13. 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.

Numbers(1)	Numbers(2)	Numbers(3)	Numbers(4)	…

Arrays can be one-dimensional (like vectors), two-dimensional (like matrices) 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 –numbers(1), numbers(2), numbers(3), numbers(4), and numbers(5).

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 (the extent) 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

Vector and Matrix Multiplication

The following table describes the vector and matrix multiplication functions:

Function	Description
dot_product(vector_a, vector_b)	This function returns a scalar product of two input vectors, which must have the same length.

matmul (matrix_a, matrix_b)	It returns the matrix product of two matrices, which must be consistent, i.e. have the dimensions like (m, k) and (k, n)

Example

The following example demonstrates dot product:

```
program arrayDotProduct

   real, dimension(5) :: a, b
   integer:: i, asize, bsize

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

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

   do i = 1, bsize
      b(i) = i*2
   end do

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

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

   Print*, 'Vector Multiplication: Dot Product:'
   Print*, dot_product(a, b)

end program arrayDotProduct
```

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

```
1.00000000
2.00000000
3.00000000
4.00000000
5.00000000
2.00000000
4.00000000
6.00000000
8.00000000
10.0000000
Vector Multiplication: Dot Product:
110.000000
```

Example

The following example demonstrates matrix multiplication:

```
program matMulProduct

   integer, dimension(3,3) :: a, b, c
   integer :: i, j

   do i = 1, 3
      do j = 1, 3
         a(i, j) = i+j
      end do
   end do

   print *, 'Matrix Multiplication: A Matrix'

   do i = 1, 3
      do j = 1, 3
         print*, a(i, j)
      end do
   end do

   do i = 1, 3
```

```
      do j = 1, 3
         b(i, j) = i*j
      end do
   end do

   Print*, 'Matrix Multiplication: B Matrix'

   do i = 1, 3
      do j = 1, 3
         print*, b(i, j)
      end do
   end do

   c = matmul(a, b)
   Print*, 'Matrix Multiplication: Result Matrix'

   do i = 1, 3
      do j = 1, 3
         print*, c(i, j)
      end do
   end do

end program matMulProduct
```

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

```
Matrix Multiplication: A Matrix
2
3
4
3
4
5
4
5
```

```
6
 Matrix Multiplication: B Matrix
1
2
3
2
4
6
3
6
9
Matrix Multiplication: Result Matrix
20
40
60
26
52
78
32
64
96
```

Reduction

The following table describes the reduction functions:

Function	Description
all(mask, dim)	It returns a logical value that indicates whether all relations in mask are .true., along with only the desired dimension if the second argument is given.
any(mask, dim)	It returns a logical value that indicates whether any relation in mask is .true., along with only the desired dimension if the second argument is given.

count(mask, dim)	It returns a numerical value that is the number of relations in mask which are .true., along with only the desired dimension if the second argument is given.
maxval(array, dim, mask)	It returns the largest value in the array array, of those that obey the relation in the third argument mask, if that one is given, along with only the desired dimension if the second argument dim is given.
minval(array, dim, mask)	It returns the smallest value in the array array, of those that obey the relation in the third argument mask, if that one is given, along with only the desired dimension if the second argument DIM is given.
product(array, dim, mask)	It returns the product of all the elements in the array array, of those that obey the relation in the third argument mask, if that one is given, along with only the desired dimension if the second argument dim is given.
sum (array, dim, mask)	It returns the sum of all the elements in the array array, of those that obey the relation in the third argument mask, if that one is given, along with only the desired dimension if the second argument dim is given.

