12

Output of Results

"Why, sometimes I've believed as many as six impossible things before breakfast."

Lewis Carroll, *Through the Looking-Glass and What Alice Found There*

"All the persons in this book are real and none is fictitious even in part."

Flann O'Brien, *The Hard Life*

Aims

The aims here are to introduce the facilities for producing neat output and to show how to write results to a file, rather than to the terminal. In particular:

- The A, I, E, F, and X layout or edit descriptors.
- The OPEN, WRITE, and CLOSE statements.

12 Output of Results

When you have used PRINT * a few times it becomes apparent that it is not always as useful as it might be. The data are written out in a way which makes some sense, but may not be especially easy to read. Real numbers are written out with all their significant places, which is very often rather too many, and it is often difficult to line up the columns for data which are notionally tabular. It is possible to be much more precise in describing the way in which information is presented by the program. To do this, we use FORMAT statements. Through the use of the FORMAT we can:

- Specify how many columns a number should take up.
- Specify where a decimal point should lie.
- Specify where there should be white space.
- Specify titles.

The FORMAT statement has a label associated with it; through this label, the PRINT statement associates the data to be written with the form in which to write them.

12.1 Integers — I format or edit descriptor

Integer format is reasonably straightforward, and offers clues for formats used in describing other numbers. I3 is an integer taking three columns. The number is right justified, a bit of jargon meaning that it is written as far to the right as it will go, so that there are no trailing or following blanks. Consider the following example:

```
PROGRAM ch1201
INTEGER :: T
   PRINT *,' '
   PRINT *,' Twelve times table'
   PRINT *,' '
   DO T=1,12
      PRINT 100, T,T*12
      100 FORMAT(' ',I3,' *  12 = ',I3)
   END DO
END PROGRAM ch1201
```

The first statement of interest is

```
      PRINT 100, T,T*12
```

The 100 is a statement label. There must be a format statement with this label in the program. The variables to be written out are T and 12*T.

The second statement of interest is

```
100 FORMAT(' ',I3,' *  12 = ',I3)
```

Inside the brackets we have

''	Print out what occurs between the quote marks, in this case one space.
,	The comma separates items in the FORMAT statement.
I3	Print out the first variable in the PRINT statement right justified in three columns
,	Item separator.
'* 12 = '	Print out what occurs between the quote characters.
,	Item separator
I3	Print out the second variable (in this case an expression) right justified in three colums.

All of the output will appear on one line.

Now consider the following example:

```
program ch1202
implicit none
integer :: big=10
integer :: i
  do i=1,40
    print 100,i,big
    100 format(1x,i3,2x,i12)
    big=big*10
  end do
end program ch1202
```

The new feature in the format statement is the 1x and 2x edit descriptor. This is another way of getting white space into the output, and in this case one space and two spaces, respectively.

This program will loop and the variable big will overflow, i.e., go beyond the range of valid values for a 32-bit integer. Does the program crash or generate a run time error? This is the output from the NAG f95 compiler and the Intel Fortran 95 compiler.

```
 1              10
 2             100
 3            1000
 4           10000
 5          100000
 6         1000000
 7        10000000
 8       100000000
 9      1000000000
10      1410065408
11      1215752192
12      -727379968
13      1316134912
14       276447232
15     -1530494976
16      1874919424
17      1569325056
18     -1486618624
19     -1981284352
20      1661992960
21      -559939584
22     -1304428544
23      -159383552
24     -1593835520
25      1241513984
26      -469762048
27      -402653184
28       268435456
29     -1610612736
30      1073741824
31     -2147483648
32                0
33                0
34                0
35                0
36                0
37                0
38                0
39                0
40                0
```

Is there a compiler switch to trap this kind of error?

12.2 Reals — F format or edit descriptor

The F format can be seen as an extension of the integer format, but here we have to deal with the decimal point. The form of the F format specifies where the decimal point will occur, and how many digits follow it. Thus, F7.4 means:

- There is a total width of seven.
- There is a decimal point
- There are four digits after the decimal point.

This means that since the decimal point is also written out, there may be up to two digits before the decimal point. As in the case of the integer, any minus sign is part of the number, and would take up one column. Thus, the format F7.4 may be used for numbers in the range

–9.9999 to 99.9999

Let us look at the last example more closely. When a number is written out, it is rounded; that is to say, if we write out 99.99999 in an F7.4 format, the program will try to write out 100.0000! This is bad news, since we have not left enough room for all those digits before the decimal point. What happens? Asterisks will be printed. In the example above, a number out of range of the format's capabilities would be printed as:

```
*******
```

What would a format of F7.0 do? Again, seven columns have been set aside to accommodate the number and its decimal point, but this time no digits follow the point.

```
    99.
-21375.
```

are examples of numbers written in this format. With an F format, there is no way of getting rid of the decimal point.

The numbers making up the parts of the descriptors must all be positive integers. The definition of a real format is therefore F followed by two integer numbers, separated by a decimal point. The first integer must exceed the second, and the second must be greater than or equal to zero. The following are valid examples:

```
F4.0
F6.2
F12.2
F16.8
```

but these are *not* valid:

```
F4.4
F6.8
F-3.0
F6
F.2
```

The program in Section 12.2.1 illustrates the use of both I format and F format.

12.2.1 Metric and imperial conversion

```
program ch1203
implicit none
integer :: fluid
real :: litres
real :: pints
  do fluid=1,10
    litres = fluid / 1.75
    pints  = fluid * 1.75
    print 100 , pints,fluid,litres
    100 format(' ',F7.3,' ',I3,' ',F7.3)
  end do
end program ch1203
```

Pints will be printed out in F7.3 format, fluid will be printed out in I3 format and litres will be printed out in F7.3 format.

