Handout Seven

November 19, 2013

1 Third look at arrays

You have seen how to declare arrays and you have made use of 2D ‘`REAL`’ arrays to accommodate matrices in your matrix library module, where the first dimension indexes the ‘rows’ of the matrix and the second dimension indexes the ‘columns’.‘Handout Five’ looked at array terminology size, rank, extent, shape and conformable. Referencing arrays and array construction were also introduced.

1.1 1D Array Construction

In ‘Handout Five’ we looked at assigning values to a one dimensional array using array constructors. This approach is revisited in this section to demonstrate other types of array constructors. Consider the example below.

```
PROGRAM arraycon
!** Examples of array construction

  IMPLICIT NONE

  INTEGER, PARAMETER :: a=3, b=4, c=6,  d=b+c
  INTEGER, DIMENSION(a) :: arr1
  INTEGER, DIMENSION(b) :: arr2
  INTEGER, DIMENSION(c) :: arr3
  INTEGER, DIMENSION(d) :: arr4
  INTEGER :: i

  arr1=(/1,2,3/)
  arr2=(/4,5,6,7/)

  !** arr3 constructed with elements from arr2
  arr3=(/arr2(2),arr2(3),arr2(4),8,9,10/)


  PRINT*,"arr1 : ",arr1
  PRINT "(2/)" !** Print two blank lines

  PRINT*,"arr2 : ",arr2
  PRINT "(2/)"

  PRINT*,"arr3 : ",arr3
  PRINT "(2/)"

  !** constructed  from the above three arrays
  !** with examples of array ranges

```

```
32    arr4=(/arr1,arr2(1:2),arr3(2:6)/)
33
34   PRINT*,"1st construct gives : ",arr4
35   PRINT "(2/)"
36
37   !** constructor inside an array constructor
38
39   arr4=(/arr1,(/4,5,6,7/),arr3(4:6)/)
40
41   PRINT*,"2nd construct gives : ",arr4
42
43 END PROGRAM arraycon
```

Exercise One : In a '`handout7/exercise1`' directory type in the above code and check you understand how the arrays are constructed by running the code and examining the results. Lines '`13`' and '`14`' show array constructors you have seen before. Line '`17`' demonstrates the use of individual elements of arrays to construct a new array. Lines '`32`' and '`39`' show the use of whole arrays and sub-sections of arrays (array strides) to construct a new array.

1.2 2D Array Construction and the'`RESHAPE`' Intrinsic Function

The above section is limited for arrays of rank one. For arrays of higher dimensions the '`RESHAPE`' intrinsic function must be used in conjunction with the above. The '`RESHAPE`' function takes as its first argument the 'source' array and as its second argument it takes a rank one array whose elements dictate the required shape of the array to be returned. So for example the code

```
INTEGER, DIMENSION(10) :: aa !** 1D array
REAL, DIMENSION(2,5) :: bb
aa=(/1,2,3,4,5,6,7,8,9,10/)

bb=RESHAPE(aa,(/2,5/))
CALL outmat(bb)  !*** Your own "outmat" subroutine
```

the output of the above code would be

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

The important points to note here are that, as requested, the result matrix has two rows and five columns. The one dimensional array '`aa`' has been reshaped and then assigned to the array '`bb`'. This has been done by filling in the the first column then the second then the third and so on this is referred to as 'column major'. There are actually two optional arguments to the '`RESHAPE`' function we will only look at one of them and that is the keyword argument '`ORDER`'. The default order is '`ORDER=(/1,2/)`' and results in the the 'column major' ordering. The reverse is specifying '`ORDER=(/2,1/)`' and would result in the source array being copied into the result array filling in the result array row by row, this is referred to as 'row major'. Consider the following bit of code.

```
INTEGER, DIMENSION(10) :: aa !** 1D array
REAL, DIMENSION(2,5) :: bb
aa=(/1,2,3,4,5,6,7,8,9,10/)

bb=RESHAPE(aa,(/2,5/),ORDER=(/2,1/))
CALL outmat(bb)
```

the output of the above code would be

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

Exercise Two : In a '`handout7/exercise2`' directory write a piece of code to create a rank one array of sixteen '`REAL`' numbers. Construct the array so that it holds the numbers one to sixteen in numeric order, ie. index one holds the number one and index two holds the number to etc. Then '`RESHAPE`' the rank one array into a rank two array to represent the matrix below.

$$\begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix}$$

Now use the rank one array and '`RESHAPE`' it into the transpose of the above matrix. Check your answers using your own '`transmat`' function.

