Fortran 90 Arrays

Program testing can be used to show the presence of bugs,
but never to show their absence

Edsger W. Dijkstra

The `DIMENSION` Attribute: 1/6

- **A Fortran 90 program uses the `DIMENSION` attribute to declare arrays.**
- **The `DIMENSION` attribute requires three components in order to complete an array specification, *rank*, *shape*, and *extent*.**
- **The *rank* of an array is the number of "indices" or "subscripts." The maximum rank is 7 (*i.e.*, seven-dimensional).**
- **The *shape* of an array indicates the number of elements in each "dimension."**

The `DIMENSION` Attribute: 2/6

- **The rank and shape of an array is represented as $(s_1,s_2,\ldots,s_n)$, where n is the rank of the array and s_i $(1\leq i \leq n)$ is the number of elements in the i-th dimension.**
  - **(7) means a rank 1 array with 7 elements**
  - **(5,9) means a rank 2 array (*i.e.*, a table) whose first and second dimensions have 5 and 9 elements, respectively.**
  - **(10,10,10,10) means a rank 4 array that has 10 elements in each dimension.**

The `DIMENSION` Attribute: 3/6

- **The *extent* is written as *m*:*n*, where *m* and *n* ($m \le n$) are `INTEGER`s. We saw this in the `SELECT CASE`, substring, etc.**
- **Each dimension has its own extent.**
- **An extent of a dimension is the range of its index. If *m*: is omitted, the default is 1.**
  - **`-3:2` means possible indices are -3, -2 , -1, 0, 1, 2**
  - **`5:8` means possible indices are 5,6,7,8**
  - **`7` means possible indices are 1,2,3,4,5,6,7**

The `DIMENSION` Attribute: 4/6

- **The `DIMENSION` attribute has the following form:**

  `DIMENSION`(extent-1, extent-2, …, extent-*n*)

- **Here, extent-*i* is the extent of dimension *i*.**
- **This means an array of dimension *n* (*i.e.*, *n* indices) whose *i*-th dimension index has a range given by extent-*i*.**
- **Just a reminder: Fortran 90 only allows maximum 7 dimensions.**
- **Exercise: given a `DIMENSION` attribute, determine its shape.**

The `DIMENSION` Attribute: 5/6

- Here are some examples:
  - `DIMENSION(-1:1)` is a 1-dimensional array with possible indices -1,0,1
  - `DIMENSION(0:2,3)` is a 2-dimensional array (*i.e.*, a table). Possible values of the first index are 0,1,2 and the second 1,2,3
  - `DIMENSION(3,4,5)` is a 3-dimensional array. Possible values of the first index are 1,2,3, the second 1,2,3,4, and the third 1,2,3,4,5.

The **DIMENSION** Attribute: 6/6

- **Array declaration is simple. Add the `DIMENSION` attribute to a type declaration.**
- **Values in the `DIMENSION` attribute are usually `PARAMETER`s to make program modifications easier.**

```
INTEGER, PARAMETER :: SIZE=5, LOWER=3, UPPER = 5
INTEGER, PARAMETER :: SMALL = 10, LARGE = 15
REAL, DIMENSION(1:SIZE) :: x
INTEGER, DIMENSION(LOWER:UPPER,SMALL:LARGE) :: a,b
LOGICAL, DIMENSION(2,2) :: Truth_Table
```

Use of Arrays: 1/3

- **Fortran 90 has, in general, three different ways to use arrays: referring to *individual array element*, referring to the *whole array*, and referring to a *section of an array*.**
- **The first one is very easy. One just starts with the array name, followed by `()` between which are the *indices* separated by `,`.**
- **Note that each index must be an `INTEGER` or an expression evaluated to an `INTEGER`, and the value of an index must be *in the range of the corresponding extent*. But, Fortran 90 won't check it for you.**

Use of Arrays: 2/3

- **Suppose we have the following declarations**

```
INTEGER, PARAMETER :: L_BOUND = 3, U_BOUND = 10
INTEGER, DIMENSION(L_BOUND:U_BOUND) :: x
```

```
DO i = L_BOUND, U_BOUND
  x(i) = i
END DO
```

array `x()` has 3,4,5,…, 10

```
DO i = L_BOUND, U_BOUND
  IF (MOD(i,2) == 0) THEN
    x(i) = 0
  ELSE
    x(i) = 1
  END IF
END DO
```

array `x()` has 1,0,1,0,1,0,1,0

Use of Arrays: 3/3

- **Suppose we have the following declarations:**

```
INTEGER, PARAMETER :: L_BOUND = 3, U_BOUND = 10
INTEGER, DIMENSION(L_BOUND:U_BOUND,            &
                   L_BOUND:U_BOUND) :: a
```

```
DO i = L_BOUND, U_BOUND
  DO j = L_BOUND, U_BOUND
    a(i,j) = 0
  END DO
  a(i,i) = 1
END DO
```

generate an identity matrix

```
DO i = L_BOUND, U_BOUND
  DO j = i+1, U_BOUND
    t      = a(i,j)
    a(i,j) = a(j,i)
    a(j,i) = t
  END DO
END DO
```

Swapping the lower and upper diagonal parts (*i.e.*, the *transpose* of a matrix)

The Implied `DO`: 1/7

- Fortran has the implied `DO` that can generate efficiently a set of values and/or elements.
- The implied DO is a variation of the `DO`-loop.
- The implied DO has the following syntax:

```
(item-1, item-2, …,item-n, v=initial,final,step)
```

- Here, `item-1`, `item-2`, …, `item-n` are variables or expressions, `v` is an `INTEGER` variable, and `initial`, `final`, and `step` are `INTEGER` expressions.
- “`v=initial,final,step`” is exactly what we saw in a `DO`-loop.

The Implied `DO`: 2/7

- The execution of an implied `DO` below lets variable `v` to start with `initial`, and step though to `final` with a step size `step`.

