CHAPTER 5
Array in Fortran Programming

Outline

- Array basics
- Array Declaration
- Dynamic Size Array Declaration
- Using Array elements
- Array Initialization
- Multidimensional Array
- Multidimensional Array Initialization
- Implied do loop
- Using whole arrays and array subset
- Array I/O
- Masked Array Assignment
- Arrays As Arguments in FORTRAN subroutine and function

Array basics

- Array is a collection of variables of the same type that are referred through a common name
- Arrays can be of any data type.
- Array is a simple structure, capable of storing many variables of the same type in a single data structure.
- Each element in an array can be accessed by an index (subscript)
- all array elements consist of contiguous memory location hence have ordered physical memory address

Array basics

- The lowest address (by default with index 1) corresponds to the first and the highest (index value SIZE) corresponds to the last element of the array
- The subscript can be expressed as an integer constant or expression that can be evaluated to integer value
- It can have from one to several dimension

Array basics

- Array is very power full structure in Fortran since
  1. It provide a mechanism to apply the same algorithm for a number of data of the same type by using looping technique

     for example, computing the square root of 100 positive integer
  2. It enable us to manipulate and perform calculation with individual elements of array one by one, with a whole arrays at once, or with various subsets of the array

Array Declaration

- Before using any array, the array variable has to be declared
- The declaration should indicate at least the data type of the elements and the number of the elements in the array.
- The array elements may have any of the data types possible in Fortran like INTEGER, REAL, LOGICAL, CHARACTER, COMPLEX, etc

Array Declaration

- The basic form of declaring an array variable **ArrVariable** which has **SIZE** elements in which each element with data type of **DType** is shown bellow

Dtype, **DIMENSION(SIZE)**:: **ArrVariable**

- **Example**

  REAL, *DIMENSION(**100**)::**salary***

  CHARACTER(len=20) DIMENSION(***100***)::***Name***

- Array elements will be accessed via the group variable name followed by a subscript enclosed by parenthesis

Array Declaration

- ArrVariable(i) refers to one of the element with subscript i
- The range of the subscript is from 1 to SIZE by default
- Hence, the first element for salary and Name can be referenced as salary(1) and Name(1) respectively
- Similarly the ith element is salary(i) and Name(i) respectively
- The number of elements of the array variable defines the **extent** of the array.

Array Declaration

- Array can be declared by stating the starting and ending index (subscript range)
- This may be important to give identical subscript as real the application refers to
- For example, to describe events which happened at the past (-ve index), present (zero index), the future events (+index)

Array Declaration

The Syntax

Dtype, DIMENSION**(START:END)::**ArrVariable

- ➢ This declares ArrVariable as an array of End-Start+1 elements in which the first and last element index is start and end respectively

Dtype, DIMENSION**(SIZE)::**ArrVariable

- ➢ In this case Start is 1 and End is SIZE

- ➢ **START**, **END** and/or **SIZE** information can be described using integer constant or named constant defined using the PARAMETER statement

Array Declaration

Examples

```
LOGICAL, DIMENSION (-1:1)::status
COMPLEX, DIMENSION(-100:100)::fourierTransform
INTEGER::DIMENSION(0:200)::myData
REAL::DIMENSION(200)::Salary
integer, Parameter:: size=100
real, dimension(size)::tax
```

Dynamic Size Array Declaration

- Static allocation of memory have some limitations during execution of the program
- Some of the limitations include
    - The array may fail to hold the entire data the user have at run time (under estimation of array size)
    - The array may under utilizes the allocated memory space (which is not a problem but may lead the computer to end up with shortage of memory for later allocation)
- Array can have dynamic size whose number of elements will be determined while the program is running
- This allows us allocate sufficiently large memory size as required dynamically
- Dynamic memory allocation allow us to get the extent dynamically from inputs and use it to define the array structure

Dynamic Size Array Declaration

- Dynamic allocation of array memory space is possible using the allocate and deallocate command

Syntax during declaration

```
Dtype, dimension(:), allocatable :: ArrVariable
```

Syntax during allocation

```
Allocate ( ArrVariable(size_expression))
```

Syntax during deallocation

```
Deallocate (ArrVariable)
```

Dynamic Size Array Declaration

```
program ArrayTest
    implicit none
    integer, Parameter:: size=100
    integer:: max
    real, dimension(100):: salary
    real, dimension(size)::tax
    real, dimension(:), allocatable::data
    print *, "Enter max "
    read *, max
    allocate (data(max+1))
    !work using the three variables here
    deallocate (data)
end program
```

Using Array elements

- Array elements are ordinary variable just as any other variable

- Hence, we can use them in an ***input*** statement, ***output*** statement, as an ***operand*** in the expression, as ***lvalue*** in assignment statement, etc

- Moreover, array can be used as a group in Fortran programming.

