Chapter 8
Whole Array and Additional Array Features

A good notation has a subtlety and suggestiveness which at times make it seem almost like a live teacher.

Bertrand Russell

Aims
The aims of the chapter are:

- To look more formally at the terminology required to precisely describe arrays.
- To introduce ways in which we can manipulate whole arrays and parts of arrays (sections).
- To introduce the concept of array element ordering and physical and virtual memory.
- To introduce ways in which we can initialise arrays using array constructors.
- To introduce the `where` statement and array masking.
- To introduce the `forall` statement and construct.
- Physical and virtual memory
- Type declaration statement summary.

8.1 Terminology

Fortran supports an abundance of array handling features. In order to make the description of these features more precise a number of additional terms have to be covered and these are introduced and explained below.

- Rank - The number of dimensions of an array is called its rank. A one dimensional array has rank 1, a two dimensional array has rank 2 and so on.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_8

- Bounds - An array's bounds are the upper and lower limits of the index in each dimension.
- Extent - The number of elements along a dimension of an array is called the extent.

```
integer, dimension(-10:15):: current
```

  has bounds –10 and 15 and an extent of 26.
- Size - The total number of elements in an array is its size.
- Shape - The shape of an array is determined by its rank and its extents in each dimension.
- Conformable - Two arrays are said to be conformable if they have the same shape, that is, they have the same rank and the same extent in each dimension.

8.2 Array Element Ordering

Array element ordering states that the elements of an array, regardless of rank, form a linear sequence. The sequence is such that the subscripts along the first dimension vary most rapidly, and those along the last dimension vary most slowly. This is best illustrated by considering, for example, a rank 2 array `a` defined by

```
real , dimension(1:4,1:2) :: a
```

`a` has 8 real elements whose array element order is a(1, 1), a(2, 1), a(3, 1), a(4, 1), a(1, 2), a(2, 2), a(3, 2), a(4, 2) i.e., mathematically by column and not row. We will look more formally at this later in this chapter.

8.3 Whole Array Manipulation

The examples of arrays so far have shown operations on arrays via array elements. One of the significant features of modern Fortran is its ability to manipulate arrays as whole objects. This allows arrays to be referenced not just as single elements but also as groups of elements. Along with this ability comes a whole host of intrinsic procedures for array processing. These procedures are mentioned in Chap. 12, and listed in alphabetical order with examples in Appendix D.

8.4 Assignment

An array name without any indices can appear on both sides of assignment and input and output statements. For example, values can be assigned to all the elements of an array in one statement:

```
real, dimension(1:12):: rainfall
rainfall=0.0
```

The elements of one array can be assigned to another:

```
integer, dimension(1:50) :: a,b
.
.
a=b
```

Arrays `a` and `b` must be conformable in order to do this.

The following example is illegal since `x` is rank 1 and extent 20, whilst `z` is rank 1 and extent 41.

```
real, dimension(1:20) :: x
real, dimension(1:41) :: z
x=50.0
z=x
```

But the following is legal because both arrays are now conformable, i.e., they are both of rank 1 and extent 41:

```
real , dimension (-20:20) :: x
real , dimension (1:41) :: y
x=50.0
y=x
```

8.5 Expressions

All the arithmetic operators available to scalars are available to arrays, but care must be taken because mathematically they may not make sense.

```
real , dimension (1:50) :: a,b,c,d,e
c=a+b
```

adds each element of `a` to the corresponding element of `b` and assigns the result to `c`.

```
e=c*d
```

multiplies each element of `c` by the corresponding element of `d`. This is not vector multiplication. To perform a vector dot product there is an intrinsic procedure

dot_product, and an example of this is given in a subsequent section on array constructors.

For higher dimensions

```
real ,dimension (1:10,1:10) :: f,g,h
f=f**0.5
```

takes the square root of every element of f.

```
h=f+g
```

adds each element of f to the corresponding element of g.

```
h=f*g
```

multiplies each element of f by the corresponding element of g. The last statement is not matrix multiplication. An intrinsic procedure matmul performs matrix multiplication; further details are given in Appendix D.