Example

The following example demonstrates the concept:

```
program arrayReduction
   real, dimension(3,2) :: a
   a = reshape( (/5,9,6,10,8,12/), (/3,2/) )
   Print *, all(a>5)
   Print *, any(a>5)
   Print *, count(a>5)
   Print *, all(a>=5 .and. a<10)

end program arrayReduction
```

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

```
F
T
```

```
5
F
```

Example

The following example demonstrates the concept:

```
program arrayReduction
implicit none

   real, dimension(1:6) :: a = (/ 21.0, 12.0,33.0, 24.0, 15.0, 16.0 /)
   Print *, maxval(a)
   Print *, minval(a)
   Print *, sum(a)
   Print *, product(a)

end program arrayReduction
```

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

```
33.0000000
12.0000000
121.000000
47900160.0
```

Inquiry

The following table describes the inquiry functions:

Function	Description
allocated(array)	It is a logical function which indicates if the array is allocated.
lbound(array, dim)	It returns the lower dimension limit for the array. If dim (the dimension) is not given as an argument, you get an integer vector, if dim is included, you get the integer value with exactly that lower dimension limit, for which you asked.

shape(source)	It returns the shape of an array source as an integer vector.
size(array, dim)	It returns the number of elements in an array. If dim is not given, and the number of elements in the relevant dimension if dim is included.
ubound(array, dim)	It returns the upper dimensional limits.

Example

The following example demonstrates the concept:

```
program arrayInquiry

   real, dimension(3,2) :: a
   a = reshape( (/5,9,6,10,8,12/), (/3,2/) )

   Print *, lbound(a, dim=1)
   Print *, ubound(a, dim=1)
   Print *, shape(a)
   Print *, size(a,dim=1)

end program arrayInquiry
```

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

```
1
3
3 2
3
```

Construction

The following table describes the construction functions:

Function	Description
merge(tsource, fsource, mask)	This function joins two arrays. It gives the elements in tsource if the condition in mask is .true. and fsource if the condition in mask is .false. The two fields tsource and fsource have to be of the same type and the same shape. The result also is of this type and shape. Also mask must have the same shape.
pack(array, mask, vector)	It packs an array to a vector with the control of mask. The shape of the logical array mask, has to agree with the one for array, or else mask must be a scalar. If vector is included, it has to be an array of rank 1 (i.e. a vector) with at least as many elements as those that are true in mask, and have the same type as array. If mask is a scalar with the value .true. then vector instead must have the same number of elements as array.
spread(source, dim, ncopies)	It returns an array of the same type as the argument source with the rank increased by one. The parameters dim and ncopies are integer. if ncopies is negative the value zero is used instead. If source is a scalar, then spread becomes a vector with ncopies elements that all have the same value as source. The parameter dim indicates which index is to be extended. it has to be within the range 1 and 1+(rank of source), if source is a scalar then dim has to be one. The parameter ncopies is the number of elements in the new dimensions.
unpack(vector, mask, array)	It scatters a vector to an array under control of mask. The shape of the logical array mask has to agree with the one for array. The array vector has to have the rank 1 (i.e. it is a vector) with at least as many elements as those that are true in mask, and also has to have the same type as array. If array is given as a scalar then it is considered to be an array with the same shape as mask and the same scalar elements everywhere. The result will be an array with the same shape as mask and the same type as vector. The values will be those from vector that are accepted, while in the

	remaining positions in array the old values are kept.

Example

The following example demonstrates the concept:

```
program arrayConstruction
implicit none
   interface
      subroutine write_array (a)
         real :: a(:,:)
      end subroutine write_array

      subroutine write_l_array (a)
         logical :: a(:,:)
      end subroutine write_l_array
   end interface

   real, dimension(2,3) :: tsource, fsource, result
   logical, dimension(2,3) :: mask

   tsource = reshape( (/ 35, 23, 18, 28, 26, 39 /), &
                   (/ 2, 3 /) )
   fsource = reshape( (/ -35, -23, -18, -28, -26, -39 /), &
                   (/ 2,3 /) )
   mask = reshape( (/ .true., .false., .false., .true., &
                .false., .false. /), (/ 2,3 /) )
   result = merge(tsource, fsource, mask)
   call write_array(tsource)
   call write_array(fsource)
   call write_l_array(mask)
   call write_array(result)
end program arrayConstruction

subroutine write_array (a)
   real :: a(:,:)
   do i = lbound(a,1), ubound(a,1)
```

```
      write(*,*) (a(i, j), j = lbound(a,2), ubound(a,2) )
   end do
   return
end subroutine write_array



subroutine write_l_array (a)
   logical :: a(:,:)
   do i = lbound(a,1), ubound(a,1)
      write(*,*) (a(i, j), j= lbound(a,2), ubound(a,2))
   end do
   return
end subroutine write_l_array
```