Class Project Five (I) : A small addition to your library module. Add another transpose function so that it takes the transpose of a matrix **without** using any '`DO`' loops and does the transpose operation itself in one line using the reshape function. Call this new version of your '`transmat`' function '`transmat2`'.

1.3 Some Array Intrinsics

- '`MAXLOC(<source-array>)`' : Accepts as its argument an array and returns an array of rank one. The number of elements in the return array is equal to the rank of the source array argument. Each element of the return array holds the index for the corresponding rank of the source array which together index the element of largest numeric value.

$$\text{aa} = \begin{pmatrix} 1 & 2 & 9 & 13 \\ 4 & 6 & 1 & 4 \\ 3 & 2 & 6 & 15 \\ 8 & 8 & 32 & 3 \end{pmatrix}$$

Then the statement ‘ind=MAXLOC(aa)’ will put the array ‘(/4,3/)’ into the rank one, two element array ‘ind’. This is because the maximum value in the array ‘aa’ is ‘32’ and it is in row four and column three. So ‘PRINT*,aa(ind(1),ind(2))’ would print the value ‘32’.

- ‘MINLOC(<source-array>)’ : The same as ‘MAXLOC’ except that it returns the location of the minimum value.
- ‘MAXVAL(<source-array>)’ : Returns the maximum value in the source-array, note the result will be a scalar of the same type as the source array.
- ‘MINVAL(<source-array>)’ : Returns the minimum value in the source-array, note the result will be a scalar of the same type as the source array.
- ‘SUM(<source-array>)’ : Returns the sum of ALL the elements in the source-array, note the result will be a scalar of the same type as the source array.
- ‘PRODUCT(<source-array>)’ : Returns the product of ALL the elements in the source-array, note the result will be a scalar of the same type as the source array.

Class Project Five (II) : In your Matrix library code, if you have not already done so, use the ‘MAXVAL’ intrinsic function to find the ‘maximum’ value of a vector in the function you wrote to calculate the infinity norm. HINT you will also need to use the ‘ABS’ function. Also in your ‘power method’ procedure, in order to calculate the dominant eigenvalue, you were told to use the any index ($i \in \{1, 2, ..., n\}$) of the ‘y’ and ‘x’ vectors to calculate the next estimate of the eigenvalue. ie

```
eigenval=y(1)/x(1)
```

Although this will work, for the given matrix, it is not a good idea, what if ‘x(1)’ was zero? Use the ‘MAXLOC’ function to return the index of the maximum value in ‘y’ and use the value, at this index location, to calculate the eigenvalue. HINT remember that the ‘MAXLOC’ function will return a rank one, single element array for a source-array of one dimension. So you need to create a one dimension, single element array to hold the result from ‘MAXLOC()’.

---

2 Dynamic Allocation of Arrays

So far all of your arrays have had their sizes ‘explicitly’ declared in your codes. It has not been possible for you to ‘read in’ the required size of your arrays during program execution and therefore make your program more general. This can be done in Fortran and arrays can be ‘allocated’ the memory they require at run-time. When they are no longer required they can be ‘deallocated’ and the memory freed. You can regard the process of ‘Dynamic Allocation’ in your code as three separate stages, declaration, allocation and deallocation. Allocatable arrays, once allocated, behave **exactly** the same in your code as arrays declared with explicit sizes for each extent (standard arrays) and they can be passed through argument lists to subroutines and functions.

2.1 Declaration

Allocatable arrays are declared in a similar fashion to standard arrays. There are two differences, no array bounds are given as their size is not known at the declaration stage. Their size, for each extent, is assumed by using a colon ':'. Secondly they have the attribute 'ALLOCATABLE' in their attribute list.

```
REAL, DIMENSION(:), ALLOCATABLE :: vec1
REAL, DIMENSION(:,:), ALLOCATABLE :: mat1
```

The first declares a rank one array of type 'REAL'. The second declares a rank two array of type 'REAL'. Note neither have been allocated any memory yet! The rank of the array is of course the same as the number of colons given in the 'DIMENSION' attribute. These two arrays can not be used in the code until an 'ALLOCATE' statement is executed to assign them some memory. This is achieved by specifying the size of each extent of the array using the 'ALLOCATE' statement.