- Note that, while accessing an array element by subscript, you should be careful to use only valid index otherwise, it will result out of bound error

Array Initialization

- Array must be initialized to be used
- Simple declaration of array as we have seen before never initialize known value for its elements
- Hence, we should make sure that the array which is going to be processed has relevant value before attempting any operation on it
- Array can be initialized in various ways

Array Initialization

- Some of the initialization techniques include
  - ***Initialization during declaration***
  - ***Initialization by assignment statement***
  - ***initialization using the READ statement***

Array Initialization during declaration

- Initial value may be loaded into the array at compilation time by declaring their values in type declaration statements
- Hence the values will be loaded at compilation time
- The syntax for assigning an array variable during declaration is

  Dtype, DIMENSION(SIZE)::ArrVariable=ArrayConstant

OR using a DATA statement

- Array constant is a comma separated list of constants enclosed by opening and ending deliminators **(/** and **/)** respectively

Array Initialization during declaration

- The lists can be expressed as a ***list of constant values*** of the target array element values or using ***implied do loops***
- Example from the first type
  - ***INTEGER, DIMESION(10)::data = (/ 1,2,3,4,5,6,7,8,9,10 /)***

- Initializing, this way an array of very large size can be made using implied do loop

Array Initialization during declaration

- Implied do loop
  - It is a structure that loop for a number of times
  - Each time its counter will change by the step value
  - A set of expression that depends on the counter produce their outputs
  - At the end of the iteration, the list of the outputs generated will be used as an initialization list
  - The general syntax of an implied do loop is

    (arg1, arg2, ..., index = istart, iend, increment)
  - For each value of index from istart to iend with a step of increment, arg1, arg2,... will be generated and used to suit for their purpose in the statement

Array Initialization during declaration

- **Implied do loop example**

  INTEGER, DIMENSION(100):: data=(/ (i,i=1,100,1) /)

  - will initialize the 100 elements from 1 to 100 respectively
  - the implied loop iterate 100 time and generate i each time and keep each value within the array constant deliminators
  - hence can be used for its purpose (i.e. initializing the array)

- Other purpose of implied do loop will be discussed soon

Array Initialization during declaration

```
program ArrayTest
    implicit none
    integer:: i
    integer, dimension(100):: salary=(/ (i*1.0,(i*i)*1.0, i=1,50,1) /)
    character (len=9), dimension(7), parameter:: weekDays= &
    (/    'Sunday   ', 'Monday   ', 'Tuesday  ', 'Wednesday', &
          'Thursday ', 'Friday   ', 'Saturday ' /)
    do i =1, 10
          print *, salary(i)
    end do
    do i =1, 7
          print *, weekDays(i)
    end do
end program
```

Array Initialization by DATA statement

- Data statements is used to initialize variables and arrays at the time of declaration.

- The form of the data statement is

  DATA list-of-variables /list-of-values/, ...

Array Initialization by DATA statement

```
program ArrayTest
    implicit none
    integer:: i
    integer, dimension(10):: digits
    character (len=9), dimension(7):: weekDays
    data digits/0,1,2,3,4,5,6,7,8,9/, weekDays /'Sunday   ', 'Monday   ', &
                'Tuesday  ', 'Wednesday', 'Thursday ', 'Friday   ', 'Saturday '/
    do i =1, 10
        print *, digits(i)
    end do
    do i =1, 7
        print *, weekDays(i)
    end do
end program
```

Array Initialization by assignment statement

- The other way of assigning an array is via assignment statement
- The assignment statement usually integrated with looping structures
- The following statement shows declaration followed by initialization by assignment statement

```
INTEGER,DIMENSION(100):: data
    DO I=1,100
        data(i) = i
    END DO
```

