati in argomento a procedure, è possibile evitare di dichiararne le dimensioni, esplicitando **solo il rango**.

Si usa pertanto l'attributo `DIMENSION` (o la forma compatta), ma i valori vengono sostituiti dai due punti `:`

```
SUBROUTINE compute(n, a, b, c)
   INTEGER :: n
   REAL, DIMENSION(n,2*n) :: a, b      ! Arrays passed by reference
   REAL, DIMENSION(:,:) :: c           ! Assumed-shape array
   REAL, DIMENSION(2*n) :: temp        ! Automatic array
```

ARRAY - INIZIALIZZAZIONE

1. Assegnazione diretta

```
REAL, DIMENSION(3) :: a
a(1) = 0.0; a(2) = 1.0; a(3) = 2.0
```

2. Costrutti DO

```
REAL, DIMENSION(3) :: a
DO i = 1, 3
   a(i) = REAL(i)
END DO
```

ARRAY - INIZIALIZZAZIONE

3. Notazione vettoriale

```
REAL, DIMENSION(3) :: a = 0.0
A(2:5) = 0.1                          !< Accesso a sezioni di array
M(2:4,2:4) = 2.0                      !<
```

4. *Array constructor* (solo per array di rango 1)

```
INTEGER, DIMENSION(5) :: A = (/4,4,2,1,2/)   ! Old (still valid)
INTEGER, DIMENSION(5) :: A = [4,4,2,1,2]     ! Since Fortran2003
```

ARRAY - INIZIALIZZAZIONE

Esempio 1: array dentro *array constructor*

```
INTEGER :: VECTOR_X(3) = [4, 1, 5]
INTEGER :: A(6)
A(1:6) = [3, 1, VECTOR_X, 9]
```

Esempio 2: *do-loop* implicito (anche in fase di dichiarazione)

```
INTEGER :: i  !< Some compilers need this
INTEGER, DIMENSION(6) :: A = [ (i, i=1,6) ]
```

ARRAY - INIZIALIZZAZIONE

Esempio 3: indicizzazione (solo array di rango 1):

```
INTEGER, DIMENSION(6) :: ip = [5, 2, 1, 3, 6, 4]
REAL, DIMENSION(6) :: a
a(ip) = [5., 1., 3., 4., 9., 1.]

WRITE(*,'(6F5.1)') a     ! OUTPUT ?
```

ARRAY - INIZIALIZZAZIONE

Esempio 3: indicizzazione (solo array di rango 1):

```
INTEGER, DIMENSION(6) :: ip = [5, 2, 1, 3, 6, 4]
REAL, DIMENSION(6) :: a
a(ip) = [5., 1., 3., 4., 9., 1.]

WRITE(*,'(6F5.1)') a     ! OUTPUT ?

3.0  1.0  4.0  1.0  5.0  9.0
```

ARRAY - INIZIALIZZAZIONE

Esempio 3: indicizzazione (solo array di rango 1):

```
INTEGER, DIMENSION(6) :: ip = [5, 2, 1, 3, 6, 4]
REAL, DIMENSION(6) :: a
a(ip) = [5., 1., 3., 4., 9., 1.]

WRITE(*,'(6F5.1)') a     ! OUTPUT ?

3.0  1.0  4.0  1.0  5.0  9.0
```

NB: Attenzione che non si ripetano due indici nell'array `ip`: il compilatore non segnala errore.

ARRAY - INIZIALIZZAZIONE

Per array di rango > 1, *l'array constructor* non funziona, ma si può utilizzare una appropriata funzione intrinseca:

```
RESHAPE(SOURCE,SHAPE)
```

Che permette di rimodellare un vettore (SOURCE) secondo una qualsiasi possibile forma SHAPE.

```
SOURCE = [ 1, 2, 3, 4, 5, 6 ]
Y = RESHAPE (SOURCE ,  [ 3, 2 ] )
```

```
      1  4
Y =   2  5
      3  6
```

NOTAZIONE VETTORIALE

Il Fortran90 permette di lavorare con vettori e matrici considerandoli nella loro **globalità**, a differenza del Fortran77 che obbliga a lavorare elemento per elemento.

