

# Fortran Arrays

"The Angry Penguin", used under creative commons licence from Swantje Hess and Jannis Pohlmann.



"All parts should go together without forcing. You must remember that the parts you are reassembling were disassembled by you. Therefore, if you can't get them together again, there must be a reason. By all means, do not use a hammer."

IBM Maintenance Manual 1925

# Arrays

- Idea of an array is very simple
  - Pack together a set of N items of the same type and allow access to each individual element by a numerical index, c.f a vector of elements
  - Arrays are guaranteed to be contiguous in the underlying memory
- Are also higher **rank** arrays (2D, 3D etc.) but we'll start with simple 1D arrays
- You can declare an array anywhere you can declare a normal variable

# Arrays

```
PROGRAM array_test
  IMPLICIT NONE
  !Array runs from 1->10
  !Different to most other languages
  !that run 0->9
  INTEGER, DIMENSION(10) :: array
  INTEGER :: i

  !SIZE is an intrinsic function that
  !tells me how large my array is
  DO i = 1, SIZE(array)
    array(i) = i
  END DO

  !Can print whole arrays although large
  !arrays are hard to read
  PRINT *, array

END PROGRAM array_test
```

# Arrays

- Creating a fixed dimension array is very simple
  - Give your variable the **DIMENSION** attribute
  - Inside the brackets of the **DIMENSION** attribute specify the number of elements in the array
- Previous example showed how to set the elements of the array to the numbers 1,2,3, ... using a loop
  - There are other ways

# Array literals

```
PROGRAM array_test
  IMPLICIT NONE
  !Array runs from 1->10
  !Different to most other languages
  !that run 0->9
  INTEGER, DIMENSION(10) :: array
  INTEGER :: i

  !Can also use an array literal
  !Surround a comma separated list of
  !numbers with square brackets
  array = [1,2,3,4,5,6,7,8,9,10]

  !Alternate (older) style of array literal uses (/ /)
  array = (/1,2,3,4,5,6,7,8,9,10/)

  !Can print whole arrays although large
  !arrays are hard to read
  PRINT *, array

END PROGRAM array_test
```

# Implied DO loops

```
PROGRAM array_test
  IMPLICIT NONE
  !Array runs from 1->10
  !Different to most other languages
  !that run 0->9
  INTEGER, DIMENSION(10) :: array
  INTEGER :: i

  !Can use an IMPLIED DO LOOP
  !to set up this array (c.f. linspace etc.)
  array = [(i,i=1,SIZE(array))]

  !Can print whole arrays although large
  !arrays are hard to read
  PRINT *, array

END PROGRAM array_test
```

# Arrays

- By default Fortran arrays run from 1 to the number of elements in the array
- Most other languages tend to run from 0 to  $n - 1$
- Unlike most other languages you can freely change this by specifying (lower\_bound:upper\_bound) in **DIMENSION**

# Arrays

```
PROGRAM array_test
  IMPLICIT NONE
  !Runs 1->10
  INTEGER, DIMENSION(10) :: array1
  !Runs 1->10 but now made explicit
  INTEGER, DIMENSION(1:10) :: array2
  !Runs 0->9
  INTEGER, DIMENSION(0:9) :: array3
  !Runs -5->4
  INTEGER, DIMENSION(-5:4) :: array4

  PRINT *, 'Array1 :', SIZE(array1), LBOUND(array1), UBOUND(array1)
  PRINT *, 'Array2 :', SIZE(array2), LBOUND(array2), UBOUND(array2)
  PRINT *, 'Array3 :', SIZE(array3), LBOUND(array3), UBOUND(array3)
  PRINT *, 'Array4 :', SIZE(array4), LBOUND(array4), UBOUND(array4)

END PROGRAM array_test
```

# Arrays

```
PROGRAM array_test
  IMPLICIT NONE
  !Runs 1->10
  INTEGER, DIMENSION(10) :: array1
  !Runs 1->10 but now made explicit
  INTEGER, DIMENSION(1:10) :: array2
  !Runs 0->9
  INTEGER, DIMENSION(0:9) :: array3
  !Runs -5->4
  INTEGER, DIMENSION(-5:4) :: array4

  PRINT *, 'Array1 :', SIZE(array1), LBOUND(array1), UBOUND(array1)
  PRINT *, 'Array2 :', SIZE(array2), LBOUND(array2), UBOUND(array2)
  PRINT *, 'Array3 :', SIZE(array3), LBOUND(array3), UBOUND(array3)
  PRINT *, 'Array4 :', SIZE(array4), LBOUND(array4), UBOUND(array4)

END PROGRAM array_test
```

Array1 :	10	1	10
Array2 :	10	1	10
Array3 :	10	0	9
Array4 :	10	-5	4

# Arrays

- Note the intrinsic functions
  - SIZE - Total number of elements in an array
  - LBOUND - Lower bound of array, technically returns an array!
  - UBOUND - Upper bound of array, technically returns an array!