- Initialize data elements from 1 to 100

Array Initialization by assignment statement

- The above initialization is equivalent to the following

```
INTEGER,DIMENSION(10):: data
data = (/ 1,2,3,4,5,6,7,8,9,10 /)
```

OR

```
INTEGER,DIMENSION(10):: data
data = (/ (i, i = 1,10) /)
```

Array Initialization by reading statement

- Another way of initializing an array in Fortran is through assigning the values directly from the input device by reading

```
INTEGER, DIMENSION(10)::data
read(*,*) (data(i), i=1,10)
```

OR

```
INTEGER, DIMENSION(10)::data
DO i=1,10
    read(*,*) data(i)
END DO
```

EXERCISE

1. write a program that reads 100 numbers from the file (create your own file) and sort them and rewrite them on a new file

1. Write a program two equal dimensional vectors of size 100 each from a file and produce the correlation of them on the separate file

1. Write a program that takes two vectors of the same dimension from the file and compute their dot product, cross product, norm of their sum and their difference. Do it in the most efficient way

1. Write a program that implements the discrete fourier transform of a given sequence of data in a file. The file first entry is an integer that shows the non zero sequence elements that exist in the file

1. Write a program that will check whether a given N-side polygon is concave or not (you can keep the information that you may need in the file in the format that you like)

Multidimensional Array

- Array in Fortran can have more than a single dimension
- Two dimensional array is common in science to manipulate matrix (tables) and related information
- Higher order array is also common in theoretical science and social science field
- For example, we need to keep information about the temperature of a place at different amplitude, latitude, longitude, season, moisture and wind
- Hence it worth to know how to define N dimension array in Fortran programming

Multidimensional Array

- To define multidimensional array variable, we only need to specify the sizes information for all the dimensions in the DIMENSION statement

- For example, a matrix of size 4x8 with each element having data type integer can be defined as

```
INTEGER, DIMENSION(4,8):: matrix
```

- Four dimensional array in which each dimension having the same extent 10 and each element is a real number can be defined as

```
REAL, DIMENSION(10,10,10,10)::values
```

- The elements can be accessed by specifying the appropriate subscript for each dimension

Dynamic Multidimensional Array

- You can declare allocatable N-Dimensional array as

Dtype DIMENSION(:,:,:, ...), allocatable ::ArrVariables

Example

Integer DIMENSION(:,:), allocatable ::MyTable

Dynamic Multidimensional Array

- **To allocate memory to N-Dimensional array**

Allocate(ArrVariable(range1, range2, ...))

where range is start:end, or size (to mean 1:size)

Example:

Allocate(MyTable(0:10, 0:20))

!Note the possible range values can be any expression that can be evaluated to integer

Dynamic Multidimensional Array

- **To de-allocate the allocated memory**

Deallocate(arrVariable)

Example:

Deallocate(MyTable)

Multidimensional Array

- Example that shows how to declare, allocate and de-allocate two dimensional memory dynamically

```
program ArrayTest
    implicit none
    integer, dimension(:,:), allocatable::data;
    allocate(data(4,3));
    !we can use data as any other two dimensional memory
    print *, shape(data);
    :
    :
    deallocate(data);
end program
```

Multidimensional Array

- The elements are ordered in the RAM in a single dimensional space, column wise (the first column first and the second follows and so on)

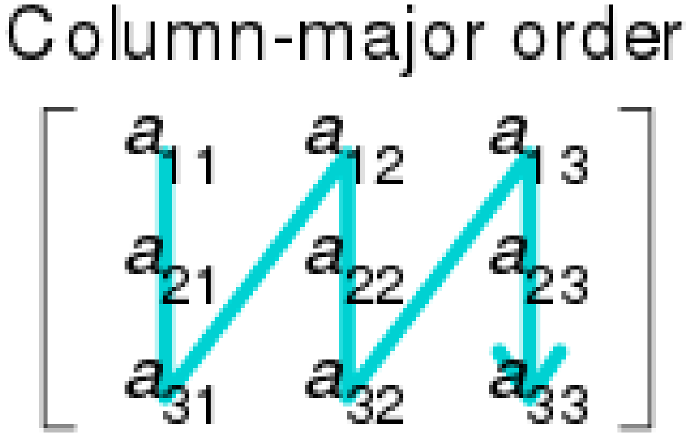
- For example, the first element in the matrix definition is matrix(1,1) and the last is matrix(3,3)