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

```
35.0000000   18.0000000   26.0000000
23.0000000   28.0000000   39.0000000
-35.0000000  -18.0000000  -26.0000000
-23.0000000  -28.0000000  -39.0000000
T F F
F T F
35.0000000   -18.0000000  -26.0000000
-23.0000000  28.0000000   -39.0000000
```

Reshape

The following table describes the reshape function:

Function	Description
reshape(source, shape, pad, order)	It constructs an array with a specified shape shape starting from the elements in a given array source. If pad is not included then the size of source has to be at least product (shape). If pad is included, it has to have the same type as source. If order is included, it has to be an integer array with the same shape as shape and

	the values must be a permutation of (1,2,3,...,n), where n is the number of elements in shape , it has to be less than, or equal to 7.

Example

The following example demonstrates the concept:

```
program arrayReshape
implicit none

interface
   subroutine write_matrix(a)
   real, dimension(:,:) :: a
   end subroutine write_matrix
   end interface

   real, dimension (1:9) :: b = (/ 21, 22, 23, 24, 25, 26, 27, 28, 29 /)
   real, dimension (1:3, 1:3) :: c, d, e
   real, dimension (1:4, 1:4) :: f, g, h

   integer, dimension (1:2) :: order1 = (/ 1, 2 /)
   integer, dimension (1:2) :: order2 = (/ 2, 1 /)
   real, dimension (1:16) :: pad1 = (/ -1, -2, -3, -4, -5, -6, -7, -8, &
                                & -9, -10, -11, -12, -13, -14, -15, -16 /)

   c = reshape( b, (/ 3, 3 /) )
   call write_matrix(c)

   d = reshape( b, (/ 3, 3 /), order = order1)
   call write_matrix(d)

   e = reshape( b, (/ 3, 3 /), order = order2)
   call write_matrix(e)

   f = reshape( b, (/ 4, 4 /), pad = pad1)
   call write_matrix(f)
```

```
   g = reshape( b, (/ 4, 4 /), pad = pad1, order = order1)
   call write_matrix(g)

   h = reshape( b, (/ 4, 4 /), pad = pad1, order = order2)
   call write_matrix(h)

end program arrayReshape



subroutine write_matrix(a)
   real, dimension(:,:) :: a
   write(*,*)

   do i = lbound(a,1), ubound(a,1)
      write(*,*) (a(i,j), j = lbound(a,2), ubound(a,2))
   end do
end subroutine write_matrix
```

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

```
21.0000000  24.0000000  27.0000000
22.0000000  25.0000000  28.0000000
23.0000000  26.0000000  29.0000000

21.0000000  24.0000000  27.0000000
22.0000000  25.0000000  28.0000000
23.0000000  26.0000000  29.0000000

21.0000000  22.0000000  23.0000000
24.0000000  25.0000000  26.0000000
27.0000000  28.0000000  29.0000000

21.0000000  25.0000000  29.0000000   -4.00000000
22.0000000  26.0000000  -1.00000000  -5.00000000
23.0000000  27.0000000  -2.00000000  -6.00000000
24.0000000  28.0000000  -3.00000000  -7.00000000
```

```
21.0000000  25.0000000  29.0000000   -4.00000000
22.0000000  26.0000000  -1.00000000  -5.00000000
23.0000000  27.0000000  -2.00000000  -6.00000000
24.0000000  28.0000000  -3.00000000  -7.00000000

21.0000000  22.0000000  23.0000000   24.0000000
25.0000000  26.0000000  27.0000000   28.0000000
29.0000000  -1.00000000 -2.00000000  -3.00000000
-4.00000000 -5.00000000 -6.00000000  -7.00000000
```

Manipulation

Manipulation functions are shift functions. The shift functions return the shape of an array unchanged, but move the elements.

