Array Concepts

Carlos Cruz

NASA GSFC Code 606 (ASTG)
Greenbelt, Maryland 20771

carlos.a.cruz@nasa.gov

October 24, 2018

Agenda

Array Concepts

- Terminology
- Declarations
- Syntax
- Expressions
- Intrinsic functions
- Allocatable arrays

Array Terminology

```
1  real, dimension(15)        :: A
2  real, dimension(-4:0,0:2) :: B
3  real C(5,3), D(0:4,0:2)
```

A:
  rank=1, size=15, shape=15
B:
  rank=2, size=15, shape=5x3
C:
  rank=2, size=15, shape=5x3
D:
  rank=2, size=15, shape=5x3

B,C,D are conformable

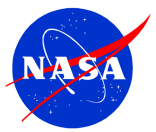
- *rank* : number of dimensions
- *bounds* : upper and lower limits of indices
- *extent* : number of elements in dimensions
- *size* : total number of elements
- *shape* : rank and extents
- *conformable* : same shape, B and C and D

Array Declarations

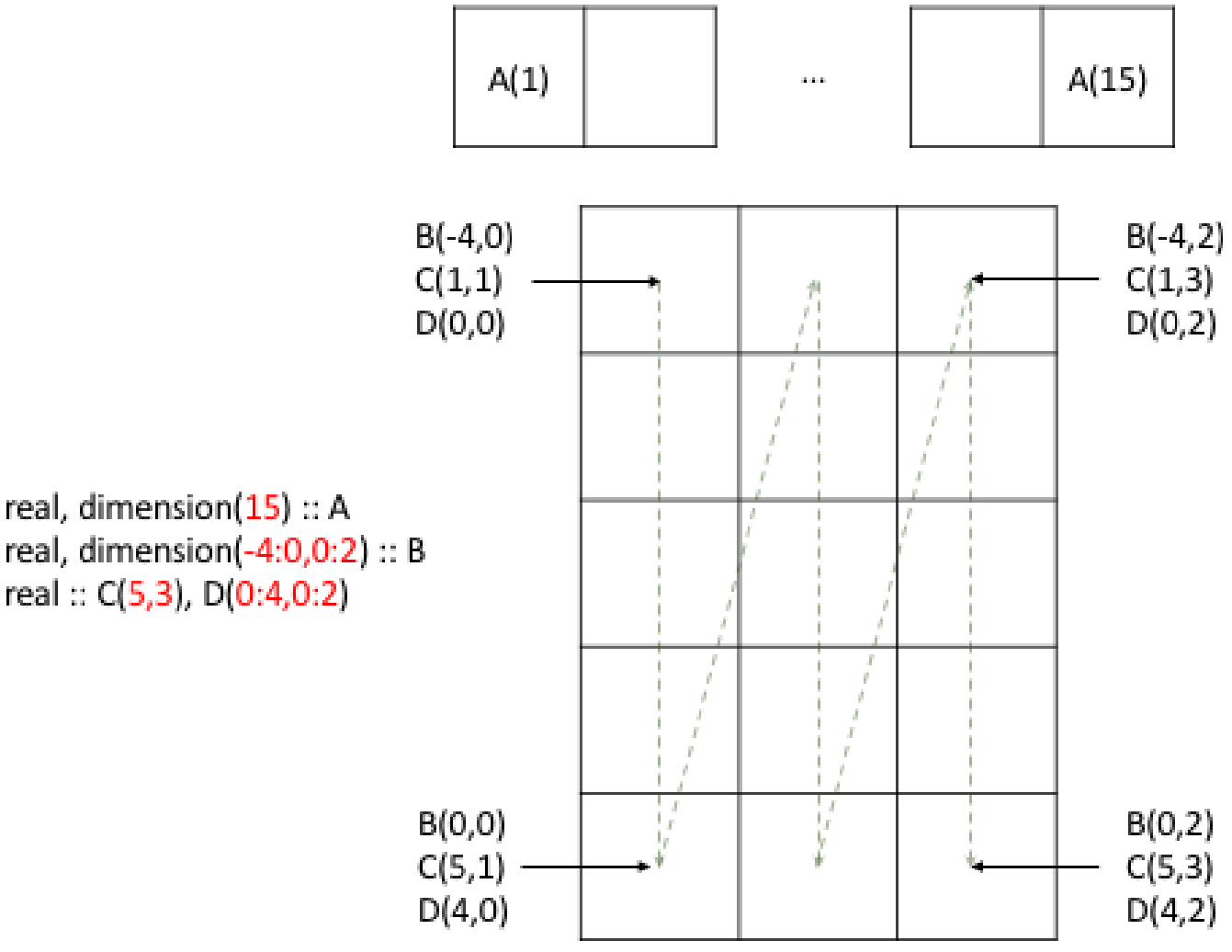
Literals and constants can be used in array declarations,

