OUTLINE

1 REVIEW

2 MORE ARRAYS
   • Using Arrays
   • Why do we need dynamic arrays?
   • Using Dynamic Arrays

3 MODULES
   • Global Variables
   • Interface Blocks
   • Modular Programming

4 FINAL REVIEW
THE STORY SO FAR...

- Create arrays and new types of data
- Advanced input/output
- How to structure the flow of your program
- Debug your programs
**Multidimensional Arrays**

- We can easily declare multidimensional arrays using the command

  \[
  \text{REAL :: a}(10,5), \text{b}(5,10)
  \]

- Again we access elements of the array using subscripts, now separated by a comma

  \[
  \begin{align*}
  \text{a}(1,2) &= \text{b}(1,1) + 10. \\
  \text{a}(10,:) &= (/ 1,2,3,4,5 /) \\
  \text{a}(1:5,1) &= (/ 1,2,3,4,5 /)
  \end{align*}
  \]

- We can use : to indicate all elements in a dimension or a subsection of them
THE SHAPE OF THE ARRAY

- Fortran has some intrinsic functions to use with arrays
- In order for the functions to work the arrays they must be conformal

```fortran
REAL :: a(3,2), b(2,3), c(2,2)
! assign values to a and b
a(1,1) = ...
! use function MATMUL
c = MATMUL(a,b)
```

- The rank is the number of dimensions
- The shape of an array is a vector of the extent of each dimension
- If arrays have the same shape we can add and multiply them together
PASSING AN ARRAY TO A SUBPROGRAM

- We can pass an array to a subroutine or a function as an argument

```plaintext
PROGRAM test
REAL :: a(5), x, y
x = my_func(a)
CALL my_sub(a, x, y)
```

- The subroutine and function must know the size of the array

```plaintext
SUBROUTINE my_sub (a, x, y)
REAL, INTENT(INOUT) :: a(5), x, y
a(1) = ...
```

- We cannot at least in this simple way return an array from a function
As your programs become more and more complex, the time needed to compile them will increase

We therefore want to specify as much as we can at runtime

So far we can only change the size of an array in the code

**Dynamic** array allocation allows the size of arrays to be set at runtime
We declare a array without size in the following way

```fortran
REAL, ALLOCATABLE :: a(:), two_dim_a(:, :)
```

Then to specify the size of the array at runtime we write

```fortran
PRINT *, 'Input size of a and two_dim_a
READ *, n, m, p
ALLOCATE ( a(n), two_dim_a(m, p), STAT=error )
```

and deallocate or delete the memory space with the statement

```fortran
DEALLOCATE (a, two_dim_a)
```
If we pass a dynamic array to a subprogram we must tell it about the size of the array

CALL my_sub(a,n)

and inside the subroutine

SUBROUTINE my_sub(a,n)
INTEGER, INTENT(IN) :: n
! declare n before a
REAL, INTENT(INOUT) :: a(n)
DO i = 1,n
   a(i) = ...
END DO
DATA FOR ALL TO SEE

- We often wish to allow lots of our subprograms to see the same piece of data
- This could be:
  - The number of points used in a finite difference scheme
  - The precision of the data variables
  - A mathematical constant such as $\pi$
- To include a module (and let the program have access to the variables) - we simply write the following statement at the top of each program

```plaintext
USE module_name
```
Example Module

```
MODULE global_data
!  double precision real number parameter
INTEGER, PARAMETER :: DP = KIND(1.0D0)
!  mathematical constant pi
REAL(DP), PARAMETER :: &
PI_D=3.14159265358979323846264338327_dp
!  number of points in finite difference
INTEGER :: no_of_points
END MODULE global_data
```

We can then set real numbers to double precision with

```
REAL(dp) :: x, y, z
```
CONNECTING PROGRAMS

- We use interface blocks to tell the compiler how to connect together functions
- An interface can allow us to return an array from a function
- or assume the shape of an array on entry to a function
- We can also use it to overload functions
- See “Fortran 90 Programming” by Ellis, Philips and Lahey for more information on overloading
**Example - Function using assumed array shape**

### Main Program

```fortran
PROGRAM test
!
interface at the top

INTERFACE
  FUNCTION add_all_elements(a)
    IMPLICIT NONE
    REAL, INTENT(IN) :: a(:)
    REAL :: add_all_elements
  END FUNCTION
END INTERFACE

! create allocatable array with n elements
REAL :: a(:)
...
ALLOCATE(a(n))
PRINT *, add_all_elements(a)
END PROGRAM test
```
EXAMPLE - FUNCTION USING ASSUMED ARRAY SHAPE

FUNCTION

FUNCTION add_all_elements(a)
  IMPLICIT NONE
  REAL, INTENT(IN) :: a(:)
  REAL :: add_all_elements
  INTEGER :: n, i
  n = size(a)
  add_all_elements = 0.
  DO i=1,n
    add_all_elements = add_all_elements + a(i)
  END DO
END FUNCTION
If we put the function in previous example inside a module, the module will create the interface for us.

We use the keyword **CONTAINS** to put functions or subroutines into a module.

```fortran
MODULE array_ops
  CONTAINS
  FUNCTION add_all_elements(a)
    IMPLICIT NONE
    REAL, INTENT(IN) :: a(:)
    ...
  END FUNCTION
END MODULE
```
Sometimes we may wish to connect functions or subroutines to a data type.

For example, there are many functions we could associate with the data type point...

such as the distance function.

so we can put the function into the module along with the data type.

when each data type is called it will have functions associated with it.
WHAT YOU CAN DO NOW...

- Able to produce, compile and run a fortran program
- Use subroutines and functions to structure your program
- Control the flow of your program
- Read and write from files and control the way data is formatted
- Use simple modules to declare global variables