Fortran - Arrays

Arrays, Multidimensional Array, Dynamic Arrays

Victor Eijkhout, Carrie Arnold, Charlie Dey

Arrays

a definition

a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Arrays

An example

```
program array1
implicit none
integer :: i
real, dimension(5) :: A = (/ 1, 2, 3, 4, 5 /)
-or-
real, dimension(5) :: A = [ 1.1, 2.2, 3.3,
4.4, 5.5 ]

do i=1,5
   print *, A(i)
end do

end program array1
```

What's different from C/C++?

- index starts at 1
- `()`'s instead of `[]`'s (old way)
- explicit declarations require '`/`' at the beginning and end of the series (old way)

Arrays

Reading and writing

```
program array2
implicit none
integer :: i, n=5
real, dimension(n) :: A

do i=1,5
   A(i)= i*i
end do

do i=1,5
   print *, A(i)
end do

print *, A

end program array2
```

What's different from C/C++?

- index starts at 1
- `()`'s instead of `[]`'s
- explicit declarations require '`/`' at the beginning and end of the series
- you can reference an array by the array variable.
- size of an array can be a parameter

TACC

Exercise 1.

Write a program that creates an array of 100 random numbers between 0 and 100

Run the following code, modify it so it meets the exercise criteria

```
// this code generates an array of 100 random numbers
program test_random_number
implicit none
real, dimension(100) :: r
    call random_number(r)
    print *, r
end program
```

Arrays

As an argument to a Function

```
program with_fct
implicit none
integer, parameter :: n = 10
real, dimension (n) :: a

! Calculate Average
aver = average(n, a)    ! Function
                        !   call
! Read more data (n, a2)
open …; read …; close …

! Calculate Average again
aver2 = average(n, a2)

contains
```

```
real function average(n, x)
integer              :: n, i
real, dimension(n)   :: x
real                 :: sum

sum = 0.

do i=1, n
  sum = sum + x(i)
enddo

average = sum / real(n)

end function average

end program
```

Exercise 2.

Using your random array generators,
Write 2 functions that take an array as an argument

- one function that finds the maximum value and the index of the maximum value
- one function that finds the minimum value and the index of minimum value

Exercise 3.

Using your random array generator,
Write a function that will sort your randomly generated array from smallest to largest, by traversing your array and swapping values of adjacent indices if a(i) > a(i+1)

Do this by writing a function that takes an array and 2 index locations and swaps the values of the array at the 2 index locations.

How can you test that your array is sorted?

Exercise 4.

Using Exercise 3, write a test function which will take your "sorted" array as an argument and tests it to verify that the array is indeed sorted, this function will return a logical.

Multi-dimensional arrays

the definition from C/C++ (Row major)

```
int a [ 3 ] [ 4 ];
```

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Arrays

Run this code.

```
program array3
implicit none
integer :: i, j, k=0
integer, dimension(5,5) :: A

do i=1, 5
   do j=1, 5
      k = k + 1
      A(i,j)= k
   end do
end do

print *, A

end program array3
```

You might think it'd print all of Row 1, then Row 2, then Row 3, all unformatted.

Arrays

Run this code.

```
program array3
implicit none
integer :: i, j, k=0
integer, dimension(5,5) :: A

do i=1, 5
   do j=1, 5
      k = k + 1
      A(i,j)= k
   end do
end do

print *, A

end program array3
```

you should see something like:

```
 1          6         11         16         21          2
 7         12         17         22          3          8
13         18         23          4          9         14
19         24          5         10         15         20
25
```

Arrays

Run this code.

```
program array3
implicit none
integer :: i, j, k=0
integer, dimension(5,5) :: A

do i=1, 5
   do j=1, 5
      k = k + 1
      A(i,j)= k
   end do
end do

print *, A

end program array3
```

you should see something like:

```
 1          6          11          16          21          2
 7         12          17          22           3          8
13         18          23           4           9         14
19         24           5          10          15         20
25
```

What is this telling us?

Arrays

Run this code.

```
program array3
implicit none
integer :: i, j, k=0
integer, dimension(5,5) :: A

do i=1, 5
   do j=1, 5
      k = k + 1
      A(i,j)= k
   end do
end do

print *, A

end program array3
```

you should see something like:

```
 1        6       11       16       21        2
 7       12       17       22        3        8
13       18       23        4        9       14
19       24        5       10       15       20
25
```

What is this telling us?
Fortran is Column major!

```
 1        6       11       16       21
 2        7       12       17       22
 3        8       13       18       23
 4        9       14       19       24
 5       10       15       20       25
```

Arrays

More about arrays, multi-dimension arrays.

```
integer, parameter      :: n = 3
integer, parameter      :: m = 7

real, dimension(n,m)    :: a
real, dimension(0:2,4,8) :: b
```