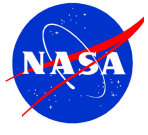
```
1      real, dimension(15)         :: A ! static arrays
2      real, dimension(-4:0,0:2)   :: B
3
4      integer, dimension(20)      :: N
5
6      integer, parameter :: UB = 5
7      real, dimension(0:UB-1)     :: Y
8      real, dimension(1+UB*UB,10) :: Z
```

- Default lower bound is 1
- Bounds can begin and end anywhere
- Arrays can be zero-sized
- Up to 7 dimensions



Array visualization





Array Syntax

Using the earlier declarations, references can be made to:

- whole arrays (conformable)
  - A = 0 ← sets whole array A to zero
  - B = C + 1 ← adds one to all elements, of C and then assigns each element to the corresponding element of B
- elements
  - A(1) = 0.0 ← sets one element to zero
  - B = A(3) + C(5,1) ← sets whole array B to the sum of two elements
- array sections
  - A(2:6) = 0 ← sets section of A to zero
  - B(-1:0,1:2) = C(1:2,2:3) + 1 ← adds one to the subsection of C and assigns it to the subsection of B

SSAI

Array Expressions

Arrays can be treated like a single variable in that:

- can use intrinsic operators between conformable arrays (or sections)
  - B = C * D - B**2
- elemental intrinsic functions can be used
  - B = sin(C) + cos(D)

Array Expressions

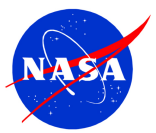
An array can be subscripted by a *subscript-triplet* giving rise to a sub-array of the original. The general form is:

(start:end:stride)

the section starts at *start* and ends at or before *end*.*stride* is the increment by which the locations are selected.

start, *end*, *stride* must all be scalar integer expressions. Thus, these are all valid:

```
1  A(m:m) = 0       ! m to m, 1 element array
2  A(m:n:k) = 0     ! m to n step k
3  A(8:3:-1) = 0    ! 8 to 3 backwards
4  A(8:3) = 2       ! step 1 => zero size
5  A(m::4) = 1      ! default UPB, step 4
6  A(::2) = 1.0     ! default LWB and UPB
7  A(m**2:n*k/3) = 1.0
```



Array Inquiry (1)

Consider the declaration:

```
1  real, dimension(-10:10,23,14:28) :: A
```

Then:

- LBOUND(SOURCE[,DIM]) – lower bounds of an array (or bound in an optionally specified dimension).
  - LBOUND(A) is (/-10,1,14/) (array).
  - LBOUND(A,1) is -10 (scalar)
- UBOUND(SOURCE[,DIM]) – upper bounds of an array (or bound in an optionally specified dimension).

Array Inquiry (2)

```
1  real, dimension(-10:10,23,14:28) :: A
```

- SHAPE(SOURCE) – shape of an array.
  - SHAPE(A) is (/21,23,15/) (array).
  - SHAPE((/4/)) is (/1/) (array)
- SIZE(SOURCE[,DIM]) – total number of array elements (in an optionally specified dimension).
  - SIZE(A,1) is 21.
  - SIZE(A) is 7245

Array Constructors

Used to give arrays or sections of arrays specific values. For example,

```
1      implicit none
2      integer                          :: i
3      integer, dimension(10)           :: ints
4      character(len=5), dimension(3) :: colors
5      real, dimension(4)               :: heights
6      heights = (/5.10, 5.6, 4.0, 3.6/)
7      colors = (/'RED  ','GREEN','BLUE '/)
8      ! note padding so strings are 5 chars
9      ints    = (/ 100, (i, i=1,8), 100 /)
```

Then:

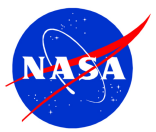
- constructors and array sections must conform.
- must be 1D.
- for higher rank arrays use RESHAPE intrinsic

Array Constructors in Initialization Statements

Named array constants can be created

```
integer, dimension(3), parameter :: &
    unit_vec = (/1,1,1/)
character(len=*), dimension(3),  parameter :: &
    lights =  (/'red  ','blue ','green'/)
real, dimension(3,3), parameter :: &
    unit_matrix = reshape( &  ! using reshape
    (/1,0,0,0,1,0,0,0,1/),(/3,3/))
```

In the second statement all strings must be same length.



RESHAPE

RESHAPE is a general intrinsic function which delivers an array of a specific shape:

RESHAPE(original_shape , new_shape)

e.g.:

```
1  A = RESHAPE((/1,2,3,4/),(/2,2/))
```

A is filled in array element order and looks like:

```
1     3
2     4
```

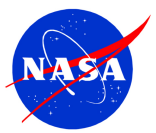
DATA

Use the DATA when other methods are tedious and/or impossible.