12.2.2 Overflow and underflow

Consider the following program:

```
program ch1204
implicit none
integer :: i
real     :: small = 1.0
real     :: big   = 1.0
  do i=1,50
    print 100,i,small,big
    100 format(' ',i3,' ',f7.3,' ',f7.3)
    small=small/10.0
    big=big*10.0
  end do
end program ch1204
```

In this program the variable small will underflow and big will overflow. The output from the Intel compiler is:

```
  1    1.000    1.000
  2    0.100   10.000
  3    0.010  100.000
  4    0.001  *******
  5    0.000  *******
  6    0.000  *******
  7    0.000  *******
  8    0.000  *******
  9    0.000  *******
 10    0.000  *******
 11    0.000  *******
 12    0.000  *******
 13    0.000  *******
 14    0.000  *******
 15    0.000  *******
 16    0.000  *******
 17    0.000  *******
 18    0.000  *******
 19    0.000  *******
 20    0.000  *******
 21    0.000  *******
 22    0.000  *******
 23    0.000  *******
 24    0.000  *******
 25    0.000  *******
 26    0.000  *******
 27    0.000  *******
 28    0.000  *******
 29    0.000  *******
 30    0.000  *******
 31    0.000  *******
 32    0.000  *******
 33    0.000  *******
 34    0.000  *******
 35    0.000  *******
 36    0.000  *******
 37    0.000  *******
 38    0.000  *******
 39    0.000  *******
 40    0.000   Infini
```

```
41    0.000   Infini
42    0.000   Infini
43    0.000   Infini
44    0.000   Infini
45    0.000   Infini
46    0.000   Infini
47    0.000   Infini
48    0.000   Infini
49    0.000   Infini
50    0.000   Infini
```

When the number is too small for the format, the printout is what you would probably expect. When the number is too large, you get asterisks. When the number actually overflows the Intel compiler tells you that the number is too big and has overflowed. However the program ran to completion and did not generate a run time error.

12.3 Reals — E format or edit descriptor

The exponential or scientific notation is useful in cases where we need to provide a format which may encompass a wide range of values. If likely results lie in a very wide range, we can ensure that the most significant part is given. It is possible to give a very large F format, but alternatively, the E format may be used. This takes a form such as

```
E10.4
```

which looks something like the F, and may be interpreted in a similar way. The 10 gives the total *width* of the number to be printed out, that is, the number of columns it will take. The number after the decimal point indicates the number of positions to be written after the decimal point. Since all exponent format numbers are written so that the number is between 0.1 and 0.9999..., with the exponent taking care of scale shifts, this implies that the first four significant digits are to be printed out.

Taking a concrete example, 1000 may be written as 10**3, or as 0.1 * 10**4. This gives us the two parts: 0.1 gives the significant digits (in this case only one significant digit), while the 10**4 gives the exponent, namely 4 or +4. In a form that looks more like Fortran, this would be written .1E+04, where the E+04 means 10**4.

There is a minimum *size* for an exponential format. Because of all the extra bits and pieces it requires:

- The decimal point.

- The sign of the entire number.
- The sign of the exponent.
- The magnitude of the exponent.
- The E.

The width of the number less the number of significant places should not be less than 6. In the example given above, E10.4 meets this requirement. When the exponent is in the range 0 to 99, the E will be printed as part of the number; when the exponent is greater, the E is dropped, and its place is taken by a larger value; however, the sign of the exponent is always given, whether it is positive or negative. The sign of the whole number will usually only be given when it is negative. This means that if the numbers are always positive, the *rule of six* given above can be modified to a *rule of five*. It is safer to allow six places over, since, if the format is insufficient, all you will get are asterisks.

The most common mistake with an E format is to make the edit descriptor too small, so that there is insufficient room for all the *padding* to be printed. Formats like E8.4 just don't work (on output anyway). The following four are valid E formats on output:

```
E9.3
E11.2
E18.7
E10.4
```

but the next five would not be acceptable as output formats, for a variety of reasons:

```
E11.7
E6.3
E4.0
E10
E7.3
```

12.3.1 Simple E format example

This is the same as ch1204 except that we have replaced the F formatting with E formatting:

```
program ch1205
implicit none
integer :: i
real    :: small = 1.0
real    :: big   = 1.0
```

```
   do i=1,50
     print 100,i,small,big
     100 format(' ',i3,' ',e10.4,' ',e10.4)
     small=small/10.0
     big=big*10.0
   end do
end program ch1205
```

We now have three ways to print out floating point numbers and each has its use. The PRINT * is very useful when developing programs.

12.4 Spaces

You have seen two ways of generating spaces on output. The first is to use ' characters to enclose blanks in the format statement. The second is to use the X edit descriptor. Consider the following.

```
PRINT 100, ALPHA,BETA
100 FORMAT(1X,F10.4,10X,F10.3)
```

The 10X is read rather like any of the other format elements — logically it should have been X10, to correspond to I10 or F10.4, but that would be allowing intuition to run away with you. Clearly the X3J3 committee felt it important that Fortran should have inconsistencies, just like a natural language.

Remember that these blanks are in addition to any generated as a result of the leading blanks on numbers (if any are present). If you wish to leave a single space, you must still precede the X by a number (in this case, 1); simply writing X is illegal. The general form is therefore a positive integer followed by X.

