9

Arrays 1
Some Fundamentals

"Thy gifts, thy tables, are within my brain
Full charactered with lasting memory."

William Shakespeare, *The Sonnets*

"Here, take this book, and peruse it well:
The iterating of these lines brings gold."

Christopher Marlowe, *The Tragical History of Doctor Faustus*

Aims

The aims of the chapter are to introduce the fundamental concepts of arrays and DO loops, in particular:

- To introduce the idea of tables of data and some of the formal terms used to describe them:
  - Array.
  - Vector.
  - List and linear list.
- To discuss the array as a random access structure where any element can be accessed as readily as any other and to note that the data in an array are all of the same type.
- To introduce the twin concepts of data structure and corresponding control structure.
- To introduce the statements necessary in Fortran to support and manipulate these data structures.

9 Arrays 1: Some Fundamentals

9.1 Tables of data

Consider the examples below.

9.1.1 Telephone directory

A telephone directory consists of the following kinds of entries:

Name	Address	Number
Adcroft A.	61 Connaught Road, Roath, Cardiff	223309
Beale K.	14 Airedale Road, Balham	745 9870
Blunt R.U.	81 Stanlake Road, Shepherds Bush	674 4546
...		
...		
...		
Sims Tony	99 Andover Road,Twickenham	898 7330

This *structure* can be considered in a variety of ways, but perhaps the most common is to regard it as a *table* of data, where there are three columns and as many rows as there are entries in the telephone directory.

Consider now the way we extract information from this table. We would scan the name column looking for the name we are interested in, and then read along the row looking for either the address or telephone number, i.e., we are using the name to *look up* the item of interest.

9.1.2 Book catalogue

A catalogue could contain:

Author(s)	Title	Publisher
Carroll L.	Alice through the Looking Glass	Penguin
Steinbeck J.	Sweet Thursday	Penguin
Wirth N.	Algorithms plus Data Structures = Program	Prentice-Hall

Again, this can be regarded as a *table* of data, having three columns and many rows. We would follow the same procedure as with the telephone directory to extract the information. We would use the *Name* to *look up* what books are available.

9.1.3 Examination marks or results

This could consist of:

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

This can again be regarded as a *table* of data. This example has seven columns and five rows. We would again *look up* information by using the Name.

9.1.4 Monthly rainfall

The following data are the monthly average rainfall for London:

Month	Rainfall
January	3.1
February	2.0
March	2.4
April	2.1
May	2.2
June	2.2
July	1.8
August	2.2
September	2.7
October	2.9
November	3.1
December	3.1

In this table there are two columns and twelve rows. To find out what the rainfall was in July, we scan the table for July in the Month column and locate the value in the same row, i.e., the rainfall figure for July.

These are just some of the many examples of problems where the data that are being considered have a tabular structure. Most general purpose languages therefore have mechanisms for dealing with this kind of structure. Some of the special names given to these structures include:

- Linear list.
- List.
- Vector.
- Array.

The term used most often here, and in the majority of books on Fortran programming, is *array*.

9.2 Arrays in Fortran

There are three key things to consider here:

- The ability to refer to a set or group of items by a single name.
- The ability to refer to individual items or members of this set, i.e., look them up.
- The choice of a control structure that allows easy manipulation of this set or array.

9.3 The DIMENSION attribute

The DIMENSION attribute defines a variable to be an array. This satisfies the first requirement of being able to refer to a set of items by a single name. Some examples are given below:

```
REAL , DIMENSION(1:100) :: Wages
INTEGER , DIMENSION(1:10000) ::  Sample
```

For the variable Wages it is of type REAL and an array of dimension or size 100, i.e., the variable array Wages can hold up to 100 real items.

For the variable Sample it is of type INTEGER and an array of dimension or size 10,000, i.e., the variable Sample can hold up to 10,000 integer items.

9.4 An index

An index enables you to refer to or select individual elements of the array. In the telephone directory, book catalogue, exam marks table and monthly rainfall examples we used the name to index or look up the items of interest. We will give concrete Fortran code for this in the example of monthly rain fall.

9.5 Control structure

The statement that is generally used to manipulate the elements of an array is the DO statement. It is typical to have several statements controlled by the DO statement, and the block of repeated statements is often called a *DO loop*. Let us look at two complete programs that highlight the above.