DATA variable / list / ...

e.g.:

```
1  INTEGER :: a(4), b(2,2), c(10)
2  DATA a /4,3,2,1/
3  DATA a /4*0/ ! * is not multiplication operator!
4  DATA b(1,:) /0,0/
5  DATA b(2,:) /1,1/
6  DATA (c(i),i=1,10,2) /5*1/
7  DATA (c(i),i=2,10,2) /5*2/
```



Allocatable Arrays

Fortran allows arrays to be created on-the-fly; these are known as *allocatable* arrays and use *dynamic heap* storage. Allocatable arrays are

- declared like explicit-shape arrays but without the extents and with the ALLOCATABLE attribute.
  - integer, dimension(:), ALLOCATABLE :: ages
  - real, dimension(:,:), ALLOCATABLE :: speed
- given a size in an ALLOCATE statement which assigns an area of memory to the object.
  - ALLOCATE(ages(1:10), STAT=ierr)
  - ALLOCATE(speed(-lwb:upb,-50:0),STAT=ierr)
- the optional STAT= field reports on the success of the storage request. If the INTEGER variable ierr is zero the request was successful otherwise it failed.

Deallocating Arrays

Heap storage can be reclaimed using the DEALLOCATE statement:

```
DEALLOCATE(ages,STAT=ierr)
```

- it is an error to deallocate an array without the ALLOCATE attribute or one that has not been previously allocated space.
- there is an intrinsic function, ALLOCATED, which returns a scalar LOGICAL values reporting on the status of an array.

```
if (ALLOCATED(ages)) DEALLOCATE(ages,STAT=ierr)
```

- the STAT= field is optional but its use is recommended
- if a procedure containing an allocatable array which does not have the SAVE attribute is exited without the array being DEALLOCATE d then this storage becomes inaccessible

Beware of memory leaks

It is the program that takes responsibility for allocating and deallocating storage to **static** variables.
However, when using dynamic arrays this responsibility falls to the **programmer**.

- Storage allocated to local variables (in say a subroutine or function - more on this later) must be **deallocated** before the exiting the procedure.
  - When leaving a procedure all local variables are deleted from memory and the program releases any associated storage for use elsewhere, however any storage allocated through the ALLOCATE statement will remain "in use" even though it has no associated variable name!
- Storage allocated, but no longer accessible, cannot be released or used elsewhere in the program and is said to be in an "undefined" state This reduction in the total storage available to the program called is a "memory leak".

Masked Array Assignment - Where Statement

This is achieved using WHERE

WHERE (I /= 0) A = B/ I

the LHS of the assignment must be array valued and the mask, (the logical expression,) and the RHS of the assignment must all conform.
For example, if

$$B = \begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{pmatrix}$$

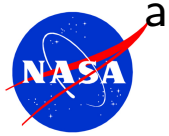
and

$$I = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

then

$$A = \begin{pmatrix} 0.5 & - \\ - & 2.0 \end{pmatrix}$$

Only the elements, corresponding to the non-zero elements of I, have been assigned to.

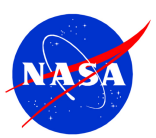


Where Construct

There is a block form of masked assignment

```
1   WHERE(A > 0.0)
2       B = LOG(A)
3       C = SQRT(A)
4    ELSEWHERE
5       B = 0.0
6    ENDWHERE
```

- the mask must conform to the RHS of each assignment; A, B and C must conform.
- WHERE ... END WHERE is not a control construct and cannot currently be nested.
- the STAT= field is optional but its use is recommended
- the execution sequence is as follows: evaluate the mask, execute the WHERE block (in full) then execute the ELSEWHERE block
- the separate assignment statements are executed sequentially but the individual elemental assignments within each statement are (conceptually) executed in parallel



Vector-valued subscripts

A 1D array can be used to subscript an array in a dimension. Consider

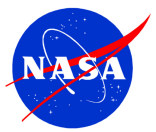
```
integer, dimension(5) :: V = (/1,4,8,12,10/)
integer, dimension(3) :: W = (/1,2,2/)
```

- A(V) is A(1), A(4), A(8), A(12), and A(10).
- the following are valid assignments:

```
A(V) = 3.5
C(1:3,1) = A(W)
```

- it would be invalid to assign values to A(W) as A(2) is referred to twice
- only 1D vector subscripts are allowed, for example

```
A(1) = SUM(C(V,W))
```



Conclusion

Arrays make Fortran a very powerful language, especially for computationally intensive program development.

- Array syntax allows easy (intuitive) initialization
- Dynamic memory allocation enables customizable sizing of arrays
- Many array intrinsic functions (most not discussed here)

Exercise

We will solve a computational problem that converts temperatures in Fahrenheit to Kelvin

$$K = \frac{5}{9}(F - 32) + 273.15 \tag{1}$$

Using a text editor open the file exercise.F90 and *complete the empty code blocks*. You will:

- Perform the conversion
- Print the F and K values
- Print the log of F (careful!)

Then build the executable and run the code using the provided Makefile.