Arrays

Array is a collection of similar entities. Indices are used to point to specific elements of arrays.

Declaration of a one-dimensional array

```
real x(10)
real :: x(10)
real, dimension(10) :: x
```

Lower limit of the index is 1 by default, but can be changed:

```
real x(0:10)
real, dimension(-10:10) :: y
```

Arrays can be initialised:

```
real, dimension(3) :: x = (/ 0.0, 0.5, 1.0 /)
real, dimension(-10:10) :: y = 0.0
```

NB: initialised arrays are stored statically and initialised only once.

Local arrays are not initialised every time a procedure is invoked. To initialise them every time use assignment statements.

Multidimensional arrays:

```
real y(2,3)
real, dimension(-1:1,0:100) :: u
```

These correspond to matrices

```
y(1,1)  y(1,2)  y(1,3)
y(2,1)  y(2,2)  y(2,3)
```

```
u(-1,0) u(-1,1) ... u(-1,100)
u( 0,0) u( 0,1) ... u( 0,100)
u( 1,0) u( 1,1) ... u( 1,100)
```

Maximum number of dimensions is 7.

Memory allocation different from other languages:

```
 real y(0:1,0:2)

y(0,0) y(0,1)  y(0,2)
y(1,0) y(1,1)  y(1,2)
```

This is stored in the memory as

```
y(0,0) y(1,0) y(0,1) y(1,1) y(0,2) y(1,2)
```

In C the array `y[2][3]` would be stored as

```
y[0][0] y[0][1] y[0][2] y[1][0] y[1][1]  y[1][2]
```

The storage order is not important for small arrays. However, if the arrays are large, they may affect the execution time (discussed later).

Array operations

Operands of arithmetic operations can be whole arrays:

```
real, dimension (100) :: x, y, z, u
x=y+z
u=x*z
```

Operators operate on corresponding elements; * is not a matrix product.

Operands must have the same size and dimensions.

Both sides of the assignment operator must have the same size and dimensions.

Exception: a scalar conforms to all arrays.

No difference between row and column vectors.

```
real x(100), y(100), a
y=1.0
x=y+a
```

Intrinsic functions can operate on arrays:

```
real x(100), y(100)
x=0.1*(/ (i, i=1,100) /)
y=sqrt(x)
```

Intrinsic functions for handling arrays:

```
real x(100), x0, x1
integer, dimension(1):: k0, k1, i0, i1

k0 = lbound(x)  ! lower bound of the index
k1 = ubound(x)  ! upper bound

x0 = minval(x)  ! smallest value
x1 = maxval(x)  ! greatest value

i0 = minloc(x)  ! index of the smallest value
i1 = maxloc(x)  ! index of the greatest value
```

Note that lbound, ubound, minloc and maxloc return an array, since they work also for multidimensional arrays.

```
real x(2,3)
integer lo(2), hi(2), i1(2)
x(1,:) = (/ 1, 2, 3 /)
x(2,:) = (/ 2, 5, 1 /)
lo=lbound(x)  !  lo = (/ 1, 1/)
hi=ubound(x)  !  hi = (/ 2, 3/)
i1=maxloc(x)  !  i1 = (/ 2, 2/)

real, dimension (10:14) :: &
    x=(/ 0.0, 1.0, 5.0, 2.0, 1.0 /)
integer, dimension(1):: l, n, i

l = lbound(x)  !  l = (/ 10 /)
n = ubound(x)  !  n = (/ 14 /)
i = maxloc(x)  !  i = (/ 3 /)
```

Vector and matrix operations:

```
real, dimension (10,10) :: a, b, c
real, dimension(10) :: v1, v2
real z, s, p

s = sum(a)            ! sum of elements
p = product(v1)         ! product of elements
z = dot_product(v1, v2)  ! scalar product
c = matmul (a, b)       ! matrix product
```

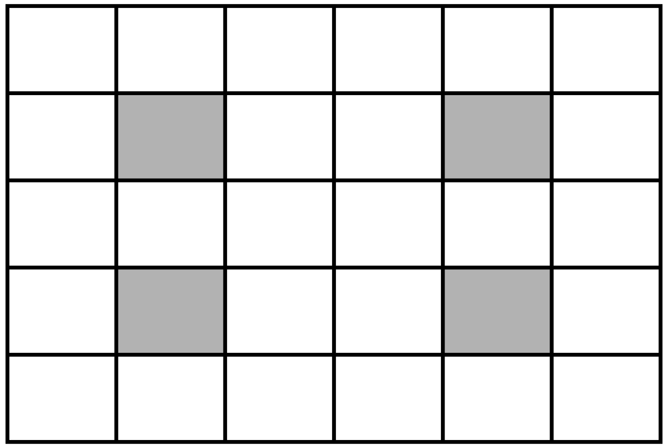
Array sections

An operand can also be a part of an array, an array section.

```
real  a(100), b(100), c(100), d(10,100)
integer i,j

b(1:10) = c(51:60)
a=d(1,:)
c=a(100:1:-1)+b

i=4 ; j=20
d(1:3,1:5) = d(i:i+4:2, j:j+4)

d(:,1) = d(5, 10:19)
```

Array section is an arbitrary rectangular equally spaced grid.

```
real a(5,6), x(2,2)
...
x = a(2:5:2, 2:6:3)
```



The right hand side of an assignment is always evaluated before the assignment.

```
integer  a(3)
a= (/ 0, 1, 2 /)
a (2:3) = a (1:2)  ! a=(/ 0, 0, 1 /)
```

This behaves in a different way than the loop

```
do i=2,3
  a(i) = a (i-1)
end do
```

`where`-statement:

```
where (x /= 0.0)  y=1/x
```

```
where (x > 0.0)
  y=1/x
  z=log(x)
end where
```

```
where (x > 0.0)
  y=1/x
  z=log(x)
elsewhere
  y=0.0
  z=0.0
end where
```