12.5 Characters — A format or edit descriptor

This is perhaps the simplest output of all. Since you will already have declared the length of a character variable in your declarations,

```
CHARACTER (10) :: B
```

when you come to write out B, the length is known — thus you need only specify that a character string is to be output:

```
PRINT 100,B
100 FORMAT(1X,A)
```

If you feel you need a little extra control, you can append an integer value to the A, like A10 (A9 or A1), and so on. If you do this, only the first 10 (9 or 1) char-

acters are written out; the remainder are ignored. Do note that 10A1 and A10 are not the same thing. 10A1 would be used to print out the first character of ten character variables, while A10 would write out the first 10 characters of a single character variable. The general form is therefore just A, but if more control is required, this may be followed by a positive integer.

The following program is a simple rewrite of a program from Chapter 7.

```
PROGRAM ch1206
!
! This program reads in and prints out
! your first name
!
IMPLICIT NONE
CHARACTER (20) :: First_name
!
   PRINT *,' Type in your first name.'
   PRINT *,' up to 20 characters'
   READ *,First_Name
   PRINT 100,First_Name
   100 FORMAT(1x,A)
!
END PROGRAM ch1206
```

12.5.1 Headings

A simple heading is given in the program below:

```
program ch1207
implicit none
integer :: fluid
real :: litres
real :: pints
   PRINT *,' Pints        Litres'
   do fluid=1,10
     litres = fluid / 1.75
     pints  = fluid * 1.75
     print 100 , pints,fluid,litres
     100 format(' ',f7.3,' ',i3,' ',f7.3)
   end do
end program ch1207
```

12.6 Mixed type output in a FORMAT statement

The following example shows how to mix and match character, integer and real output in one FORMAT statement:

```
PROGRAM ch1208
IMPLICIT NONE
CHARACTER (LEN=15) :: Firstname
INTEGER :: age
REAL :: weight
CHARACTER (LEN=1) :: sex
   PRINT *,' Type in your first name '
   READ *,Firstname
   PRINT *,' type in your age in years'
   READ *,age
   PRINT *,' type in your weight in kilos'
   READ *,weight
   PRINT *,' type in your sex (f/m)'
   READ *,sex
   PRINT *,' your personal details are'
   PRINT *
   PRINT 100
   100 FORMAT(4x,'first name', 4x , 'age' , 1x , &
       'weight' , 2x , 'sex')
   PRINT 200 , firstname, age , weight , sex
   200 FORMAT(1x , a , 2x , i3 , 2x , f5.2 , 2x, a)
END PROGRAM ch1208
```

Take care to match up the variables with the appropriate edit descriptors. You also need to count the number of characters and spaces when lining up the heading.

12.7 Common mistakes

It must be stressed that an integer can only be printed out with an I format, and a real with an F (or E) format. You cannot use integer variables or expressions with F or E edit descriptors or real variables and expressions with I edit descriptors. If you do, unpredictable results will follow. There are (at least) two other sorts of errors you might make in writing out a value. You might try to write out something which has never actually been assigned a value; this is termed an indefinite value. You might find that the letter I is written out. In passing, note that many loaders and link editors will preset all values to zero — i.e., unset (indefinite) values are actually set to zero. On better systems there is generally some way of turning this facility off, so that undefined is really indefinite. More often than not, indefinite values are the result of mistyping rather than of never setting values. It is not un-

common to type O for 0, or 1 for either I or l. The other likely error is to try to print out a value greater than the machine can calculate — *out of range* values. Some machines will print out such values as R, but some will actually print out something which looks right, and such *overflow* and *underflow* conditions can go unnoticed. Be wary.

12.8 OPEN (and CLOSE)

One of the particularly powerful features of Fortran is the way it allows you to manipulate files. Up to now, most of the discussion has centred on reading from and writing to the terminal. It is also possible to read and write to one or more files. This is achieved using the OPEN, WRITE, READ and CLOSE statements. In a later chapter we will consider *reading* from files but here we will concentrate on *writing*.

12.8.1 The OPEN statement

This statement sets up a file for either reading or writing. A typical form is

```
OPEN (UNIT=1,FILE='DATA')
```

The file will be known to the operating system as DATA (or will have DATA as the first part of its name), and can be written to by using the UNIT number. This statement should come *before* you first read from or write to the file DATA.

It is not possible to write to the file DATA directly; it must be referenced through its unit number. Within the Fortran program you write to this file using a statement such as

```
WRITE(UNIT=1,FMT=100) XVAL,YVAL
```

or

```
WRITE(1,100) XVAL,YVAL
```

These two statements are equivalent. Besides opening a file, we really ought to CLOSE it when we have finished writing to it:

```
CLOSE(UNIT=1)
```

In fact, on many systems it is not obligatory to OPEN and CLOSE all your files. Almost certainly, the terminal will not require this, since INPUT and OUTPUT units will be there by default. At the end of the job, the system will CLOSE all your files. Nevertheless, explicit OPEN and CLOSE cannot hurt, and the added clarity generally assists in understanding the program.

The following program contains all of the above statements:

```
program ch1209
implicit none
integer :: fluid
real :: litres
real :: pints
   open (unit=1,file='ch1209.txt')
   write(unit=1,fmt=200)
   200 format(' Pints        Litres')
   do fluid=1,10
      litres = fluid / 1.75
      pints  = fluid * 1.75
      write(unit=1,fmt=100) , pints,fluid,litres
      100 format(' ',f7.3,' ',i3,' ',f7.3)
   end do
   close(1)
end program ch1209
```

12.8.2 Writing

PRINT is always directed to the file OUTPUT; in the case of interactive working, this is the terminal. This is not a very flexible arrangement. WRITE allows us to direct output to any file, including *OUTPUT*. The basic form of the WRITE is

```
WRITE(6,100) X,Y,Z
```

or

```
WRITE(UNIT=6,FMT=100) X,Y,Z
```

The latter form is more explicit, but the former is probably the one most widely used. We have an example here of the use of positionally dependent parameters in the first case and equated keywords in the second. With the exceptions of the PRINT statement and the READ * form of the READ, all of the input/output statements allow the unit number and the format labels to be specified either by an equated keyword (or specifier) or in a positionally dependent form. If you use the explicit UNIT= and FMT= it does not matter what order the elements are placed in, but if you omit these keywords, the unit number must come first, followed by the format label.

