Fortran 2008

Advanced Fortran 2008: Array syntax and advanced language features

Fiona Reid
f.reid@epcc.ed.ac.uk

Lecture 3 - overview

- Outline
  - Array declarations and array I/O
  - Array sub-objects and assignments
  - Conditional operations – **`where`** and **`forall`** statements
  - Array intrinsic procedures
  - Static, assumed, automatic and dynamic arrays
  - Pure and elemental procedures
  - Advantages/ disadvantages of Fortran array syntax
  - Derived data types
  - Pointers

Array declaration

- Recall, arrays are declared with **`dimension`** attribute

```
implicit none
integer, dimension(4) :: n4
integer, dimension(1:4) :: n4  ! same as previous line
```

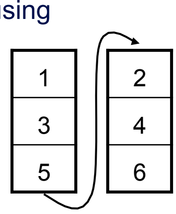
- Provides 4 elements
  - Elements: **`n4(1), n4(2), n4(3), n4(4)`**
  - First element is, by default, 1 *** not **`0`** ***
- Arrays can have more than one dimension

```
complex, dimension(1:10, 1:20) :: z
```

- Terminology
  - Number of dimensions is the *rank* (here 2)
  - Number of elements in given dimension is the *extent*
  - Sequence of the extents is the *shape* (here 10, 20)

Array I/O

- Recall that arrays are stored in memory by columns
- Can write out the **`3`** (rows) by **`2`** (columns) array **`a`** using




```
real, dimension(3,2) :: a
write(*,*) a
```

- The output will be: **`1, 3, 5, 2, 4, 6`**
- If we wish to write out the array in the order we typically use in matrix notation (maths) we need to use

```
do i = 1,3
  write(*,*)(a(i,j), j = 1,2) ) ! implied do loop
end do
```

- Output:

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

Array sub-objects

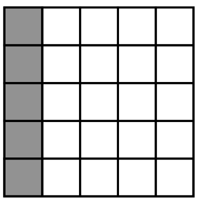
- Consider the array

```
real, dimension(1:10) :: a
```

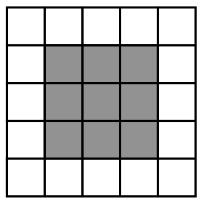
- Array in whole, or in part, can be addressed using a triplet:
  - `a([lower] : [upper] [:stride])`
- Examples

```
– a(7:)        ! a(7), a(8), a(9), a(10)
– a(2:10:2)    ! a(2), a(4), a(6), a(8), a(10)
– a(:)         ! whole array
– a(10:2:-2)   ! a(10), a(8), a(6), a(4), a(2)
```

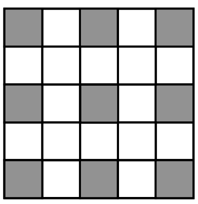

2-d array sub-objects



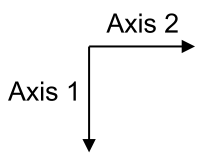
`a(:,1)`



`a(2:4,2:4)`



`a(1::2,1::2)`



Elemental assignments

- Array assignments and arithmetic
  - `a(1:10) = 0.0          ! assign 0 to each element`
  - `a(:) = a(:) + 1.0      ! add 1 to each element`
  - `a = 0.0                ! array or scalar? depends`
    `                             ! on declaration of a`
- Can use a constructor `(/ … /)`
  - `real, dimension(2) :: s = (/ -1.0, 1.0 /)`
  - `a(1:5) = (/ 1.0, -1.0, 1.0, -1.0, 1.0 /)`
- With an implied do loop…
  - `real, dimension(10000) :: &`
    `                    a = (/ (i, i = 1, 10000) /)`
  - sets `a(1) = 1.0, a(2) = 2.0, …`

Conformable arrays

- Arrays used in Fortran 2008 expressions must be *conformable*
  - They must have the same shape, that is,
  - The same rank (number of dimensions)
  - The same extent in every dimension
- For example

```
integer, dimension(10)   :: a, b
integer, dimension(10,2) :: c
a(:) = b(:)                   ! correct
a(1:5) = b(6:10)              ! correct
a(1:3) = b(1:4)               ! clearly wrong
c(:, 1) = a(:)                ! correct
c(:, :) = b(:)                ! clearly wrong
                              ! different ranks
                              ! will not compile
```