2.2 Allocation

In order for any 'ALLOCATABLE' arrays to be used they must first be 'ALLOCATED' with a 'size' so some memory can be reserved to store their values. This is done using the Fortran 'ALLOCATE' statement. You can 'ALLOCATE' the memory for 'ALLOCATABLE' arrays anywhere in your code but this must be done **before** you attempt to use the array in your code.

```
ALLOCATE(vec1(16)) !** Allocate space for 16 elements
ALLOCATE(mat1(4,4))!** Allocate space for 4 rows and 4 cols
```

2.3 Deallocation

When you do not need your allocatable array anymore it is **good programming practice** to 'DEALLOCATE' the array and free the reserved memory. This is done using the 'DEALLOCATE' command.

```
DEALLOCATE(vec1) !** Free space used by vec1
DEALLOCATE(mat1) !** Free space used by mat1
```

Exercise Three : Some of your earlier codes could have benefited from the use of allocatable arrays, for example the code you wrote to multiply together two matrices in class project part three. Go back and amend your main program for this code to use dynamic memory allocation for all the arrays.

3 Fortran 90 Keyword & Optional arguments.

3.1 Keyword arguments

Up to now, when we call a procedure, the arguments in the calling statement map onto the 'dummy' arguments in the procedure's header argument list by order of appearance. That is the first argument in the calling statement is assigned to the first argument in the procedure header's argument list, the second to the second and so on. ie. in the call

```
CALL addnumbers(a,b,c)
```

with the procedure header for 'addnumbers' being

```
SUBROUTINE addnumbers(num1,num2,num3)

  REAL, INTENT(IN)  :: num1,num2
  REAL, INTENT(OUT) :: num3
```

When the 'CALL' statement is reached and program execution moves into the subroutine 'addnumbers' 'num1' gets passed the value of 'a', 'num2' gets passed the value of 'b', 'num3' gets passed the value of 'c'. There is a way in Fortran 90 of making this more flexible and allowing the arguments to be listed in any order. This is done using 'keyword' arguments and they work as follows. In the calling statement argument list the value to be passed is equated to the name of the dummy argument. So the following section of code works identically to the above section,

```
CALL addnumbers(num3=c,num1=a,num2=b)
```

So because we are using keywords the strict ordering of the arguments can be relaxed. There is a strict rule to be remembered here though, as soon as one argument is passed as a 'keyword' argument then ALL subsequent arguments (going left to right) must also be passed as keyword arguments!

3.2 Optional arguments

So far when you use a procedure, all the data objects that appear in the argument list in the procedure header must then also appear in the argument list of the calling statement. So for example in the subroutine defined as,

```
SUBROUTINE addnumbers(a,b,c,d,e,f)
```

When you call this subroutine the same number of arguments ie. 'a,b,c,d,e,f' must be included in the calling statement ie,

```
CALL addnumbers(a,b,c,d,e,f)
```

There is a way in Fortran 90 to make arguments optional, that is to define the arguments such that they do not have to be included in the calling statements argument list. This is done by giving them the attribute 'OPTIONAL' in the dummy argument declaration statement. The only rule here is that **all** optional arguments must appear **after** all non-optional arguments in the procedure's argument list. It is **good** programming practice when using 'optional' arguments to pass them through as 'keyword' arguments as it makes the code more readable and clearer to understand. Consider the rather useless subroutine '`addnumbers`',

```
SUBROUTINE addnumbers(num1,num2,num3,add,minus)

  REAL, INTENT(IN) :: num1,num2
  REAL, INTENT(OUT) :: num3
  LOGICAL, INTENT(IN), OPTIONAL :: add, minus

  IF  (PRESENT(add)) THEN
     IF (add .EQV. .TRUE.) num3=num1+num2
  ELSE IF (PRESENT(minus))  THEN
     IF (minus .EQV. .TRUE.) num3=num1-num2
  ENDIF

END SUBROUTINE addnumbers
```

Exercise Four : In a '`handout7/exercise4`' Code up the above subroutine in a module called '`add_mod.f90`'. Use the '`CALL`' statements below, from within a main program called '`main.f90`', to help understand how the code works.

```
CALL addnumbers(a,b,c)
CALL addnumbers(a,b,c,minus=.TRUE.)
CALL addnumbers(a,b,c,add=.FALSE.,minus=.TRUE.)
CALL addnumbers(a,num3=c,num2=b,minus=.TRUE.)
```

Note the above calls to '`addnumbers`' are all valid but not necessarily sensible i.e. the first call would do nothing.