Chapter 7
Arrays 2: Further Examples

Sir, In your otherwise beautiful poem (The Vision of Sin) there is a verse which reads Every moment dies a man, every moment one is born. Obviously this cannot be true and I suggest that in the next edition you have it read Every moment dies a man, every moment 1 1/16 is born. Even this value is slightly in error but should be sufficiently accurate for poetry.

Charles Babbage in a letter to Lord Tennyson

Aims

The aims of the chapter are to extend the concepts introduced in the previous chapter and in particular:

- To set an array size at run time - allocatable arrays.
- To introduce the idea of an array with more than one dimension and the corresponding control structure to permit easy manipulation of higher-dimensioned arrays.
- To introduce an extended form of the dimension attribute declaration, and the corresponding alternative form to the do statement, to manipulate the array in this new form.
- To introduce the do loop as a mechanism for the control of repetition in general, not just for manipulating arrays.
- To formally define the block do syntax.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_7

7.1 Varying the Array Size at Run Time

The earlier examples set the array size in the following two ways:

- Explicitly using a numeric constant
- Implicitly using a parameterised variable

In both cases we knew the size of the array at the time we compiled the program. We may not know the size of the array at compile time and Fortran provides the allocatable attribute to accommodate this kind of problem.

7.1.1 Example 1: Allocatable Arrays

Consider the following example.

```
program ch0701
!
! This program is a simple variant of ch0602.
! The array is now allocatable
! and the user is prompted for the
! number of people at run time.
!
  implicit none
  integer :: number_of_people
  real :: total = 0.0, average = 0.0
  integer :: person
  real, dimension (:), allocatable :: weight

  print *, ' How many people?'
  read *, number_of_people
  allocate (weight(1:number_of_people))
  do person = 1, number_of_people
    print *, ' type in the weight for person ', &
      person
    read *, weight(person)
    total = total + weight(person)
  end do
  average = total/number_of_people
  print *, ' The total of the weights is ', &
    total
  print *, ' Average Weight is ', average
  print *, ' ', number_of_people, &
    ' Weights were '
```

```
    do person = 1, number_of_people
      print *, weight(person)
    end do
  end program ch0701
```

The first statement of interest is the type declaration with the `dimension` and `allocatable` attributes, e.g.,

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

The second is the `allocate` statement

```
  allocate(weight(1:number_of_people))
```

where the value of the variable `number_of_people` is not known until run time. This is known in Fortran as a deferred shape array.

7.2 Higher-Dimension Arrays

There are many instances where it is necessary to have arrays with more than one dimension. Consider the examples below.

7.2.1 Example 2: Two Dimensional Arrays and a Map

Considerthe representation of the height of an area of land expressed as a two dimensional table of numbers e.g., we may have some information represented in a simple table as follows:

```
             Longitude
             1        2       3
Latitude
  1        10.0     40.0    70.0
  2        20.0     50.0    80.0
  3        30.0     60.0    90.0
```

The values in the array are the heights above sea level. The example is obviously artificial, but it does highlight the concepts involved. For those who have forgotten their geography, lines of latitude run east–west (the equator is a line of latitude) and lines of longitude run north–south (they go through the poles and are all of the same length). In the above table therefore the latitude values are ordered by row and the longitude values are ordered by column.

A program to manipulate this data structure would involve something like the following:

```
program ch0702
! Variables used
! Height - used to hold the heights above sea
! level
! Long - used to represent the longitude
! Lat - used to represent the latitude
! both restricted to integer values.
! Correct - holds the correction factor
  implicit none
  integer, parameter :: n = 3
  integer :: lat, long
  real, dimension (1:n, 1:n) :: height
  real, parameter :: correct = 10.0

  do lat = 1, n
    do long = 1, n
      print *, ' type in value at ', lat, ' ', &
        long
      read *, height(lat, long)
    end do
  end do
  do lat = 1, n
    do long = 1, n
      height(lat, long) = height(lat, long) + &
        correct
    end do
  end do
  print *, ' Corrected data is '
  do lat = 1, n
    do long = 1, n
      print *, height(lat, long)
    end do
  end do
end program ch0702
```

Note the way in which indentation has been used to highlight the structure in this example. Note also the use of a textual prompt to highlight which data value is expected. Running the program highlights some of the problems with the simple i/o used in the example above. We will address this issue in the next example.