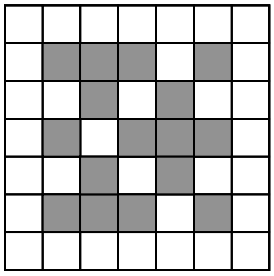
Conditional operations

- A conditional ('`if`') type construct for arrays

```
where (logical array-expression)
     array assignments
[else where] (logical array-expression)
     array assignments
end where
```

- Example: can create a mask for all elements of `a > 0.0`
  - `where` uses a logical mask equivalent to applying an "`if`" on each element of the array
  - Also an `else where ()` clause

```
real, dimension(7,7) :: a,b
where(a > 0.0)
  a = 1.0/a    ! LHS and RHS must be
  b = a*b      ! the same shape
elsewhere
  b = 0.0
end where
```



Forall

- Consider

```
do i = 1, n
  a(i, i) = x(i)
end do
```

  - performed serially (one iteration after the other)
- Instead can use Fortran 95 statement `forall`

```
forall (i = 1:n) a(i, i) = x(i)
```

  - performed conceptually 'at the same time' – in parallel
- Several distinct stages involved in the `forall`
  - Computation of valid index set (`1:n` above)
  - Define active index set (`1:n` above)
  - Evaluate right hand sides (`x(1), x(2), …, x(n)`)
  - Assign to left hand sides (`a(1,1), a(2,2), …, a(n,n)`)

Forall

- More complicated structures are allowed

```
forall (i = 1:n)
  where (a(i, :) == 0) a(i, :) = i
  b(i, :) = i / a(i, :)
end forall
```

- Useful for expressing parallelism in code – see e.g. HPF
- **Please note**: **`forall`** is nearly always less efficient than using array syntax – only use when array syntax is not appropriate

Fortran intrinsic procedures

- Many intrinsic procedures are pure elemental
  - Elemental means they act on each element of array independently
  - Pure means they have no side-effects (more on this later)
  - Examples include: **`sin`**, **`cos`**, **`min`**, **`max`**, **`exp`**, **`sqrt`**, **`log`**…
- Consider

```
real, dimension(5) :: a
a = (/ 1.0, 4.0, 9.0, 16.0, 25.0 /)
a = sqrt(a)
```

  - **`a = sqrt(a)`** finds the square root of all the elements of array **`a`**
  - after application **`a(1)=1.0`**, **`a(2)=2.0`**, **`a(3)=3.0`**, **`a(4)=4.0`** and **`a(5)=5.0`**

Fortran intrinsic procedures

- Array specific inquiry functions

  size, lbound, ubound, shape
  - Used to obtain information about the bounds of an array
  - Functions **size**, **lbound** and **ubound** have an optional, **dim** argument, e.g. **size(array [,dim])**
- Consider the two dimensional array, **a**, declared with

```
real, dimension(20,10) :: a
```

  - **size(a, dim = 1)** gives the size of the array along the first dimension, i.e. = 20 for this example; **size** returns integer
  - **lbound(a, dim = 1)** gives the lower bound of the array along the first dimension, i.e. = 1; **lbound** returns integer or array
  - **ubound(a, dim = 2)** gives the upper bound of the array along the second dimension, i.e. = 10
  - **shape(a)** gives the shape of the array, i.e. 20 10

Fortran intrinsic procedures

- Array transformation functions
  - Array construction functions: **spread, pack, reshape, …**
  - Vector and matrix multiplication: **dot_product, matmul**
  - Reduction functions: **sum, product, any, maxval, minval…**
  - Geometric location functions: **minloc maxloc**
  - Array manipulation functions: **transpose, cshift, …**
- Complete list – see Metcalf and Reid or the Standard documents

Example: reshape

- The **`reshape`** array intrinsic function allows the shape of an array to be altered
  - Syntax **`reshape (source, shape)`** e.g.

```
a(1:8) = (/ 1, 2, 3, 4, 5, 6, 7, 8 /)
b = reshape(a, (/ 4, 2 /) )
```

  - Visually

a =

1	2	3	4	5	6	7	8

$\xrightarrow{\texttt{reshape}}$ **b =**

1	5
2	6
3	7
4	8

Reduction operations

- Typically involve every element, i.e, a global operation

```
integer, dimension(1024)    :: a
integer, dimension(4, 1024) :: b
integer :: ival
ival = sum(a)                     ! Sum of all elements
ival = count(a == 0)              ! Number of zero elements
ival = product(b, dim = 1)        ! Product in first dim
ival = minval(a, mask = a < 0)    ! Smallest negative
```

