Arrays

Introduction to Arrays

More often than not, you will be working with more than just individual data values. Instead, you will have an entire list or set of data that is all the same type. And you definitely don't want to work with them this way:

```
real :: rh_1, rh_2, rh_3, rh_4, rh_5, ...
  ...
rh_1 = 88.2; rh_2 = 74.8; rh_3 = 55.4; ...
```

In Fortran, a collection of values of the same type is called an **array**. This allows us to do this instead:

```
real, dimension(1000) :: rh
rh(1) = 88.2; rh(2) = 74.8; rh(3) = 55.4
print*, rh(1:3)
print*, rh
```

The numbers in parenthesis that specify the location of an item within an array are called **subscripts** (borrowed from mathematics). It's customary to refer to the expression **x(3)** as "**x sub 3**".

We can use variables as subscripts, too!

```
i = 1
print*, rh(i)
```

So you might imagine writing a subroutine that would print out all of our rh records. Let's take a look! **rhvals.f90**

Array Terminology and Declarations

The preferred method of declaring arrays is to use the dimension attribute in a type statement:

```
real, dimension(15) :: x
real, dimension(1:5,1:3) :: y, z
```

The above are **explicit-shape** arrays.

Some terminology:

* **Rank** = number of dimensions
  * The rank of X is 1; rank of Y and Z is 2.
* **Bounds** = upper and lower limits of indices
  * The bounds of X are 1 and 15; bounds of Y and Z are 1 and 5 and 1 and 3.
* **Extent** = number of elements in dimension
  * The extent of X is 15; extents of Y and Z are 5 and 3.

- **Size** = total number of elements
  - Size of X, Y and Z is 15.
- **Shape** = rank and extents
  - Shape of X is 15; shape of Y and Z is 5,3.
- **Conformable** = same shape
  - Y and Z are conformable.

More on Declarations

* Literals and constants can be used in array declarations (with some caveats).
* The default lower bound is 1.
* Bounds can begin and end anywhere (i.e., you can use zero as a subscript as well as negative subscripts).

Examples:

```
real, dimension(1:100) :: r     is the same as  real, dimension(100) :: r
real, dimension(1:10, 1:10) :: s
real :: t(10,10)
real, dimension(-10:-1) :: u
real, dimension(2,5,-1:8) :: x  ! this has a rank of 3, extents of 2, 5 and 10,
  ! a shape of (/ 2, 5, 10 /), and a size of 100

integer, parameter :: lda = 5
real, dimension(0:lda-1) :: y
real, dimension(lda,lda+1,lda+2) :: big_array
```

* Declarations using colons for the subscripts may be used for a dummy argument of a procedure. This indicates that the shape of the dummy array is to be taken from the actual argument used when the procedure is called. This is known as an **assumed-shape array**.
    * **Example:** rhvals2.f90
* The declaration of arrays may also use values of other dummy arguments to establish extents. These are called **automatic arrays**.

Example:

```
subroutine s2 (dummy_list, n, dummy_array)
    real, dimension(:) :: dummy_list
    real, dimension(size(dummy_list)) :: local_list
    real, dimension(n,n) :: dummy_array, local_array
    real, dimension(2*n + 1) :: longer_local_list
```

Array Syntax

We can reference:

- whole arrays

```
a = 0.0          ! set all elements of the array a to zero
b = c + d        ! adds the elements of c and d together, assign result to b
```

- individual elements

```
a(1) = 0.0          ! set just one element of the array to zero
b(0,0) = a(3) + c(5,1)
```

- array sections

```
a(2:4) = 0.0          ! set a(2), a(3) and a(4) to zero
b(-1:0,1:2) = c(1:2,2:3) + 1.0  ! adds one to the subsection of c
```

Array-valued Expressions and Assignment

Arrays are now first-class objects, and array-valued expressions are evaluated element-wise, which saves writing many simple loops:

```
real, dimension(512,1024) :: raw, background, exposure, result, std_err
...
result = (raw - background) / exposure
```

Similarly, all appropriate intrinsic functions operate element by element if given an array as an argument:

```
std_err = SQRT(raw) / exposure
```

Note that the arrays must be conformable for these operations to be valid.

```
background = 0.1 * exposure + 0.125   ! can include scalar constants and
                                        variables
```

Array Sections

We can select a portion of an array to use in a particular computation with subscript-triplets. The general form is

[<bound1>] : [<bound2>] [:<stride>]

Examples:

```
x(:)                    ! the whole array
x(3:9) or x(3:9:1)      ! x(3) to x(9) in steps of 1
x(m:n) or x(m:n:k)      ! use integer variables as bounds and stride
x(8:3:-1)               ! x(8) to x(3) in steps of -1
x(m:)                   ! from x(m) to default upper bound
x(:n)                   ! from default lower bound to x(n)
x(::2)                  ! from lower bound to upper bound in steps of 2
x(m:m)                  ! one element section
```

Slice assignment: can involve overlapping slices

```
a(2:10) = a(1:9)        ! shift up one element
b(1:9) = b(3:11)        ! shift down two elements
```

Vector subscripts may also be used:

```
integer, dimension(4) :: mysub = (/ 32, 16, 17, 18 /)
real, dimension(100) :: vector
...
write(*,*) vector(mysub)
```

Note that vector subscripts can only be used on the left-hand side of an assignment if there are no repeated values in the list of subscripts.

