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
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:

```
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:

```
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)

A(3, 1:5:2)

A(2:6:2, 7)
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)
```
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, the compute for that dimension only. And the result is an integer scalar. Otherwise, the result is an integer vector.
MINVAL(x) ! The minimum of all elements of x
MAXVAL(x) ! The maximum of all elements of x

These return a scalar of the same type as x

MINLOC(x) ! The indices of the minimum
MAXLOC(x) ! The indices of the maximum

These return an integer vector, just like 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)**

$\text{MATMUL}(x,y)$ 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
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

```
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

```
marks = [ 10, 25, 32, 54, 56, 60 ]
```
Variable expressions are okay in constructors

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

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
Array Constructors (3)

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:

```plaintext
REAL, PARAMETER :: a(3) = (/ 1.23, 4.56, 7.89 /)
REAL :: b(3) = (/ 1.23, 4.56, 7.89 /)
b = exp(b)
```
Other types can be initialized in the same way

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

Initialization expressions are allowed

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

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
Multiple Dimensions

Constructors cannot be nested - e.g., NOT:

\[
\text{REAL, DIMENSION(3,4) :: xvals = } & \\
(\text{/ (}/ 1.1, 2.1, 3.1 /), (}/ 1.2, 2.2, 3.2 /), & \\
(}/ 1.3, 2.3, 3.3 /), (}/ 1.4, 2.4, 3.4 /)/) \\
\]

They construct only rank one arrays

Use the \text{RESHAPE} intrinsic function to construct higher rank arrays. We’ll cover this later if time permits.
Allocatable Arrays (1)

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

\[
\text{INTEGER, DIMENSION}(:,:,\text{ALLOCATABLE :: counts}}
\]
\[
\text{REAL, DIMENSION}(:,:,(:,:,\text{ALLOCATABLE :: values}}
\]

They become defined when space is allocated

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

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

```
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
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

```plaintext
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