Vettori e matrici devono essere **conformi** per poter essere coinvolti in **operazioni vettoriali**.

E' essenziale ricordare che per calcolare il risultato di un'assegnazione vettori e matrici sono **valutati prima che l'assegnazione abbia luogo**, elemento per elemento. Ovvero si lavora sempre per elementi, con un ciclo DO implicito.

Le **funzioni intrinseche** possono avere **risultato vettoriale**.

NOTAZIONE VETTORIALE

Esempi:

```
REAL, DIMENSION(10,10) :: a, b, c
```

Queste 3 forme sono equivalenti

```
a = 0.0
a(:,:) = 0.0
a(1:10, 1:10) = 0.0
```

Queste 3 forme sono equivalenti

```
c = a * b
c(:,:) = a(:,:) * b(:,:)
c(1:10, 1:10) = a(1:10, 1:10) * b(1:10, 1:10)
```

NOTAZIONE VETTORIALE

Esempi:

```
integer :: i
integer :: array1(10)       ! 1D integer array of 10 elements
integer :: array2(10, 10)   ! 2D integer array of 100 elements

array1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  ! Array constructor
array1 = [(i, i = 1, 10)]                 ! Implied do loop constructor
array1(:) = 0                             ! Set all elements to zero
array1(1:5) = 1                           ! Set first five elements to one
array1(6:) = 1                            ! Set all elements after five to one

print *, array1(1:10:2)    ! Print out elements at odd indices
print *, array2(:,1)       ! Print out the first column in a 2D array
print *, array1(10:1:-1)   ! Print an array in reverse
```

NOTAZIONE VETTORIALE

Esempi:

```
REAL, DIMENSION(10, -5:5, 3) :: a, b
REAL, DIMENSION(-4:5, 1:5, 3) :: c
```

Si possono specificare solo gli elementi di indice dispari

```
a(1:10:2, -3:5:2, 1:3:2) = &
        b(1:10:2, -3:5:2, 1:3:2) * c(::2, 1:5, ::2)
```

Oppure un solo piano

```
a(5, -5:, :) = b(3, -5:, :)
```

Oppure solo una parte di piano

```
a(5, 1:5, :) = b(5, 1:5, :) + c(0, :, :)
```

Ovvero solo alcune righe di un piano

```
a(1:10:3, 0, 1:3) = c(1:4, 1, :)
```

NOTAZIONE VETTORIALE

La notazione vettoriale è una sintassi molto efficace, ma è necessario tener ben presente come vengono eseguite le operazioni.

Il principio è che tutto ciò che compare alla destra del segno di assegnazione è **interamente** calcolato, **prima** che l'assegnazione venga fatta:

```
a(2:n) = a(2:n) + b(1:n-1)
```

equivale a:

```
DO i = 2, n
   a(i) = a(i) + b(i-1)
END DO
```

NOTAZIONE VETTORIALE

```
b(2:n) = a(2:n) + b(1:n-1)
```

equivale a:

```
DO i = 2, n
       t(i) = a(i) + b(i-1)
END DO
DO i = 2, n
       b(i) = t(i)
END DO
```

e non come si potrebbe erroneamente pensare a:

```
DO i = 2, n
       b(i) = a(i) + b(i-1)
END DO
```

ARRAY A DIMENSIONE ZERO

In Fortran sono ammessi gli array, e le sezioni di array, a dimensione nulla. Ciò permette di evitare di trattare alcuni casi limite con eccezioni e codice extra.

```
DO i = 1, n
   x(i) = b(i) / a(i, i)
   b(i+1:n) = b(i+1:n) - a(i+1:n, i) * x(i)  ! b(n+1:n) is fine
ENDDO
```

Attenzione1: *selezionare un* subarray a sezione nulla è diverso da accedere ad un elemento oltre il limite dell'array.

```
INTEGER, DIMENSION(4) :: b
b(5:4) = 0   ! Ok, zero-sized array, no assignment is made.
b(5) = 0     ! Not ok, out-of-bounds access.
```