Array Constructors

A way of assigning an array a set of values along one dimension only. The constructor is delimited by **(/** and **/)**, and the elements are separated by commas.

```
x(1:4) = (/ 1.2, 3.5, 1.1, 1.5 /)          ! a scalar expression
x(1:4) = (/ 1.2, aval, 1.1, bval /)        ! also a scalar expression
x(1:4) = (/ a(i,1:2), a(i+1,2:3) /)        ! an array expression
x(1:4) = (/ (sqrt(real(i)),i=1,4) /)       ! an implied do list
```

An **implied do list** is a list of expressions followed by something that is like an iterative control in a do statement.

```
x(1:4) = (/ sqrt(1.0), sqrt(2.0), sqrt(3.0), sqrt(4.0) /)      ! equivalent
```

You can use them for other purposes, too:

```
print *, (a(i,i), i = 1,n)
```

And they are valid in an array declaration:

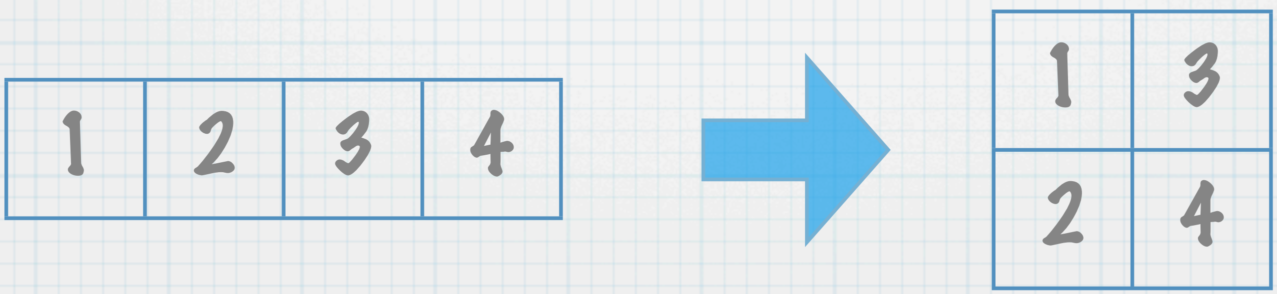
```
real, dimension(4) :: x = (/ 1.2, 3.5, 1.1, 1.5 /)
```

The **reshape** intrinsic function can be used to define rank-two and greater arrays using array constructors:

```
RESHAPE (source, shape)
```

Example:

```
integer, dimension(2,2) :: a
a = reshape( (/ 1,2,3,4 /), (/ 2,2 /) )
```



Allocatable Arrays

Fortran 90 allows arrays to be created on-the-fly; these are known as **deferred-shape** arrays.

* Declaration: (note allocatable attribute, fixed rank)

```
integer, dimension(:), allocatable :: ages
real, dimension(:,:), allocatable :: speed
```

* Allocation:

```
read*, isize
allocate(ages(isize), stat=ierr)
if (ierr /= 0) print*, “ages: allocation failed”

allocate(speed(0:isize-1,10), stat=ierr)
if (ierr /= 0) print*, “speed: allocation failed”
```

Deallocating Arrays

Heap storage can be reclaimed using the **DEALLOCATE** statement:

```
if (allocated(ages)) deallocate(ages, stat=ierr)
```

* You'll get an error if you try to deallocate an array without the allocate attribute or an array that has not previously been allocated space.
* If a procedure containing an allocatable array which does not have the save attribute is exited without being deallocated, then this storage becomes inaccessible.

The WHERE statement and construct

Used to assign values to only those elements of an array where is logical condition is true.

* Single statement form:

```
where (a < 0) b = 0          ! a and b must be arrays of the same shape
```

* Block form:

```
where (c /= 0)       ! c /= 0 is a logical
   a = b / c         ! a and b must conform to c
elsewhere
   a = 0             ! the elements of a are set to 0 where they have not
                     ! been set to b/c.
   c = 1             ! the 0 elements of c are set to 1
end where
```

* All statements within a **WHERE** construct must be array assignments.

* The assignments are executed in the order they are written: first those in the **WHERE** block, then those in the **ELSEWHERE** block.

* **WHERE** constructs may not be nested.

Element Renumbering in Expressions

The elements in an expression no longer have the same subscripts as the elements in the arrays that make up the expression. They are renumbered with 1 as the lower bound in each dimension.

```
y(0:7) + z(-7:0)          ! result is an array with subscripts 1, 2, 3, ..., 8

integer, dimension(0:6), parameter :: v = (/ 3, 7, 0, -2, 2, 6, -1 /)
maxloc(v)            ! result is 2
maxloc(v(2:6))       ! result is 4 because the largest entry (6) is in the
                     ! 4th position of the section v(2:6)
```