- Hence, the ordering or the array above become

(1,1)→(2,1)→(3,1) →

(1,2)→(2,2)→(3,2) →

(1,3)→(2,3)→(3,3) →


Column-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

Multidimensional Array

- The rank of an array is the number of dimension an array definition has.
- One dimensional array has rank of 1, two dimensional array has rank of two, and so on.
- The shape of an array is defined by the rank and the extent of the array
- The array defined bellow has
-         -an extent value 20x10
-         -a rank value 2
-         -a shape of 2x20x10

```
REAL, DIMENSION(20,10)::Table
```

Multidimensional array initialization

- Like the one dimensional array, we can initialize two or more dimensional array
  - *By assignment statement (the same with 1D)*
  - *By read statement (the same with 1D)*
  - *at the time of declaration*
    - *Using data statement*
    - *Using listing methods with reshape function*
      - *With out implied do loop*
      - *With implied do loop*

Multidimensional array initialization

Using Assignment statement

```
program ArrayTest
    implicit none
    integer:: i,j
    integer, dimension(2,3):: salary, temp
    do i =1, 2
        do j =1, 3
            salary(i,j) = i * j
        end do
    end do
    temp = salary;
    do i =1, 2
        do j =1, 3
            print *, temp(i,j)
        end do
    end do
end program
```

Multidimensional array initialization

Using Read statement (Type 1)

```
program ArrayTest
    implicit none
    integer:: i,j
    integer, dimension(2,3):: salary
    do i =1, 2
        do j =1, 3
            read*, salary(i,j)
        end do
    end do
    do i =1, 2
        do j =1, 3
            print *, salary(i,j)
        end do
    end do
end program
```

Multidimensional array initialization

Using Read statement (Type2)

```
program ArrayTest
    implicit none
    integer:: i,j
    integer, dimension(2,3):: salary
    read *,  (( salary(i,j), j = 1,3), i =1,2)
    do i =1, 2
         do j =1, 3
              print *, salary(i,j)
         end do
    end do
    print *, salary
end program
```

Multidimensional array initialization during declaration

Using data statement (Type 1)

```
program ArrayTest
    implicit none
    integer:: i,j
    integer, dimension(2,3):: salary
    data salary/1,2,3,4,5,6/
    do i =1, 2
        do j =1, 3
            print *, salary(i,j)
        end do
    end do
End
 program
!output 1,3,5,2,4,6
```

Multidimensional array initialization during declaration

Using data statement (Type 2)

```
program ArrayTest
    implicit none
    integer:: i,j
    integer, dimension(2,3):: salary
    data salary/6*1/
    do i =1, 2
         do j =1, 3
              print *, salary(i,j)
         end do
    end do
end program
```

Output

```
1
1
1
1
1
```

Multidimensional array initialization during declaration

- ***Using listing methods with reshape function***
  - The RESHAPE function takes two argument array constant list and shape information
  - The form of the RESHAPE function is

    RESHAPE(ArrayConstantList, shape)

  - Where the array constant list is enclosed in (/ and /) delimiters and the shape is specified by the list of extents to each dimension separated by comma and enclosed in (/ and /)
  - The rank will be automatically computed from the extent list

Multidimensional array initialization during declaration

- *Using listing methods using reshape function*
  - The array constant list can be generated by the implied do loop or just manually with out the implied do loop

  - The following are some examples of the reshape statement and its effects

INTEGER, **DIMESION(4,3)**::data = **RESHAPE((/** 1,1,1,1,2,2,2,2,3,3,3,3 **/)** , **(/** 4, 3 **/))**

  - The reshape statement will change the shape of the array constant from 1x12 into 4x3 which is **conformable** to the array variable

Multidimensional array initialization during declaration

- *Using listing methods with reshape function*

```
integer, dimension(40,3)::data = &
    RESHAPE((/ ((j, i=1,40), j=1,3) /) , (/ 40,3 /))
```