CONTROLLO DELL'ACCESSO

Attenzione2: per ottimizzare, non sempre i compilatori impongono a *runtime* il controllo che gli accessi agli elementi degli array siano legali:

```
REAL, DIMENSION(5) :: a
a(12) = 3.          ! Illegal access
WRITE(*,*) a(0)     ! Illegal access
```

In fase di sviluppo e/o *debugging*, è quindi bene forzare questi controlli, così che ogni accesso illegale produca un errore a *runtime*.

```
gfortran -g –fcheck=bounds [...]
ifort -g -check bounds [...]
nvfortran –g –Mbounds [...]
```

REFERENZE

1. S.J. Chapman, ‘Fortran for Scientists and Engineers’, Fourth Edition (2018), Chaps. 6, 8.
2. M. Metcalf et al., ‘Modern Fortran explained’, Fifth Edition (2018), Chap. 7.
3. Arrays and strings — Fortran Programming Language (fortran-lang.org).
4. Python Fortran Rosetta Stone — Fortran Programming Language (fortran-lang.org).

Funzioni intrinseche per array

Introduction to modern Fortran

Tommaso Gorni
t.gorni@cineca.it

COSA VIENE PRESENTATO

- Classificazione delle funzioni intrinseche per array
- Funzioni matematiche e algebriche
- Funzioni di riduzione
- Funzioni interrogative
- Funzioni costruttive
- Funzioni di ricerca
- Funzioni di manipolazione
- Il costrutto WHERE

PROCEDURE INTRINSECHE

Fortran possiede attualmente più di 200 procedure intrinseche, la maggior parte delle quali è composta da **funzioni per operare su array**.

Tali funzioni possono essere classificate in base a come operano:

- **Funzioni elementari** (*elemental functions*): progettate per operare su scalari, se operano su array eseguono la stessa operazione elemento per elemento.
- **Funzioni interrogative** (*inquiry functions*): restituiscono proprietà del loro argomento che non dipendono dal suo valore (la taglia di un array, …).
- **Funzioni trasformative** (*transformational functions*): ogni elemento del risultato dipende da più elementi del, o degli, array di input (somma, prodotti matriciali, …).

FUNZIONI INTRINSECHE

Alternativamente, si possono classificare in base al tipo di operazione che svolgono sugli array:

- **Funzioni matematiche e algebriche** (*mathematical and algebraic functions*).
- **Funzioni di riduzione** (*reduction functions*).
- **Funzioni interrogative** (*inquiry functions*).
- **Funzioni costruttive** (*construction functions*).
- **Funzioni di ricerca** (*location functions*).
- **Funzioni di manipolazione** (*manipulation functions*).

NB: tutte le funzioni intrinseche di Fortran sono

- Generiche: è possibile passare alla stessa funzione array di interi, *float* a singola o doppia precisione, o complessi.
- Pure: non presentano effetti collaterali (non alterano gli array in input o variabili globali, né eseguono I/O.)

FUNZIONI MATEMATICHE

Tutte le funzioni matematiche in Fortran sono implementate come funzioni elementari: se applicate ad un array, restituiscono un array conforme dove la funzione è stata calcolata elemento per elemento.

Esempi:

```
sin(array)          ! Sine value for each element of array
sinh(array)         ! Hyperbolic sine
log(array)          ! Natural logarithm
sqrt(array)         ! Square root
erf(array)          ! Error function
gamma(array)        ! Gamma function
bessel_j0(array)    ! Bessel function of first kind and order zero
```

FUNZIONI ALGEBRICHE: PRODOTTO SCALARE

Sintassi con operazione esplicita

```
REAL a(N), b(N), c
c = 0.0
DO i = 1, N
       c = c + a(i) * b(i)
ENDDO
```

Sintassi con utilizzo di funzione intrinseca Fortran

```
REAL, DIMENSION(N) :: a, b
REAL :: c = 0.0
c = DOT_PRODUCT(a,b)     ! Transformational function
```

FUNZIONI ALGEBRICHE: PRODOTTO DI MATRICI

Sintassi con operazione esplicita

```
REAL a(N,L), b(L,M), c(N,M)
DO j = 1, M
   DO i = 1, N
      c(i,j) = 0.0
      DO k = 1, L
         c(i,j) = c(i,j) + a(i,k) * b(k,j)
      ENDDO
   ENDDO
ENDDO
```