- Ordered collection of elements
- Each element has an index
- Index may start at any integer number, not only 1
- Array element may be of intrinsic or derived type
- Array size refers to the number of elements
- The number of dimensions is the rank
- The size along a dimension is called an extent
- Array shape is the sequence of extents

```
size of a: 21
rank of a: 2
extent of a, first dimension: 3
shape of a: (3,7)

size of b:96
rank of b:3
extent of b, first dimension: 3
shape of b: (3,4,8)
```

TACC

Multi-dimensional arrays

recall from C++

```
#include <iostream>


using namespace std;

int multiplyByC(int arr[][4], int rows, int cols, int C)
{

   for (int i = 0; i < rows; i++)
   {
      for (int j = 0; j < cols; j++)
      {
         arr[i][j] *= C;
      }
   }
   return 0;
}
```

```
int main ()
{
   int a[3][4];

   for ( int i = 0; i < 3; i++ )
      for ( int j = 0; j < 4; j++ )
         a[i][j] = i+j;

   multiplyByC(a, 3, 4, 5);

   for ( int i = 0; i < 3; i++ )
      for ( int j = 0; j < 4; j++ ) {
         cout << a[i][j]<< endl;
      }

   return 0;
}
```

Multi-dimensional arrays

in Fortran as a Subroutine

```
program mArray
implicit none
integer :: i, j
integer, dimension(3,4) :: a, b

do i=1, 3
   do j=1, 4
      A(i,j) = i+j
   end do
end do

call multiplyByC(a, b, 3, 4, 5);

print *, b
```

```
contains


subroutine multiplyByC(arr, ans, rows, cols, C)
implicit none
integer :: rows, cols, C
integer, dimension(rows, cols) :: arr, ans


   ans = arr * C


end subroutine

end program
```

Multi-dimensional arrays

in Fortran as a Function

```
program mArray
implicit none
integer :: i, j
integer, dimension(3,4) :: a, b

do i=1, 3
   do j=1, 4
      A(i,j) = i+j
   end do
end do

b =  multiplyByC(a, 3, 4, 5);

print *, b
```

```
contains

function multiplyByC(arr,rows, cols, C)
implicit none
integer :: rows, cols, C
integer, dimension(rows, cols) :: arr, multiplyByC

   multiplyByC = arr * C

end function

end program
```

Exercise 5.

Write a subroutine or function that creates a 100x100 identity matrix , a matrix where the diagonal values are 1's and the rest of the values - the upper and lower triangles - are 0's

Exercise 6.

- Using your random number generator, create 2 random 100x100 matrices.
- Write a subroutine or function that multiplies the 2 matrices together and puts the result in a third matrix.
- Test your matrix multiplication subroutine by multiplying your random matrix with the same size identity matrix, the result will be the same as the original matrix.

Matrix Multiplication Algorithm:

- Input: matrices A and B
- Let C be a new matrix of the appropriate size
- For i from 1 to n:
    - For j from 1 to p:
        - Let sum = 0
        - For k from 1 to m:
            - Set sum ← sum + A[i][k] × B[k][j]
        - Set C[i][j] ← sum
- Return C

Array Slicing, Array Shortcuts

```
real, dimension(5) :: A

 a(1:3)     // Elements selected: a(1), a(2), a(3)
 a(1:5:2)   // Elements selected: a(1), a(3), a(5)
 a(:)       // Elements selected: a(1), a(2), a(3), a(4),
            // a(5)
```

- Variables on the left and the right have to be conformable in size and shape
  i.e. number of elements and rank
- Scalars are conformable
- Strides can be used

Array Slicing, Array Shortcuts

```
REAL, DIMENSION(5,5)  :: A

 a(2, (/1, 3, 5/))      // a(2,1)  a(2,3)  a(2,5)

 a(2, 1:3)              // a(2,1)  a(2,2)  a(2,3)
 a(2, 1:3:2)            // a(2,1)  a(2,3)  a(2,5)
 a(2, :)                // a(2,1)  a(2,2)  a(2,3)  a(2,4)
                         //(2,5)
```

- Variables on the left and the right have to be conformable in size and shape
  i.e. number of elements and rank
- Scalars are conformable
- Strides can be used

TACC

Array Slicing, Array Shortcuts

```
real                    :: x
real, dimension(10)     :: a, b
real, dimension(10,10)  :: c, d

a        = b
c        = d
a(1:10)  = b(1:10)
a(2:3)   = b(4:5)
a(1:10)  = c(1:10,2)
a        = x
c        = x

a(1:3)   = b(1:5:2)

a = c(:,1)
a = c(:,5)
a = c(1,:)
a = c(5,:)
```

What's being done on each line?