8.6 Example 1: Rank 1 Whole Arrays in Fortran

Consider the following example, which is a solution to a problem set earlier, but is now addressed using some of the whole array features of Fortran

```
program ch0801
  implicit none
  integer, parameter :: n = 12
  real, dimension (1:n) :: rainfall_ins = 0.0
  real, dimension (1:n) :: rainfall_cms = 0.0
  integer :: month

  print *, &
    ' Input the rainfall values in inches'
  read *, rainfall_ins
  rainfall_cms = rainfall_ins*2.54
  do month = 1, n
    print *, ' ', month, ' ', rainfall_ins(month &
      ), ' ', rainfall_cms(month)
  end do
end program ch0801
```

The statements

```
real , dimension(1:n) :: rainfall_ins=0.0
real , dimension(1:n) :: rainfall_cms=0.0
```

are examples of whole array initialisation. Each element of the arrays is set to 0.0.

The statement

```
read *, rainfall_ins
```

is an example of whole array i/o, where we no longer have to use a do loop to read each element in.

Finally, we have the statement

```
rainfall_cms = rainfall_ins * 2.54
```

which is an example of whole array arithmetic and assignment.

8.7 Example 2: Rank 2 Whole Arrays in Fortran

Here is a two-dimensional example:

```
program ch0802
! This program reads in a grid of temperatures
! (degrees Fahrenheit) at 25 grid references
! and converts them to degrees Celsius
  implicit none
  integer, parameter :: n = 5
  real, dimension (1:n, 1:n) :: fahrenheit, &
    celsius
  integer :: long, lat
!
! read in the temperatures
!
  do lat = 1, n
    print *, ' For Latitude= ', lat
    do long = 1, n
      print *, ' For Longitude', long
      read *, fahrenheit(lat, long)
    end do
  end do
!
! Conversion applied to all values
!
```

```
  celsius = 5.0/9.0*(fahrenheit-32.0)
  print *, celsius
  print *, fahrenheit
end program ch0802
```

Note the use of whole arrays in the print statements. The output does look rather messy though, and also illustrates array element ordering.

8.8 Array Sections

Often it is necessary to access part of an array rather than the whole, and this is possible with Fortran's powerful array manipulation features.

8.8.1 Example 3: Rank 1 Array Sections

Consider the following:

```
program ch0803
  implicit none
  integer, dimension (-5:5) :: x
  integer :: i

  x(-5:-1) = -1
  x(0) = 0
  x(1:5) = 1
  do i = -5, 5
    print *, ' ', i, ' ', x(i)
  end do
end program ch0803
```

The statement

```
  x(-5:-1) = -1
```

is working with a section of an array. It assigns the value –1 to elements `x(-5)` through `x(-1)`.

The statement

```
  x(1:5)   =  1
```

is also working with an array section. It assigns the value 1 to elements `x(1)` through `x(5)`.

8.8.2 Example 4: Rank 2 Array Sections

In Chap. 6 we gave an example of a table of examination marks, and this is given again below:

Name	Physics	Maths	Biology	History	English	French
Fowler L.	50	47	28	89	30	46
Barron L.W	37	67	34	65	68	98
Warren J.	25	45	26	48	10	36
Mallory D.	89	56	33	45	30	65
Codd S.	68	78	38	76	98	65

The following program reads the data in, scales column 3 by 2.5 as the Biology marks were out of 40 (the rest are out of 100), calculates the averages for each subject and for each person and prints out the results.

```
program ch0804
  implicit none
  integer, parameter :: nrow = 5
  integer, parameter :: ncol = 6
  real, dimension (1:nrow, 1:ncol) :: &
    exam_results = 0.0
  real, dimension (1:nrow) :: people_average = &
    0.0
  real, dimension (1:ncol) :: subject_average = &
    0.0
  integer :: r, c

  do r = 1, nrow
    read *, exam_results(r, 1:ncol)
  end do
  exam_results(1:nrow, 3) = 2.5* &
    exam_results(1:nrow, 3)
  do r = 1, nrow
    do c = 1, ncol
      people_average(r) = people_average(r) + &
        exam_results(r, c)
    end do
  end do
```

```
  people_average = people_average/ncol
  do c = 1, ncol
    do r = 1, nrow
      subject_average(c) = subject_average(c) + &
        exam_results(r, c)
    end do
  end do
  subject_average = subject_average/nrow
  print *, ' People averages'
  print *, people_average
  print *, ' Subject averages'
  print *, subject_average
end program ch0804
```

The statement

```
  read *, exam_results(r,1:ncol)
```

uses sections to replace the implied do loop in the earlier example, takes column 3 of the two dimensional array `exam_results`, multiplies it by 2.5 (as a whole array) and overwrites the original values.

