

# Arrays in Fortran

Victor Eijkhout, Susan Lindsey

Fall 2019

# Fortran dimension

Preferred way of creating arrays through `dimension` keyword:

```
real(8), dimension(100) :: x,y
```

One-dimensional arrays of size 100.

Older mechanism works too:

```
integer :: i(10,20)
```

Two-dimensional array of size  $10 \times 20$ .

These arrays are statically defined, and only live inside their program unit.

# 1-based Indexing

Array indexing in Fortran is 1-based by default:

```
integer,parameter :: N=8  
real(4),dimension(N) :: x  
do i=1,N  
  ... x(i) ...
```

Note the use of `parameter`: compile-time constant

# Lower bound

Unlike C++, Fortran can specify the lower bound explicitly:

```
real,dimension(-1:7) :: x
do i=-1,7
  ... x(i) ...
```

Safer:

Code:

```
real,dimension(-1:7) :: array
integer :: idx
!! ...
do idx=lbound(array,1),ubound(array,1)
  array(idx) = 1+idx/10.
  print *,array(idx)
end do
```

Output

[arrayf] lubound:

```
0.899999976
1.000000000
1.100000002
1.200000005
1.299999995
1.399999998
1.500000000
1.600000002
1.700000005
```

# Array initialization

```
real,dimension(5) :: real5 = [ 1.1, 2.2, 3.3, 4.4, 5.5 ]  
!! ...  
real5 = [ (1.01*i,i=1,size(real5,1)) ]  
!! ...  
real5 = (/ 0.1, 0.2, 0.3, 0.4, 0.5 /)
```

# Array sections example

Use the colon notation to indicate ranges:

```
real(4),dimension(4) :: y
```

```
real(4),dimension(5) :: x
```

```
x(1:4) = y
```

```
x(2:5) = x(1:4)
```

# Use of sections

Code:

```
real(8),dimension(5) :: x = &  
    [.1d0, .2d0, .3d0, .4d0, .5d0]  
!! ...  
x(2:5) = x(1:4)  
print '(f5.3)',x
```

Output

[arrayf] sectionassign:

```
0.100  
0.100  
0.200  
0.300  
0.400
```

# Exercise 1

Code out the above array assignment with an explicit, indexed loop. Do you get the same output? Why? What conclusion do you draw about internal mechanisms used in array sections?



# Strided sections

Code:

```
integer,dimension(5) :: &  
    y = [0,0,0,0,0]  
integer,dimension(3) :: &  
    z = [3,3,3]  
!! ...  
y(1:5:2) = z(:)  
print '(i3)',y
```

Output

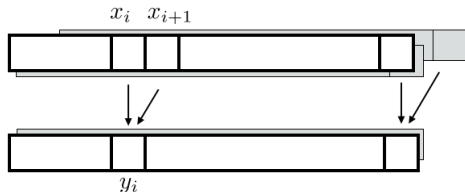
[arrayf] sectionmg:

3  
0  
3  
0  
3

# Index arrays

```
integer,dimension(4) :: i = [2,4,6,8]  
real(4),dimension(10) :: x  
print *,x(i)
```

## Exercise 2



Code  $\forall_i: y_i = (x_i + x_{i+1})/2$ :

- First with a do loop; then
- in a single array assignment statement by using sections.

Initialize the array  $x$  with values that allow you to check the correctness of your code.

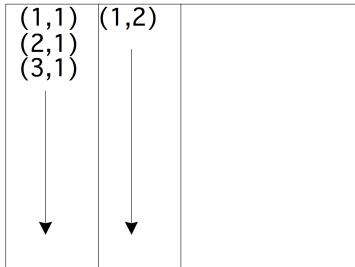
# Multi-dimension arrays

```
real(8),dimension(20,30) :: array  
array(i,j) = 5./2
```

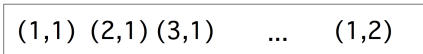
# Array layout

Sometimes you have to take into account how a higher rank array is laid out in (linear) memory:

Fortran column major



Physical:



'First index varies quickest'

# Array sections in multi-D

```
real(8),dimension(10) :: a,b  
a(1:9) = b(2:10)
```

or

```
logical,dimension(25,3) :: a  
logical,dimension(25)   :: b  
a(:,2) = b
```

You can also use strides.