9.6 Monthly rainfall

Let us look at this earlier example in more depth now. Consider the following:

Month	Associated integer representation	Array and index	Rainfall value
January	1	RainFall(1)	3.1
February	2	RainFall(2)	2.0
March	3	RainFall(3)	2.4
April	4	RainFall(4)	2.1
May	5	RainFall(5)	2.2
June	6	RainFall(6)	2.2
July	7	RainFall(7)	1.8
August	8	RainFall(8)	2.2
September	9	RainFall(9)	2.7
October	10	RainFall(10)	2.9
November	11	RainFall(11)	3.1
December	12	RainFall(12)	3.1

Most of you should be familiar with the idea of the use of an integer as an alternate way of representing a month, e.g., in a date expressed as 1/3/2000, for 1st March 2000 (anglicised style) or January 3rd (americanised style). Fortran, in common with other programming languages, only allows the use of integers as an index into an array. Thus when we write a program to use arrays we have to map between whatever construct we use in everyday life as our index (names in our examples of telephone directory, book catalogue, and exam marks) to an integer

representation in Fortran. The following is an example of an assignment statement showing the use of an index:

```
RainFall(1)=3.1
```

We saw earlier that we could use the DIMENSION attribute to indicate that a variable was an array. In the above example Fortran statement our array is called RainFall. In this statement we are assigning the value 10.4 to the first element of the array; i.e., the rainfall for the month of January is 10.4. We use the index 1 to represent the first month. Consider the following statement:

```
SummerAverage = (RainFall(6) + RainFall(7) + &
                 RainFall(8))/3
```

This statement says take the values of the rainfall for June, July and August, add them up and then divide by 3, and assign the result to the variable SummerAverage, thus providing us with the rainfall average for the three summer months — Northern Hemisphere of course.

9.6.1 Example 1: Rainfall

The following program reads in the 12 monthly values from the terminal, computes the sum and average for the year, and prints the average out.

```
PROGRAM ch0901
IMPLICIT NONE
REAL :: Total=0.0, Average=0.0
REAL , DIMENSION(1:12) :: RainFall
INTEGER :: Month
  PRINT *,' Type in the rainfall values'
  PRINT *,' one per line'
  DO Month=1,12
    READ *, RainFall(Month)
  ENDDO
  DO Month=1,12
    Total = Total + RainFall(Month)
  ENDDO
  Average = Total / 12
  PRINT *,' Average monthly rainfall was'
  PRINT *, Average
END PROGRAM ch0901
```

RainFall is the *array name*. The variable Month in brackets is the index. It takes on values from 1 to 12 inclusive, and is used to pick out or select elements of the

array. The index is thus a variable and this permits dynamic manipulation of the array at *run time*. The general form of the DO statement is

```
DO Counter = Start, End, Increment
```

The block of statements that form the loop is contained between the DO statement, which marks the beginning of the block or loop, and the ENDDO statement, which marks the end of the block or loop.

In this program, the DO loops take the form:

```
DO Month=1,12          start
   ...                       body
ENDDO                  end
```

The body of the loop in the program above has been indented. This is *not* required by Fortran. However it is good practice and will make programs easier to follow.

The number of times that the DO loop is executed is governed by the last part of the DO statement, i.e., by the

```
Counter = Start, End, Increment
```

Start as it implies, is the initial value which the counter (or index, or control variable) takes. Each time the loop is executed, the value of the counter will be increased by the value of *increment,* until the value of *end* is reached. If *increment* is omitted, it is assumed to be 1. No other element of the DO statement may be omitted. In order to execute the statements within the loop (the *body*) it must be possible to reach *end* from *start.* Thus zero is an illegal value of *increment*. In the event that it is not possible to reach *end,* the loop will not be executed and control will pass to the statement after the end of the loop.

In the example above, both loops would be executed 12 times. In both cases, the first time around the loop the variable MONTH would have the value 1, the second time around the loop the variable MONTH would have the value 2, etc., and the last time around the loop MONTH would have the value 12.

9.7 People's weights

In the table below we have ten people, with their names as shown. We associate each name with a number — in this case we have ordered the names alphabetically, and the numbers therefore reflect their ordering. WEIGHT is the *array name*. The number in brackets is called the *index* and it is used to pick out or select elements of the array. The table is read as the first element of the array WEIGHT has the value 85, the second element has the value 76*, etc.*

Person	Associated integer representation	Array and index	Associated value
Andy	1	Weight(1)	85
Barry	2	Weight(2)	76
Cathy	3	Weight(3)	85
Dawn	4	Weight(4)	90
Elaine	5	Weight(5)	69
Frank	6	Weight(6)	83
Gordon	7	Weight(7)	64
Hannah	8	Weight(8)	57
Ian	9	Weight(9)	65
Jatinda	10	Weight(10)	76

9.7.1 Example 2: Setting array size with a parameter

In the first example we so-called *hard coded* the number 12, which is the number of months, into the program. It occurred four times. Modifying the program to work with a different number of months would obviously be tedious and potentially error prone.

