Fortran 90  Arrays

- To store lists of data, all of the same type, use arrays.
  - *e.g.* lists of exam marks, or the elements of a position vector \((x, y, z)\).
  - A matrix \[
  \begin{bmatrix}
  a_{11} & a_{12} \\
  a_{21} & a_{22}
  \end{bmatrix}
  \] is a two-dimensional array.
  - In Fortran all elements of an array are of the *same type* and have the same name. They are distinguished by a subscript or array index. Subscripts are consecutive integers.
  - Arrays are named in the same way and have the same types as scalar variables.
  - Arrays are declared, like variables, in type statements. These give their dimensions (sizes).
Arrays (cont.)

- e.g. REAL, DIMENSION(20) :: L, B
  REAL, DIMENSION(4,20) :: F
  DOUBLE PRECISION, DIMENSION(2,3) :: A, B
  INTEGER, DIMENSION(31) :: Exam_Marks

- The first array element is 1, i.e. L(1) to L(20), etc.

- You may specify the first and last elements (lower and upper bounds) in each array dimension.
  e.g. REAL, DIMENSION(-5:14) :: L would define L(-5) to L(14). (but generally why bother?)

- These declaration statements reserve space in computer memory.

- Array sizes must be specified by INTEGER constants. The maximum number of dimensions (rank) is 7.

- The shape of an array is a list (1 dimensional array) of the number of elements in each dimension. The SHAPE function gives this. Try PRINT *, SHAPE(array_name). Empty for a scalar.
Arrays (cont.)

• Each array element is a variable just like any scalar variable.
  – Array elements (also known as subscripted variables) may be used anywhere in executable statements where a scalar variable of the same type would be used.
  – e.g. `DOUBLE PRECISION :: S, Y
          S = X(1) + X(2) + X(3)
          Y = A(2,1) + B(1,3)
          X(1) = Y + S`

  – Each subscript may be represented by an INTEGER expression
  – e.g. `INTEGER :: J, K
          J = 1
          S = X(J) + X(J+1) + X(J+2)
          K = 2
          Y = A(K,J) + B(J+K)`
Arrays (cont.)

- **WARNING**
  A computed subscript **must** be within the declared bounds of each dimension of the array. The compiler cannot check this. If not, an "out of bounds" run-time error may be produced. The Salford compiler has an option which will include code in the compiled program to check the bounds. If run-time checking is not included the program may simply read from or write to some other location in computer memory which could create havoc!!

- **Another WARNING!**
  Arrays and functions are accessed in the same way so must be declared correctly, arrays with dimensions, subscripts like scalar variables. An undeclared array may be mistaken for a function and produce an incomprehensible error message.
Arrays  Named constants in declarations

• If you have a lot of arrays of the same or related sizes you can use the PARAMETER attribute to create a named constant to use in array declarations.
  – INTEGER, PARAMETER :: ISIZE = 100
  REAL, DIMENSION(ISIZE) :: ARRAY1, ARRAY2
  REAL, DIMENSION(2*ISIZE) :: ARRAY3
  – If later you want to change the size of these arrays you only have to change one statement, not many.
Arrays  Array constants

• An array constructor is a list of constants of the appropriate type between the symbol pairs (/ and /)
  – *e.g.* an INTEGER array ARR of size 5 could have its elements set to 1, 2, …, 5 by
  – \[ \text{ARR} = (/ 1, 2, 3, 4, 5 /) \]
  – If some, or all, of the elements are related in a regular way we can use an *implied DO-loop* (same structure as the DO variable control in a DO statement)
  – *e.g* \[ \text{ARR} = (/ (I, I=2, 8, 2), 0 /) \]
  – The list of values in an array constructor **must contain exactly** the same number of values as the size of the array to which it is assigned.
Arrays  Initializing

• Like scalar variables arrays and array elements must be initialized.
  – In a declaration statement
    INTEGER, DIMENSION(5) :: &
    ARR = (/ 1, 3, 6, 7, 9 /)
  – In an assignment statement
    ARR = (/ 1, 2, 3, 4, 5 /)
  – In either case, if all elements of the array are to be set to the same value we may assign a scalar constant
    ARR = 0
  – Set elements individually, for instance using DO-loops for initializing and processing arrays.
    ARR(1) = 0
  – Nested DO-loops can process multi-dimensional arrays.
**Arrays**  **Initializing** *(cont.)*

