FORTRAN - ARRAYS

Arrays can store 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.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Numbers1 Numbers2 Numbers3 Numbers4 ...

Arrays can be one- dimensional *likevectors*, two-dimensional *likematrices* and Fortran allows you to create up to 7-dimensional arrays.

Declaring Arrays

Arrays are declared with the **dimension** attribute.

For example, to declare a one-dimensional array named number, of real numbers containing 5 elements, you write,

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

The individual elements of arrays are referenced by specifying their subscripts. The first element of an array has a subscript of one. The array numbers contains five real variables –numbers1, numbers2, numbers3, numbers4, and numbers5.

To create a 5 x 5 two-dimensional array of integers named matrix, you write:

```
integer, dimension (5,5) :: matrix
```

You can also declare an array with some explicit lower bound, for example:

```
real, dimension(2:6) :: numbers
integer, dimension (-3:2,0:4) :: matrix
```

Assigning Values

You can either assign values to individual members, like,

```
numbers(1) = 2.0
```

or, you can use a loop,

```
do i=1,5
   numbers(i) = i * 2.0
end do
```

One dimensional array elements can be directly assigned values using a short hand symbol, called array constructor, like,

```
numbers = (/1.5, 3.2,4.5,0.9,7.2 /)
```

please note that there are no spaces allowed between the brackets '('and the back slash '/'

Example

The following example demonstrates the concepts discussed above.

```
program arrayProg

   real :: numbers(5) !one dimensional integer array
   integer :: matrix(3,3), i , j !two dimensional real array

   !assigning some values to the array numbers
   do i=1,5
      numbers(i) = i * 2.0
   end do

   !display the values
   do i = 1, 5
      Print *, numbers(i)
   end do

   !assigning some values to the array matrix
   do i=1,3
      do j = 1, 3
         matrix(i, j) = i+j
      end do
   end do

   !display the values
   do i=1,3
      do j = 1, 3
         Print *, matrix(i,j)
      end do
   end do

   !short hand assignment
   numbers = (/1.5, 3.2,4.5,0.9,7.2 /)

   !display the values
   do i = 1, 5
      Print *, numbers(i)
   end do

end program arrayProg
```

When the above code is compiled and executed, it produces the following result:

```
 2.00000000
 4.00000000
 6.00000000
 8.00000000
 10.0000000
          2
          3
          4
          3
          4
          5
          4
          5
          6
 1.50000000
 3.20000005
 4.50000000
0.899999976
 7.19999981
```

Some Array Related Terms

The following table gives some array related terms:

Term	Meaning
Rank	It is the number of dimensions an array has. For example, for the array named matrix, rank is 2, and for the array named numbers, rank is 1.
Extent	It is the number of elements along a dimension. For example, the array numbers has extent 5 and the array named matrix has extent 3 in both dimensions.
Shape	The shape of an array is a one-dimensional integer array, containing the number of elements *theextent* in each dimension. For example, for the array matrix, shape is 3, 3 and the array numbers it is 5.
Size	It is the number of elements an array contains. For the array matrix, it is 9, and for the array numbers, it is 5.

Passing Arrays to Procedures

You can pass an array to a procedure as an argument. The following example demonstrates the concept:

```
program arrayToProcedure
implicit none

   integer, dimension (5) :: myArray
   integer :: i

   call fillArray (myArray)
   call printArray(myArray)

end program arrayToProcedure


subroutine fillArray (a)
implicit none

   integer, dimension (5), intent (out) :: a

   ! local variables
   integer :: i
   do i = 1, 5
      a(i) = i
   end do

end subroutine fillArray


subroutine printArray(a)

   integer, dimension (5) :: a
   integer::i

   do i = 1, 5
      Print *, a(i)
   end do

end subroutine printArray
```

When the above code is compiled and executed, it produces the following result:

```
1
2
3
4
5
```

In the above example, the subroutine fillArray and printArray can only be called with arrays with dimension 5. However, to write subroutines that can be used for arrays of any size, you can rewrite it using the following technique:

```
program arrayToProcedure
implicit  none

   integer, dimension (10) :: myArray
   integer :: i

   interface
      subroutine fillArray (a)
         integer, dimension(:), intent (out) :: a
         integer :: i
      end subroutine fillArray

      subroutine printArray (a)
         integer, dimension(:) :: a
         integer :: i
      end subroutine printArray
   end interface

   call fillArray (myArray)
   call printArray(myArray)

end program arrayToProcedure


subroutine fillArray (a)
implicit none
   integer,dimension (:), intent (out) :: a

   ! local variables
   integer :: i, arraySize
   arraySize = size(a)

   do i = 1, arraySize
      a(i) = i
   end do

end subroutine fillArray


subroutine printArray(a)
implicit none

   integer,dimension (:) :: a
   integer::i, arraySize
   arraySize = size(a)

   do i = 1, arraySize
     Print *, a(i)
   end do

end subroutine printArray
```

Please note that the program is using the **size** function to get the size of the array.

When the above code is compiled and executed, it produces the following result:

```
1
2
3
4
5
6
7
8
```

```
9
10
```

Array Sections

So far we have referred to the whole array, Fortran provides an easy way to refer several elements, or a section of an array, using a single statement.

To access an array section, you need to provide the lower and the upper bound of the section, as well as a stride *increment*, for all the dimensions. This notation is called a **subscript triplet:**

```
array ([lower]:[upper][:stride], ...)
```

When no lower and upper bounds are mentioned, it defaults to the extents you declared, and stride value defaults to 1.

The following example demonstrates the concept:

```
program arraySubsection

   real, dimension(10) :: a, b
   integer:: i, asize, bsize

   a(1:7) = 5.0 ! a(1) to a(7) assigned 5.0
   a(8:) = 0.0  ! rest are 0.0
   b(2:10:2) = 3.9
   b(1:9:2) = 2.5

   !display
   asize = size(a)
   bsize = size(b)

   do i = 1, asize
      Print *, a(i)
   end do

   do i = 1, bsize
      Print *, b(i)
   end do

end program arraySubsection
```

When the above code is compiled and executed, it produces the following result:

```
5.00000000
5.00000000
5.00000000
5.00000000
5.00000000
5.00000000
5.00000000
0.00000000E+00
0.00000000E+00
0.00000000E+00
2.50000000
3.90000010
2.50000000
3.90000010
2.50000000
3.90000010
2.50000000
3.90000010
2.50000000
3.90000010
```

Array Intrinsic Functions

Fortran 90/95 provides several intrinsic procedures. They can be divided into 7 categories.

- Vector and matrix multiplication
- Reduction
- Inquiry
- Construction
- Reshape
- Manipulation
- Location