UNIT=6 means that the output will be written to the file given the unit number 6. In the next chapter we will cover the way in which you may associate file names and unit numbers, but, for the moment, we will assume that the default is being used. The name of the file, as defined by the system, will depend on the particular system you use; a likely name is something like DATA06, TAPE6, or FILE0006. One *easy* way to find out (apart from asking someone) is to create such a file from

a program and then look at the names of your files after the program has finished. A great many of computing's minor complexities can be clarified by simple experimentation.

FMT=100 simply gives the label of the format to be used.

The overworked asterisk may be used, either for the unit or for the format:

UNIT=* will write to OUTPUT (the terminal)

FMT=* will produce output controlled by the list of variables, often called *list directed output.*

The following three statements are therefore equivalent:

```
WRITE(UNIT=*,FMT=*) X,Y,Z
WRITE(*,*) X,Y,Z
PRINT*,X,Y,Z
```

There are other controls possible on the WRITE, which will be elaborated later.

12.9 Repetition

Often we need to print more than one number on a line and want to use the same layout descriptor. Consider the following:

```
PRINT 100,A,B,C,D
```

If each number can be written with the same layout descriptor, we can abbreviate the FORMAT statement to take account of the pattern:

```
100 FORMAT(1X,4F8.2)
```

is equivalent to

```
100 FORMAT(1X,F8.2,F8.2,F8.2,F8.2)
```

as you might anticipate. If the pattern is more complex, we can extend this approach:

```
PRINT 100,I,A,J,B,K,C
100 FORMAT(1X,3(I3,F8.2))
```

Bracketing the description ensures that we repeat the whole entity:

```
100 FORMAT(1X,3(I3,F8.2))
```

is equivalent to

```
100 FORMAT(1X,I3,F8.2, I3,F8.2, I3,F8.2)
```

Repetition with brackets can be rather more complex. In order to give some overview of formatted Fortran output, it is helpful to delve a little into the history of the language. Many of the attributes of Fortran can be traced back to the days of single-user mainframes (with often a fraction of the power of many contemporary microcomputers and workstations). These would generally take input from punched cards (the traditional 80-column Hollerith card), and would generate output on a line printer. In this sort of environment, the individual punched card had a significance which lines in a file do not have today. Each card could be seen as a single entity — a physical record unit. The *record* was seen as an element of a subdivision within a file. Even then, there was some confusion between the notion of physical records and files split into logically distinct subunits, since these subunits might also be termed records. The present Fortran standard merely says that a record *does not necessarily correspond to a physical entity*, although *a punched card is usually considered to be a record.* This leaves us sitting at our terminals in a bemused state, especially since we may have no idea what a punched card looks like (an ideal state of affairs!).

It is important to have some notion of a record, since most of the formal definitions dealing with output (and input) are couched in terms of records. Every time an input or output statement is executed your nominal position in the file changes. If we think in terms of individual records (which may be cards), the notions of *current, preceding* and *next* record seem fairly straightforward. The current record is simply the one we have just read or written, and the other definitions follow naturally.

The situation becomes less clear when we realise that a single output statement may generate many lines of output:

```
WRITE(UNIT=6,FMT=101)   A,B,C
101  FORMAT(1X,F10.4)
```

writes out three separate lines. Looking at the output alone, there is no way to distinguish this from the output generated by

```
WRITE(UNIT=6,FMT=101)   A
WRITE(UNIT=6,FMT=101)   B
WRITE(UNIT=6,FMT=101)   C
101  FORMAT(1X,F10.4)
```

In the latter case we would probably be happy to consider each line a *record,* although in the previous example we might swither between considering all three lines (generated by a single statement) a single record or three records. Consider the first of these two examples more closely; each time the format is exhausted — that is to say, each time we run out of format description, we start again on a new

line (a new record). A new record is begun as each F10.4 is begun. The correct interpretation is therefore that three records have been written.

The same sort of thing happens in more complex FORMAT statements:

```
WRITE(UNIT=6,FMT=105) X,I,Y
105 FORMAT(1X,F8.4,I3,(F8.4))
```

would write out a single record containing a real, an integer and a real. Using the same format statement with WRITE (UNIT=6, FMT=105) X,I,Y,Z would write out two records. The first containing the values of X, I and Y and the second containing only Z. If there were still more values

```
WRITE(UNIT=6,FMT=105) X,I,Y,Z,A
```

would print out three records. The group in brackets — the (F8.4) — is repeated until we run out of items.

12.10 Some more examples

Since it is the last open bracket which determines the position at which the format is repeated, simply writing

```
WRITE(UNIT=6,FMT=100)   A,I,B,C,J
100 FORMAT(1X,F8.4,I3,F8.2)
```

would imply that A, I and B would be written on one line then, returning to the last open brackets (in this case the only open brackets), a new record (or line) is begun to write out C and J. A statement like

```
100 FORMAT(1X,(F8.4),I3,F8.2)
```

would return to the (F8.4) group, and then continue to the I3 and F8.2 before repeating again (if necessary). The same thing happens if the (F8.4) had no brackets around it. On the other hand

```
100 FORMAT(1X,(F8.4),I3,(F8.2))
```

contains superfluous brackets around the F8.4, since the repeat statement will never return to that group. Are you confused yet? This all seems very esoteric, and really, we have only hinted at the complexity which is possible. It is seldom that you have to create complex FORMAT statements, and clarity is far more important than brevity.