`(item-1, item-2, …,item-n, v=initial,final,step)`

- The result is a sequence of items.
- `(i+1, i=1,3)` generates 2, 3, 4.
- `(i*k, i+k*i, i=1,8,2)` generates `k`, `1+k` (`i` = 1), `3*k`, `3+k*3` (`i` = 3), `5*k`, `5+k*5` (`i` = 5), `7*k`, `7+k*7` (`i` = 7).
- `(a(i),a(i+2),a(i*3-1),i*4,i=3,5)` generates `a(3)`, `a(5)`, `a(8)` , `12` (`i`=3), `a(4)`, `a(6)`, `a(11)`, `16` (`i`=4), `a(5)`, `a(7)`, `a(14)`, `20`.

The Implied `DO`: 3/7

- Implied `DO` may be nested.

  `(i*k,(j*j,i*j,j=1,3), i=2,4)`

- In the above, `(j*j,i*j,j=1,3)` is nested in the implied `i` loop.

- Here are the results:
  - When `i` = 2, the implied `DO` generates

    `2*k, (j*j,2*j,j=1,3)`

  - Then, `j` goes from 1 to 3 and generates

$$2*k,\ \underbrace{1*1,\ 2*1}_{j\ =\ 1},\ \underbrace{2*2,\ 2*2}_{j\ =\ 2},\ \underbrace{3*3,\ 2*3}_{j\ =\ 3}$$

The Implied **DO**: 4/7

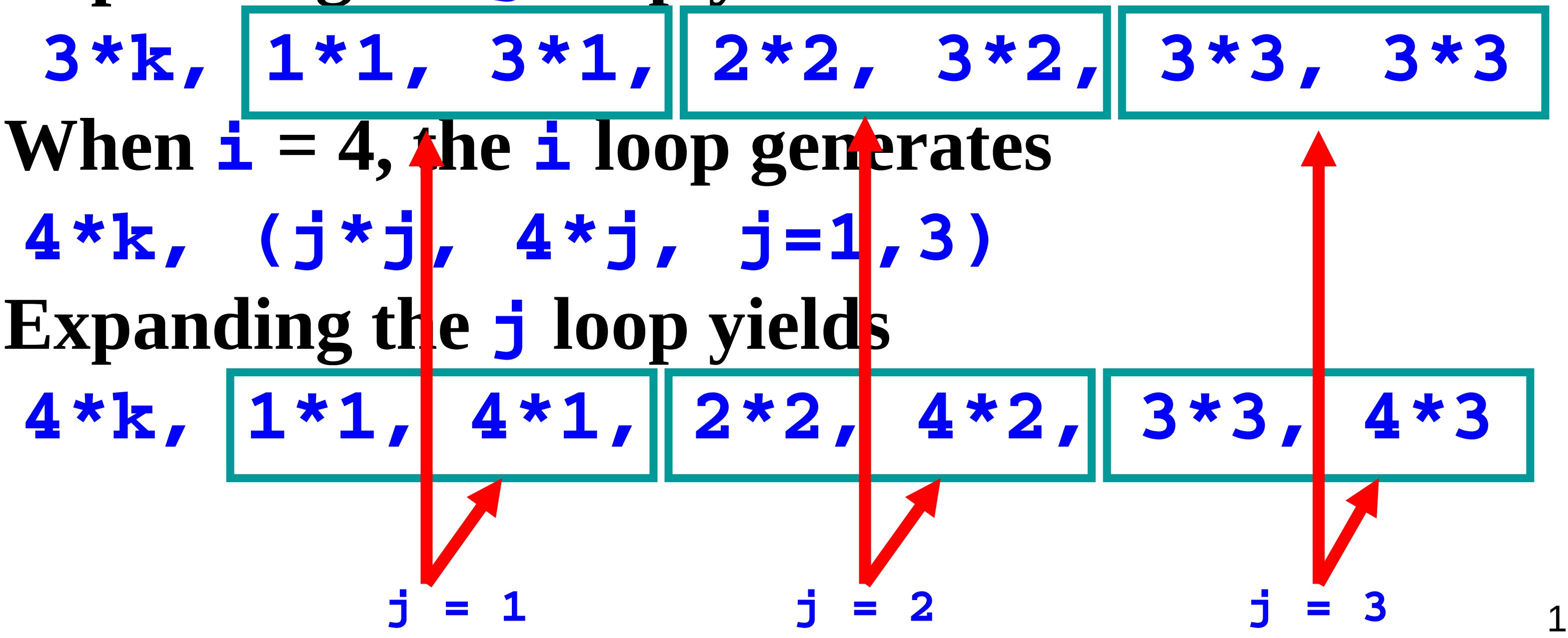
- Continue with the previous example

  `(i*k,(j*j,i*j,j=1,3), i=2,4)`

- When `i` = 3, it generates the following:

  `3*k, (j*j,3*j,j=1,3)`

- Expanding the `j` loop yields:



  `3*k, 1*1, 3*1, 2*2, 3*2, 3*3, 3*3`

- When `i` = 4, the `i` loop generates

  `4*k, (j*j, 4*j, j=1,3)`

- Expanding the `j` loop yields

  `4*k, 1*1, 4*1, 2*2, 4*2, 3*3, 4*3`

`j = 1` `j = 2` `j = 3`

The Implied `DO`: 5/7

- The following generates a multiplication table:

  `((i*j,j=1,9),i=1,9)`

- When `i` = 1, the inner `j` implied `DO`-loop produces 1*1, 1*2, …, 1*9
- When `i` = 2, the inner `j` implied `DO`-loop produces 2*1, 2*2, …, 2*9
- When `i` = 9, the inner `j` implied `DO`-loop produces 9*1, 9*2, …, 9*9

The Implied `DO`: 6/7

- **The following produces all upper triangular entries, *row-by-row*, of a 2-dimensional array:**