- This generate i 40 times for each j which runs from 1 to 3

- Hence data becomes a two dimensional array with dimension 40x3

Multidimensional array initialization during declaration

➢*The following is a programming example that initialize two dimensional array using listing methods using reshape function with out implied do loop*

```
program ArrayTest
    implicit none
    integer:: i,j
    integer, dimension(4,3)::data = &
            reshape( (/1,2,3,4,5,6,7,8,9,10,11,12/), (/4,3/))
    do i =1, 4
        print *, (data(i,j), j=1,3)
    end do
end program
```

Multidimensional array initialization during declaration

- *The following is a programming example that initialize two dimensional array using listing methods using reshape function with a simple implied do loop*

```
program ArrayTest
       implicit none
       integer:: i,j
       integer, dimension(40,30)::data = &
              RESHAPE(  (/ (j, j = 1,1200) /) , (/40,30/) )

       do i =1, 10
              print *, (data(i,j), j=1,10)
       end do

end program
```

Multidimensional array initialization during declaration

- *The following is a programming example that initialize two dimensional array using listing methods using reshape function with a nested implied do loop*

```
program ArrayTest
     implicit none
     integer:: i,j

     integer, dimension(4,3)::data = &
               reshape( (/ ((i, j = 1,4), i=1,3) /), (/4,3/) )

     do i =1, 4
          print *, (data(i,j), j=1,3)
     end do
end program
```

Implied do loop

- Implied do loops are important construct in Fortran programming to generate inputs to initialize array during declaration

- Moreover, they can be used in input and output statements

- The following are some more examples and their interpretation

Implied do loop

- Example one:

```
program ArrayTest
    implicit none
    integer:: i,j
    integer, dimension(4,3)::data = &
                reshape( (/ ((i, j = 1,4), i=1,3) /), (/4,3/) )
    do i =1, 4
         print *, (data(i,j), j=1,3)
    end do
end program
```

Output

```
1               2          3
1               2          3
1               2          3
1               2          3
```

Implied do loop

- Example two:

```
program ArrayTest
    implicit none
    integer:: i,j
    integer, dimension(4,3)::data = &
                reshape( (/ ((i, j = 1,4), i=1,3) /), (/4,3/) )
    print *,  ((data(i,j), j = 1,3), i =1,4)
end program
```

Output

```
1  2   3   1   2   3   1   2   3   1   2   3
```

Implied do loop

- Example three:

```
program ArrayTest
    implicit none
    integer:: i,j
    integer, dimension(4,3)::data
    read *,  (( data(i,j), j = 1,3), i =1,4)
    print *,  ((data(i,j), j = 1,3),  '<--->',i =1,4)
end program
```

Input

```
0    1    2    3    4    5    6    7    8    9    10   11   12
```

Output

```
0    1    2    ←→      2    3    4    ←→      5    6    7    ←→      8    9    10
```

Implied do loop

- Example four:

```
program ArrayTest
    implicit none
    integer:: i,j
    integer, dimension(4,3)::data
    read *,  (( data(i,j), j = 1,3), i =1,4)
    print *,  data
end program
```

Input

```
1   2   3   4   5   6   7   8   9   10   11   12
```

Output

```
1   4   7   10   2   5   8   11   3   6   9   12
```

Hence The input was organized as

```
1      2      3
4      5      6
7      8      9
10     11     12
```

Using whole arrays and array subset

- One of the most important property of Fortran for Engineers and Scientists is the capability of processing the whole or section of the array at once

- We can use the whole array in arithmetic operations

- Arithmetic operation on a two array variable of the same shape is possible

- This is equivalent to arithmetic that is made element by element

Using whole arrays and array subset

- Practice your self by adding, multiplying, subtracting, and diving two array of the same shape

- Practice also your self by applying exponentiation of one array with some scalar constant

Using whole arrays and array subset

- Two arrays can also be used in assignment statement if they have the same shape and element type.

- Two arrays of the same shape are called **conformable**

- Two array may not have the same subscript range to be conformable.

    -Scalar constants and scalar variables are conformable with array.