When patterned or repeated output is used, we may want to stop when there are no more numbers to write out. Take the following example:

```
WRITE(UNIT=1,FMT=100) A,B,C,D
100 FORMAT(1X,4(F6.1,','))
```

This will give output which looks like

```
 37.4,   29.4,   14.2,   -9.1,
```

The last comma should not be there. We can suppress these unwanted elements by using the colon:

```
100 FORMAT(1X,4(F6.1:','))
```

which would then give us

```
 37.4,   29.4,   14.2,   -9.1
```

Since we run out of data at the fourth item, D, the output following is not written out. It is a small point, but it does look a lot tidier. There are other ways of achieving the same thing.

This helps to illustrate another point, namely that you may have formats which are more extensive than the lists which reference them:

```
WRITE(UNIT=1,FMT=100) A,B,C
WRITE(UNIT=1,FMT=100) X,Y
100 FORMAT(1X,6F8.2)
```

Both WRITE statements use the format provided, although they write out different numbers of data, and neither uses up the whole format.

12.11 Implied DO loops and array sections for array output

The following program shows how to use both implied DO loops and array sections to output an array in a neat fashion:

```
program ch1210
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
integer :: r,c
   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
   do r=1,nrow
      print 100 , (exam_results(r,c),c=1,ncol) ,&
         people_average(r)
      100 format(1x,6(1x,f5.1),'  = ',f6.2)
   end do
   print *,'  ====  ====  ====  ====  ====  ====  ====='
   print 110, subject_average(1:ncol)
   110 format(1x,6(1x,f5.1))
end program ch1210
```

The print 100 uses an implied DO loop and the print 110 uses an array section.

Take care when using whole arrays. Consider the following program:

```
PROGRAM ch1211
REAL , DIMENSION(10,10) :: Y
INTEGER :: NROWS=6
INTEGER :: NCOLS=7
INTEGER :: I,J
INTEGER :: K=0

   DO I=1,NROWS
      DO J=1,NCOLS
         K=K+1
         Y(I,J)=K
      END DO
   END DO

   WRITE(UNIT=*,FMT=100)Y
```

```
  100 FORMAT(1X,10F10.4)

END PROGRAM ch1211
```

There are several points to note with this example. Firstly, this is a whole array reference, and so the entire contents of the array will be written; there is no scope for fine control. Secondly, the order in which the array elements are written is according to Fortran's array element ordering, i.e., the first subscript varying 1 to 10 (the array bound), with the second subscript as 1, then 1 to 10 with the second subscript as 2 and so on; the sequence is

```
Y(1,1)   Y(2,1)   Y(3,1)       Y(10,1)
Y(1,2)   Y(2,2)   Y(3,2)       Y(10,2)
   .
   .
Y(1,10)  Y(2,10)               Y(10,10)
```

Thirdly we have defined values for part of the array. This program behaves differently with the following compilers:

- Sun Fortran 90.
- NagWare F95.
- Compaq F95.

If you have access to more than one compiler then try out this example.

12.12 Formatting for a line printer

There is one extension to format specifications which is relevant to line printers. Fortran defines four special characters which have an effect on standard line printers when they occur in the first character position of a line. This means that a lineprinter which is not under your immediate control can be used to produce neat output by sending a file to be printed on it. This has a variety of names including, *spooling*, *queueing* and *routing* depending on the system. You should check with your local system for the exact mechanism to achieve this.

The special characters are +, 0, 1 and blank. To be used, they must be the first character of the output in each line — as if they were to be printed in column 1. In fact, a standard line printer never prints a character that occurs in column 1 at all.

Whenever a WRITE statement is begun, the printer *advances* to a new record; i.e., a new line is begun before any data are transferred. If the first character is a *special character,* then this will be interpreted by the line printer. If the first character

to be printed is a blank, the printer continues printing on that line. The first character is also known as the *carriage control character.*

The blank is a *do nothing special* control. It signifies that the line is to be printed as it is.

The zero indicates that you wish to leave an extra line; this is often useful in spacing out results to make the output more readable.

The 1 makes the output skip down to the top of the next page. This is clearly useful for separating logically distinct chunks of output. If you obtain a line printer listing of your compiled program, each segment will start at the top of a new page.

The plus is a *no advance* or *overprint* character. It suppresses the effect of the line advance which a WRITE generates. No new line is begun and the previous line is overprinted with the new. Overprinting can be useful especially when you wish to print out grey scale maps but its use is rather restricted. In particular, it can be a dangerous control character. If you have a format starting with a plus in a loop, you can make the printer overprint again and again and again . . . and again and again, until it has hammered itself into a pulp. This is not a good idea.

Similarly, accidental use of the 1 as a control character in a loop will give you lots of blank pages. It is just a bit embarrassing to be presented with a 6 inch stack of paper which is (almost) blank, because you had a 1 repeatedly in column 1.

12.12.1 Mechanics of carriage control

The following are all quite reasonable ways of generating the blank in column 1:

```
WRITE(UNIT=6,FMT=100)A
100 FORMAT(' ',F10.4)
```

or

```
WRITE(UNIT=6,FMT=100)A
100 FORMAT(1X,F10.4)
```

or

```
WRITE(UNIT=6,FMT=100)A
100 FORMAT(' THE ANSWER IS ',F10.4)
```

Note, however, that

```
WRITE(UNIT=6,FMT=100)A
100 FORMAT(F8.4)
```

could result in problems. If A contained the value 100.2934, the result on a line printer would be

```
00.2934
```

printed at the top of a new page. The 1 is taken as carriage control, and the rest of the line then printed.