  `((a(p,q),q = p,n),p = 1,n)`

- **When `p` = 1, the inner `q` loop produces `a(1,1)`, `a(1,2)`, …, `a(1,n)`**
- **When `p`=2, the inner `q` loop produces `a(2,2)`, `a(2,3)`, …., `a(2,n)`**
- **When `p`=3, the inner `q` loop produces `a(3,3)`, `a(3,4)`, …, `a(3,n)`**
- **When `p`=`n`, the inner `q` loop produces `a(n,n)`**

The Implied `DO`: 7/7

- **The following produces all upper triangular entries, *column-by-column*:**

```
((a(p,q),p = 1,q),q = 1,n)
```

- **When `q`=1, the inner `p` loop produces `a(1,1)`**
- **When `q`=2, the inner `p` loop produces `a(1,2)`, `a(2,2)`**
- **When `q`=3, the inner `p` loop produces `a(1,3)`, `a(2,3)`, …, `a(3,3)`**
- **When `q`=`n`, the inner `p` loop produces `a(1,n)`, `a(2,n)`, `a(3,n)`, …, `a(n,n)`**

Array Input/Output: 1/8

- **Implied `DO` can be used in `READ(*,*)` and `WRITE(*,*)` statements.**
- **When an implied `DO` is used, it is equivalent to execute the I/O statement with the generated elements.**
- **The following prints out a multiplication table**

```
WRITE(*,*)((i,“*”,j,“=“,i*j,j=1,9),i=1,9)
```

- **The following has a better format (*i.e.*, 9 rows):**

```
DO i = 1, 9
   WRITE(*,*) (i, “*”, j, “=“, i*j, j=1,9)
END DO
```

Array Input/Output: 2/8

- **The following shows three ways of reading `n` data items into an one dimensional array `a()`.**
- **Are they the same?**

(1)

```
READ(*,*) n,(a(i),i=1,n)
```

(2)

```
READ(*,*) n
READ(*,*) (a(i),i=1,n)
```

(3)

```
READ(*,*) n
DO i = 1, n
   READ(*,*) a(i)
END DO
```

Array Input/Output: 3/8

- **Suppose we wish to fill `a(1)`, `a(2)` and `a(3)` with 10, 20 and 30. The input may be:**

```
3   10   20   30
```

- ***Each `READ` starts from a new line!***

(1)

```
READ(*,*) n,(a(i),i=1,n)
```

OK

(2)

```
READ(*,*) n
READ(*,*) (a(i),i=1,n)
```

Wrong! `n` gets 3 and the second `READ` fails

(3)

```
READ(*,*) n
DO i = 1, n
   READ(*,*) a(i)
END DO
```

Wrong! `n` gets 3 and the three `READ`s fail

Array Input/Output: 4/8

- **What if the input is changed to the following?**

```
3
10   20   30
```

(1)

```
READ(*,*) n,(a(i),i=1,n)
```

OK

(2)

```
READ(*,*) n
READ(*,*) (a(i),i=1,n)
```

OK. Why????

(3)

```
READ(*,*) n
DO i = 1, n
   READ(*,*) a(i)
END DO
```

Wrong! `n` gets 3, `a(1)` has 10; but, the next two `READ`s fail

Array Input/Output: 5/8

- **What if the input is changed to the following?**

```
3
10
20
30
```

(1)

```
READ(*,*) n,(a(i),i=1,n)
```

OK

(2)

```
READ(*,*) n
READ(*,*) (a(i),i=1,n)
```

OK

(3)

```
READ(*,*) n
DO i = 1, n
   READ(*,*) a(i)
END DO
```

OK

Array Input/Output: 6/8

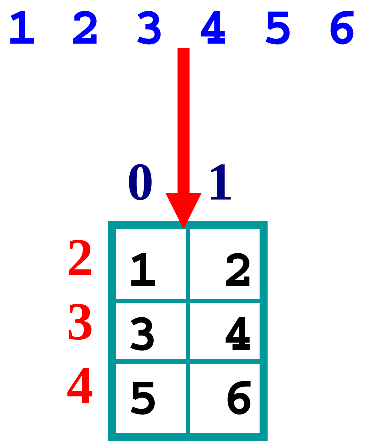
- Suppose we have a two-dimensional array `a()`:

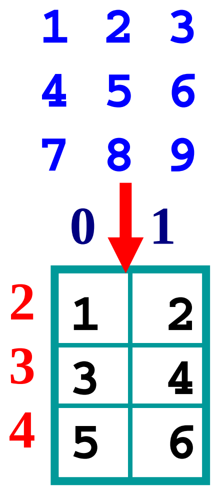
  `INTEGER, DIMENSION(2:4,0:1) :: a`

- Suppose further the **READ** is the following:

  `READ(*,*) ((a(i,j),j=0,1),i=2,4)`

- What are the results for the following input?



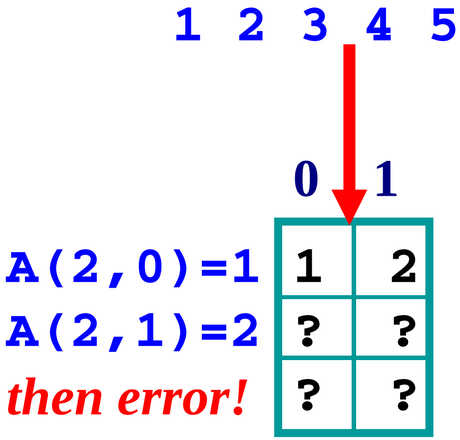


Array Input/Output: 7/8

- **Suppose we have a two-dimensional array `a()`:**

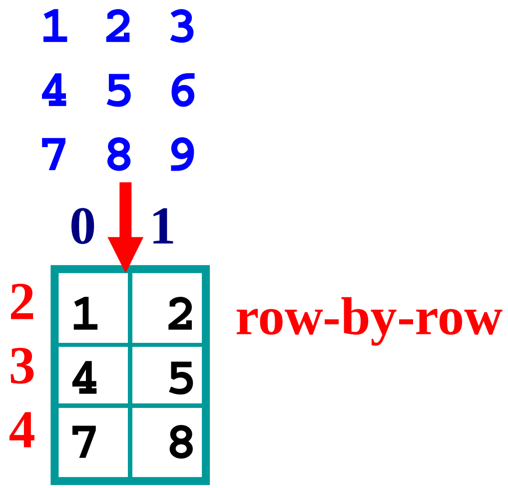
```
INTEGER, DIMENSION(2:4,0:1) :: a
DO i = 2, 4
  READ(*,*) (a(i,j),j=0,1)
END DO
```