Sintassi con utilizzo di funzione intrinseca fortran

```
REAL a(N,L), b(L,M), c(N,M)
c = MATMUL(a,b)                    ! Transformational function
```

FUNZIONI DI RIDUZIONE

Le funzioni di riduzione si applicano ad array multidimensionali di rango `N`, e ritornano un valore scalare. Se è specificato l'argomento scalare opzionale `DIM`, l'operazione viene eseguita solo lungo la dimensione specificata e la funzione restituisce un array di rango `N-1`. L'argomento opzionale `MASK` è un array logico conforme all'array di input. Nel caso sia fornito, l'operazione coinvolge solo gli elementi dell'array per i quali `MASK` è `.TRUE.`.

`ALL(MASK[,DIM])`	`.TRUE.` solo se tutti gli elementi sono `.TRUE.`
`ANY(MASK[,DIM])`	`.TRUE.` se almeno un elemento è `.TRUE.`
`COUNT(MASK[,DIM])`	numero degli elementi `.TRUE.`
`MAXVAL(ARRAY[,DIM][,MASK])`	valore del massimo degli elementi
`MINVAL(ARRAY[,DIM][,MASK])`	valore del minimo degli elementi
`PRODUCT(ARRAY[,DIM][,MASK])`	prodotto degli elementi
`SUM(ARRAY[,DIM][,MASK])`	somma degli elementi

Tutte le funzioni di riduzione sono trasformative per definizione.

FUNZIONI DI RIDUZIONE

Esempio 1: ricerca del massimo

```
REAL, DIMENSION(100,100,100) :: a

valmax = MAXVAL( a, MASK=(a<1.0))
```

Esempio 2: calcolo del valore medio

```
REAL, DIMENSION(100,100,100) :: a

valmed = SUM(a,MASK=(a>0.0))/COUNT(MASK=(a>0.0))
```

FUNZIONI DI RIDUZIONE

Esempio 3: uso di ALL

```
LOGICAL :: test1, test2, test3
REAL, DIMENSION(3,2) :: a = RESHAPE( [5,9,6,10,8,12],[3,2] )

test1 = ALL(a > 5)                ! false
test2 = ALL(a < 20)               ! true
test3 = ALL(a >= 5 .AND. test2)   ! true
```

FUNZIONI INTERROGATIVE

Queste funzioni permettono di **conoscere le proprietà** degli array.

`ALLOCATED(ARRAY)`	`.TRUE.` se la memoria è associata (solo `ALLOCATABLE`)
`LBOUND(ARRAY[,DIM])`	limiti inferiori per gli indici, per ogni dimensione
`SHAPE(SOURCE)`	forma della matrice (si applica anche agli scalari)
`SIZE(ARRAY[,DIM])`	dimensione della matrice
`UBOUND(ARRAY[,DIM])`	limiti superiori per gli indici, per ogni dimensione

FUNZIONI INTERROGATIVE

Esempio 1: Uso di SIZE

```
REAL, DIMENSION(3,2) :: a

num = SIZE(a)          ! num=6
num = SIZE(a, DIM=1)   ! num=3
num = SIZE(a, DIM=2)   ! num=2
```

E' possibile usare SIZE anche nelle dichiarazioni:

```
REAL, DIMENSION(lda,n) :: a
REAL, DIMENSION(SIZE(a,1),SIZE(a,2)) :: temp
```

FUNZIONI INTERROGATIVE

Esempio 2: estremi e forma di un array multidimensionale

```
REAL DIMENSION(2:3,2:5) :: a
WRITE(*,*) LBOUND(a) ! [2,2]
WRITE(*,*) UBOUND(a) ! [3,5]
WRITE(*,*) SHAPE(a)  ! [2,4]
WRITE(*,*) SIZE(a)   ! 8
```

