Array Concepts
Fortran 90 uses the **DIMENSION** attribute to declare arrays. The most common examples are:

```fortran
INTEGER, DIMENSION(30) :: days_in_month
CHARACTER(LEN=10), DIMENSION(250) :: names
REAL, DIMENSION(350,350) :: box_locations
```

In Fortran the **starting index** defaults to a value of 1 (not 0 as is common in many other languages - C/C++/Python)
BUT you can specify a lower bound different than 1. It will just default to 1 if you omit it.

The syntax is `<lower bound>:<upper bound>` where the bound values are `INTEGER`s.

```plaintext
INTEGER, DIMENSION(0:99) :: arr1, arr2, arr3
CHARACTER(LEN=10), DIMENSION(1:250) :: names
REAL, DIMENSION(-10:10,-10:10) :: pos1, pos2
REAL, DIMENSION(0:5,1:7,2:9,1:4,-5:-2) :: pos1, pos2
```
Alternative Form

The same **base type** but different **bounds**:

\[
\begin{align*}
\text{INTEGER} & : \ arr1(0:99), \ arr2(0:99), \ days\_in\_month(1:12) \\
\text{REAL} & : \ box\_locations(1:350), \ pos1(-10:10,-10:10)
\end{align*}
\]

Don’t mix the two forms!
REAL :: A(0:99), B(3,6:9,5)

• The **rank** of an array is the number of dimensions. **The maximum number of dimensions is 7!**
  
  A has rank 1 and B has rank 3

• The **bounds** are the upper and lower limits.
  
  A has bounds 0:99 and B has bounds 1:3, 6:9 and 1:5

• The **extent** of an array dimension is the range of its index or indices. (upperbound-lowerbound+1)
The size of an array is the total number of elements. 
A has size 100 and B has size 60

The shape of an array is its rank and extents. 
A has shape (100) and B has shape (3,4,5)

Arrays are **conformable** if they share the same shape. The bounds do not have to be the same.
Array References

In general, there are **three** different ways to reference arrays:

- **individual** array elements  \( \text{arr}{1}(5), \text{myintval}(-10) \)
- **entire** array  \( \text{arr}{1} \) or \( \text{arr}{1}(:) \)
- **array section**  \( \text{arr}{1}(5:24), \text{arr}{1}(-10:-7) \)
An array index can be any integer expression e.g., \texttt{months(j)} selects the jth month

\begin{verbatim}
INTEGER, DIMENSION(-50:50) :: val
DO i = -50,50
  val(i) = 2*i
END DO
\end{verbatim}

Sets \texttt{val} to -100, -98, ..., 98, 100
Index Expressions

Set the even elements to the odd indices and vice versa

INTEGER, DIMENSION(1:80) :: series
DO K = 1,40
  series(2*K) = 2*K-1
  series(2*K-1) = 2*K
END DO

You can go completely overboard, too

series(int(1.0+80.0*cos(123.456))) = 42
Example of Arrays: Sorting

Sort a list of numbers into ascending order.
The top level algorithm is:

1. Read the numbers and store them in an array.
2. Sort them into ascending order of magnitude.
3. Print them out in sorted order.
Selection Sort

This is **NOT** how to write a general sort. It takes $O(N^2)$ time compared to $O(N \log(N))$.

For each location $J$ from 1 to $N-1$
   For each location $K$ from $J+1$ to $N$
      If the value at $J$ exceeds that at $K$
         Then swap them
   End of loop
End of loop

Let’s take a look: *sort10.f90*
Using Arrays as Objects

Set all the elements of an array to a single value

```
INTEGER, DIMENSION(1:50) :: series
series = 0
```

You can use entire arrays as simple variables provided they are conformable

```
REAL, DIMENSION(200) :: arr1, arr2
arr1 = arr2 + 1.23*exp(arr1/4.56)
```

The RHS and any LHS indices are evaluated, and then the RHS is assigned to the LHS.
Array sections create an aliased subarray. It is a simple variable with a value:

```fortran
INTEGER :: arr1(100), arr2(50), arr3(100)
arr1(1:63) = 5; arr1(64:100) = 7
arr2 = arr1(1:50) + arr3(51:100)
```

Even this is legal but it forces a copy:

```fortran
arr1(26:75) = arr1(1:50) + arr1(51:100)
```
Array Sections

A(1:6, 1:8)

A(1:3, 1:4)

A(2:5, 7)
Short Form

Existing array bounds may be omitted
Especially useful for multidimensional arrays

If we have `REAL, DIMENSION(1:6, 1:8) :: A`

- `A(3:, :4)` is the same as `A(3:6, 1:4)`
- `A, A(:, :)` and `A(1:6, 1:8)`

- `A(6, :)` is the 6th row as a 1-D vector
- `A(:, 3)` is the 3rd column as a 1-D vector
- `A(6:6, :)` is the 6th row as a 1x8 matrix
- `A(:, 3:3)` is the 3rd columns as a 6x1 matrix
Conformability of Sections