# Query functions

- Bounds: lbound, ubound
- size
- Can be used per dimension, or overall giving array of bounds/sizes.

Code:

```
integer,dimension(8) :: x
integer,dimension(5,3:7) :: y
!! ...
print *,size(x)
print *,size(y)
print *,size(y,2)
print *,lbound(y)
print *,ubound(y,1)
```

Output

[arrayf] query:

8	
25	
5	
1	3
5	

# Pass array: subprogram

Note declaration as `dimension(:)`

actual size is queried

```
real(8) function arraysum(x)
  implicit none
  real(8),intent(in),dimension(:) :: x
  real(8) :: tmp
  integer i

  tmp = 0.
  do i=1,size(x)
    tmp = tmp+x(i)
  end do
  arraysum = tmp
end function arraysum
```



# Pass array: main program

Passing array as one symbol:

**Code:**

```
real(8),dimension(:) :: x(N) &  
    = [ (i,i=1,N) ]  
real(8),dimension(:) :: y(0:N-1) &  
    = [ (i,i=1,N) ]  
  
sx = arraysum(x)  
sy = arraysum(y)  
print '("Sum of one-based array:",/,4x,  
    f6.3)', sx  
print '("Sum of zero-based array:",/,4x  
    ,f6.3)', sy
```

**Output**

**[arrayf] arraypass1d:**

Sum of one-based array:  
55.000

Sum of zero-based array:  
55.000

# Array allocation

```
real(8), dimension(:), allocatable :: x,y
```

```
n = 100
```

```
allocate(x(n), y(n))
```

You can deallocate the array when you don't need the space anymore.

# Array intrinsics

- MaxVal finds the maximum value in an array.
- MinVal finds the minimum value in an array.
- Sum returns the sum of all elements.
- Product return the product of all elements.
- MaxLoc returns the index of the maximum element.

*i* = MAXLOC( *array* [, *mask* ] )

- MinLoc returns the index of the minimum element.
- MatMul returns the matrix product of two matrices.
- Dot\_Product returns the dot product of two arrays.
- Transpose returns the transpose of a matrix.
- Cshift rotates elements through an array.

## Exercise 3

The 1-norm of a matrix is defined as the maximum of all sums of absolute values in any column:

$$\|A\|_1 = \max_j \sum_i |A_{ij}|$$

while the infinity-norm is defined as the maximum row sum:

$$\|A\|_\infty = \max_i \sum_j |A_{ij}|$$

Write Fortran **Functions** that compute these, and test them.

## Exercise 4

Compare implementations of the matrix-matrix product.

1. Write the regular  $i, j, k$  implementation, and store it as reference.
2. Use the DOT\_PRODUCT function, which eliminates the  $k$  index. How does the timing change? Print the maximum absolute distance between this and the reference result.
3. Use the MATMUL function. Same questions.
4. Bonus question: investigate the  $j, k, i$  and  $i, k, j$  variants. Write them both with array sections and individual array elements. Is there a difference in timing?

Does the optimization level make a difference in timing?

# Timer routines

```
integer :: clockrate, clock_start, clock_end
call system_clock(count_rate=clockrate)
!! ...
call system_clock(clock_start)
!! ...
call system_clock(clock_end)
print *, "time:", (clock_end-clock_start)/REAL(clockrate)
```

# Operate where

```
where ( A<0 ) B = 0
```

Full form:

```
WHERE ( logical argument )  
    sequence of array statements  
ELSEWHERE  
    sequence of array statements  
END WHERE
```

# Do concurrent

The `do concurrent` is a true do-loop. With the `concurrent` keyword the user specifies that the iterations of a loop are independent, and can therefore possibly be done in parallel:

```
do concurrent (i=1:n)
  a(i) = b(i)
  c(i) = d(i+1)
end do
```

(Do not use `for all`)