- **What are the results for the following input?**



```
1 2 3 4 5 6
```

	0	1
A(2,0)=1 A(2,1)=2	1	2
	?	?
then error!	?	?



```
1 2 3
4 5 6
7 8 9
```

	0	1
2	1	2
3	4	5
4	7	8

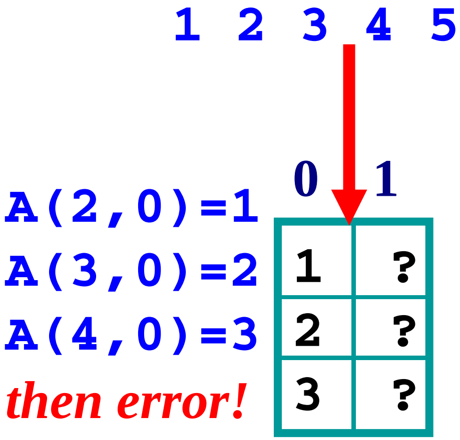
row-by-row

Array Input/Output: 8/8

- **Suppose we have a two-dimensional array `a()`:**

```
INTEGER, DIMENSION(2:4,0:1) :: a
DO j = 0, 1
  READ(*,*) (a(i,j),i=2,4)
END DO
```

- **What are the results for the following input?**



```
1 2 3 4 5 6
```

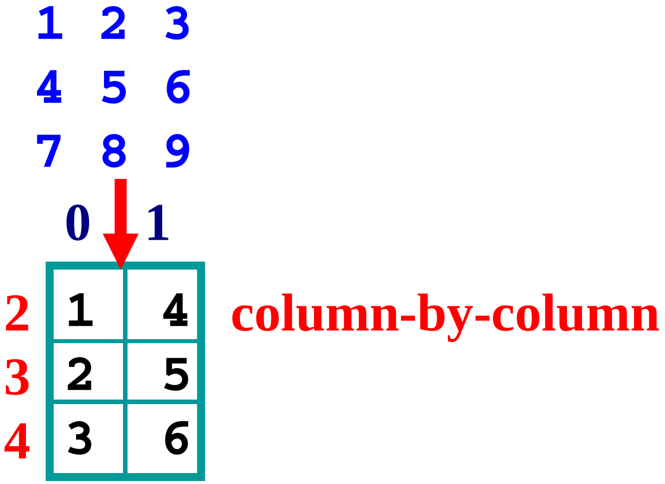
A(2,0)=1

A(3,0)=2

A(4,0)=3

then error!

0	1
1	?
2	?
3	?



```
1 2 3
4 5 6
7 8 9
```

	0	1
2	1	4
3	2	5
4	3	6

column-by-column

Matrix Multiplication: 1/2

- **Read a $l \times m$ matrix $A_{l \times m}$ and a $m \times n$ matrix $B_{m \times n}$, and compute their product $C_{l \times n} = A_{l \times m} \bullet B_{m \times n}$.**

```
PROGRAM Matrix_Multiplication
   IMPLICIT  NONE
   INTEGER, PARAMETER :: SIZE = 100
   INTEGER, DIMENSION(1:SIZE,1:SIZE) :: A, B, C
   INTEGER              :: L, M, N, i, j, k
   READ(*,*) L, M, N              ! read sizes <= 100
   DO i = 1, L
      READ(*,*) (A(i,j), j=1,M) ! A() is L-by-M
   END DO
   DO i = 1, M
      READ(*,*) (B(i,j), j=1,N) ! B() is M-by-N
   END DO
   …… other statements ……
END PROGRAM Matrix_Multiplication
```

Matrix Multiplication: 2/2

- **The following does multiplication and output**

```
DO i = 1, L
   DO j = 1, N
      C(i,j) = 0   ! for each C(i,j)
      DO k = 1, M  ! (row i of A)*(col j of B)
         C(i,j) = C(i,j) + A(i,k)*B(k,j)
      END DO
   END DO
END DO
```

```
DO i = 1, L        ! print row-by-row
   WRITE(*,*) (C(i,j), j=1, N)
END DO
```

Arrays as Arguments: 1/4

- **Arrays may also be used as arguments passing to functions and subroutines.**
- **Formal argument arrays may be declared as usual; however, Fortran 90 recommends the use of *assumed-shape arrays*.**
- **An assumed-shape array has its lower bound in each extent specified; but, the upper bound is not used.**