The inner loop is said to be nested within the outer one. It is very common to encounter problems where nesting is a natural way to express the solution. Nesting is permitted to any depth. Here is an example of a valid nested do loop:

```
do         ! Start of outer loop
  do           ! Start of inner loop
    .
    .
  enddo         ! End of inner loop
enddo         ! End of outer loop
```

This example introduces the concept of two indices, and can be thought of as a row and column data structure.

7.2.2 Example 3: Sensible Tabular Output

The first example had the values printed in a format that wasn't very easy to work with. In this example we introduce a so-called implied do loop, which enables us to produce neat and humanly comprehensible output:

```
program ch0703
! Variables used
! Height - used to hold the heights above sea
! level
! Long - used to represent the longitude
! Lat - used to represent the latitude
! both restricted to integer values.
  implicit none
  integer, parameter :: n = 3
  integer :: lat, long
  real, dimension (1:n, 1:n) :: height
  real, parameter :: correct = 10.0

  do lat = 1, n
    do long = 1, n
      read *, height(lat, long)
      height(lat, long) = height(lat, long) + &
        correct
    end do
  end do
  do lat = 1, n
    print *, (height(lat,long), long=1, n)
```

```
  end do
end program ch0703
```

The key statement in this example is

```
print * , (height(lat,long),long=1,n)
```

This is called an implied do loop, as the `longitude` variable takes on values from 1 through 3 and will write out all three values on one line.

We will see other examples of this statement as we go on.

7.2.3 Example 4: Average of Three Sets of Values

This example extends the previous one. Now we have three sets of measurements and we are interested in calculating the average of these three sets. The two new data sets are:

```
 9.5    39.5     69.5
19.5    49.5     79.5
29.5    59.5     89.5
```

and

```
10.5    40.5     70.5
20.5    50.5     80.5
30.5    60.5     90.5
```

and we have chosen the values to enable us to quickly check that the calculations for the averages are correct.

This program also uses implied do loops to read the data, as data in files are generally tabular:

```
program ch0704
! Variables used
! h1,h2,h3
! used to hold the heights above sea level
! h4
! used to hold the average of the above
! Long - used to represent the longitude
! Lat - used to represent the latitude
! both restricted to integer values.
  implicit none
```

```
  integer, parameter :: n = 3
  integer :: lat, long
  real, dimension (1:n, 1:n) :: h1, h2, h3, h4

  do lat = 1, n
    read *, (h1(lat,long), long=1, n)
  end do
  do lat = 1, n
    read *, (h2(lat,long), long=1, n)
  end do
  do lat = 1, n
    read *, (h3(lat,long), long=1, n)
  end do
  do lat = 1, n
    do long = 1, n
      h4(lat, long) = (h1(lat,long)+h2(lat,long) &
        +h3(lat,long))/n
    end do
  end do
  do lat = 1, n
    print *, (h4(lat,long), long=1, n)
  end do
end program ch0704
```

The original data was accurate to three significant figures. The output from the above has spurious additional accuracy. We will look at how to correct this in the later chapter on output.

7.2.4 Example 5: Booking Arrangements in a Theatre or Cinema

A theatre or cinema consists of rows and columns of seats. In a large cinema or a typical theatre there would also be more than one level or storey. Thus, a program to represent and manipulate this structure would probably have a 2-d or 3-d array. Consider the following program extract:

```
program ch0705
  implicit none
  integer, parameter :: nr = 5
  integer, parameter :: nc = 10
  integer, parameter :: nf = 3
  integer :: row, column, floor
```

```
  character *1, dimension (1:nr, 1:nc, 1:nf) :: &
    seats = ' '

  do floor = 1, nf
    do row = 1, nr
      read *, (seats(row,column,floor), column=1 &
        , nc)
    end do
  end do
  print *, ' Seat plan is'
  do floor = 1, nf
    print *, ' Floor = ', floor
    do row = 1, nr
      print *, (seats(row,column,floor), column= &
        1, nc)
    end do
  end do
end program ch0705
```

Note here the use of the term `parameter` in conjunction with the integer declaration. This is called an entity orientated declaration. An alternative to this is an attribute-orientated declaration, e.g.,

```
integer :: nr,nc,nf
parameter :: nr=5,nc=10,nf=3
```

and we will be using the entity-orientated declaration method throughout the rest of the book. This is our recommended method as you only have to look in one place to determine everything that you need to know about an entity.