Array Slicing, Array Shortcuts

best practices

- Always access slices as V(:, 1), V(:, 2), or V(:, :, 1), e.g. the colons should be on the left.
    - That way the stride is contiguous and it will be faster.
    - When you need some slice in your algorithm, always setup the array in a way, so that you call it as above. If you put the colon on the right, it will be slow.

```
dydx = matmul(C(:, :, i), y) ! fast
dydx = matmul(C(i, :, :), y) ! slow
```

TACC

Array Slicing, Array Shortcuts

best practices

- the “fortran storage order” is:
  - smallest/fastest changing/innermost-loop index first,
  - largest/slowest/outermost-loop index last (“Inner-most are left-most.”).
  - So the elements of a 3D array A(N1,N2,N3) are stored, and thus most efficiently accessed as:

```
do i3 = 1, N3
    do i2 = 1, N2
        do i1 = 1, N1
            A(i1, i2, i3)
        end do
    end do
end do
```

Exercise 7.

- Create a 100x100 matrices
    - set all elements initially equal to 1
    - slice your matrix such that
        - elements in rows 1 through 50 and column 1 through 50 are set to 1
        - elements in rows 1 through 50 and column 51 through 100 are set to 2
        - elements in rows 51 through 100 and column 1 through 50 are set to 3
        - elements in rows 51 through 100 and column 51 through 100 are set to 4

Exercise 8.

- Code y(i) = (x(i) + x(i+1))/2 in a **single** array statement.
  Initialize the array x with values that allow you to check the correctness of your code.

Hint, you may want to start with a do-loop or an implicit do-loop
then see if you can combine your code into one statement.

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.

Homework.

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?

Homework.

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)
```

Dynamically Allocated Arrays

Sometimes you need to allocate memory for an array that is not static in size.

new declaration option: `allocatable`

new command: `allocate, deallocate`

Dynamically Allocated Arrays

```
program alloc_array
implicit none

real, dimension(:),   allocatable :: x_1d  ! Attribute
real, dimension(:,:), allocatable :: x_2d  ! allocatable

...
read n, m
...

allocate(x_1d(n), stat=ierror)             ! Check the
if (ierror /= 0) stop 'error x_1d'         ! error status

allocate(x_2d(n,m), stat=ierror)
if (ierror /= 0) stop 'error x_2d'


deallocate(x)                              ! optional
```

- Declaration and allocation in two steps
- Declare an array as allocatable
  - Use colons (:) as placeholders
- Allocate/deallocate in the executable part

Dynamically Allocated Arrays

```
subroutine sub(n)
real, dimension(:),   allocatable :: x_1d

...
allocate(x_1d(n), stat=ierror)              ! Check the
if (ierror /= 0) stop 'error x_1d'          ! error status
...


end
```

What are your thoughts here?

What happens to the allocated memory space when you leave the subroutine?

Does this work or does it produce a memory leak?

Dynamically Allocated Arrays

```
subroutine sub(n)
real, dimension(:),   allocatable :: x_1d

...
allocate(x_1d(n), stat=ierror)              ! Check the
if (ierror /= 0) stop 'error x_1d'          ! error status
...

deallocate(x_1d)
end
```

What are your thoughts here?

Dynamically allocated arrays are automatically deallocated, when you leave the scope

Nevertheless, it does not hurt to put a deallocate statement yourself

Dynamically Allocated Arrays

```
program main
real, dimension(:), allocatable :: x_1d

allocate(x_1d(1000000000), state=ierror)
if (ierror /= 0) stop 'error x_1d'

call sub(x_1d)


contains
subroutine sub(x_1d)
...
...
end subroutine


end program
```

What about this problem?

We have an array being dynamically allocated, we don't know the size, but we want to pass it to an function or subroutine.

Dynamically Allocated Arrays

```
program main
real, dimension(:), allocatable :: x_1d

allocate(x_1d(1000000000), state=ierror)
if (ierror /= 0) stop 'error x_1d'

call sub(x_1d)


contains

subroutine sub(x_1d)
real, dimension(:) :: x
...
end subroutine


end program
```

What about this problem?

We have an array being dynamically allocated, we don't know the size, but we want to pass it to an function or subroutine.

We allow the program to assume the size using (:) and proceed as normal.

TACC

Exercise 8.

- set 2 integer arguments, `n` and `m`
    - create a function that:
    - pass 2 integers argument, `n` and `m`,
        - dynamically allocate an array to build an `n` x `m` matrix
    - Fill your array using random numbers
        - if the random number is even make the element = zero.
        - if the random number is odd make the element = one.
- using nested do-loops, print out the matrix in an easy to read format