- Logical reductions: **`any, all`**

```
logical, dimension ::  c(n)
if (all(c)) …                ! Global logical and
if (all(b(1,:) == a(:))) …   ! True if all elements equal
if (any(c)) …                ! Global logical or
if (any(b < 0.0)) …          ! True if any element < 0.0
```

Shift Operations

- Move whole array in particular direction
- For example, a cyclic shift one place to the right would produce the following result

Source

1	2	3	4
5	6	7	8

→

Result

4	1	2	3
8	5	6	7

- Also "end-off" shift; provide boundary conditions
- Many efficient parallel algorithms can be implemented in terms of shifts
- Useful for, e.g., image processing, cellular automata

Cshift

Intrinsic for regular movement of data

Axis 2

Axis 1

src

1	2	3
4	5	6
7	8	9

→

res1

3	1	2
6	4	5
9	7	8

```
res1 = cshift(src, shift=-1, dim=2)
```

is roughly equivalent to

```
res1(i,j) = src(i,j-1)
```

Pure/elemental procedures (F95 onwards)

- Procedures with no side-effects can be declared *pure*
  - No function dummy arguments are altered, i.e. all variables, intent(in)
  - No variables accessed by host association are altered
  - No I/O, stop

```
pure real function eqn_of_state(t, p)
```

- Elemental functions
  - declared with scalar dummy arguments but can be called with array actual arguments
  - must be pure

```
elemental real function eqn_of_state(t, p)
  real, intent(in) :: t, p

a          = eqn_of_state(t0, p0)
b(:)       = eqn_of_state(t1(:), p1(:))
c(:, 2:4) = eqn_of_state(t2(:, 2:4), p2(:, 2:4))
```


Static arrays

- So far have looked only at static arrays
  - Fixed size at declaration
  - Number of elements cannot be altered during program execution
  - Can be wasteful in terms of storage, often defined to be larger than required
  - In Fortran 2008 these are known as **explicit-shape** arrays
  - Declared using

```
real, dimension(1:10) :: a        ! 1D array
real, dimension(1:10,1:5) :: b    ! 2D array
```

Assumed & automatic arrays

- Fortran allows array sizes to flexible
  - Size is determined on entering a sub-program
  - Can only be used within a sub-program
  - Allows flexibility and re-use of sub-program
  - The dimensions of these arrays are *known* at compile time
  - In Fortran 2008 known as **assumed-shape** or **automatic** arrays
  - Declared using

```
real, dimension(:) :: a          ! 1D array
real, dimension(:,:) :: b        ! 2D array
real, dimension(:,:,:) :: c      ! 3D array
```


Assumed & automatic arrays

```
subroutine reverse(aa)
  implicit none
  real, dimension(:), intent(inout) :: aa ! assumed
  real :: work(size(aa))   ! automatic
  integer :: i,imax

  imax = int(size(work))
  do i = 1,imax
    work(imax-i+1) = aa(i)
  end do
  aa = work        ! Copy back for output
end subroutine reverse
```

- Size of **aa** is *assumed* - determined by the calling program
- Size of **work** is *automatic* - as it depends on **aa**

Array dummy arguments

- Ranks of actual and dummy arguments must agree
- Shape may be *assumed*

```
subroutine swap(a, b)
  real, dimension(:), intent(inout) :: a, b
```

  – Array inquiry functions such as **`size(a)`** and **`lbound(a)`**, **`ubound(a)`** make it easier to write generic routines

- Local arrays which may need to vary are *automatic*

```
subroutine swap(a, b)
  real, dimension(:), intent(inout) :: a, b
  real, dimension(size(a))          :: work
  work = a
  a = b
  b = work
end subroutine swap
```

  – Care with lower and upper bounds

Dynamic arrays

- Dynamic arrays
  – Size can be allocated during execution
  – Very flexible but may slow run-time performance
  – Lack of array bounds checking during compilation
  – In Fortran 2008 these are known as **allocatable** arrays
  – Declared using

```
real, dimension(:), allocatable   :: names ! 1D array
real, dimension(:,:), allocatable  :: grid  ! 2D array
```

  – Storage space for dynamic array is allocated using **`allocate`**

```
allocate(work(n,2*n,3*n))
```

  – Storage space is deallocated using **`deallocate`**

```
deallocate(work)
```

  – Beware, deallocated data is lost permanently!