In this example we parameterise the size of the array and reduce the effort involved in modifying the program to work with a different number of people:

```
PROGRAM ch0902
! The program reads up to number_of_people weights
! into the array Weight
! Variables used
! Weight, holds the weight of the people
! Person, an index into the array
! Total, total weight
! Average, average weight of the people
! Parameters used
! NumberOfPeople ,10 in this case.
! The weights are written out so that
! they can be checked
!
IMPLICIT NONE
INTEGER , PARAMETER :: Number_Of_People = 10
REAL ::  Total = 0.0, Average = 0.0
```

```
INTEGER :: Person
REAL , DIMENSION(1:Number_of_People) :: Weight
  DO Person=1,Number_Of_People
    PRINT *, ' Type in the weight for person ',Person
    READ *,Weight(Person)
    Total = Total + Weight(Person)
  ENDDO
  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)
  ENDDO
END PROGRAM ch0902
```

9.8 Summary

The DIMENSION attribute declares a variable to be an array, and must come at the start of a program unit, with other *declarative* statements. It has two forms and examples of both of them are given below. In the first case we explicitly specify the upper and lower limts:

```
REAL , DIMENSION(1:Number_of_People) :: Weight
```

In the second case the lower limit defaults to 1

```
REAL , DIMENSION(Number_of_People) :: Weight
```

The latter form will be seen in legacy code, especially Fortran 77 code suites.

The PARAMETER attribute declares a variable to have a fixed value that cannot be changed during the execution of a program. In our example above note that this statement occurs before the other declarative statements that depend on it. To recap the statements covered so far, the order is summarised below.

PROGRAM	First statement	
INTEGER REAL CHARACTER	*Declarative*	In any order and the DIMENSION and PARAMETER attributes are added here
Arithmetic assignment PRINT * READ * DO ENDDO	*Executable*	In any order
END PROGRAM	Last statement	

We choose individual members using an index, and these are always of integer type in Fortran.

The DO loop is a very convenient control structure for manipulating arrays, and we use indentation to clearly identify loops.

9.9 Problems

1. Compile and run examples 1 and 2 from this chapter.

2. Using a DO loop and an array rewrite the program which calculated the average of five numbers (Question 3 in Chapter 8) and increase the number of values read in from five to ten.

3.1 Modify the program that calculates the total and average of people's weights to additionally read in their heights and calculate the total and average of their heights. Use the data given below, which have been taken from a group of first year undergraduates:

```
Height      Weight
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
```

3.2 Your body mass index is given by your weight (in kilos) divided by your height (in metres) squared. Calculate and print out the BMI for each person.

Grades of obesity according to Garrow as follows:

Grade 0 (desirable) 20–24.9

Grade 1 (overweight) 25–29.9

Grade 2 (obese) 30–40

Grad 3 (morbidly obese) >40

Ideal BMI range,

Men, Range 20.1–25 kg/m^2

Women, Range 18.7–23.8 kg/m^2

3.3 When working on either a UNIX system or a PC in a DOS box it is possible to use the following characters to enable you to read data from a file or write output to a file when running your program:

Character	**Meaning**
<	read from file
>	write to file

On a typical UNIX system we could use

```
a.out < data.dat > results.txt
```

to read the data from the file called data.dat and write the output to a file called results.txt.

On a PC in a DOS box the equivalent would be

```
program.exe < data.dat > results.txt
```

This is a quick and dirty way of developing programs that do simple I/O; we don't have to keep typing in the data and we also have a record of the behaviour of the

program. Rerun the program that prints out the BMI values to write the output to a file called results.txt. Examine this file in an editor.

4. Modify the program that read in your name to read in ten names. Use an array and a DO loop. When you have read the names into the array write them out in reverse order on separate lines.

Hint: Look at the formal syntax of the DO statement.

5. Modify the rainfall program (which assumes that the measurement is in inches) to convert the values to centimetres. One inch equals 2.54 centimetres. Print out the two sets of values as a table.

Hint: Use a second array to hold the metric measurements.

6. Combine the programs that read in and calculate the average weight with the one that reads in peoples names. The program should read the weights into one array and the names into another. Allow 20 characters for the length of a name. Print out a table linking names and weights.

7. In an earlier chapter we used the following formula to calculate the period of a pendulum:

```
T = 2 * PI * (LENGTH / 9.81) ** .5
```

Write a program that uses a DO loop to make the length go from 1 to 10 metres in 1-metre increments.

Produce a table with two columns, the first of lengths and the second of periods.

10

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.

10 Arrays 2: Further Examples

10.1 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. Consider the following example.

```
PROGRAM ch1001
!
! This program is a simple variant of ch0902.
! 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)
   ENDDO
   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)
   ENDDO
END PROGRAM ch1001
```

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 where the value of the variable Number_of_people is not known until run time, e.g.,

```
ALLOCATE(Weight(1:Number_Of_People))
```

We will look more formally at these statements in Chapter 11.