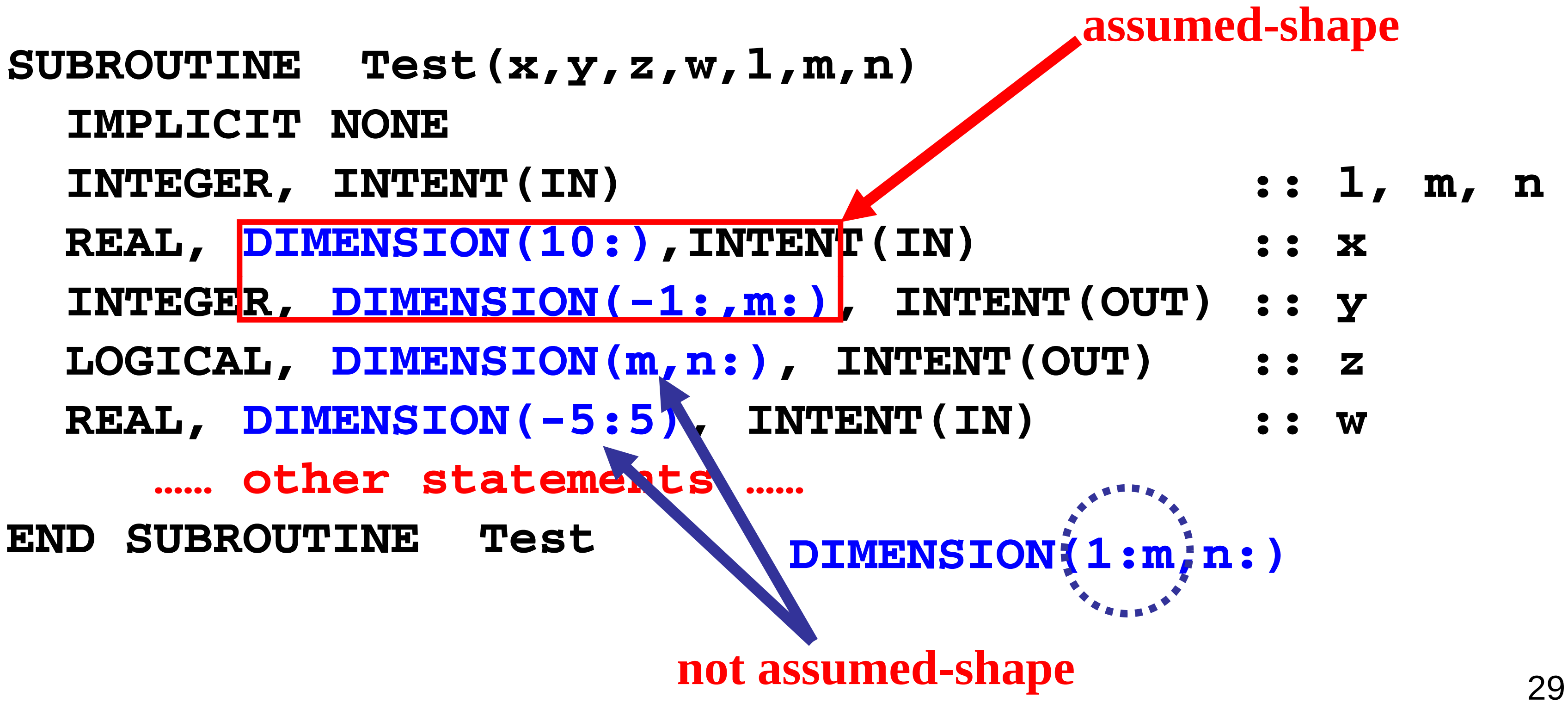
formal arguments

```
REAL, DIMENSION(-3:,1:), INTENT(IN)  :: x, y
INTEGER, DIMENSION(:), INTENT(OUT)   :: a, b
```

assumed-shape

Arrays as Arguments: 2/4

- **The extent in each dimension is an expression that uses *constants* or other *non-array formal arguments* with `INTENT(IN)` :**



```
SUBROUTINE  Test(x,y,z,w,l,m,n)
  IMPLICIT NONE
  INTEGER, INTENT(IN)                      :: l, m, n
  REAL, DIMENSION(10:),INTENT(IN)          :: x
  INTEGER, DIMENSION(-1:,m:), INTENT(OUT)  :: y
  LOGICAL, DIMENSION(m,n:), INTENT(OUT)    :: z
  REAL, DIMENSION(-5:5), INTENT(IN)        :: w
     …… other statements ……
END SUBROUTINE  Test
```