- *Example:*

  ```fortran
  INTEGER :: I, J
  REAL, DIMENSION(20) :: FTOTAL
  REAL, DIMENSION(4,20) :: F
  DO J = 1,20
      FTOTAL(J) = 0.
      DO I = 1,4
          FTOTAL(J) = FTOTAL(J) + F(I,J)
      END DO
      PRINT *, J, FTOTAL(J)
  END DO
  ```

- This fragment might be a subroutine which has been given the array *F* containing a table of values and prints the sum of each row (or column) of the table.
- In Fortran 90 we could put *FTOTAL = 0.* outside the loops.
Arrays  Input/Output

• Consider a matrix operation to solve simultaneous equations

\[
\begin{bmatrix}
Y_1 \\
Y_2 \\
Y_3
\end{bmatrix} =
\begin{bmatrix}
A_{11} & A_{12} & A_{13} \\
A_{21} & A_{22} & A_{23} \\
A_{31} & A_{32} & A_{33}
\end{bmatrix}
\begin{bmatrix}
X_1 \\
X_2 \\
X_3
\end{bmatrix}
\]

• To read in A and X and print Y
  
  REAL, DIMENSION(3) :: X, Y
  REAL, DIMENSION(3,3) :: A
  INTEGER :: I, J
  READ *, A  All 9 values in A are read in column order
    A(1,1), A(2,1), A(3,1), A(1,2), etc.
  READ *, X  will read X(1), X(2), X(3)
Arrays  Input/Output (cont.)

- :  
  Y = 0.  
  DO I = 1,3  
    DO J = 1,3  
      Y(I) = Y(I) + A(I,J)*X(J)  
    END DO  
  END DO  
  PRINT *, Y  

Prints Y(1), Y(2), Y(3)

- Fortran always stores by **columns** - the first subscript varies more rapidly than the second, and so on.
- Data in/out Col. 1  Col. 2  etc.
Arrays  Input/Output (cont.)

• I/O of individual and grouped elements:
  – *e.g.* `READ *, X(1), X(3)`
  – If we print a matrix using `PRINT *, A` the values will be output as shown, only starting a new line when one is full.

```
A(1,1)        A(1,2)        A(1,3)
A(2,1)        A(2,2)        A(2,3)
A(3,1)        A(3,2)        A(3,3)
```

• Memory storage works the same way!
  – Stores by 1\textsuperscript{st} subscript, then 2\textsuperscript{nd}
  – Important for speed / efficiency
Arrays  Input/Output (cont.)

– To print the matrix on 3 lines:
– DO I = 1,3
   PRINT *, (A(I,J), J=1,3)
END DO
– A(I,1), A(I,2) and A(I,3) will be printed on the same line, equivalent to
   PRINT *, A(I,1), A(I,2), A(I,3)
and so on for J=2 and 3

• Use implied DO-loops to READ 2-dimensional arrays so you can read by rows (not columns).
Arrays    Operations

• An array element can be used like a scalar variable.
  – Alter an array element's subscript to make it refer to a different location.

• A new, very important feature of Fortran 90:

• In Fortran 90 an array can be processed as a single object.
  – Any operation between two scalar variables can be performed on arrays, provided they are conformable.

• Conformable arrays have the same shape
  – same rank (same number of dimensions)
  – same extent in each dimension (but upper and lower bounds need not be)

• A scalar, including a constant, is conformable with any array.
Arrays  Operations (cont.)

- Operations are carried out element by element.
- REAL, DIMENSION(20) :: A, B, C
- C = A*B
  - DO I = 1,20
  -   C(I) = A(I)*B(I)
  END DO

Array operation
Equivalent explicitly on individual elements (old style)

- C = 10.*A
  All elements operated on by same scalar

- B = 5.
  All elements set to same value.

- C = SIN(A)
  All elements operated on by elemental intrinsic function.
Arrays   Subprograms

• Example: Matrix multiplication is rather common so it would be useful to put it in a reusable subroutine.
  – PROGRAM MATMLPY_TEST
     REAL, DIMENSION(3) :: X, Y
     REAL, DIMENSION(3,3) :: A
     READ *, A
     READ *, X
     CALL MATMLPY(Y, A, X, 3, 3)
     PRINT *, Y
     END PROGRAM MATMLPY_TEST

  – Array names are used as arguments in the subroutine call and the array dimensions (3,3) are automatically passed to the subroutine.
Arrays  Subprograms (cont.)