7.3 Additional Forms of the Dimension Attribute and Do Loop Statement

7.3.1 Example 6: Voltage from –20 to +20 Volts

Consider the problem of an experiment where the independent variable voltage varies from –20 to +20 volts and the current is measured at 1-volt intervals. Fortran has a mechanism for handling this type of problem:

```
program ch0706
  implicit none
  real, dimension (-20:20) :: current
```

```
    real :: resistance
    integer :: voltage

    print *, ' type in the resistance'
    read *, resistance
    do voltage = -20, 20
      current(voltage) = voltage/resistance
      print *, voltage, ' ', current(voltage)
    end do
  end program ch0706
```

We appreciate that, due to experimental error, the voltage will not have exact integer values. However, we are interested in representing and manipulating a set of values, and thus from the point of view of the problem solution and the program this is a reasonable assumption. There are several things to note.

This form of the dimension attribute

```
dimension(first:last)
```

is of considerable use when the problem has an effective index which does not start at 1.

There is a corresponding form of the do statement which allows processing of problems of this nature. This is shown in the above program. The general form of the do statement statement is therefore:

```
do counter=start, end, increment
```

where `start`, `end` and `increment` can be positive or negative. Note that zero is a legitimate value of the dimension limits and of a do loop index.

7.3.2 Example 7: Longitude from –180 to +180

Consider the problem of the production of a table linking time difference with longitude. The values of longitude will vary from –180 to +180 degrees, and the time will vary from +12 hours to –12 hours. A possible program segment is:

```
program ch0707
  implicit none
  real, dimension (-180:180) :: time = 0
  integer :: degree, strip
  real :: value

  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
  end program ch0707
```

7.3.3 Notes

The values of the time are not being calculated at every degree interval.

The variable `time` is a real variable. It would be possible to arrange for the time to be an integer by expressing it in either minutes or seconds.

This example takes no account of all the wiggly bits separating time zones or of British Summer Time or Daylight Saving Time.

What changes would you make to the program to accommodate +180? What is the time at –180 and +180?

7.4 The Do Loop and Straight Repetition

7.4.1 Example 8: Table of Liquid Conversion Measurements

Consider the production of a table of liquid measurements. The independent variable is the litre value; the gallon and US gallon are the dependent variables. Strictly speaking, a program to do this does not have to have an array, i.e., the do loop can be used to control the repetition of a set of statements that make no reference to an array. The following shows a complete but simple conversion program:

```
program ch0708
  implicit none
!
! 1 us gallon = 3.7854118 litres
! 1 uk gallon = 4.545 litres
!
  integer :: litre
  real :: gallon, usgallon

  do litre = 1, 10
```

```
    gallon = litre/4.545
    usgallon = litre/3.7854118
    print *, litre, ' ', gallon, ' ', usgallon
  end do
end program ch0708
```

Note here that the do statement has been used only to control the repetition of a block of statements — there are no arrays at all in this program.

This is the other use of the do statement. The do loop thus has two functions — its use with arrays as a control structure and its use solely for the repetition of a block of statements.

7.4.2 Example 9: Means and Standard Deviations

In the calculation of the mean and standard deviation of a list of numbers, we can use the following formulae. It is not actually necessary to store the values, nor to accumulate the sum of the values and their squares. In the first case, we would possibly require a large array, whereas in the second, it is conceivable that the accumulated values (especially of the squares) might be too large for the machine. The following example uses an updating technique which avoids these problems, but is still accurate. The do loop is simply a control structure to ensure that all the values are read in, with the index being used in the calculation of the updates:

```
program ch0709
! variables used are
! mean - for the running mean
! ssq - the running corrected sum of squares
! x - input values for
which
! mean and sd required
! w - local work variable
! sd - standard deviation
! r - another work variable
  implicit none
  real :: mean = 0.0, ssq = 0.0, x, w, sd, r
  integer :: i, n

  print *, ' enter the number of readings'
  read *, n
  print *, ' enter the ', n, &
    ' values, one per line'
  do i = 1, n
    read *, x
```

```
      w = x - mean
      r = i - 1
      mean = (r*mean+x)/i
      ssq = ssq + w*w*r/i
    end do
    sd = (ssq/r)**0.5
    print *, ' mean is ', mean
    print *, ' standard deviation is ', sd
  end program ch0709
```

7.5 Summary

Arrays can have up to fifteen dimensions.

Do loops may be nested, but they must not overlap.