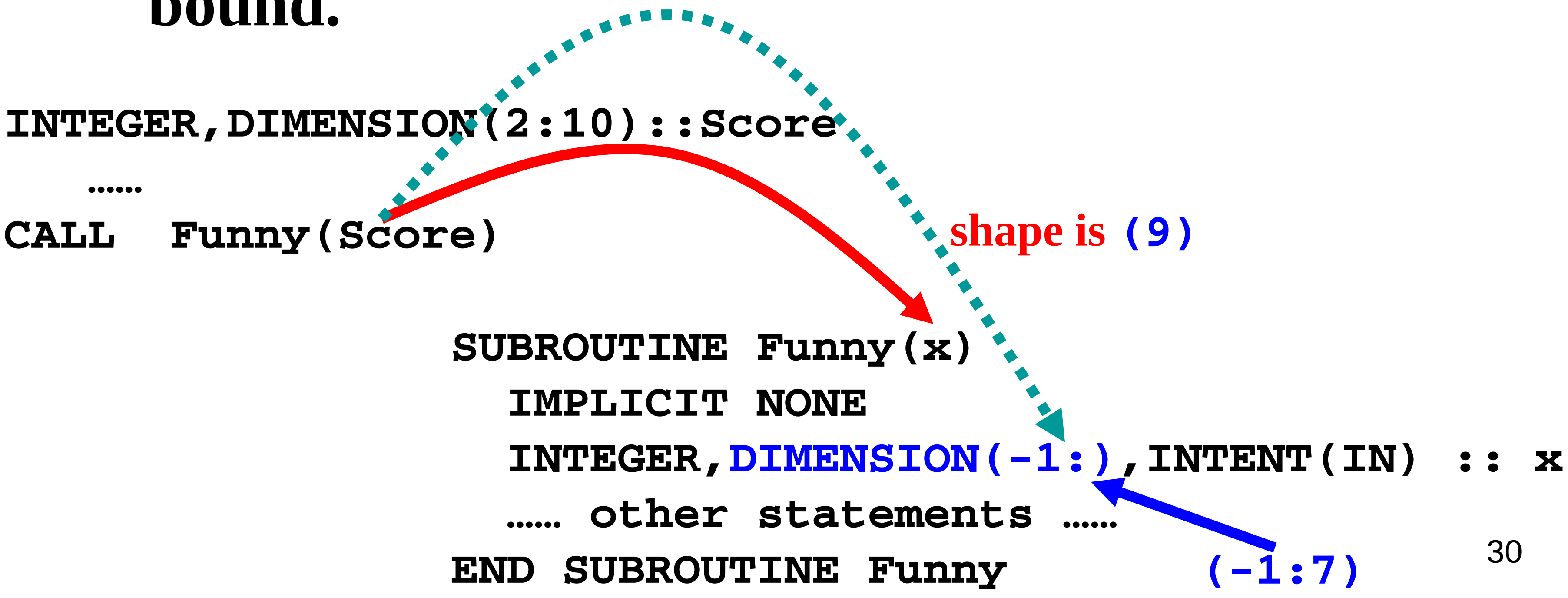
assumed-shape

`DIMENSION(1:m,n:)`

not assumed-shape

Arrays as Arguments: 3/4

- **Fortran 90 automatically passes *an array* and *its shape* to a formal argument.**
- **A subprogram receives the shape and uses the lower bound of each extent to recover the upper bound.**



```
INTEGER,DIMENSION(2:10)::Score
   ……
CALL  Funny(Score)
```

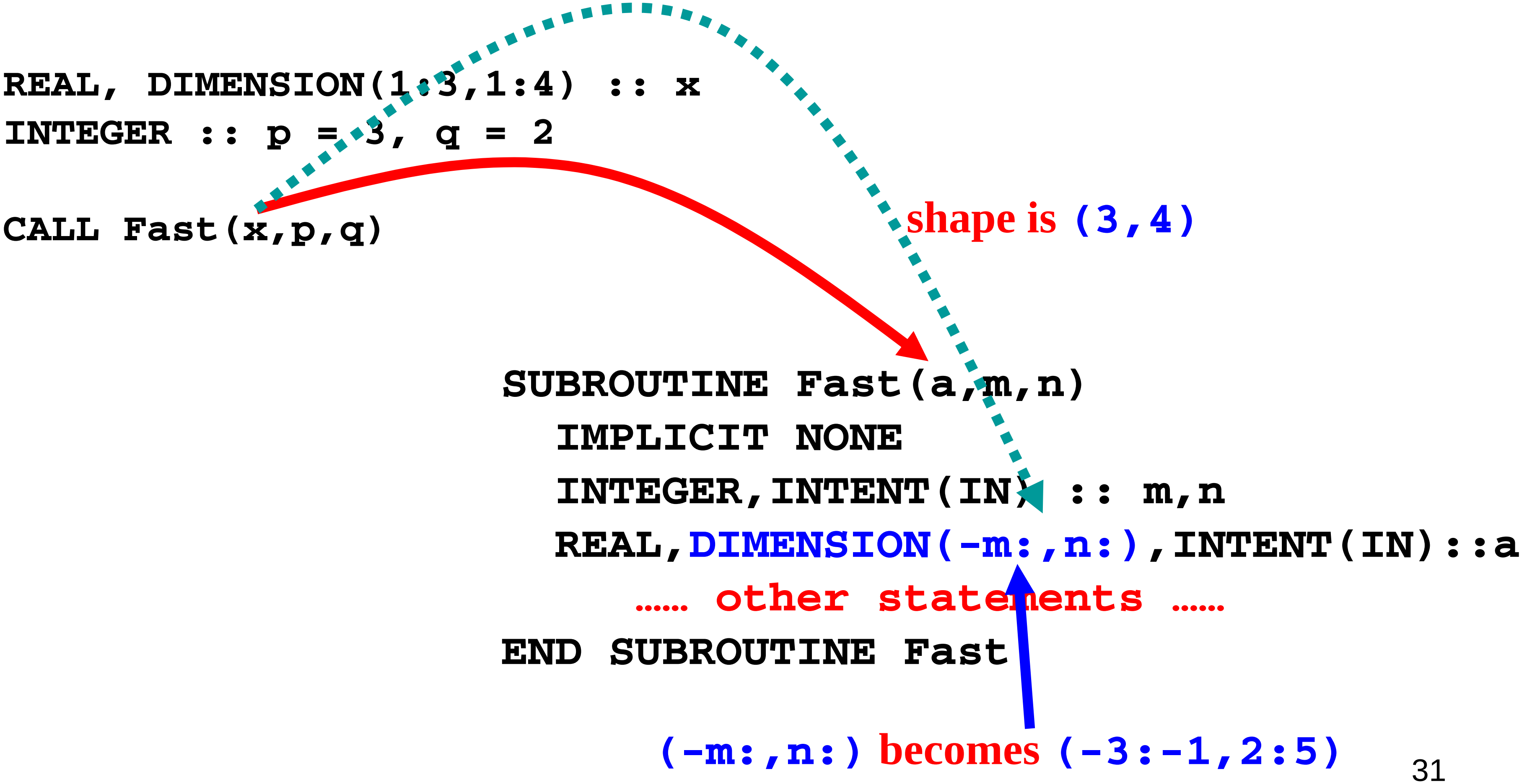
shape is `(9)`

```
SUBROUTINE Funny(x)
   IMPLICIT NONE
   INTEGER,DIMENSION(-1:),INTENT(IN) :: x
   …… other statements ……
END SUBROUTINE Funny
```

`(-1:7)`

Arrays as Arguments: 4/4

- **One more example**



```
REAL, DIMENSION(1:3,1:4) :: x
INTEGER :: p = 3, q = 2

CALL Fast(x,p,q)
```