- SUBROUTINE MATMLPY(Q, R, P, M, N)
  INTEGER :: M, N ← Array dimensions       Dummy
  REAL :: P(M), Q(N), R(M,N)               } variables
  INTEGER :: I, J                             } Local variables
  DO I = 1,N
    Q(I) = 0.0
    DO J = 1,M
      Q(I) = Q(I) + R(I,J)*P(J)
    END DO
  END DO
END SUBROUTINE MATMLPY

- In fact this one is so useful that it's available as an intrinsic function in Fortran 90.  \( Y = \text{MATMUL}(A, X) \)
- Function results can be arrays.
- Dimensions can be automatically passed to the subprogram.
Arrays Subprograms (cont.)

- Notice how the method was generalized to work with rectangular matrices by passing both dimensions of the array.

\[
\begin{bmatrix}
Y_1 \\
Y_2 \\
Y_3
\end{bmatrix} = \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22} \\
A_{31} & A_{32}
\end{bmatrix} \begin{bmatrix}
X_1 \\
X_2
\end{bmatrix}
\]
Arrays  Subprograms (cont.)

- Subroutine parameter passing:
  - Main       Subroutine       Main
  - \( 3 \Rightarrow M \)       \text{INTENT}
  - \( 3 \Rightarrow N \)       \text{IN}
  - address of \( X \) \( \Rightarrow P \)       \text{IN}
  - " " \( A \) \( \Rightarrow R \)       \text{IN}
  - " " \( Y \) \( \rightarrow Q \Rightarrow Y \) address       \text{OUT}
  - The subroutine doesn't have to reserve space in memory for
dummy variables and arrays. The CALL tells it where to find them
in the main program space.
  - The \text{INTENT} attribute may be used in dummy argument
declarations to show direction of data flow. (Also \text{INOUT})
    - Limits data flow to one direction (or both)
  - \text{e.g. INTEGER, INTENT (IN) :: M, N}
Arrays   Subprograms (cont.)

• Array dimensions
  – May be INTEGER dummy variables and passed as integer expressions (constants or variables).
  – If array itself is also a dummy argument, as in the example, they must evaluate to values within array dimensions declared in main program.
    • i.e., it has to fit to pass back
  – Dummy array dimensions may also define completely new arrays in the subprogram.

  – Often used for temporary work space required by subprogram, amount required depending on arguments passed to it.

  – Such methods, and others in Fortran 90, allow the re-use of subprograms with different array sizes.
Arrays  Memory allocation

• Memory for arrays that are not local to a subprogram is allocated by the calling program, ultimately by the main program.
• But you may not know the actual size of your datasets, or they may vary.
  – To change dimensions you must edit source code and recompile.
    • (Note advantage of PARAMETER attribute.)
    – OK for small programs, not so good for large ones.

• Allocate enough space to handle maximum amount of data.
  – May not always be needed. : inefficient
Arrays Memory allocation (cont.)

Physical dimensions of array
Logical dimensions
(i.e. what is needed at the time)

Depending on how data is organised, you may need to pass both sets of dimensions to subprogram.

e.g. array is (7,8), matrix is (3,3)

Data in memory is Col 1 Col 2 etc

So physical length of column would be needed to access x.
Arrays  Memory allocation (cont.)

• Trivial example!
  – Given an integer \( n \) and real values \( x_1, x_2, \ldots, x_n \), write a program to output the \( x \)-values in reverse order \( x_n, x_{n-1}, \ldots, x_1 \).
    Assume \( n \leq 1000 \)
  – **PROGRAM REVERSE**
    INTEGER :: N, I
    REAL, DIMENSION(1000) :: X
    READ *, N, (X(I),I=1,N)
    DO I = N,1,-1
        PRINT *, X(I)
    END DO
    END PROGRAM REVERSE
  – Quite a lot of space in memory may be wasted by dimensioning \( X \) to 1000.
Allocatable Arrays

- **Fortran 90: A better way.**
  - The **ALLOCATABLE** attribute allows the shape and size of an array to be deferred until run time.

```fortran
REAL, ALLOCATABLE, DIMENSION(:) :: X
READ *, N
ALLOCATE ( X(N) )
READ *, X
```

- The *rank* of an allocatable (or deferred-shape) array is specified by the number of colons in the **DIMENSION** attribute.
  - e.g. **REAL, ALLOCATABLE, DIMENSION(:,:) :: Y**

- The **ALLOCATE** statement may be used to allocate dimensions to more than one array,
  - e.g. **ALLOCATE ( X(N), Y(M,N) )**