The statement

```
  exam_results(1:nrow,3) = &
    2.5 * exam_results(1:nrow,3)
```

uses array sections in the arithmetic and the assignment.

8.9 Array Constructors

Arrays can be given initial values in Fortran using array constructors. Some examples are given below.

8.9.1 Example 5: Rank 1 Array Initialisation — Explicit Values

```
program ch0805
  implicit none
  integer, parameter :: n = 12
  real :: total = 0.0, average = 0.0
```

```
  real, dimension (1:n) :: rainfall = (/ 3.1, &
    2.0, 2.4, 2.1, 2.2, 2.2, 1.8, 2.2, 2.7, 2.9, &
    3.1, 3.1 /)
  integer :: month

  do month = 1, n
    total = total + rainfall(month)
  end do
  average = total/n
  print *, ' Average monthly rainfall was'
  print *, average
end program ch0805
```

The statement

```
real , dimension(1:n) :: rainfall = &
(/3.1,2.0,2.4,2.1,2.2,2.2,1.8,2.2,2.7,2.9,3.1,3.1/)
```

provides initial values to the elements of the array `rainfall`.

8.9.2 Example 6: Rank 1 Array Initialisation Using an Implied Do Loop

The next example uses a simple variant:

```
program ch0806
  implicit none
!
! 1 us gallon = 3.7854118 litres
! 1 uk gallon = 4.545 litres
!
  integer, parameter :: n = 10
  real, parameter :: us = 3.7854118
  real, parameter :: uk = 4.545
  integer :: i
  integer, dimension (1:n) :: litre = [ (i,i=1,n &
    ) ]
  real, dimension (1:n) :: gallon, usgallon

  gallon = litre/uk
  usgallon = litre/us
  print *, ' Litres Imperial USA'
```

```
  print *, ' Gallon Gallon'
  do i = 1, n
    print *, litre(i), ' ', gallon(i), ' ', &
      usgallon(i)
  end do
end program ch0806
```

The statement

```
integer , dimension(1:n) :: litre=[(i,i=1,n)]
```

initialises the 10 elements of the `litre` array to the values 1,2,3,4,5,6,7,8,9,10 respectively.

8.9.3 Example 7: Rank 1 Arrays and the `dot_product` Intrinsic

This example uses an array constructor and the intrinsic procedure `dot_product`.

```
program ch0807
  implicit none
  integer, dimension (1:3) :: x, y
  integer :: result

  x = [ 1, 3, 5 ]
  y = [ 2, 4, 6 ]
  result = dot_product(x, y)
  print *, result
end program ch0807
```

and result has the value 44, which is obtained by the normal mathematical dot product operation, 1*2 + 3*4 + 5*6.

The general form of the array constructor is `[list of expressions]` or `(/ a list of expressions /)` where each expression is of the same type.

8.9.4 Initialising Rank 2 Arrays

To construct arrays of higher rank than one the intrinsic function `reshape` must be used. An introduction to intrinsic functions is given in Chap. 12, and an alphabetic

list with a full explanation of each function is given in Appendix D. To use it in its simplest form:

```
matrix = reshape ( source, shape)
```

where `source` is a rank 1 array containing the values of the elements required in the new array, `matrix`, and `shape` is a rank 1 array containing the shape of the new array `matrix`.

We consider the rank 1 array `b=(1,3,5,7,9,11)`, and we wish to store these values in a rank 2 array `a`, such that `a` is the matrix:

$$a = \begin{pmatrix} 1 & 7 \\ 3 & 9 \\ 5 & 11 \end{pmatrix}$$

The following code extract is needed:

```
integer, dimension(1:6) :: b
integer, dimension(1:3, 1:2) :: a
  b = (/1,3,5,7,9,11/)
  a = reshape(b,(/3,2/))
```

Note that the elements of the source array `b` must be stored in the array element order of the required array `a`.