Function	Description
cshift(array, shift, dim)	It performs circular shift by shift positions to the left, if shift is positive and to the right if it is negative. If array is a vector the shift is being done in a natural way, if it is an array of a higher rank then the shift is in all sections along the dimension dim. If dim is missing it is considered to be 1, in other cases it has to be a scalar integer number between 1 and n (where n equals the rank of array). The argument shift is a scalar integer or an integer array of rank n-1 and the same shape as the array, except along the dimension dim (which is removed because of the lower rank). Different sections can therefore be shifted in various directions and with various numbers of positions.
eoshift(array, shift, boundary, dim)	It is end-off shift. It performs shift to the left if shift is positive and to the right if it is negative. Instead of the elements shifted out new elements are taken from boundary. If array is a vector the shift is being done in a

	natural way, if it is an array of a higher rank, the shift on all sections is along the dimension dim. if dim is missing, it is considered to be 1, in other cases it has to have a scalar integer value between 1 and n (where n equals the rank of array). The argument shift is a scalar integer if array has rank 1, in the other case it can be a scalar integer or an integer array of rank n-1 and with the same shape as the array array except along the dimension dim (which is removed because of the lower rank).
transpose (matrix)	It transposes a matrix, which is an array of rank 2. It replaces the rows and columns in the matrix.

Example

The following example demonstrates the concept:

```
program arrayShift
implicit none

   real, dimension(1:6) :: a = (/ 21.0, 22.0, 23.0, 24.0, 25.0, 26.0 /)
   real, dimension(1:6) :: x, y
   write(*,10) a

   x = cshift ( a, shift = 2)
   write(*,10) x

   y = cshift (a, shift = -2)
   write(*,10) y

   x = eoshift ( a, shift = 2)
   write(*,10) x

   y = eoshift ( a, shift = -2)
   write(*,10) y
```

```
   10 format(1x,6f6.1)

end program arrayShift
```

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

```
21.0  22.0  23.0  24.0  25.0  26.0
23.0  24.0  25.0  26.0  21.0  22.0
25.0  26.0  21.0  22.0  23.0  24.0
23.0  24.0  25.0  26.0   0.0   0.0
0.0    0.0  21.0  22.0  23.0  24.0
```

Example

The following example demonstrates transpose of a matrix:

```
program matrixTranspose
implicit none

   interface
      subroutine write_matrix(a)
         integer, dimension(:,:) :: a
      end subroutine write_matrix
   end interface

   integer, dimension(3,3) :: a, b
   integer :: i, j

   do i = 1, 3
      do j = 1, 3
         a(i, j) = i
      end do
   end do

   print *, 'Matrix Transpose: A Matrix'

   call write_matrix(a)
   b = transpose(a)
```

```
   print *, 'Transposed Matrix:'

   call write_matrix(b)
end program matrixTranspose

subroutine write_matrix(a)
   integer, dimension(:,:) :: a
   write(*,*)

   do i = lbound(a,1), ubound(a,1)
      write(*,*) (a(i,j), j = lbound(a,2), ubound(a,2))
   end do
end subroutine write_matrix
```

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

```
Matrix Transpose: A Matrix
1  1  1
2  2  2
3  3  3

Transposed Matrix:
1  2  3
1  2  3
1  2  3
```

Location

The following table describes the location functions:

Function	Description
maxloc(array, mask)	It returns the position of the greatest element in the array array, if mask is included only for those which fulfil the conditions in mask, position is returned and the result is an integer vector.

minloc(array, mask)	It returns the position of the smallest element in the array array, if mask is included only for those which fulfil the conditions in mask, position is returned and the result is an integer vector.

Example

The following example demonstrates the concept:

```
program arrayLocation
implicit none

   real, dimension(1:6) :: a = (/ 21.0, 12.0,33.0, 24.0, 15.0, 16.0 /)
   Print *, maxloc(a)
   Print *, minloc(a)

end program arrayLocation
```

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

```
3
2
```

14. Fortran – Dynamic Arrays

A **dynamic array** is an array, the size of which is not known at compile time, but will be known at execution time.