The `dimension` attribute allows limits to be specified for a block of information which is to be treated in a common way. The limits must be integer, and the second limit must exceed the first, e.g.,

```
real , dimension(-123:-10) :: list
real , dimension(0:100,0:100) :: surface
real , dimension(1:100) :: value
```

The last example could equally be written

```
real , dimension(100) :: value
```

where the first limit is omitted and is given the default value 1. The array `list` would contain 114 values, while `surface` would contain 10201.

A `do` statement and its corresponding `enddo` statement define a loop. The `do` statement provides a starting value, terminal value, and optionally, an increment for its index or counter.

The increment may be negative, but should never be zero. If it is not present, the default value is 1. It must be possible for the terminating value to be reached from the starting value.

The counter in a do loop is ideally suited for indexing an array, but it may be used anywhere that repetition is needed, and of course the index or counter need not be used explicitly.

The formal syntax of the block do construct is

```
[ do-construct-name : ] do [label] [ loop-control ]
  [execution-part-construct ]
[ label ] end-do
```

where the forms of the loop control are

```
[ , ] scalar-variable-name =
scalar-numeric-expression ,
scalar-numeric-expression
[ , scalar-numeric-expression ]
```

and the forms of the end-do are

```
end do [ do-construct-name ]
continue
```

and [] identify optional components of the block do construct. This statement is looked at in much greater depth in Chap. 13.

We have introduced the concept of a deferred-shape array. Arrays do not need to have their shape specified at compile time, only their rank. Their actual shape is deferred until runtime. We achieve this by the combined use of the allocatable attribute on the variable declaration and the allocate statement, which makes Fortran a very flexible language for array manipulation.

7.6 Problems

7.1 Compile and run all the examples in this chapter, except example 5. This is covered in Problem 7.8.

7.2 Modify the first example to convert the height in feet to height in metres. The conversion factor is one 1 foot equals 0.305 m.

Hint: You can either overwrite the height array or introduce a second array.

7.3 The following are two equations for temperature conversion

```
c =  5 /9 * (t-32)
f = 32 + 9 /5 * t
```

Write a complete program where t is an integer do loop variable and loop from –50 to 250. Print out the values of c, t and f on one line. What do you notice about the c and f values?

7.4 Write a program to print out the 12 times table. Typical output would be of the form:

```
1  *  12  =  12
2  *  12  =  24
3  *  12  =  36
```

etc.

Hint: You don't need to use an array here.

7.5 Write a program to read the following data into a two-dimensional array:

```
1  2  3
4  5  6
7  8  9
```

Calculate totals for each row and column and produce output similar to that shown below:

```
 1   2   3  6
 4   5   6  15
 7   8   9  24
12  15  18
```

Hint 1: Example ch0602 shows how to sum over a loop.

Hint 2: You need to introduce two one-dimensional arrays to hold the row and column totals. You need to index over the rows to get the column totals and over the columns to get the row totals.

7.6 Modify the above to produce averages for each row and column as well as the totals.

7.7 Using the following data from Problem 6.4 in Chap. 6:

```
1.85    85
1.80    76
1.85    85
1.70    90
1.75    69
1.67    83
1.55    64
1.63    57
1.79    65
1.78    76
```

Use the program that evaluated the mean and standard deviation to do so for these heights and weights.

In the first case use the program as is and run it twice, first with the heights then with the weights.

What changes would you need to make to the program to read a height and a weight in a pair?

Hint: You could introduce separate scalar variables for the heights and weights.

7.8 Example 5 looked at seat bookings in a cinema or theatre. Here is an example of a sample data file for this program

```
P P P P P P P P P P
P P P C C C C P P P
C C C E E P P P P P
C C C C C C C C C C
E E E P P P P P P P
C C E E P P C C E E
P P P P P P P P P P
P P P C C C C P P P
C C C E E P P P P P
C C C C C C C C C C
E E E P P P P P P P
C C E E P P C C E E
P P P P P P P P P P
P P P C C C C P P P
C C C E E P P P P P
```

The key for this is as follows:

```
C = Confirmed Booking
P = Provisional Booking
E = Seat Empty
```

Compile and run the program. The output would benefit from adding row and column numbers to the information displayed. We will come back to this issue in a subsequent chapter on output formatting.

The data are in a file on the web and the address is given below.

```
https://www.fortranplus.co.uk
```

Problem 6.6 in the last chapter shows how to read data from a file.