8.9.5 Example 8: Initialising a Rank 2 Array

The following example illustrates the additional forms of the `reshape` function that are used when the number of elements in the source array is less than the number of elements in the destination. The complete form is

```
reshape(source, shape, pad, order)
```

`pad` and `order` are optional. See Appendix D for a complete explanation of `pad` and `order`:

```
program ch0808
  implicit none
  integer, dimension (1:2, 1:4) :: x
  integer, dimension (1:8) :: y = (/ 1, 2, 3, 4, &
    5, 6, 7, 8 /)
  integer, dimension (1:6) :: z = (/ 1, 2, 3, 4, &
```

```
      5, 6 /)
    integer :: r, c

    print *, ' Source array y'
    print *, y
    print *, ' Source array z'
    print *, z
    print *, ' Simple reshape sizes match'
    x = reshape(y, (/2,4/) )
    do r = 1, 2
      print *, (x(r,c), c=1, 4)
    end do
    print *, &
      ' Source 2 elements smaller pad with 0'
    x = reshape(z, (/2,4/), (/0,0/) )
    do r = 1, 2
      print *, (x(r,c), c=1, 4)
    end do
    print *, &
      ' As previous now specify order as 1*2'
    x = reshape(z, (/2,4/), (/0,0/), (/1,2/) )
    do r = 1, 2
      print *, (x(r,c), c=1, 4)
    end do
    print *, &
      ' As previous now specify order as 2*1'
    x = reshape(z, (/2,4/), (/0,0/), (/2,1/) )
    do r = 1, 2
      print *, (x(r,c), c=1, 4)
    end do
  end program ch0808
```

8.10 Miscellaneous Array Examples

The following are examples of some of the flexibility of arrays in Fortran.

8.10.1 Example 9: Rank 1 Arrays and a Stride of 2

Consider the following example:

```
program ch0809
  implicit none
  integer :: i
  integer, dimension (1:10) :: x = (/ (i,i=1,10) &
    /)
  integer, dimension (1:5) :: odd = (/ (i,i=1,10 &
    ,2) /)
  integer, dimension (1:5) :: even

  even = x(2:10:2)
  print *, ' x'
  print *, x
  print *, ' odd'
  print *, odd
  print *, ' even'
  print *, even
end program ch0809
```

The statement

```
integer , dimension(1:5)  :: odd=(/(i,i=1,10,2)/)
```

steps through the array 2 at a time.

The statement

```
  even=x(2:10:2)
```

shows an array section where we go from elements two through ten in steps of two. The 2:10:2 is an example of a subscript triplet in Fortran, and the first 2 is the lower bound, the 10 is the upper bound, and the last 2 is the increment. Fortran uses the term stride to mean the increment in a subscript triplet.

8.10.2 Example 10: Rank 1 Array and the Sum Intrinsic Function

The following example is based on ch0805. It uses the `sum` intrinsic to calculate the sum of all the values in the `rainfall` array.

```
program ch0810
  implicit none
  real :: total = 0.0, average = 0.0
  real, dimension (12) :: rainfall = (/ 3.1, 2.0 &
```

```
    , 2.4, 2.1, 2.2, 2.2, 1.8, 2.2, 2.7, 2.9, &
    3.1, 3.1 /)

  total = sum(rainfall)
  average = total/12
  print *, ' Average monthly rainfall was'
  print *, average
end program ch0810
```

The statement

```
  total = sum(rainfall)
```

replaces the statements below from the earlier example.

```
  do month=1,n
    total = total + rainfall(month)
  enddo
```

In this example the `sum` intrinsic function adds up all of the elements of the array `rainfall`.

So we have three ways of processing arrays:

- Element by element.
- Using sections.
- On a whole array basis.

The ability to use sections and whole arrays when programming is a major advance of the element by element processing supported by Fortran 77.