# Arrays

- You've seen that you can use numerical literal values to specify array bounds, but what else can you use?
  - Literal expressions (i.e.  $1+10$  is fine)
  - You can use INTEGER, PARAMETER variables to define bounds and expressions involving them
  - You can use INTEGER dummy variables to a function to define bounds
  - **ABSOLUTELY NOT normal variables or expressions containing normal variables**

# Allocatable Arrays

- You can have arrays that change size depending on normal variables and they are called **Allocatable Arrays**
  - Technical reasons for why they are different but we won't go into that
- Another attribute is added to the variable declaration
  - **ALLOCATABLE**
  - Also put a ":" into the **DIMENSION** attribute to indicate that the size isn't known yet

# Allocatable Arrays

- There are then two intrinsic functions that allocate and deallocate memory for allocatable arrays
  - **ALLOCATE(array\_name(range\_spec))**
  - **DEALLOCATE(array\_name)**
- **range\_spec** can be anything that is valid for a constant size array but can use normal variables
  - So you can specify upper and lower bounds etc
  - **range\_spec** is evaluated when the ALLOCATE command happens

# Allocatable Arrays

```
PROGRAM array_test
  IMPLICIT NONE
  INTEGER, DIMENSION(:), ALLOCATABLE :: array
  INTEGER :: array_count

  array_count = 10
  ALLOCATE(array(array_count))
  PRINT *, 'V1 :', SIZE(array), LBOUND(array), UBOUND(array)
  DEALLOCATE(array)
  ALLOCATE(array(0:array_count-1))
  PRINT *, 'V2 :', SIZE(array), LBOUND(array), UBOUND(array)
  DEALLOCATE(array)

END PROGRAM array_test
```

V1 :	10	1	10
V2 :	10	0	9

# Allocatable Arrays

- Once an array is allocated you have to deallocate it before you can allocate it again
  - This is **good** because most compilers will detect this so it means that you can't leak memory by accidentally reallocating memory without freeing it
- Allocatable arrays defined in functions are deallocated when you leave the function (unless they have the **SAVE** attribute (see later for this))
  - Different rules apply for arrays that are passed into a function as an argument

# Allocatable Arrays

- Allocatable arrays have an “allocated” property that you can check with the “**ALLOCATED**” function
- Before you allocate an array is not allocated
- After you allocate it it is allocated
- After you deallocate it it is not allocated
- Lots of array functions like SIZE etc. will give strange answers on unallocated arrays

# Allocatable Arrays

```
PROGRAM alloc_test

    INTEGER, DIMENSION(:), ALLOCATABLE :: array

    PRINT *, 'Before allocation ', ALLOCATED(array)
    ALLOCATE(array(10))
    PRINT *, 'After allocation ', ALLOCATED(array)
    DEALLOCATE(array)
    PRINT *, 'After deallocation ', ALLOCATED(array)

END PROGRAM alloc_test
```

```
Before allocation  F
After allocation   T
After deallocation F
```

# Implicit Allocation

- When an allocatable array has another array assigned to it with **=** it will reallocate itself to the size of that array
- This is why **array = [array,new\_element]** that we introduced at the start works, so long as **array** is an allocatable array
- You create a new element containing all of the existing elements of the array and then a new element, and then assign it back to the array which is reallocated to be size of the newly formed array
- This only applies to assignment to the whole array, assigning to a **subsection** (see later) doesn't change the array's size (even if the subsection is the whole array)

# Arrays and Subprograms

- You pass an array into a subprogram by specifying an array type variable as a dummy argument to the subprogram in the normal way
  - Just add the **DIMENSION** attribute to the dummy variable
  - Even if you are working with **ALLOCATABLE** arrays you only **have** to flag the dummy variable in a subprogram with **ALLOCATABLE** if you want to allocate or deallocate the array inside the subprogram
    - There are other reasons that you might want to do that

# Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(array_in)
    INTEGER, DIMENSION(10), INTENT(IN) :: array_in

    PRINT *, 'Array is : ', SIZE(array_in), &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod
```

```
PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(10) :: array
    CALL test_array(array)