FUNZIONI COSTRUTTIVE

`MERGE(TSOURCE,FSOURCE,MASK)` è una funzione elementare che si applica a 3 matrici conformi. Le prime 2 devono essere dello stesso tipo, `MASK` dev'essere di tipo logico. Si genera una matrice conforme a `MASK` e di tipo uguale alle prime due; il valore di ogni elemento proviene da `TSOURCE` se l'elemento corrispondente di `MASK` è `.TRUE.`, altrimenti proviene da `FSOURCE`.

```
integer :: m(2,2) = reshape([1,2,3,4],shape(m))

integer :: n(2,2) = reshape([4,3,2,1],shape(n))

integer :: a(2,2) = merge(m, n, m<n)
```

$$m = \begin{matrix} 1 & 3 \\ 2 & 4 \end{matrix} \qquad n = \begin{matrix} 4 & 2 \\ 3 & 1 \end{matrix} \qquad a = \begin{matrix} 1 & 2 \\ 2 & 1 \end{matrix}$$

FUNZIONI COSTRUTTIVE

SPREAD(SOURCE,DIM,NCOPIES) è una funzione trasformativa che genera una matrice duplicando NCOPIES volte il vettore SOURCE lungo la dimensione DIM.

Esempio: uso di SPREAD

```
REAL, DIMENSION(3):: a=[2,3,4]
REAL, DIMENSION(3,3) :: b, c

b = SPREAD(a, DIM=2, NCOPIES=3)
c = SPREAD(a, DIM=1, NCOPIES=3)
```

$$b = \begin{matrix} 2 & 3 & 4 \\ 2 & 3 & 4 \\ 2 & 3 & 4 \end{matrix}$$

$$c = \begin{matrix} 2 & 2 & 2 \\ 3 & 3 & 3 \\ 4 & 4 & 4 \end{matrix}$$

FUNZIONI DI RICERCA

MAXLOC(ARRAY[,MASK]) è una funzione trasformativa che genera un array con la posizione dell'elemento di valore massimo della matrice ARRAY, eventualmente considerando i soli elementi corrispondenti a .TRUE. di MASK.

MINLOC(ARRAY[,MASK]) è una funzione trasformativa che genera un array con la posizione dell'elemento di valore minimo della matrice ARRAY, eventualmente considerando i soli elementi corrispondenti a .TRUE. di MASK.

```
              0.1   0.2   0.3   0.4
a(3,4) =      1.0   2.0   4.0   3.0        MAXLOC(a) : [3,4]
              10.   20.   30.   40.
```

FUNZIONI DI RICERCA E RIDUZIONE

Alcuni esempi di riepilogo:

```
            0.1  0.2  0.3  0.4      MAXVAL(a)         : 40.0
a(3,4) =    1.0  2.0  4.0  3.0      MAXLOC(a)         : [3,4]
            10.  20.  30.  40.      MAXVAL(a,DIM=2) : [0.4,4.0,40.0]
```

```
              0.1 0.2 0.3    1.1 1.2 1.3   2.1 2.2 2.3
c(3,3,3) =    1.0 2.0 3.0    1.1 2.1 3.1   1.2 2.2 3.2
              10. 20. 30.    11. 21. 31.   12. 22. 32.
```

```
MAXVAL(c) : 32.0

                          2.1 2.2 2.3    SHAPE(MAXVAL(c,DIM=3)) : [3,3]
MAXVAL(c,DIM=3) :         1.2 2.2 3.2    MAXLOC(MAXVAL(c,DIM=3) : [3,3]
                          12. 22. 32.
```

FUNZIONI DI MANIPOLAZIONE

CSHIFT(ARRAY,SHIFT[,DIM]) è una funzione trasformativa che genera una matrice conforme a ARRAY, spostando gli elementi di ARRAY in modo circolare lungo la dimensione specificata da DIM della quantità indicata in SHIFT.

Esempio:

```
    1.0  4.0  7.0  10.0                          3.0  6.0  9.0  12.0
a = 2.0  5.0  8.0  11.0      CSHIFT(a,2,dim=1) = 1.0  4.0  7.0  10.0
    3.0  6.0  9.0  12.0                          2.0  5.0  8.0  11.0
```

SHIFT può anche essere un array:

```
CSHIFT(a, [0,2,0], DIM=2) =
```

?