8.10.3 Example 11: Rank 2 Arrays and the Sum Intrinsic Function

This example is based on the earlier exam results program:

```
program ch0811
  implicit none
  integer, parameter :: nrow = 5
  integer, parameter :: ncol = 6
  real, dimension (1:nrow*ncol) :: results = (/ &
    50, 47, 28, 89, 30, 46, 37, 67, 34, 65, 68, &
    98, 25, 45, 26, 48, 10, 36, 89, 56, 33, 45, &
    30, 65, 68, 78, 38, 76, 98, 65 /)
  real, dimension (1:nrow, 1:ncol) :: &
```

```
    exam_results = 0.0
  real, dimension (1:nrow) :: people_average = &
    0.0
  real, dimension (1:ncol) :: subject_average = &
    0.0

  exam_results = reshape(results, (/nrow,ncol/), &
    (/0.0,0.0/), (/2,1/) )
  exam_results(1:nrow, 3) = 2.5* &
    exam_results(1:nrow, 3)
  subject_average = sum(exam_results, dim=1)
  people_average = sum(exam_results, dim=2)
  people_average = people_average/ncol
  subject_average = subject_average/nrow
  print *, ' People averages'
  print *, people_average
  print *, ' Subject averages'
  print *, subject_average
end program ch0811
```

This example has several interesting array features:

- We initialise a rank 1 array with the values we want in our exam marks array. The data are laid out in the program as they would be in an external file in rows and columns.
- We use `reshape` to initialise our exam marks array. We use the fourth parameter `(/2,1/)` to populate the rank 2 array with the data in row order.
- We use `sum` with a `dim` of 1 to compute the sums for the subjects.
- We use `sum` with a `dim` of 2 to compute the sums for the people.

8.10.4 *Example 12: Masked Array Assignment and the* `where` *Statement*

Fortran has array assignment both on an element by element basis and on a whole array basis. There is an additional form of assignment based on the concept of a logical mask.

Consider the example of time zones given in Chap. 7. The `time` array will have values that are both negative and positive. We can then associate the positive values with the concept of east of the Greenwich meridian, and the negative values with the concept of west of the Greenwich meridian e.g.:

```
program ch0812
  implicit none
```

```
  real, dimension (-180:180) :: time = 0
  integer :: degree, strip
  real :: value
  character (len=1), dimension (-180:180) :: &
    direction = ' '

  do degree = -180, 165, 15
    value = degree/15.
    do strip = 0, 14
      time(degree+strip) = value
    end do
  end do
  do degree = -180, 180
    print *, degree, ' ', time(degree)
  end do
  where (time>0.0)
    direction = 'E'
    elsewhere (time<0.0)
    direction = 'W'
  end where
  print *, direction
end program ch0812
```

8.10.5 Notes

The arrays must be conformable, i.e., in our example `time` and `direction` are the same shape.

The selective assignment is achieved through the `where` construct.

Both the `where` and `elsewhere` blocks can be executed.

The formal syntax is:

```
where (array logical expression)
  ...
elsewhere (array logical expression)
  ...
end where
```

The first array assignment is executed where `time` is positive and the second is executed where `time` is negative. For further coverage of logical expressions see Chaps. 13 and 16.

8.11 Array Element Ordering in More Detail

Fortran compilers will store arrays in memory according to the array element ordering scheme. Section 9.5.3.2 of the Fortran 2018 standard provides details of this. Table 8.1 summarises the information for rank 1, 2 and 3 arrays.

Table 8.1 Array element ordering in Fortran

Rank	Subscript bounds	Subscript list	Subscript order value
1	j1:k1	s1	1 + (s1 − j1)
2	j1:k1, j2:k2	s1, s2	1 + (s1 − j1) + (s2 − j2)*d1
3	j1:k1, j2:k2, j3 − k3	s1, s2, s3	1 + (s1 − j1) + (s2 − j2)*d1 + (s3 − j3)*d2*d1

8.11.1 Example 13: Array Element Ordering

Here is a short program illustrating the above for a 2*5 array.

```
program ch0813
  implicit none
  integer :: j1 = 1
  integer :: k1 = 2
  integer :: j2 = 1
  integer :: k2 = 5
  integer :: s1
  integer :: s2
  integer :: d1
  integer :: position

  d1 = k1 - j1 + 1
  print *, ' Row  Column    Position'
  do s1 = j1, k1
    do s2 = j2, k2
      position = 1 + (s1-j1) + (s2-j2)*d1
      print 100, s1, s2, position
100   format (3x, i2, 6x, i2, 10x, i2)
    end do
  end do

end program ch0813
```

and here is the output.

```
 Row  Column    Position
   1       1           1
   1       2           3
   1       3           5
   1       4           7
   1       5           9
   2       1           2
   2       2           4
   2       3           6
   2       4           8
   2       5          10
```

So for rank 2 arrays the array element ordering is by column, not row.

