# Arrays: declaring them

- An array is an ordered sequence of many instances of the same data type.
- Computing equivalent of a vector or matrix.
- Are declared using the `DIMENSION` keyword:
  `<DTYPE>, DIMENSION( <id> ) :: <var>`
- the index descriptor `<id>` is a comma-separated list describing the index range in each dimension:
  - a single value $n$ means $n$ values in that dimension, with indices $\{1, 2, ..., n\}$
  - a pair a colon-separated values $n : m$ means $(m - n + 1)$ values in that dimension, with indices $\{n, (n + 1), ..., m\}$

# Examples of array declarations

```
! declare v1 to be a vector of 5 reals
!    with indices 1, 2, ..., 5
  REAL, DIMENSION(5) :: v1

! declare m1 to be a 2 x 2 array of integers
!    with indices 1, 2; 1, 2
  INTEGER, DIMENSION(2, 2) :: m1

! declare v2 to be a vector of 4 reals (of
!    KIND value 8) with indices -1, 0, 1, 2
  REAL(8), DIMENSION(-1:2) :: v2
```

# Accessing array elements

- We access a particular element of an array via its subscript values.
- So to assign to a variable `x` the value in the $m$-th row and $n$-th column of a matrix `A`, we would code:

```
x = A(m, n) ! assign array elt to x
```

- The index specifiers (e.g. $m$, $n$ above) should lie within the ranges given in the declaration.
- Sub-array can be accessed using a range-specifier (as in a declaration).

# Example: array initialisation

```
PROGRAM array_prog1
  IMPLICIT NONE
  REAL(8), DIMENSION(5) :: v1 ! assign REAL(8) vector of 5 e
  INTEGER :: i
  v1 = (/ 1.1, 1.2, 1.3, 1.4, 1.5 /) ! direct assigment
  PRINT*, v1 ! note low precision

  DO i = 1, 5                        ! assign via DO loop
    v1(i) = 1.0_8 + i * 0.1_8 ! note high precision
  END DO

  PRINT*, v1 ! note high precision
END PROGRAM array_prog1
```

# Example: array initialisation

```
PROGRAM array_prog2
  IMPLICIT NONE
  INTEGER, DIMENSION(2, 2) :: my_matrix
  INTEGER :: i, j

  DO i = 1, 2
    DO j = 1, 2
      my_matrix(i, j) = i + i + j - 2
    END DO
  END DO

  PRINT*, my_matrix
END PROGRAM array_prog2
```

# Example: array initialisation

```
PROGRAM array_prog3
  IMPLICIT NONE
  REAL, DIMENSION(5) :: v1
  REAL, DIMENSION(3) :: v2
  INTEGER :: i
  v1 = (/ 1.1, 1.2, 1.3, 1.4, 1.5 /) ! direct assigment

  v2 = v1(2:4)

  PRINT*, 'v1 = ', v1
  PRINT*, 'v2 = ', v2
END PROGRAM array_prog3
```

# Example array program

```
PROGRAM fibprog3
  IMPLICIT NONE
  INTEGER(8), DIMENSION(50) :: x ! declare array of length 5
  INTEGER :: n                   ! declare the iterator
  x(1) = 1                       ! initial conditions
  x(2) = 1                       ! initial conditions
  DO n = 3, 50                   ! begin loop
    x(n) = x(n-1) + x(n-2)       ! iteration step
  END DO                         ! end of loop
  PRINT*, x                      ! print array
END PROGRAM fibprog3
```

# Array operations

Given a function whose argument and return type are the same, Fortran allows that function to be applied to an array, all of whose elements are of that type; the given function is applied to each of the elements of the array.

```
PROGRAM array_prog4
  IMPLICIT NONE
  INTEGER, DIMENSION(3) :: v1 = (/ 1, 2, 3 /)
  REAL, DIMENSION(4) :: v2 = (/ 0.0, 0.1, 0.2, 0.3 /)

  v1 = v1**3
  PRINT*, v1

  v2 = SIN(v2)
  PRINT*, v2
END PROGRAM array_prog4
```

# Array operations

```
PROGRAM array_prog5
  IMPLICIT NONE
  INTEGER, DIMENSION(3) :: v1, v2, v3, v4

  v1 = (/ 1, 2, 3 /)  ! assign v1
  v2 = (/ 4, 5, 6 /)  ! assign v2

  v3 = v1 + v2 ! add corr. elts of v1, v2
  PRINT*, v3

  ! NB following line is NOT scalar / vector product
  v4 = v1 * v2 ! mult corr elts of v1, v2
  PRINT*, v4

END PROGRAM array_prog5
```

# The WHERE keyword

The WHERE keyword is used when you want to want to apply an operation to elements of an array dependent on a condition.

```
PROGRAM array_prog6
  IMPLICIT NONE
  INTEGER, DIMENSION(5) :: v1 = (/ -2, -1, 0, 1, 2 /)
  INTEGER, DIMENSION(5) :: v2
  v2 = ABS(v1) ! perform ABS on each elt of v1
  WHERE ( v1 < 0 ) v1 = -v1 ! equivalent to line above
  PRINT*, v1
  PRINT*, v2
END PROGRAM array_prog6
```

# Allocatable arrays

- In some circumstances the size of the array won't be known before runtime
  e.g. if the size of the array has to be read in from the user.
- This is done using the keywords `allocatable` and `allocate` .