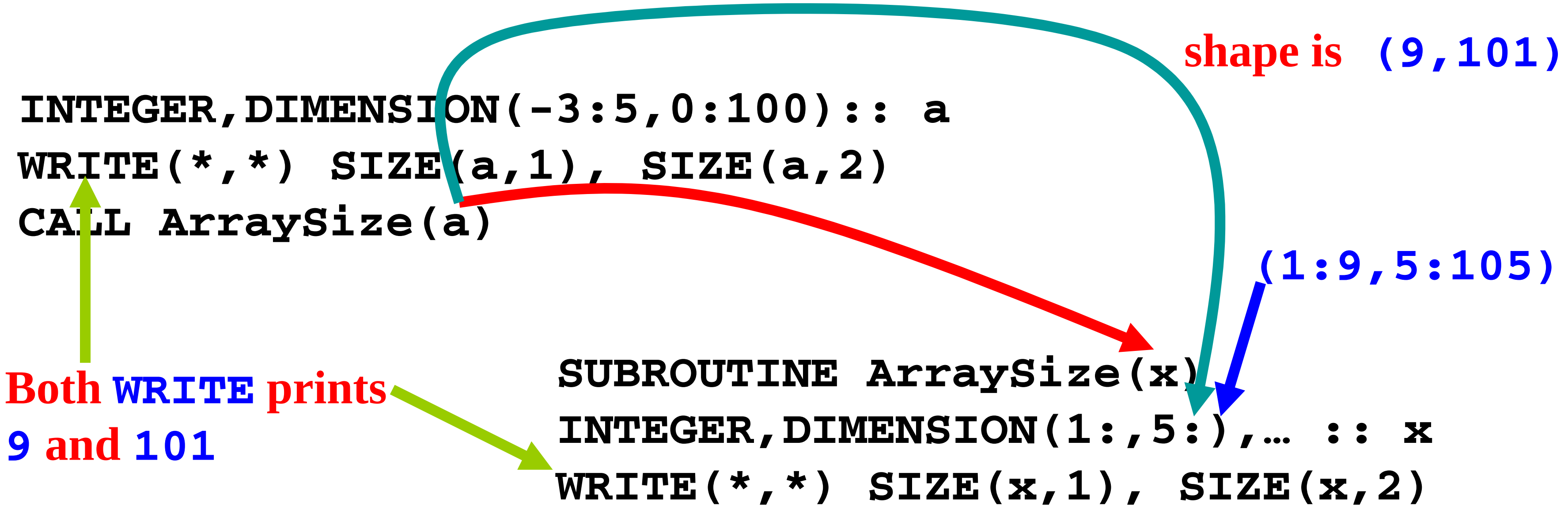
shape is `(3,4)`

```
SUBROUTINE Fast(a,m,n)
   IMPLICIT NONE
   INTEGER,INTENT(IN) :: m,n
   REAL,DIMENSION(-m:,n:),INTENT(IN)::a
      …… other statements ……
END SUBROUTINE Fast
```

`(-m:,n:)` becomes `(-3:-1,2:5)`

The `SIZE()` Intrinsic Function: 1/2

- How do I know the shape of an array?
- Use the `SIZE()` intrinsic function.
- `SIZE()` requires two arguments, an array name and an `INTEGER`, and returns the size of the array in the given "dimension."



```
INTEGER,DIMENSION(-3:5,0:100):: a
WRITE(*,*) SIZE(a,1), SIZE(a,2)
CALL ArraySize(a)
```

shape is `(9,101)`

`(1:9,5:105)`

```
SUBROUTINE ArraySize(x)
INTEGER,DIMENSION(1:,5:),… :: x
WRITE(*,*) SIZE(x,1), SIZE(x,2)
```

Both `WRITE` prints `9` and `101`

The **SIZE()** Intrinsic Function : 2/2

```
INTEGER,DIMENSION(-1:1,3:6) :: Empty
CALL Fill(Empty)
DO i = -1,1
  WRITE(*,*) (Empty(i,j),j=3,6)
END DO
```

shape is (3,4)

```
SUBROUTINE  Fill(y)
  IMPLICIT NONE
  INTEGER,DIMENSION(1:,1:),INTENT(OUT)::y
  INTEGER :: U1, U2, i, j
  U1 = SIZE(y,1)
  U2 = SIZE(y,2)
  DO i = 1, U1
    DO j = 1, U2
      y(i,j) = i + j
    END DO
  END DO
END SUBROUTINE Fill
```

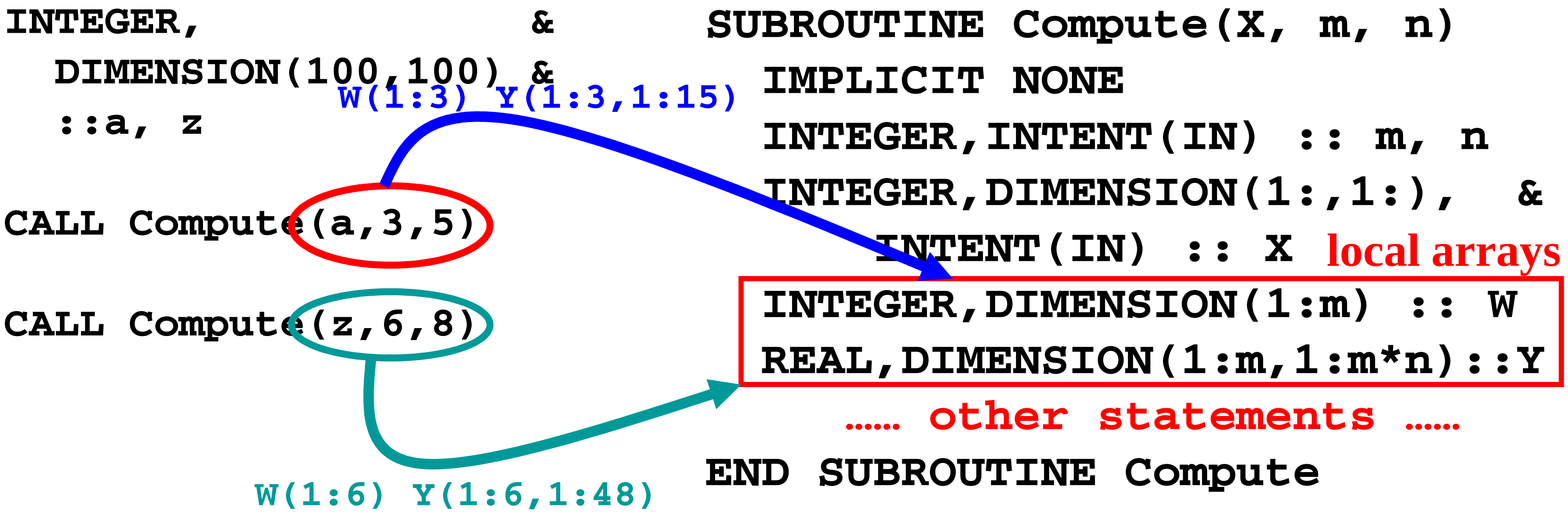
(1:3,1:4)