Accidentally printing zeros in column 1 is a little more difficult, but

```
WRITE(UNIT=6,FMT=100)I
100 FORMAT(I1)
```

might just do it. Don't.

Remember that this only applies to line printer output, and not to the terminal. Since Fortran only defines four characters as carriage control, you will find that anything else in column 1 will give unpredictable results. On some systems, a fair number of alternatives may be defined by the installation, and they may do something useful. On other systems, they may do something, but they may also fail to print the rest of the line. This can be very perplexing. Beware.

12.12.2 Generating a new line on both line printers and terminals

There are several ways of generating new lines, other than with a 0 in column 1 of your line printer output. A more general approach, which works on both terminals and line printers, is through the oblique or slash, /. Each time this is encountered in a FORMAT statement, a new line is begun.

```
PRINT 101,A,B
101 FORMAT(1X,F10.4/1X,F10.4)
```

would give output like

```
 100.2317
-4.0021
```

This is the same as (F10.4) would have given, but clearly it opens up lots of possibilities for formatting output more tidily:

```
PRINT 102,NVAL,XMAX,XMIN
102 FORMAT(' NUMBER OF VALUES READ IN WAS: ',I10/ &
          ' MAXIMUM VALUE IS: ',F10.4/            &
          ' MINIMUM VALUE IS: ',F10.4)
```

may be easier to read than using only one line, and it is certainly more compact to write than using three separate print statements. It is not necessary to separate / by commas, although if you do nothing catastrophic will happen.

You may also begin a format description with a /, in order to generate an extra line or even generate lots of lines with lots of slashes; e.g.,

```
WRITE(UNIT=6,FMT=103)A,B
103 FORMAT(//1X,F10.4,4(/),1X,F10.4)
```

will leave two lines before printing A, and then will generate four new lines before writing B (i.e., there will be three lines between A and B — the fourth new line will contain B). While a slash by itself, or with another slash, does not have to be separated by commas from other groups, a more complex grouping, 4(/), does have to have commas and brackets to delimit it.

12.13 Timing of writing formatted files

The following example looks at the amount of time spent in different sections of a program with the main emphasis on formatted output:

```
program ch1212
implicit none
integer , parameter :: n=1000000
integer , dimension(1:n) :: x
real    , dimension(1:n) :: y
integer :: i
real :: t,t1,t2,t3,t4,t5
character*10 :: comment
  open(unit=10,file='ch1212.txt')
  call cpu_time(t)
  t1=t
  comment=' Intial '
  print 100,comment,t1
  do i=1,n
    x(i)=i
  end do
  call cpu_time(t)
  t2=t-t1
  comment = ' Integer '
  print 100,comment,t2
  y=real(x)
  call cpu_time(t)
  t3=t-t1-t2
```

```
   comment = ' real '
   print 100,comment,t2
   do i=1,n
     write(10,200) x(i)
     200 format(1x,i10)
   end do
   call cpu_time(t)
   t4=t-t1-t2-t3
   comment = ' i write '
   print 100,comment,t4
   do i=1,n
     write(10,300) y(i)
     300 format(1x,f10.0)
   end do
   call cpu_time(t)
   t5=t-t1-t2-t3-t4
   comment = ' r write '
   print 100,comment,t5
   100 format(1x,a,2x,f7.3)
end program ch1212
```

There is a call to the built-in intrinsic cpu_time to obtain timing information. Timing details for a number of compilers follow:

	Intel	**Nag**	**Salford**
Intial	0.063	0.046	0.094
Integer	0.031	0.031	0.016
real	0.031	0.031	0.016
i write	10.109	16.968	2.453
r write	12.281	101.860	3.453

Formatted output takes up a lot of time, as we are converting from an internal binary representation to an external decimal form.

12.14 Timing of writing unformatted files

The following program is a variant of the above but now the output is in unformatted or binary form:

```
program ch1213
implicit none
integer , parameter :: n=1000000
integer , dimension(1:n) :: x
real    , dimension(1:n) :: y
```

```
integer :: i
real :: t,t1,t2,t3,t4,t5
character*10 :: comment
  open(unit=10,file='ch1213.txt',form='unformatted')
  call cpu_time(t)
  t1=t
  comment=' Intial '
  print 100,comment,t1
  do i=1,n
    x(i)=i
  end do
  call cpu_time(t)
  t2=t-t1
  comment = ' Integer '
  print 100,comment,t2
  y=real(x)
  call cpu_time(t)
  t3=t-t1-t2
  comment = ' real '
  print 100,comment,t2
  write(10) x
  call cpu_time(t)
  t4=t-t1-t2-t3
  comment = ' i write '
  print 100,comment,t4
  write(10) y
  call cpu_time(t)
  t5=t-t1-t2-t3-t4
  comment = ' r write '
  print 100,comment,t5
  100 format(1x,a,2x,f7.3)
end program ch1213
```

Timing details for a number of compilers follows:

	Intel	**Nag**	**Salford**
Intial	0.063	0.062	0.172
Integer	0.031	0.030	0.031
real	0.031	0.030	0.031
i write	0.031	0.064	0.063
r write	0.031	0.062	0.063

Unformatted is very efficient in terms of time. It also has the benefit for real or floating point numbers of no information loss.

In Chapter 13 we will look at timing information reading in formatted and unformatted files.

12.15 Summary

You have been introduced in this chapter to the use of format or layout descriptors which will give you greater control over output.

The main features are:

- The I format for integer variables.
- The E and F formats for real numbers.
- The A format for characters.
- The X, which allows insertion of spaces.

Output can be directed to files as well as to the terminal through the WRITE statement.

The WRITE, together with the OPEN and CLOSE statements, also introduces the class of Fortran statements which use equated keywords, as well as positionally dependent parameters.