END PROGRAM test
```

```
Array is :           10           1           10
```

# Arrays and Subprograms

- That works fine for arrays where you know the size when you are writing the code but not if you are working with allocatable arrays where you don't know the size until the code runs
- You can write functions that take arguments that are called "deferred shape arrays" that take on any size that they need to when the code runs
  - Sounds fancy, but just replace the array size in the **DIMENSION** statement with a ":" again
- You can still use the **SIZE** etc. functions to find out how large they are

# Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(array_in)
    INTEGER, DIMENSION(:), INTENT(IN) :: array_in

    PRINT *, 'Array is : ', SIZE(array_in), &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod

PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(10) :: array
    CALL test_array(array)

END PROGRAM test
```

# Bounds problems in subprograms

- There is one really, really annoying feature of Fortran arrays that is a hold over from FORTRAN 77
- When you pass an array into a subprogram it's lower bound always maps back to one
- **JUST THE NUMBERING**
- All of your array is still there but the lowest element is now accessed through the value 1 rather than the previous lower bound

# Bounds problems in subprograms

- You can change this behaviour by
  - You explicitly set the lower bound in declaration of the dummy variable to the subprogram
  - You give the dummy variable either the **ALLOCATABLE** or **POINTER** attribute (we cover **POINTER** in the extra materials)

# Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(array_in)
    INTEGER, DIMENSION(10), INTENT(IN) :: array_in

    PRINT *, 'Array is : ', SIZE(array_in), &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod

PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(0:9) :: array
    CALL test_array(array)

END PROGRAM test
```

# Arrays and Subprograms

Array is :                    10                    1                    10

- Bounds of the array as specified are ignored!
- The array is mapped back to having a lower bound of 1

# Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(array_in)
    INTEGER, DIMENSION(0:9), INTENT(IN) :: array_in

    PRINT *, 'Array is : ', SIZE(array_in), &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod

PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(0:9) :: array
    CALL test_array(array)

END PROGRAM test
```

# Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(array_in)
    INTEGER, DIMENSION(0:9), INTENT(IN) :: array_in

    PRINT *, 'Array is : ', array_in, &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod

PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(0:9) :: array
    CALL test_array(array)

END PROGRAM test
```

```
Array is :          10          0          9
```

# Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(lower, array_in)
    INTEGER, INTENT(IN) :: lower
    INTEGER, DIMENSION(lower:lower+9), INTENT(IN) :: array_in

    PRINT *, 'Array is : ', SIZE(array_in), &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod

PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(0:9) :: array
    CALL test_array(0, array)

END PROGRAM test
```

# Arrays and Subprograms

- What about deferred shape arrays?
- Still have the lower bound problem
- You can still do the same tricks to reset the lower bounds but now you just don't specify the upper bound at all so the call looks like
  - **INTEGER, DIMENSION(lower:)**

# Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(lower, array_in)
    INTEGER, INTENT(IN) :: lower
    INTEGER, DIMENSION(lower:), INTENT(IN) :: array_in

    PRINT *, 'Array is : ', SIZE(array_in), &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod

PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(0:9) :: array
    CALL test_array(0, array)

END PROGRAM test
```

# Arrays and Subprograms

- So I mentioned that if you specify the **ALLOCATABLE** attribute to the dummy variable to a function then the bounds will be correctly passed through to the function
- This also allows you to `ALLOCATE` and `DEALLOCATE` the variable inside the function (unless it is specified `INTENT(IN)` because then you can't change it at all)
- Problem is that the function will then **ONLY** work with allocatable variables

# Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(array_in)
    INTEGER, DIMENSION(:), ALLOCATABLE, INTENT(IN) :: array_in

    PRINT *, 'Array is : ', SIZE(array_in), &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod

PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(:), ALLOCATABLE :: array
    ALLOCATE(array(0:9))
    CALL test_array(array)

END PROGRAM test
```

# Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(array_in)
    INTEGER, DIMENSION(:), ALLOCATABLE, INTENT(IN) :: array_in

    PRINT *, 'Array is : ', SIZE(array_in), &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod
```

```
PROGRAM test

USE arraymod
IMPLICIT NONE

INTEGER, DIMENSION(:), ALLOCATABLE :: array
ALLOCATE(array(0:9))
CALL test_array(array)