The **conformability** rule applies to sections, too.

```plaintext
REAL :: A(1:6, 1:8), B(0:3, -5:5), C(0:10)

A(2:5,1:7) = B(:, -3:3) ! both have shape (4,7)
A(4,2:5) = B(:, 0) + C(7:) ! all have shape (4)
C(:) = B(2,:) ! both have shape (11)
```

But these would be illegal

```plaintext
A(1:5,1:7) = B(:, -3:3) ! shapes (5,7) and (4,7)
A(1:1,1:3) = B(1,1:3) ! shapes (1,3) and (3)
```
Sections with Strides

Array sections need not be **contiguous**
Any **uniform progression** is allowed
This is **exactly** like a more compact **DO**-loop
Negative strides are allowed, too

```fortran
INTEGER :: arr1(1:100), arr2(1:50), arr3(1:50)
arr1(1:100:2) = arr2       ! Sets every odd element
arr1(100:1:-2) = arr3     ! Even elements, reversed
arr1 = arr1(100:1:-1)     ! Reverses the order of arr1
```
Strided Sections

\[ A(1:6, 1:8) \]
Array Bounds

Subscripts and sections must be within the array bounds. The following are invalid (undefined behavior):

REAL :: A(1:6, 1:8), B(0:3, -5:5), C(0:10)

A(2:5, 1:7) = B(:, -6:3)
A(7, 2:5) = B(:, 0)
C(:11) = B(2, :)

Most compilers will NOT check for this automatically! Errors will lead to overwriting, etc. and CHAOS.
Elemental Operations

Most built-in operators/functions are elemental. They act element-by-element on arrays.

```fortran
REAL, DIMENSION(1:200) :: arr1, arr2, arr3
arr1 = arr2 + 1.23*EXP(arr3/4.56)
```

Comparisons and logical operations, too:

```fortran
REAL, DIMENSION(1:200) :: arr1, arr2, arr3
LOGICAL, DIMENSION(1:200) :: flags
flags = (arr1 > EXP(arr2) .OR. + arr3 < 0.0)
```
Array Intrinsic Functions (1)

There are over 20 useful intrinsic procedures. They can save a lot of coding and debugging.

- `SIZE(x [,n]) ! The size of x (an integer scalar)`
- `SHAPE(x) ! The shape of x (an integer vector)`
- `LBOUND(x [,n]) ! The lower bound of x`
- `UBOUND(x [,n]) ! The upper bound of x`

If `n` is present then compute for that dimension only. And the result is an integer scalar. Otherwise, the result is an integer vector.
Array Intrinsic Functions (2)

\[
\begin{align*}
\text{MINVAL}(x) & \quad \text{! The minimum of all elements of } x \\
\text{MAXVAL}(x) & \quad \text{! The maximum of all elements of } x \\
\end{align*}
\]

These return a \textit{scalar} of the same \textit{type} as \(x\)

\[
\begin{align*}
\text{MINLOC}(x) & \quad \text{! The indices of the minimum} \\
\text{MAXLOC}(x) & \quad \text{! The indices of the maximum} \\
\end{align*}
\]

These return an \textit{integer vector}, just like \texttt{SHAPE}
Array Intrinsic Functions (3)

SUM(x [,n]) ! The sum of all elements of x
PRODUCT(x [,n]) ! The product of all elements of x

If n is present the compute for that dimension only

TRANSPOSE(x) means $X_{ij} \Rightarrow X_{ji}$
   It must have two dimensions but need not be square

DOT_PRODUCT(x,y) means $\sum_i X_i \cdot Y_i \Rightarrow Z$
   Two vectors, both of same length and type
Array Intrinsic Functions (4)

MATMUL(x,y) means \[ \sum_k X_{ik} \cdot Y_{kj} \Rightarrow Z_{ij} \]

2nd dimension of \( X \) must match the 1st of \( Y \)
The matrices need not be the same shape
Either \( X \) or \( Y \) may be a vector

Many more for array reshaping and array masking
Array Element Order (1)

This is also called the “storage order”

Traditional term is “column-major order”
But Fortran arrays are not laid out in columns!
Much clearer: “first index varies fastest”

REAL, DIMENSION(1:3,1:4) :: A

The elements of A are stored in this order:

A(1,1), A(2,1), A(3,1), A(1,2), A(2,2), A(3,2),
A(1,3), A(2,3), A(3,3), A(1,4), A(2,4), A(3,4)
Array Element Order (2)

Opposite to C, Matlab, Mathematica, IDL, etc.