FUNZIONI DI MANIPOLAZIONE

CSHIFT(ARRAY,SHIFT[,DIM]) è una funzione trasformativa che genera una matrice conforme a ARRAY, spostando gli elementi di ARRAY in modo circolare lungo la dimensione specificata da DIM della quantità indicata in SHIFT.

Esempio:

```
    1.0  4.0  7.0  10.0                          3.0  6.0  9.0  12.0
a = 2.0  5.0  8.0  11.0      CSHIFT(a,2,dim=1) = 1.0  4.0  7.0  10.0
    3.0  6.0  9.0  12.0                          2.0  5.0  8.0  11.0
```

SHIFT può anche essere un array:

```
                                    1.0   4.0  7.0  10.0
CSHIFT(a, [0,2,0], DIM=2) =         8.0  11.0  2.0   5.0
                                    3.0   6.0  9.0  12.0
```

FUNZIONI DI MANIPOLAZIONE

EOSHIFT(ARRAY,SHIFT[,BOUNDARY][,DIM]) è una funzione trasformativa che genera una matrice conforme a ARRAY, spostando gli elementi di ARRAY lungo la dimensione specificata da DIM della quantità indicata in SHIFT. Gli elementi che finiscono fuori dalle dimensioni di ARRAY sono persi. Le posizioni lasciate libere sono azzerate oppure riempite con gli elementi di BOUNDARY.

Esempio:

```
    1.0  5.0   9.0  13.0
a = 2.0  6.0  10.0  14.0
    3.0  7.0  11.0  15.0
    4.0  8.0  12.0  16.0
```

```
                                3.0  7.0  11.0  15.0
EOSHIFT(a,2,dim=1) =            4.0  8.0  12.0  16.0
                                0.0  0.0   0.0   0.0
                                0.0  0.0   0.0   0.0
```

```
                                0.0  0.0   0.0   0.0
EOSHIFT(a,-2,dim=1) =           0.0  0.0   0.0   0.0
                                1.0  5.0   9.0  13.0
                                2.0  6.0  10.0  14.0
```

FUNZIONI DI MANIPOLAZIONE

TRANSPOSE(MATRIX) è una funzione trasformativa che genera la trasposta della matrice MATRIX (ossia di array di rango 2).

Esempio:

```
    1.0  5.0   9.0  13.0
a = 2.0  6.0  10.0  14.0
    3.0  7.0  11.0  15.0
    4.0  8.0  12.0  16.0
```

```
                    1.0  2.0   3.0   4.0
TRANSPOSE(a) = 5.0  6.0   7.0   8.0
               9.0 10.0  11.0  12.0
              13.0 14.0  15.0  16.0
```

IL COSTRUTTO WHERE

Il costrutto `WHERE` è utilizzato per controllare quali elementi di un **array** sono oggetto di un'assegnazione sulla base del risultato di una condizione logica.

```
WHERE (mask_1)
       array = ...      ! If mask_1 = .true.
ELSEWHERE (mask_2)
       array = ...      ! If mask_1 = .false. and mask_2 = .true.
ELSEWHERE
       array = ...      ! If mask_1 = .false. and mask_2 = .false.
END WHERE
```

Dal Fortran95 i costrutti `WHERE` possono essere annidati.

IL COSTRUTTO WHERE

Esempio 1: Evitare la divisione per zero

```
REAL, DIMENSION(100,100) :: a, b

WHERE ( b > 0.0 ) a = a / b
```

Esempio 2: Evitare la divisione per zero

```
INTEGER, DIMENSION(8,8) :: a

WHERE ( a<=0 )
  a = 0
ELSEWHERE
  a = 1/a
END WHERE
```

REFERENZE

1. S.J. Chapman, ‘Fortran for Scientists and Engineers’, Fourth Edition (2018), Chaps. 8.and 9.
2. M. Metcalf et al., ‘Modern Fortran explained’, Fifth Edition (2018), Chap. 9.
3. Fortran Intrinsics — Fortran Programming Language (fortran-lang.org)

GRAZIE

Tommaso Gorni
t.gorni@cineca.it