END PROGRAM test
```

```
Array is :          10          0          9
```

# Arrays and Subprograms

- There's also one final wrinkle to putting **ALLOCATABLE** as an attribute for a dummy variable
- If you specify that the variable is **INTENT(OUT)** then it assumes that you don't want any information from outside ***so it will deallocate the array that you pass in***
- While you can see why that was the decision that the standards body came to it isn't ***obvious*** that this will happen so watch out for it

# Arrays and Subprograms

- There isn't any good solution to the lower bound problem of passing arrays to subprograms in Fortran
- I tend to favour
  - setting the lower bound to be a constant if it's the same for all arrays that will be passed to the function
  - passing the extra parameter when it isn't
- I try to avoid writing functions that "cope" with changing the lower bound to 1 since it is often confusing and inelegant
- One popular solution is to encapsulate the array in a **derived type** (see later) and pass the type to your subprogram

# Multidimensional Arrays

- Fortran is one of comparatively few languages that have multidimensional arrays as a core language feature
- Generally the number of dimensions that an array has is called its **rank** and this term is very commonly used in Fortran
- So the arrays that we have been working with are **rank 1** arrays
- 2D arrays are **rank 2** etc.

# Multidimensional Arrays

```
PROGRAM ranktest
```

```
INTEGER, DIMENSION(0:9, -1:10) :: array1  
INTEGER, DIMENSION(:, :), ALLOCATABLE :: array2
```

```
ALLOCATE(array2(-5:4, 1:10))  
array2(3, 7) = 4
```

```
END PROGRAM ranktest
```

- Syntax for higher rank arrays is very similar to rank 1 arrays
- Just comma separate the bounds and the indices and off you go!
- Access is (a, b) rather than multiple brackets (c.f. C)
  - Any valid range etc. is valid for each rank separately (e.g array(3, :))

# Multidimensional Arrays

- Fortran up to Fortran 2008 has a maximum rank of 7
  - Fortran 2008 increased this to 15 but compiler support is still patchy for this although it is improving
- Mostly 7D arrays (i.e. array (a, b, c, d, e, f, g)) will be enough
- Higher rank arrays are passed to functions in exactly the same way as rank 1 arrays
  - Just create a dummy variable **matching the rank** of the array that you want to pass in

# Multidimensional Arrays

- Functions in Fortran can only be written to take arrays of a specific rank
  - Until Fortran 2018 but at the moment it is hard to use
  - Fortran 2023 is adding more tools for doing this
- Usually simpler anyway because writing a generic routine that works for any rank of array but performs well is very hard
- If you really need to cope with this then using a 1D array with an indexing function to “mock up” higher ranks is the easiest solution

# Multidimensional Intrinsic

- The intrinsic array functions change slightly for higher rank arrays
  - SIZE - Returns the total number of elements in the array
  - LBOUND - Returns an array of the lower bounds in each rank. Optional **DIM** integer parameter to specify which rank you want the lower bound for. If **DIM** is specified then returns a simple INTEGER
  - UBOUND - See LBOUND
  - SHAPE - Returns an array of the number of elements in each rank of the array

# Array bound checking

- Most Fortran compilers can turn on *array bounds checking* that will check if you are accessing any rank outside of the specified bounds
  - Slows down the code so not on by default, usually "-C" compiler option to turn it on
- Tools like Valgrind can detect accesses that are completely outside of an array
  - Can't detect just overrunning one rank, only accessing outside array

# Whole Array Operations

- In Fortran most intrinsic operations and functions can be applied to whole arrays
  - They apply element by element
- You can validly add two arrays, multiply two arrays or take the sine of an array etc.
- Arrays must be the same size and rank for these to work (excepting automatic reallocation!)
  - Although you can do "array = scalar" to set all the array values to the single scalar value

# Whole Array Operations

```
PROGRAM whole_array

REAL, PARAMETER :: pi = 3.14159
REAL, DIMENSION(10) :: a, b
INTEGER :: i

!Set a so that the elements go pi/2, pi, 3pi/2 etc
a=[(pi*REAL(i)/2.0,i=1,SIZE(a))]
b = SIN(a)
PRINT *, b

END PROGRAM whole_array
```

```
1.000000000      2.53518169E-06  -1.000000000      -5.07036339E-06
1.000000000      7.60554485E-06  -1.000000000      -1.01407268E-05
1.000000000      1.26759078E-05
```

# Whole Array Operations

