Arrays

Arjen Markus

Deltares

June 6, 2018

Arrays as data structure

Arrays are by far the most important data structure in Fortran:

- Used to store large amounts of data of the same basic type
- Iterate over the data via DO-loops
- Use array operations
- One or more dimensions

Data of different basic types can be stored as derived types

Basic operations

Simple assignment:

```
real, dimension(20) :: array
integer             :: i

do i = 1,20
    array(i) = 10.0
enddo

! Alternatively:
array = 10.0
```

Basic operations (2)

Reversing the order:

```
real, dimension(20) :: a, b
integer             :: i

do i = 1,20
    a(i) = b(21-i)
enddo

! Alternatively:
a = b(20:1:-1)

! This works too:
a = a(20:1:-1)
```

On the last example: a "universal principle"

The last example:

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

a = a(20:1:-1)
```

works because in Fortran the right-hand side is evaluated *as if independent of the left-hand side*.

Consequences – side remark

Some of the consequences are:

- It is easier to reason about the effect.
- You may need to be careful:

```
integer, parameter :: dp = kind(1.0d0)
real(kind=dp) :: a   ! dp is the kind for double precision ...

a = 1.23456789012345 ! The constant is single precision
```

Use instead:

```
integer, parameter :: dp = kind(1.0d0)
real(kind=dp) :: a   ! dp is the kind for double precision ...

a = 1.23456789012345_dp  ! Or: a = 1.234567890123450
```

Array sections

You can specify a part of an array:

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

b = a(1,:)

! Left: elements 1, 3, 5, 7 and 9
! Right: elements 2, 4, 6, 8, 10 - same number!

b(1:10:2) = a(2:10:2,1)
```

More on array sections

There is a difference between `a` and `a(:)` (array `a` one-dimensional):

- The first is the whole array.
- The second is an array section that happens to coincide with the whole array.

The distinction is important: the whole array may be allocatable, whereas a section is not. In some contexts, an array may be (re)allocated.

Lower and upper bounds

By default the lower bound of array dimension is 1, but consider:

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

This is useful at times, as it saves index arithmetic.

Lower and upper bounds – caveat

The lower bound is *not* passed to subroutines/functions:

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

call myroutine( a )
```

In the subroutine the array's lower bound is again 1, unless specified otherwise.

Array operations

Arrays and array sections can be used in expressions:

```
real, dimension(10)    :: a, b
real, dimension(10,23) :: c

a = log(b) + c(:,12)
```

This is equivalent to:

```
real, dimension(10)    :: a, b
real, dimension(10,23) :: c
integer                :: i

do i = 1,10
    a(i) = log(b(i)) + c(i,12)
enddo
```

Array operations (2)

Arrays and array sections can be zero elements long:

```
real, dimension(10)     :: a
integer                 :: number

number = 0
a(1:number) = 10.0
```

Useful: simpler implementation of many algorithms

Memory management

Arrays can be allocatable:

```
real, dimension(:), allocatable :: a
integer                         :: number

write(*,*) 'Required size?'
read(*,*) number

allocate( a(number) )

! Or:
allocate( a(-number:number) )
```

Memory management (3)

When no longer needed they should be deallocated:

```
real, dimension(:), allocatable :: a, b, c

...

! We're done
deallocate( a, b, c )
```

Note: deallocation of allocatables is automatic in subroutines and functions.

Pointers

Fortran also defines *pointers* – you can view them as *aliases* for other variables, arrays or sections of arrays:

```
real, dimension(10), target :: x       ! Note the keyword!
real, dimension(:), pointer :: px, ps
real, pointer               :: pe

px => x              ! Point to the whole array
pe => x(2)           ! Point to a single element
ps => x(1:10:4)      ! Point to a section
```

Pointers (2)

To elaborate on the last example:

```
ps => x(1:10:4)      ! Point to a section

ps(1) <----> x(1)
ps(2) <----> x(1+4) = x(5)
...

x     = 0.0
ps    = 1.1
ps(2) = 4.0

write(*,*) x

==> 1.1 0.0 0.0 0.0 4.0 0.0 0.0 0.0 1.1 0.0
```

Pointers (3)

One use of pointers:

```
real, dimension(:), pointer :: tmp, new_values, old_values

allocate( new_values(1000), old_values(1000) )

do i = 1,100
    new_values = solve( old_values )

    ! Exchange the solutions

    tmp        => old_values
    old_values => new_values
    new_values => tmp
enddo
```

Allocatables and pointers

There is a difference between allocatables and pointers:

- Allocatables have a well-defined life-time. Automatic deallocation when returning from a routine.
- Pointers do not have this – you are yourself responsible
- Pointers can be used in recursively defined data structures as linked lists.

Different dimensions

You can use pointers to access an array with different dimensions:

```
integer, dimension(10,10), target  :: x
integer, dimension(:,:,:), pointer :: y

y(1:5,1:5,1:2) => x

write(*,*) shape(y)
```

```
===> 5 5 2
```

Arrays and routines

Pass an array as argument:

- The FORTRAN 77 way – assumed-size arrays

```
subroutine sub( array )
    real, dimension(10,*) :: array    ! Size is NOT inherited

    ...
end subroutine sub
```

- From Fortran 90 onwards – assumed-shape arrays:

```
subroutine sub( array )
    real, dimension(:,:) :: array    ! Size is inherited

    ...
end subroutine sub
```

This does require an explicit interface – easy way: use modules.

Arrays and routines: local arrays

Define an automatic array:

```
subroutine sub( array )
    real, dimension(:) :: array

    real, dimension(2*size(array)) :: work

    ...
end subroutine sub
```

Arrays and routines: return an array as the result

You can return an array result from a function:

```
function mkdouble( array ) result( r )
    real, dimension(:)             :: array

    real, dimension(size(array)) :: r

    r = 2.0 * array
end function mkdouble

...

a = mkdouble(b)
a = mkdouble(a+b)
```

Arrays and routines: matching type, rank, kind

Array arguments must match the type, rank and kind:

```
function mkdouble( array ) result( r )
    real, dimension(:)             :: array

    real, dimension(size(array)) :: r

    r = 2.0 * array
end function mkdouble

real(kind=dp), dimension(10) :: a, b

a = mkdouble(b)    ! Does not work!
```

Arrays and routines: elemental routines

Via elemental routines you can lift the restriction on the rank:

```
elemental function mkdouble( a ) result( r )
    real :: a
    real :: r

    r = 2.0 * a
end function mkdouble

real, dimension(10,20,30) :: a, b

a = mkdouble(b)   ! No need for a separate 3D version
```

Automatic (re)allocation

If an allocatable array appears on the left-hand side it may be automatically reallocated:

```
real, dimension(:), allocatable :: array

allocate( array(100) )

call random_number(array)

! Keep the values larger than 0.5

array = pack( array, array > 0.5 )

write(*,*) 'Number: ', size(array)

==> 53   (just an example)
```

Standard (intrinsic) functions

Some functions:

- Sum of array elements, number of array elements, given a condition

```
write(*,*) sum(array), sum(array, array > 0.5)
write(*,*) count(array > 0.5)
```

- Maximum value, position:

```
write(*,*) maxval(array)
write(*,*) maxloc(array)
```

Standard (intrinsic) functions and routines (2)

Some functions:

- Packing (filtering):

```
write(*,*) pack(array, array > 0.5)
```

- Size, shape:

```
write(*,*) 'Second dimension: ', size(array2d,2)
write(*,*) 'Shape:            ', shape(array3d)
```

- Random numbers between 0 and 1:

```
call random_number( array )
```

Standard (intrinsic) functions (2)

Check whether array elements fulfill a condition:

```
if ( all( x > 0.0 ) ) then
    write(*,*) 'All elements are positive'
elseif ( any( x > 0.0 ) then
    write(*,*) 'Only some elements are positive'
endif
```

Standard (intrinsic) functions and routines (3)

More functions:

- Packing (filtering):

```
write(*,*) pack(array, array > 0.5)
```

- Size, shape:

```
write(*,*) 'Second dimension: ', size(array2d,2)
write(*,*) 'Shape:            ', shape(array3d)
```

- Random numbers between 0 and 1:

```
call random_number( array )
```