Vectors and arrays

One of the biggest advantages of Fortran over other languages is that arrays are build in part of the language. Fortran supports arrays from one dimensional vectors to tensors of rank 7 (7 independent indices). Both static (dimension present at the compile time) and dynamic arrays (size of each rank can be determined at the run time) are supported. First we will discuss the static arrays and talk about dynamic memory allocation and dynamic arrays in the next section.

Static arrays must have their dimension present as a parameter (variable that cannot be modified) at the compile time and will utilize stack memory (unless changed with the compiler parameter). For example

```
integer, parameter :: n = 10
real, dimension(n) :: vector
```

will create a one dimensional variable called `vector` of `real` type and size 10. One must remember that the elements of the array are indexed starting from 1 (not 0 as in C language).

Individual elements of the array can be accessed using `()` operator with first element being `(1)` and the last one `(n)` where `n` is the size of the array, for example

```
vector(1) = 1.0
...
vector(10) = 1.0
```

Fortran also allows collective operations on all elements of the array, for example

```
vector = 1.0
```

will assign value 1.0 to every element of the array.

To use arrays with more than one dimension one needs to only change the number of parameters in the `dimension` keyword. To create a 2D array i.e. matrix in Fortran we only require simple change, e.g.

```
 integer, parameter :: n = 3
 real, dimension(n,n) :: matrix
```

Similarly to vector case above the collective element access is also possible for multidimensional arrays, e.g.

```
matrix = 0.0
```

will set all elements to 0.0. Single row of column can be accessed using

```
matrix(1,:)
```

will be the first row, and

```
matrix(:,1)
```

is a first column. One needs to remember that contrary to C language the ordering of elements are stored column-major not row-major order. This means that the index is running through columns not rows.

Example code

```
program simple_array

  implicit none

  integer, parameter :: n = 3

  real, dimension(n) :: vector
  real, dimension(n,n) :: matrix

  vector = 0.0
  print *, vector

  matrix = 1.0
  print *, matrix

end program simple_array
```

which gives the following output

```
   0.00000000       0.00000000       0.00000000
   1.00000000       1.00000000       1.00000000       1.00000000
1.00000000       1.00000000       1.00000000       1.00000000       1.00000000
```

Dynamic memory allocation

In many situations the size of the requested array is not known at the compile time and is determined at the run time. For that case Fortran allows the dynamic memory allocation of an arbitrary size.

A simple dynamically allocatable vector can be defined using

```
real, allocatable, dimension(:) :: vector
```

when the vector is needed in the code, the memory can be allocated to store its elements using

```
n = 10
allocate(vector(n))
```

before the `allocate(vector(n))` command is issued, the memory is not used to store any of the elements but only the information about the array (negligible amount comparing to elements). After the memory is allocated the array is treated exactly the same as the static array and all elements can be accessed either collectively or one element at the time. At any place in the code when the array is not needed any more programmer can deallocate memory using `deallocate` function.

```
deallocate(vector)
```

Similar for multidimensional arrays

```
integer :: n
real, allocatable, dimension(:,:) :: matrix

n = 10
allocate(matrix(n,n))
matrix = 1.0
deallocate(matrix)
```

Fortran 95 will deallocate arrays automatically when they run out of scope, however, it is a good practice to always explicitly deallocate objects that we explicitly allocated.

Memory allocation command can be called with parameter capturing the status of the process. In case of not sufficient memory in the system the programmer can request a message to easier debug the problem

```
integer :: n
integer :: AllocationStatus
real, allocatable, dimension(:,:) :: A

allocate( A(n,n), stat = AllocationStatus )
if( AllocationStatus /= 0 ) stop 'Problem with memory allocation'
```

if the status of the command has value different than 0 then the command has failed to allocate requested amount of memory. Similar for `deallocate` command

```
deallocate( A, stat = AllocationStatus )
```

Example program may look like this

```
program allocatable_arrays

  implicit none

  real, allocatable, dimension(:) :: vector
  real, allocatable, dimension(:,:) :: matrix

  integer :: n, MemoryError

  n = 10

  allocate( vector(n), stat=MemoryError )
  if(MemoryError /= 0) stop "Cannot allocate vector"

  allocate( matrix(n,n), stat=MemoryError )
  if(MemoryError /= 0) stop "Cannot allocate matrix"

  vector = 1.0
  matrix = 0.0

  deallocate( vector, stat=MemoryError )
  if(MemoryError /= 0) stop "Cannot deallocate vector"

  deallocate( matrix, stat=MemoryError )
  if(MemoryError /= 0) stop "Cannot deallocate matrix"

end program allocatable_arrays
```

Collective operations

As we have mentioned above Fortran supports operations on whole arrays. The collective operations can be performed on either whole array or on a subsection (slice of the array)

Assignment on whole and the slice may look like

```
A = 0.0
A(1,:) = 0.0
```

Binary operations

```
A = B + 2.0 * C
a = b + C(1,:)
```

where `A`, `B` and `C` are matrices and `a` and `b` are vectors.

`maxval(A,D)` function returns value of the maximum element in the array, `maxloc(A)` returns a 1D array with location of the maximum element (location of the first occurrence if found more than one time). `D` specifies dimension, if omitted the function operates on whole array.

`product(A,D)` returns a product of all elements of the array, `sum()` returns a sum of the elements. `D` is the dimension and is optional. Function

```
sum(A, DIM=1)
```

will return sum of elements along the dimension 1. All of the functions mentioned above can be also invoked with `MASK` option. `MASK` is a logical expression and the operation will be applied only to elements that satisfy this expression, for example

```
sum(A, DIM=1, MASK=A>0)
```

the above will sum along first dimension but only elements that are larger than 0.

`transpose(A)` is a matrix function for transposition of the dimensions. Matrix specific functions include also `matmul()` and `dot_product()`. Matrix multiplication can be done on a pair of matrices which inner dimension has matching size

```
C = matmul(A,B)
```

the command returns the product matrix. If the matrix dimensions do not match the command will return an error. A transposition can be done inside the `matmul` command

```
D = matmul(transpose(A),B)
```

Function `dot_product(x,y)` returns a sum of products of elements from vectors `x` and `y`. If multidimensional arrays are passed to `dot_product` they will be treated as 1D and generalized dot product will be calculated.

Example usage of `matmul()` function

```
program matmul_example

  implicit none

  real, allocatable, dimension(:,:) :: A, B, C
  integer :: n
  integer :: i, j

  n = 3

  allocate( A(n,n), B(n,n), C(n,n) )

  ! Make A and B unit matrices
  A = 0.0
  B = 0.0

  do i=1,n
    A(i,i) = 1.0
    B(i,i) = 1.0
  end do

  ! Make C a zero matrix
  C = 0.0

  ! Multiply C = A * B, C should be a unit matrix now
  C = matmul(A,B)

  do i=1,n
      print *, ( C(i,j), j=1,n )
  end do

  deallocate(A,B,C)

end program matmul_example
```

which outputs

```
   1.00000000       0.00000000       0.00000000
   0.00000000       1.00000000       0.00000000
   0.00000000       0.00000000       1.00000000
```

An example usage to `dot_product()` function

```
program dot_product_example

  implicit none

  real, allocatable, dimension(:) :: a, b
  real :: total
  integer :: n

  n = 10
  allocate( a(n), b(n) )

  a = 1.0
  b = 1.0

  total = dot_product(a,b)

  ! total should be 10
  print *,'Total (should be : ',n,' ) is : ', total

  deallocate(a,b)

end program dot_product_example
```

which will output

```
Total (should be :           10  ) is :    10.0000000
```