Dynamic arrays are declared with the attribute **allocatable**.

For example,

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

The rank of the array, i.e., the dimensions has to be mentioned however, to allocate memory to such an array, you use the **allocate** function.

```
allocate ( darray(s1,s2) )
```

After the array is used, in the program, the memory created should be freed using the**deallocate** function

```
deallocate (darray)
```

Example

The following example demonstrates the concepts discussed above.

```
program dynamic_array
implicit none

   !rank is 2, but size not known
   real, dimension (:,:), allocatable :: darray
   integer :: s1, s2
   integer :: i, j

   print*, "Enter the size of the array:"
   read*, s1, s2

   ! allocate memory
   allocate ( darray(s1,s2) )

   do i = 1, s1
      do j = 1, s2
         darray(i,j) = i*j
```

```
         print*, "darray(",i,",",j,") = ", darray(i,j)
      end do
   end do

   deallocate (darray)
end program dynamic_array
```

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

```
Enter the size of the array: 3,4
darray( 1 , 1 ) = 1.00000000
darray( 1 , 2 ) = 2.00000000
darray( 1 , 3 ) = 3.00000000
darray( 1 , 4 ) = 4.00000000
darray( 2 , 1 ) = 2.00000000
darray( 2 , 2 ) = 4.00000000
darray( 2 , 3 ) = 6.00000000
darray( 2 , 4 ) = 8.00000000
darray( 3 , 1 ) = 3.00000000
darray( 3 , 2 ) = 6.00000000
darray( 3 , 3 ) = 9.00000000
darray( 3 , 4 ) = 12.0000000
```

Use of Data Statement

The **data** statement can be used for initialising more than one array, or for array section initialisation.

The syntax of data statement is:

```
data variable / list / ...
```

Example

The following example demonstrates the concept:

```
program dataStatement
implicit none

   integer :: a(5), b(3,3), c(10),i, j
```

```
   data a /7,8,9,10,11/

   data b(1,:) /1,1,1/
   data b(2,:)/2,2,2/
   data b(3,:)/3,3,3/
   data (c(i),i=1,10,2) /4,5,6,7,8/
   data (c(i),i=2,10,2)/5*2/

   Print *, 'The A array:'
   do j = 1, 5
      print*, a(j)
   end do

   Print *, 'The B array:'
   do i = lbound(b,1), ubound(b,1)
      write(*,*) (b(i,j), j = lbound(b,2), ubound(b,2))
   end do

   Print *, 'The C array:'
   do j = 1, 10
      print*, c(j)
   end do

end program dataStatement
```

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

```
The A array:
7
8
9
10
11
The B array:
1  1  1
2  2  2
3  3  3
```

```
The C array:
4
2
5
2
6
2
7
2
8
2
```

Use of Where Statement

The **where** statement allows you to use some elements of an array in an expression, depending on the outcome of some logical condition. It allows the execution of the expression, on an element, if the given condition is true.

Example

The following example demonstrates the concept:

```
program whereStatement
implicit none

   integer :: a(3,5), i , j

   do i = 1,3
      do j = 1, 5
         a(i,j) = j-i
      end do
   end do

   Print *, 'The A array:'

   do i = lbound(a,1), ubound(a,1)
      write(*,*) (a(i,j), j = lbound(a,2), ubound(a,2))
   end do
```

```
   where( a<0 )
      a = 1
   elsewhere
      a = 5
   end where

   Print *, 'The A array:'
   do i = lbound(a,1), ubound(a,1)
      write(*,*) (a(i,j), j = lbound(a,2), ubound(a,2))
   end do

end program whereStatement
```

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

```
The A array:
0   1   2  3  4
-1  0   1  2  3
-2  -1  0  1  2
The A array:
5   5   5  5  5
1   5   5  5  5
1   1   5  5  5
```