```
ELEMENTAL SUBROUTINE double(a)
  !NOTE this function takes a single scalar NOT an array
  INTEGER, INTENT(INOUT) :: a
  a = a * 2
END SUBROUTINE double
```

- Some functions treat arrays like vector or matrices such as **DOT\_PRODUCT** or **MATMUL**
- If you want to create your own functions to operate on arrays element by element then use **ELEMENTAL** subprograms
- Rules in general are quite complicated but the idea is simple

# Array subsections

- Can also refer to subsections of an array
- Use `array(lbound:ubound)` to refer to an array subsection
  - Same for multiple rank arrays, can put in a range or a single value for any index that you like
- Can use array subsections exactly like you can arrays
  - Set them equal to things, pass them to functions, everything you can do with an array you can do with an array subsection

# Array subsections

- If you want an array section to go to the lower bound, simply don't specify the lower index
  - `array(:10)`
- If you want an array section to go to the upper bound, simply don't specify the upper index
  - `array(10:)`
- You can validly specify the whole array using just `(:)` although there are only a few reasons to (preventing implicit reallocation is one!)
- Less commonly seen you can also specify a stride
  - `array(1:100:5)` specifies every 5th element of the array from 1 to 100

# MOVE\_ALLOC

```
PROGRAM move_test
```

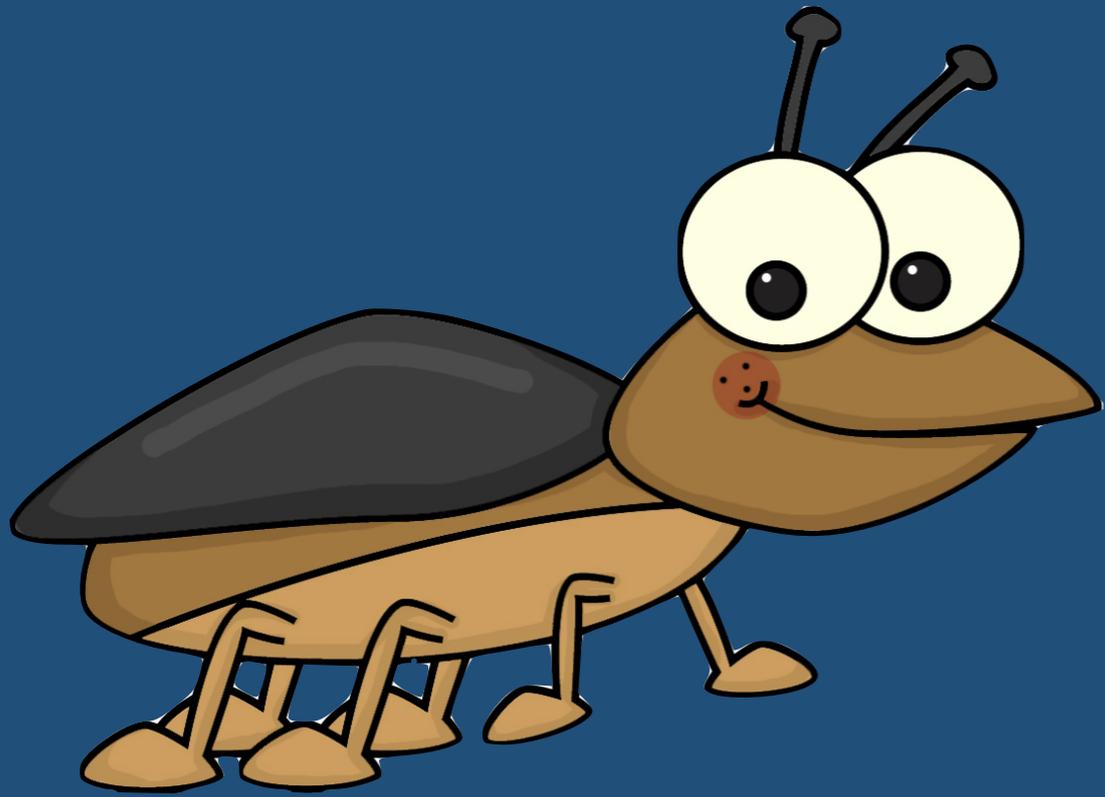
```
INTEGER, DIMENSION(:), ALLOCATABLE :: a, b  
INTEGER :: i
```

```
ALLOCATE(a(-5:10))  
DO i = -5, 10  
    a(i) = i  
END DO
```

```
CALL MOVE_ALLOC(FROM = a, TO = b)  
PRINT *, LBOUND(b), UBOUND(b)  
PRINT *, MINVAL(b), MAXVAL(b)  
PRINT *, ALLOCATED(a), ALLOCATED(b)
```

```
END PROGRAM move_test
```

- Move data between allocatable arrays
- TO must start deallocated
- FROM becomes deallocated
- Faster than setting = because memory is **moved** not **copied**



The End