The FORMAT statement and its associated layout or edit descriptor are powerful and allow repetition of patterns of output (both explicitly and implicitly).

When output is to be directed to a line printer, the following four characters:

- +
- 0
- 1
- (blank)

allow reasonable control over the layout. Care must to be taken with these characters, since it is possible to decimate forests with little effort.

12.16 Problems

1. Rewrite the temperature conversion program which was Problem 8 in Chapter 10 to actually produce the output shown.

2. Write a litres and pints conversion program to produce a similar kind of output to problem one above. Start at 0 and make the central column go up to 50. One pint is 0.568 litres.

3. Information on car fuel consumption is usually given in miles per gallon in Britain and the United States and in litres per 100 kilometres in Europe. Just to add an extra problem US gallons are 0.8 imperial gallons.

Prepare a table which allows conversion from either US or imperial fuel consumption figures to the metric equivalent. Use the PARAMETER statement where appropriate:

1 imperial gallon = 4.54596 litres
1 mile = 1.60934 kilometres

4. The two most commonly used operating systems for Fortran programming are UNIX and DOS. It is possible to use the operating system file redirection symbols < and > to read from a file and write to a file, respectively. Rerun the program in problem 1 to write to a file. Examine the file using an editor.

5. Modify any of the above to write to a file rather than the terminal. What changes are required to produce a general output which will be suitable for both the terminal and a line printer? Is this degree of generality worthwhile?

6. To demonstrate your familiarity with formats, reformat problems 1, 2 or 3 to use E formats, rather than F (or vice versa).

7. Modify the temperature conversion program to produce output suitable for a line printer. Use the local operating system commands to send the file to be printed.

8. Repeat for the litres and pints program.

9. What features of Fortran reveal its evolution from punched card input?

10. Try to create a real number greater than the maximum possible on your computer — write it out. Try to repeat this for an integer. You may have to exercise some ingenuity.

11. Check what a number too large for the output format will be printed as on your local system — is it all asterisks?

12. Write a program which stores litres and corresponding pints in arrays. You should now be able to control the output of the table (excluding headings — although this could be done too) in a single WRITE or PRINT statement. If you don't like litres and pints, try some other conversion (£ sterling to US dollars, leagues to fathoms, Scots miles to Betelgeusian pfnings). The principle remains the same.

13. Fortran is an old programming language and the text formatting functionality discussed in this chapter assumes very dumb printing devices.

The primary assumption is that we are dealing with so-called monospace fonts, i.e., that digits, alphabetic characters, punctuation, etc., all have the same width.

If you are using a PC try using:

- Notepad

and

- Word

to open your programs and some of the files created in this chapter. What happens to the layout?

If you are using Notepad look at the *Word wrap* and *set Font* options under the *edit* menu.

What fonts are available? What happens to the layout when you choose another font?

If you are using Word what fonts are available? What happens when you make changes to your file and exit Word? Is it sensible to save a Fortran source file as a Word document?

13

Reading in Data

"Winnie-the-Pooh read the two notices very carefully,
first from left to right, and afterwards,
in case he had missed some of it, from right to left."

A A Milne, *Winnie-the-Pooh*

"For Madmen Only"

Hermann Hesse, *Steppenwolf*

Aims

The aims of this chapter are to introduce some of the ideas involved in reading data into a program. In particular, using the following:

- Reading from fixed fields.
- Integers, reals and characters.
- Blanks — nulls or zeros?
- READ — extensions.
    - error handling on input.
- OPEN — associating unit numbers and file names.
    - CLOSE
    - REWIND
    - BACKSPACE

13 Reading in Data

13.1 Reading from the terminal or keyboard versus reading from files

It is unlikely that you would use fixed formats when reading numeric input from the terminal or keyboard; they are more likely to be used when reading data from a file. However the examples that follow do it. We look at reading from files later in this chapter.

13.2 Fixed fields on input

All the formats described earlier are available, and again they are limited to particular types. Integers may only be input by the I format, reals with F and E, and character (alphanumeric) with A.

13.2.1 Integers and the I format

Integers are read in with the I edit descriptor. Whereas, on output, integers appear right justified, on input they may appear anywhere in the field you have delimited. Blanks (by default) are considered not to exist for the purpose of the value read, although they do contribute to the field width. Apart from the digits 0 to 9, the only other characters which may appear in an integer field are – and +.

Consider the following 12 times table:

```
 1 *  12 =  12
 2 *  12 =  24
 3 *  12 =  36
 4 *  12 =  48
 5 *  12 =  60
 6 *  12 =  72
 7 *  12 =  84
 8 *  12 =  96
 9 *  12 = 108
10 *  12 = 120
11 *  12 = 132
12 *  12 = 144
```

The following is a program to read the first and last columns of integer data:

```
program ch1301
implicit none
integer , parameter :: n=12
integer :: i
```

```
integer , dimension(1:n) :: x
integer , dimension(1:n) :: y
  do i=1,n
    read  100,x(i),y(i)
    100 format(2x,i2,9x,i3)
    print 200,x(i),y(i)
    200 format(1x,i3,2x,i3)
  end do
end program ch1301
```

The

```
    read  100,x(i),y(i)
```

will try reading values into x(i) and y(i) using format statement

```
    100 format(2x,i2,9x,i3)
```

which will skip the first two characters on the line or record, read the first value from the next two columns, skip the next nine characters and read the last value from the next three characters.

We recommend that when working with formatted files you to use a text editor that displays the column and line details

Notepad under Windows has a status bar option under the View menu. Gvim under Windows has line and column information available. Under Redhat, vim and gedit both display line and column information. User SuSe, kedit and vim display line and column information. There should be an editor available on your system that has this option.

