Arrays in Fortran

Victor Eijkhout, Susan Lindsey

Fall 2021
last formatted: November 7, 2021

1. 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 10×20.

These arrays are statically defined, and only live inside their program unit (subroutine, function, module).

2. 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) ...
```

(Different from most other languages.)

Note the use of `parameter`: compile-time constant
Size needs to be known to the compiler.

3. 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
   1.00000000
   1.10000002
   1.20000005
   1.29999995
   1.39999998
   1.50000000
   1.60000002
   1.70000005
```

4. Array initialization

Different syntaxes:

- Explicit:

```
real,dimension(5) :: real5 = [ 1.1, 2.2, 3.3, 4.4, 5.5 ]
```

- Implicit do-loop:

```
real5 = [ (1.01*i,i=1,size(real5,1)) ]
```

- Legacy syntax

```
real5 = (/ 0.1, 0.2, 0.3, 0.4, 0.5 /)
```

(Maybe for systems that do not have the [] characters?)

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)
```

6. Use of sections

Code:

```fortran
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
```

Notes:

- `1.d0` explicit double precision: avoid loss of precision
- Format syntax will be discussed later:
  float number, 5 positions, 3 after decimal point.

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?

7. 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
```

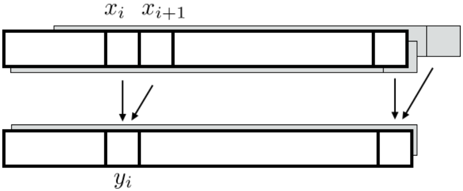
Output
[arrayf] sectionmg:

```
  3
  0
  3
  0
  3
```

8. 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.

9. Multi-dimension arrays

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

10. Reshaping array

Code:

```
real,dimension(2,2) :: x
x = reshape( [ ( 1.*i,i=1,size(x) ) ],
     shape(x) )
!  x = reshape( (/ ( 1.*i,i=1,4 ) /),
     shape(x) )
print *,x
```

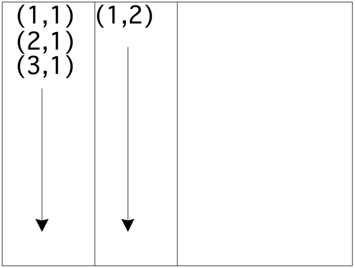
Output
[arrayf] multi:

```
   1.00000000
    2.00000000
    3.00000000
    4.00000000
```

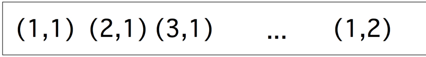
11. Array layout

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'

12. Array sections in multi-D

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

or

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

You can also use strides.

13. 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
```

14. 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
```

15. 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:",/,4
     x,f6.3)', sy
```

Output
[arrayf] arraypass1d:

```
Sum of one-based
     array:
    55.000
Sum of zero-based
     array:
    55.000
```

16. 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.

17. Array intrinsics

- `Abs` creates the matrix of pointwise absolute values.
- `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.
- `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.

18. Multi-dimensional intrinsics

- Functions such as `Sum` operate on a whole array by default.
- To restrict such a function to one subdimension add a keyword parameter `DIM`:

  ```
  s = Sum(A,DIM=1)
  ```

  where the keyword is optional.
- Likewise, the operation can be restricted to a `MASK`:

  ```
  s = Sum(A,MASK=B)
  ```

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:

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

Compute these norms using array functions as much as possible, that is, try to avoid using loops.

For bonus points, write Fortran `Function`s that compute these norms.

Optional 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)
```

19. Operate where

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

Full form:

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

20. 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`)