output

```
2 3 4 5
3 4 5 6
4 5 6 7
```

Local Arrays: 1/2

- **Fortran 90 permits to declare *local* arrays using `INTEGER` formal arguments with the `INTENT(IN)` attribute.**



```
INTEGER,                  &
  DIMENSION(100,100) &
  ::a, z

CALL Compute(a,3,5)

CALL Compute(z,6,8)
```

W(1:3) Y(1:3,1:15)

W(1:6) Y(1:6,1:48)

```
SUBROUTINE Compute(X, m, n)
  IMPLICIT NONE
  INTEGER,INTENT(IN) :: m, n
  INTEGER,DIMENSION(1:,1:),   &
      INTENT(IN) :: X
  INTEGER,DIMENSION(1:m) :: W
  REAL,DIMENSION(1:m,1:m*n)::Y
     …… other statements ……
END SUBROUTINE Compute
```

local arrays

Local Arrays: 2/2

- **Just like you learned in C/C++ and Java, memory of local variables and local arrays in Fortran 90 is allocated before entering a subprogram and deallocated on return.**
- **Fortran 90 uses the formal arguments to compute the extents of local arrays.**
- **Therefore, different calls with different values of actual arguments produce different shape and extent for the same local array. However, the rank of a local array will not change.**

The **ALLOCATBLE** Attribute

- **In many situations, one does not know exactly the shape or extents of an array. As a result, one can only declare a “large enough” array.**
- **The `ALLOCATABLE` attribute comes to rescue.**
- **The `ALLOCATABLE` attribute indicates that at the declaration time one only knows the rank of an array but not its extent.**
- **Therefore, each extent has only a colon `:`.**

```
INTEGER,ALLOCATABLE,DIMENSION(:)     :: a
REAL,ALLOCATABLE,DIMENSION(:,:)      :: b
LOGICAL,ALLOCATABLE,DIMENSION(:,:,:) :: c
```

The **ALLOCATE** Statement: 1/3

- The `ALLOCATE` statement has the following syntax:

  `ALLOCATE(array-1,…,array-n,STAT=v)`

- Here, `array-1`, …, `array-n` are array names with complete extents as in the `DIMENSION` attribute, and `v` is an `INTEGER` variable.
- After the execution of `ALLOCATE`, if `v` ≠ 0, then at least one arrays did not get memory.

```
REAL,ALLOCATABLE,DIMENSION(:) :: a
LOGICAL,ALLOCATABLE,DIMENSION(:,:) :: x
INTEGER :: status
ALLOCATE(a(3:5), x(-10:10,1:8), STAT=status)
```

The **ALLOCATE** Statement: 2/3

- **ALLOCATE** only allocates arrays with the **ALLOCATABLE** attribute.
- The extents in **ALLOCATE** can use **INTEGER** expressions. Make sure all involved variables have been initialized properly.

```
INTEGER,ALLOCATABLE,DIMENSION(:,:) :: x
INTEGER,ALLOCATABLE,DIMENSION(:)   :: a
INTEGER                            :: m, n, p
READ(*,*)  m, n
ALLOCATE(x(1:m,m+n:m*n),a(-(m*n):m*n),STAT=p)
IF (p /= 0) THEN
   …… report error here ……
```

If `m = 3` and `n = 5`, then we have `x(1:3,8:15)` and `a(-15:15)`

The **ALLOCATE** Statement: 3/3

- **`ALLOCATE` can be used in subprograms.**
- **Formal arrays are *not* `ALLOCATABLE`.**
- **In general, an array allocated in a subprogram is a local entity, and is automatically deallocated when the subprogram returns.**
- **Watch for the following odd use:**

```
PROGRAM Try_not_to_do_this
  IMPLICIT NONE
  REAL,ALLOCATABLE,DIMENSION(:) :: x
CONTAINS
  SUBROUTINE  Hey(…)
     ALLOCATE(x(1:10))
  END SUBROUTINE Hey
END PROGRAM Try_not_to_do_this
```

The **DEALLOCATE** Statement

- Allocated arrays may be deallocated by the `DEALLOCATE()` statement as shown below:

```
DEALLOCATE(array-1,…,array-n,STAT=v)
```

- Here, `array-1`, …, `array-n` are the names of allocated arrays, and `v` is an `INTEGER` variable.
- If deallocation fails (*e.g.*, some arrays were not allocated), the value in `v` is non-zero.
- After deallocation of an array, it is not available and any access will cause a program error.

```
DEALLOCATE(a, b, c, STAT=status)
```

The **ALLOCATED** Intrinsic Function

- The `ALLOCATED(a)` function returns `.TRUE.` if `ALLOCATABLE` array `a` has been allocated. Otherwise, it returns `.FALSE.`

```
INTEGER,ALLOCATABLE,DIMENSION(:) :: Mat
INTEGER :: status

ALLOCATE(Mat(1:100),STAT=status)
…… ALLOCATED(Mat) returns .TRUE. …..
…… other statements ……
DEALLOCATE(Mat,STAT=status)
…… ALLOCATED(Mat) returns .FALSE. ……
```

The End