Example: dynamic arrays

```
module work_array
  implicit none
  integer  :: n
  real, dimension(:,:,:), allocatable :: work
end module work_array

program main
  use work_array
  implicit none
  read(*,*) n
  allocate(work(n,2*n,3*n))
  . . .
  deallocate(work)
end program main
```


Pros/Cons of array syntax

- Fortran 2008 has important features for scientific computing
  - Array sub-objects and assignment
  - Operations **`where`** and **`forall`**
  - Pure and elemental procedures
  - New intrinsic transformation functions
- These allow concise code to be written
- However, overblown array syntax can
  - Harm readability (e.g., obscure underlying algorithm)
  - Have undesirable impact on performance (possible to create very complex expressions the compiler can't unravel)
- Use with discretion
- Be aware of possible performance issues

Fortran 2008

Advanced features: derived data types & pointers

Fiona Reid
f.reid@epcc.ed.ac.uk

Derived data types

- Fortran 2008* allows the use of derived data types
- In many algorithms data types can be grouped together to form an aggregate structure
- Often useful to be able to manipulate objects that are more sophisticated than the intrinsic types
- Allows linked data structures, lists, trees etc
- Derived data types are often needed in Coarray Fortran
- Imagine we wish to specify objects representing persons
  - Each person is uniquely distinguished by a name, age and ID number
- We can define a corresponding "person" data type as follows:

*allowed since Fortran 90

Derived data types

```
type person
  character (len=10) :: name
  real               :: age
  integer            :: id
end type person
```

- To declare a variable of type person use the syntax
  `type(person) :: you, me`
- Can assign values to the variable **`you`** using
  `you = person("Joe Bloggs", 21, 1234)`
- **`you`** is a variable containing 3 elements: **`name`**, **`age`** & **`id`**

Derived data types

- The elements of derived type **`person`** may be accessed by using the variable name and the element name separated by the `%` character, e.g.

```
you%name ! contains the name of you
you%age  ! contains the age of you
you%id   ! contains the id of you
```

- `%` is known as the *component selector*
- Can perform computations using derived type variables as follows:
  - Difference in age between variables **`you`** and **`me`**

```
real :: age_diff
age_diff = you%age - me%age
```

Pointers

- Pointers were included in the Fortran 90 standard
  - Included in Fortran 2008 and useful for Coarray Fortran
- Similar but not identical to pointers used in C/C++/Java
- In other languages a pointer stores an *address* rather than the actual value
- In Fortran, a pointer is an *alias* for the variable it points to
  - Pointer variables do not store addresses, instead they evaluate to the variable that they point to
  - No need to dereference pointer variables in Fortran
- Fortran pointers can also point to a section of data structure, e.g. to a row or column of an array or part of a derived data type

Pointers

- To declare a pointer variable, add the keyword, **`pointer`**, to the variable declaration, e.g.

```
real, pointer :: f
```

- We also need to identify the variables to which **`f`** can point. Use the keyword, **`target`**, to do this, e.g.

```
real, target :: a
```

- Assignment of pointer variables is achieved using the pointer assignment operator, **`=>`**, e.g.

```
f => a
```

- The memory location associated with **`a`** is now also the memory location associated with **`f`**
  - If the value of **`a`** changes then the value of **`f`** also changes and vice versa as they now share the same memory location
- Can use **`associated`** to test whether a pointer points to something

Pointers – simple example

```
program pointer_example
   implicit none
   real, target  :: a=3.141, b=1.414, c=0.866   ! initial values
   real, pointer :: r, s                  ! pointers to real values

! Pointer associations result in  a = r = 3.141 and b = s = 1.414
   r => a
   s => b

! Conventional assignments, variables a, b, r, and s will be 3.141
   s = r

! Can use pointer association to set r = c = 0.866 without changing
! a, b, or s.
   r => c

end program pointer_example
```


Pointers and arrays

- Can also have pointers to arrays

```
real, dimension(8), target :: mydata
real, dimension(:), pointer :: index
```

- Pointer arrays just have a rank
  - They are declared without any bounds
  - Bounds are picked up from the target array
  - Target and pointer ranks must match

```
mydata = (/(i, i=1,8)/)
index => mydata(1:8:2)
write(*,*)"Size of target array = ",size(mydata)  ! 8
write(*,*)"Size of pointer array = ",size(index)  ! 4
```

Summary

- This session we have looked at
  - Array operations in Fortran 2008: assignments, I/O, sub-objects, conditionals, array intrinsic functions
  - Advantages/disadvantages of using Fortran array syntax
  - Advanced features of the language
    - Derived data types
    - Pointers