# Allocatable arrays

```
PROGRAM alloc
  IMPLICIT NONE
  REAL(8), DIMENSION( : ), ALLOCATABLE :: my_vec
  INTEGER :: n, ncpts ! counter, #cpts
  PRINT*, 'Enter number of coordinates:' ! prompt
  READ*, ncpts ! get number of coordinates
  ALLOCATE( my_vec(ncpts) ) ! allocate array dim
  DO n = 1, ncpts ! assign cpts of array
    my_vec(n) = ACOS(-1.0_8) ** n
  END DO
  PRINT*, my_vec ! print array
END PROGRAM alloc
```

# Array intrinsic functions

There are lots intrinsic functions for arrays (go look at the list!).
We shall focus on

```
LBOUND   (array,   dim)
UBOUND   (array,   dim)
SIZE     (array,   dim)
```

which return the lower bound, upper bound, and extent, respectively, of the dimension `dim` of array `array` .

# Example: norm of $n$-dim vector

```
MODULE norm_mod2
  IMPLICIT NONE
CONTAINS
  REAL FUNCTION mynorm2(vec_arg) RESULT(res)
    REAL, DIMENSION(:), INTENT(IN) :: vec_arg
    REAL :: a = 0.0          ! a real variable
    INTEGER :: n, nc         ! some integers
    nc = SIZE(vec_arg, 1)    ! first index of vector
    DO n = 1, nc             ! calculate norm-squared
      a = a + vec_arg(n)**2
    END DO
    res = SQRT(a)            ! and square root it
  END FUNCTION mynorm2
END MODULE norm_mod2
```

# Example: norm of $n$-dim vector

```
PROGRAM norm5
  USE norm_mod2          ! USE MODULE norm_mod2
  IMPLICIT NONE
  REAL, DIMENSION(4) :: my_vec ! vector with 4 cpts
  REAL :: r                    ! real variable
  my_vec = (/ 0.1, 0.2, 0.3, 0.4 /) ! assign
  PRINT*, my_vec         ! print vector
  r = mynorm2(my_vec) ! get norm
  PRINT*, r              ! print norm
END PROGRAM norm5
```

# Accessing array subsets

- already seen how to access a subrange using a single subscript or a colon-separated pair
- possible to access any “hyper-rectangular” sub-array
- syntax: must supply a comma-separated list of index descriptors (one for each dimension)
- each index descriptor may be (in addition to single value or colon separated pair):
  - a colon (returns all the cpts in that dimension)
  - a colon preceded by a value:

```
my_array( i : ) ! is equiv to
my_array( i : UBOUND(my_array,1) )
```

  - a colon following by a value:

```
my_array( : i ) ! is equiv to
my_array( LBOUND(my_array,1) : i )
```

# Example: accessing rows/columns

```
PROGRAM array_prog2a
  IMPLICIT NONE
  INTEGER, DIMENSION(2, 2) :: my_matrix
  INTEGER :: i, j

  DO i = 1, 2
    DO j = 1, 2
      my_matrix(i, j) = i + i + j - 2
    END DO
  END DO

  PRINT*, my_matrix
  PRINT*, '1st row: ', my_matrix(1 , : ) ! print 1st row
  PRINT*, '2nd col: ', my_matrix( : , 2) ! print 2nd col

END PROGRAM array_prog2a
```

# Strings

- a “string” is an array of type `CHARACTER`
- Fortran has an additional (more intuitive?) way of dealing with strings.
- String length declared using the `LEN` keyword (see next slide).
- N.B. A single parameter to `CHARACTER(n)` declaration is LENgth, not KIND number.
- Substring referencing requires a colon, so to access $i$-th character of *my_string*, one would code: `my_string(i:i)`
- Concatenation operator `//`

# Example string program

```
PROGRAM string_prog1
  IMPLICIT NONE
  ! declare array in generic way; only generic array operati
  CHARACTER, DIMENSION(5) :: s1 = (/ 'H', 'e', 'l', 'l', 'o'

  ! declare string in string-specific way
  ! it allows concise initialisation and concatenation
  CHARACTER(LEN=12) :: s2 = 'Hello again!' ! declare & initi
  CHARACTER(19) :: s3 = ", what's your name?" ! LENgth 19
  CHARACTER(30) :: s4 ! uninitialised, LENgth 30

  s4 = s2(1:5) // s3  ! concatenate
  PRINT*, s4          ! print result
END PROGRAM string_prog1
```