- In this case the scalar operation will be applied to each of the elements of the array

- Try scalar operation in the same fashion as you did on the previous exercise

Using whole arrays and array subset

- Many of the Fortran intrinsic functions that are used with scalar values also accept arrays as input argument and return arrays as a result
- Example

```
INTEGER :: X = -10, Y
INTEGER, DIMENSION(2,2)::data = RESHAPE( (/-1,2,3,-4/), (/ 2,2 /) )
INTEGER, DIMENSION(2,2)::value

Y = abs(X)             !returns 10
Value = abs(data)      !returns
                                  1          3
                                  2          4
```

Using whole arrays and array subset

- One can also perform operation on the subset of the array element
- A subset of an array is called an array section
- Array section can be specified by placing an array subscript with **subscript triplet** or **vector** subscript
- A subscript triplet has a form

  start:end:step

- start is the first subscript to be included, end is the last subscript to be included, step is the filter for unwanted elements to jump in the sequence

Using whole arrays and array subset

```
INTEGER, DIMENSION(100):: data
INTEGER, DIMENSION(4)::section
```

```
section = data(1:10:3)
```

assign the variable section to the subsection of data with subscript index 1, 4, 7, 10

A subscript triplet can be written in one of the following form

Using whole arrays and array subset

- A subscript triplet can be written in one of the following form

  start:end:step

  start:end

  start:

  start : : step

  :end

  :end:step

  : : step

- if not provided, step is one, end is the last subscript of the array, start is the first subscript of the array in that dimension.

Array I/O

- Array I/O can be made using
  - ***Element by element process***
  - ***The implied do loop***
  - ***I/O of whole arrays and array subsections***
- Element by element and the implied do loop I/O is already discussed

Array I/O

- The implied do loop I/O can have the following form

  write(unit, format) (arg1,arg2,...,index = istart, iend, increment)

  read(unit, format) (arg1,arg2,...,index = istart, iend, increment)

- Write a program that read a file which contains 10 vectors each having 10 elements. Then your program should display each vector line by line on the screen

Array I/O

- Array can also be read and write in its entirety or section of it if necessary

- The following code segment gives clarification on it

```
INTEGER, DIMENSION(20,10):: data
read (*,*) data              write(*,*) data
read(*,*) data(1,:)          write(*,*) data(1,:)
read(*,*) data(1,1:5)        write(*,*) data(1,1:5)
```

- etc

Masked Array Assignment

- It is a mechanism to perform elemental function in which the type of operation will be masked by conditional statements

Syntax

[name:]   WHERE (mask expression1)

  – Array Assignment statement 1

- ELSEWHERE (mask expression2)

  – array assignment statement 2

- END WHERE [name:]

Masked Array Assignment

Example

```
real, dimension(4,2):: data = reshape ( (/ -1,0,1,-1,-2,0,1,-3/), (4,2))

WHERE (data > 0.)
    logvalue = LOG(data)
ELSEWHERE
    logvalue = -999999
END WHERE
```

Arrays As Arguments in FORTRAN subroutine and function

- If a programmer wishes to pass an *entire* array, only the name of the array *without* any subscripts is listed, both in the calling statement and in the procedure code.
- Nevertheless, (except for internal subprograms without parameters) the array must be declared as an array **both** in the calling program and inside the subprogram.
- As an example, the following program sums the values stored in an array of size 10.

Arrays As Arguments in FORTRAN subroutine and function

```
PROGRAM MAIN
        REAL :: B(10),TOTAL
        data b/1,2,3,4,5,6,7,8,9,10/
        CALL SUMARR(B,TOTAL)
        print *, "The sum = ", total
END

SUBROUTINE SUMARR(A,SUM)
        REAL, dimension(10) :: A
        real, intent(out):: SUM
        INTEGER I
        SUM = A(1)
        DO I=2,10
              SUM=SUM+A(I)
        END DO
        RETURN
END
```

Arrays As Arguments in FORTRAN subroutine and function

- A programmer may want to use the *same* subprogram for arrays of *different* sizes.

- This can cause problems since some Fortran compilers demand that the formal and actual arrays be dimensioned exactly the same.

- To solve this dilemma, two general schemes are used
  - The older scheme involves passing the array dimension as one of the arguments
  - The newer scheme makes use of Fortran-90/95/2003 assumed-shape arrays

Arrays As Arguments in FORTRAN subroutine and function

- In the first scheme, in the subprogram the array used as an argument is declared with a variable as the upper limit, for example,

REAL :: A(N) where *both* A and N are arguments passed into the subprogram.