8.12 Physical and Virtual Memory

There will be a limit to the amount of physical memory available on any computer system. To enable problems that require more than the amount of physical memory available to be solved, most implementations will provide access to virtual memory, which in reality means access to a portion of a physical disk.

Access to virtual memory is commonly provided by a paging mechanism of some description. Paging is a technique whereby fixed-sized blocks of data are swapped between real memory and disk as required.

In order to minimise paging (and hence reduce execution time) array operations should be performed according to the array element order.

Page sizes, past and present, include:

- Sun UltraSparc – 4 Kb, 8 Kb.
- DEC Alpha – 8 Kb, 16 Kb, 32 Kb, 64 Kb.
- Intel 80 $\times$ 86 – 4 Kb.
- Intel Pentium PIII – 4 Kb, 2 Mb, 4 Mb.
- AMD64 – 4 Kb, 2 Mb, 4 Mb - legacy mode
- AMD64 – 4 Kb, 2 Mb, 1 Gb - 64 bit mode
- Intel 64 and IA-32 – 4 Kb, 2 Mb, 1 Gb - depending on mode.

See the references at the end of the chapter for more details.

8.13 Type Declaration Statement Summary

It is a convenient time to introduce a summary of the syntax of type declarations. You have already seen some of these, and we will cover the rest in later chapters.

A type declaration statement normally has three components

- a type declaration
- optional attributes
- variable list

Here are details of the type declaration.

- intrinsic type specifier
- type (derived type specification)
- class (derived type specification)
- class (*)

The attribute specification is one of

- allocatable
- asynchronous
- bind
- dimension
- external
- intent
- intrinsic
- optional
- parameter
- pointer
- private
- protected
- public
- save
- target
- value
- volatile

8.14 Summary

We can now perform operations on whole arrays and partial arrays (array sections) without having to refer to individual elements. This shortens program development time and greatly clarifies the meaning of programs.

Array constructors can be used to assign values to rank 1 arrays within a program unit. The `reshape` function allows us to assign values to a two or higher rank array when used in conjunction with an array constructor.

8.15 Problems

8.1 Compile and run all the examples.

8.2 Give the rank, bounds, extent and size of the following arrays:

```
real , dimension(1:15) :: a
integer , dimension(1:3,0:4) :: b
real , dimension(-2:2,0:1,1:4) :: c
integer , dimension(0:2,1:5) :: d
```

Which two of these arrays are conformable?

8.3 Write a program to read in five rank 1 arrays, `a`, `b`, `c`, `d`, `e` and then store them as five columns in a rank 2 array `table`.

8.4 Take the first part of Problem 7.5 in Chap. 7 and rewrite it using the `sum` intrinsic function.

8.16 Bibliography

8.16.1 DEC Alpha

Bhandarkar D.P., Alpha Implementation and Architecture: Complete Reference and Guide, Digital Press,1995.

8.16.2 AMD

Visit

```
http://support.amd.com/en-us/search/tech-docs
```

for details of the AMD manuals. The following five manuals are available for download as pdf's from the above site.

- AMD64 Architecture Programmer's Manual Volume 1: Application Programming
- AMD64 Architecture Programmer's Manual Volume 2: System Programming
- AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions
- AMD64 Architecture Programmer's Manual Volume 4: 128-bit and 256 bit media instructions

- AMD64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Floating-Point Instructions

8.16.3 Intel

Visit

```
https://software.intel.com/en-us/articles/intel-sdm
```

for a list of manuals. The following three manuals are available for download as pdf's from the above site.

- Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture
- Intel 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes 2A and 2B: Instruction Set Reference, A-Z.
- Intel 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes 3A and 3B: System Programming Guide, Parts 1 and 2.