10.2 Higher-dimension arrays

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

10.2.1 A map

Consider the 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		
Latitude	1	2	3
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 C1002
! 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 :: Size = 3
INTEGER  :: Lat , Long
REAL , DIMENSION(1:Size,1:Size) :: Height
REAL , PARAMETER :: Correct = 10.0
  DO Lat = 1,Size
    DO Long = 1,Size
      PRINT *,' Type in value at ',Lat,' ',Long
      READ * , Height(Lat,Long)
    ENDDO
  ENDDO
  DO Lat = 1,Size
    DO Long = 1,Size
      Height(Lat,Long) = Height(Lat,Long) + Correct
    ENDDO
  ENDDO
  PRINT * , ' Corrected data is '
  DO Lat = 1,Size
    DO Long = 1,Size
      PRINT * , Height(Lat,Long)
    ENDDO
  ENDDO
END PROGRAM C1002
```

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.

10.2.2 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 C1003
! 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 :: Size = 3
INTEGER  :: Lat , Long
REAL , DIMENSION(1:Size,1:Size) :: Height
REAL , PARAMETER :: Correct = 10.0
  DO Lat = 1,Size
    DO Long = 1,Size
     READ * , Height(Lat,Long)
     Height(Lat,Long) = Height(Lat,Long) + Correct
    ENDDO
  ENDDO
  DO Lat = 1,Size
    PRINT * , (Height(Lat,Long),Long=1,3)
  ENDDO
END PROGRAM C1003
```

The key statement in this example is

```
PRINT * , (Height(Lat,Long),Long=1,3)
```

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.

10.2.3 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 C1004
! 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 :: Size = 3
INTEGER  :: Lat , Long
REAL , DIMENSION(1:Size,1:Size) :: H1,H2,H3,H4
  DO Lat = 1,Size
       READ * , (H1(Lat,Long), Long=1,Size)
  ENDDO
  DO Lat = 1,Size
       READ * , (H2(Lat,Long), Long=1,Size)
  ENDDO
  DO Lat = 1,Size
       READ * , (H3(Lat,Long), Long=1,Size)
  ENDDO
  DO Lat = 1,Size
    DO Long = 1,Size
      H4(Lat,Long)=( H1(Lat,Long) + H2(Lat,Long) + &
```

```
                    H3(Lat,Long) ) / Size
     ENDDO
   ENDDO
   DO Lat = 1,Size
        PRINT * , (H4(Lat,Long),Long=1,3)
   ENDDO
END PROGRAM C1004
```

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.

10.2.4 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 two-d or three-d array. Consider the following program extract:

```
PROGRAM ch1005
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)
     ENDDO
   ENDDO
   PRINT *,' Seat plan is'
   DO   Floor=1,NF
     PRINT *,' Floor = ',Floor
     DO   Row=1,NR
       PRINT *,(Seats(Row,Column,Floor),Column=1,NC)
     ENDDO
   ENDDO
END PROGRAM ch1005
```

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.

10.3 Additional forms of the DIMENSION attribute and DO loop statement

10.3.1 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 C1006
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)
  ENDDO
END PROGRAM C1006
```

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 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.

10.3.2 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 ch1007
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
    ENDDO
  ENDDO
  DO Degree=-180,180
    PRINT *,Degree,' ',Time(Degree)
  END DO
END PROGRAM ch1007
```

10.3.3 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.

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

10.4 The DO loop and straight repetition

10.4.1 Example 8: Table of temperatures

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 ch1008
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 * 0.2641925
    USGallon = Litre * 0.220022
    PRINT *,Litre, ' ',Gallon,' ',USGallon
  END DO
END PROGRAM ch1008
```

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.

10.4.2 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 ch1009
! 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
   ENDDO
   SD=(SSQ/R)**0.5
   PRINT *,' Mean is ',Mean
   PRINT *,' Standard deviation is ',SD
END PROGRAM ch1009
```

10.5 Summary

Arrays can have up to seven 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 Chapter 16.

10.6 Problems

1. Compile and run all the examples in this chapter, except example 5. This is covered separately later.

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

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

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?

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.

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 an output similar to that below:

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

Hint 1: Example ch0902 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.

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

9. Using the following data from Problem 2 in Chapter 9:

```
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.

10. 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.

- http://www.kcl.ac.uk/fortran

Problem 3.3 in the last chapter shows how to read data from a file.

11

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).
- ALLOCATABLE arrays — ways in which the size of an array can be deferred until execution time.
- 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.

11 Whole Array and Additional Array Features

11.1 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.

11.1.1 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.

11.1.2 Bounds

An array's bounds are the upper and lower limits of the index in each dimension.

11.1.3 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.

11.1.4 Size

The total number of elements in an array is its size.

11.1.5 Shape

The shape of an array is determined by its rank and its extents in each dimension.

11.1.6 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.

11.1.7 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.