You don’t often need to know the storage order
Three important cases where you do:

• **I/O of arrays**, especially unformatted
• **Array constructors** and **array constants**
• **Optimization** (caching and locality)
Arrays and sections can be included in I/O. These are expanded in array element order.

```fortran
REAL, DIMENSION(3,2) :: oxo
READ *, oxo
```

This is exactly equivalent to:

```fortran
READ *, oxo(1,1), oxo(2,1), oxo(3,1), &
       oxo(1,2), oxo(2,2), oxo(3,2)
```
Array sections can also be used

REAL, DIMENSION(100) :: nums
READ *, nums(30:50)

REAL, DIMENSION(3,3) :: oxo
READ *, oxo(:3), oxo(3:1:-1,1)

This last statement equivalent to:

READ *, oxo(1,3), oxo(2,3), oxo(3,3), 
&
  oxo(3,1), oxo(2,1), oxo(1,1)
Array Constructors (1)

Commonly used for assigning array values
An array constructor will create a temporary array

```fortran
INTEGER, DIMENSION(6) :: marks
marks = (/ 10, 25, 32, 54, 56, 60 /)
```

Constructs an array with the elements
10, 25, 32, 54, 56, 60
And then copies that array into marks

Fortran 2003 addition: Also can use square brackets

```fortran
marks = [ 10, 25, 32, 54, 56, 60 ]
```
Array Constructors (2)

Variable expressions are okay in constructors

\[ \text{marks} = (/ x, 2.0 \times y, \sin(t \times w/3.0), \ldots /) \]

They can be used anywhere an array can be
Except where you might assign to them!

All expressions must be the same type
This can be relaxed in Fortran 2003
Arrays can be used in the value list. They are flattened into array element order.

Implied DO-loops (as in I/O) allow sequences.

If \( n \) has the value 5:

\[
\text{marks} = (0.0, (k/10.0, k=2, n), 1.0)
\]

This is equivalent to:

\[
\text{marks} = (0.0, 0.2, 0.3, 0.4, 0.5, 1.0)
\]
Array constructors can be very useful for this.
All elements must be initialization expressions
i.e., ones that can be evaluated at compile time.

For rank one arrays just use a constructor:

\[
\text{REAL, PARAMETER :: } \text{a}(3) = (/ 1.23, 4.56, 7.89 /) \\
\text{REAL :: } \text{b}(3) = (/ 1.23, 4.56, 7.89 /) \\
b = \exp(b)
\]
Other types can be initialized in the same way:

```fortran
CHARACTER(LEN=4), DIMENSION(5) :: &
```

Initialization expressions are allowed:

```fortran
INTEGER, PARAMETER :: N = 3, M = 6, P = 12
INTEGER :: arr(3) = (/ N, (M/N), (P/N) /)
```
What about this?

```fortran
REAL :: arr(3) = (/ 1.0, exp(1.0), exp(2.0) /)
```

Fortran 90 does **NOT** allow this but Fortran 2003 **does**

Not just **intrinsic functions** but all sorts of things
Constructors cannot be nested - e.g., **NOT**:

```fortran
REAL, DIMENSION(3,4) :: xvals = &
(/ (/ 1.1, 2.1, 3.1 /), (/ 1.2, 2.2, 3.2 /), &
 (/ 1.3, 2.3, 3.3 /), (/ 1.4, 2.4, 3.4 /) /)
```

They construct only **rank one** arrays.

Use the **RESHAPE** intrinsic function to construct higher rank arrays.
Allocatable Arrays (1)

Arrays can be declared with an unknown shape
Use the ALLOCATABLE attribute in the type declaration

\[
\begin{align*}
\text{INTEGER, DIMENSION(:,:), ALLOCATABLE} & \quad :: \quad \text{counts} \\
\text{REAL, DIMENSION(:,:,,:), ALLOCATABLE} & \quad :: \quad \text{values}
\end{align*}
\]

They become defined when space is allocated

\[
\begin{align*}
\text{ALLOCATE(counts(1:1000000))} \\
\text{ALLOCATE(value(0:N,-5:5,M:2*N+1))}
\end{align*}
\]

You can also allocate multiple arrays in a single ALLOCATE statement
Allocatable Arrays (2)

Failures will terminate the program
You can trap most allocation failures

```fortran
INTEGER :: istat
ALLOCATE(arr(0:100,-5:5,7:14),STAT=istat)
IF (istat /= 0) THEN
    ...
ENDIF

Arrays can be deallocated using

DEALLOCATE(counts)
```
Example

INTEGER, DIMENSION(:,), ALLOCATABLE :: counts
INTEGER :: size, code

! -- Ask the user how many counts he has
PRINT *, ‘Type in the number of counts’
READ *, size

! -- Allocate memory for the array
ALLOCATE(counts(1:size), STAT=code)
IF (code /= 0.0) THEN
   PRINT *, ‘Error in allocate statement’
   ...
ENDIF
WHERE Construct (1)

Used for masked array assignment
Example: Set all negative elements of an array to zero

REAL, DIMENSION(20,30) :: array

DO j = 1,30
  DO k = 1,20
    IF (array(i,j) < 0.0) array(k,j) = 0.0
  ENDDO
ENDDO

But the WHERE statement is much more convenient

WHERE (array < 0.0) array = 0.0
WHERE Construct (2)

It has a `statement construct` form, too

Example: Set all negative elements of an array to zero

```
WHERE (array < 0.0)
array = 0.0
ELSE WHERE
array = 0.01 * array
ENDWHERE
```

Masking expressions are `LOGICAL` arrays
You can use an actual array there, if you want
Masks and assignments need the same shape