- For two and higher dimensional arrays, all the subscripts may be variables, but all these variables *must* be arguments passed from the calling program

Arrays As Arguments in FORTRAN subroutine and function

```
PROGRAM MAIN
        REAL :: B(10),TOTAL
        data B/1,2,3,4,5,6,7,8,9,10/
        CALL SUMARR(B,TOTAL,10)
        print *, "The sum = ", total
        CALL SUMARR(B(3:9),TOTAL,7)
        print *, "The sum = ", total
        CALL
        SUMARR(B(1:10:2),TOTAL,5)
        print *, "The sum = ", total
END
```

```
SUBROUTINE SUMARR(A, SUM,N)
    integer:: N
    REAL, dimension(N), intent(in):: A
    real, intent(out):: SUM
    INTEGER I
    SUM = A(1)
    DO I=2,N
            SUM=SUM+A(I)
    END DO
    RETURN
END
```

Arrays As Arguments in FORTRAN subroutine and function

- Observe the calling module calls the subroutine with various section of the array by properly specifying the number of elements

- Fortran-90/95/2003 has introduced an alternative to this method of indicating the dimension of array used as arguments.

- Such an array is called an **assumed-shape** array and is declared by using a colon (or colon with a preceding initial dimension value) in the declaration for each dimension of the array.

Arrays As Arguments in FORTRAN subroutine and function

- For example, if D is an three-dimensional array being passed to a subroutine SUB1, the array could be declared as follows:

```
SUBROUTINE SUB1(D)
INTEGER, DIMENSION (:,:,:) :: D
```

- In a case such as this, array D receives the exact values of the three dimensions from the program segment which invokes SUB1.

- This is done at compile time when the compiler checks the array used as the actual argument.

Arrays As Arguments in FORTRAN subroutine and function

- This method of generalizing the size of the array being passed to the subprogram forces the compiler to determine additional information it does not have from the code itself

- As a result, the subprogram code must be integrated with the calling program in special ways.

- This may be done either by making such a subprogram an internal subprogram or by using a module, or by including an interface block if the subprogram is written in external file

Arrays As Arguments in FORTRAN subroutine and function

- One can includes an interface block in the calling program segment by placing it *after* any variable declarations.

- The interface itself only contains the header line of the subprogram, declaration statements for the arguments, and the subprogram END statement.

Example

Arrays As Arguments in FORTRAN subroutine and function

```
PROGRAM SAMPLE1
        INTEGER, DIMENSION (4, 5, 6) :: A
        ...
        INTERFACE
              SUBROUTINE SUB1(D)
                    INTEGER, DIMENSION (:,:,:) :: D
              END SUBROUTINE
        END INTERFACE
        ...
        CALL SUB1(A)
        ...
END PROGRAM
```

Arrays As Arguments in FORTRAN subroutine and function

- An example of a subprogram which uses assumed shaped arrays as well as array-handling features of Fortran-90/95/2003 is the following subroutine which exchanges the values contained in two one-dimensional arrays (i.e., vectors).

```
SUBROUTINE SWAP(A,B)
        REAL, DIMENSION(:) :: A, B
        REAL, DIMENSION(SIZE(A)) :: WORK
        WORK = A
        A = B
        B = WORK
END SUBROUTINE
```

Arrays As Arguments in FORTRAN subroutine and function

```
PROGRAM MAIN
        INTERFACE
        SUBROUTINE SUMARR(A, SUM)
              REAL, dimension(:), intent(in):: A
              real, intent(out):: SUM
              INTEGER I
        END
        END INTERFACE
        REAL :: B(10),TOTAL
        data b/1,2,3,4,5,6,7,8,9,10/
        CALL SUMARR(B,TOTAL)
        print *, "The sum = ", total
        CALL SUMARR(B(3:9),TOTAL)
        print *, "The sum = ", total
        CALL SUMARR(B(1:10:2),TOTAL)
        print *, "The sum = ", total
END
```

```
        SUBROUTINE SUMARR(A, SUM)
            real, intent(out):: SUM
            REAL, dimension(:), intent(in):: A
            REAL, dimension(1):: s
!s is the shape of the input matrix which is
!1XDimension of the martix
            integer:: n,i
            s = shape(A);
            N= s(1)
!N is the number of elements in the vector
            print *,"Number of elements = ",N
            sum = 0
            DO I=1,N
                        SUM=SUM+A(I)
            END DO
            RETURN
        END
```