13.2.2 Reals and the F format

Real numbers may be input using a variety of formats and we will look at the F fomat in this example. Consider the following BMI data:

```
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
```

The following program will read in the data:

```
program ch1302
implicit none
integer , parameter :: n=10
real , dimension(1:n) :: h
real , dimension(1:n) :: w
real , dimension(1:n) :: bmi
integer :: i
  do i=1,n
    read 100, h(i),w(i)
    100 format(f4.2,2x,f3.0)
  end do
  bmi=w/(h*h)
  do i=1,n
    print 200,bmi(i)
    200 format(2x,f5.0)
  end do
end program ch1302
```

To read in the heights we need a total width of four columns with two after the decimal point. We then skip two spaces and read in the weights. The data in the file do not have a decimal point!

13.2.3 Reals and the E Format

An exponential format number (which may be read in F or E formats) can take a number of different forms. The most obvious is the explicit form

```
-1.2E-4
```

where all the components of the value are present — the significant digits to the left of the E, the E itself, and the exponent to the right. We can drop (almost) any two of these three components, so:

```
-1.2
-1.2E
-1.2-4
-4
```

are all valid values. Only the first two are interpreted as the same numerical value, and just giving the exponent part would be interpreted by the format as giving only the significant digits. If the exponent is to be given, there must be some significant digits as well. It is not even enough to give the E and assume that the program will interpret this as 10 to the power *exponent*.

```
E-4
```

is not an acceptable exponential format value, although

```
1E-4
```

would be.

There are opportunities for confusion with E formats.

```
READ(UNIT=*,FMT=102) X,Y
102 FORMAT(2E10.3)
```

with:

```
10.23 -2
```

would be interpreted as X taking the value 10.23E-2 and Y taking the value 0.0, while with

```
102 FORMAT(2F8.3)
```

X would be 10.23, and Y would be –2.0.

Although the decimal point may also be dropped, this might generate confusion as well. While

```
4E3
45
45E-4
45-4
```

are all valid forms, if an E format is used, a special conversion takes place. A format like E10.8, when used with integral significant digits (no decimal point), uses the 8 as a *negative* power of 10 scaling e.g.'

```
3267E05
```

converts to

```
3267*10**8*10**5
```

or

```
3267*10**3
```

or

3.267

Therefore, the interpretation of, say, 136, read in E format, would depend on the format used:

Value	Format		Interpretation
136	E10.0		136.0
136	E10.4		136.0*10**–4
		or	0.0136
136	E10.10		136.0*10**–10
		or	0.0000000136
136.	Any above		136.0

One implication of all this is that the format you use to input a variable may not be suitable to output that same variable. So given the data:

```
136
136
136
136
136.
136.
136.
136.
```

and the program

```
program ch1303
implicit none
real     :: x
  read 100,x
  100 format(e10.0)
  print *,x
  read 200,x
  200 format(e10.4)
  print *,x
  read 300,x
  300 format(e10.10)
  print *,x
  read *,x
```

```
   print *,x
   read 100,x
   print *,x
   read 200,x
   print *,x
   read 300,x
   print *,x
   read *,x
   print *,x
end program ch1303
```

We get the following output when the program is compiled with the Intel compiler:

```
    136.0000
   1.3600000E-02
   1.3600000E-08
    136.0000
    136.0000
    136.0000
    136.0000
    136.0000
```

Other compilers may give slightly different formatting of the output.

13.3 Blanks, nulls and zeros

You can control how Fortran treats blanks in input through two special format instructions, BN and BZ. BN is a shorthand form of *blanks become null,* that is, a blank is treated as if it were not there at all. BZ is therefore *blanks become zeros.*

As we have already seen, 1 4 (i.e., the two digits separated by a blank) read in I3 format would be read as 14; similarly, 14 (one-four-blank) is also 14 when the BN format is in operation. All of the blanks are ignored for the purposes of interpreting the number. They help to create the width of the number, but otherwise contribute nothing. This is the default, which will be in operation unless you specify otherwise.

The BZ descriptor turns blanks into zeros. Thus, 1 4 (one-blank-four) read in I3 format is 104, and 14 (one-four-blank) is 140.

There is one place where we must be very careful with the use of the BZ format — when using exponent format input. Consider

```
5.321E+02
```

read in (BZ,E10.3) format. We have specified a field which is ten characters wide; therefore the blank in column 10, which follows the E+02, is read as a zero, making this E+020. This is probably not what was required.

13.4 Characters

When characters are read in, it is sufficient to use the A format, with no explicit mention of the size of the character string, since this size (or length) is determined in the program by the CHARACTER declaration. This implies that any *extra* characters would not be read in. You may however read in less:

```
CHARACTER (10) :: LIST
    .
    .
   READ(UNIT=5,FMT=100)LIST
   100 FORMAT(A1)
```

would read only the first character of the input. The remaining nine characters of LIST would be set to blank.

The notion of blanks as nulls or zeros has no meaning for characters. The blank is a legitimate character and is treated as meaningful, completely distinct from the notion of a null or a zero.

A simple variant on ch1301 which uses the character variable temp to hold the text between the two numbers appears below:

```
program ch1304
implicit none
integer , parameter :: n=12
integer :: i
integer , dimension(1:n) :: x
integer , dimension(1:n) :: y
character*9 :: temp
  do i=1,n
    read  100,x(i),temp,y(i)
    100 format(2x,i2,a,i3)
    print 200,x(i),y(i)
    200 format(1x,i3,2x,i3)
  end do
end program ch1304
```

Note that in the format statement we just use the A edit descriptor and the number of characters to read is picked up from the variable declaration.

13.5 Skipping spaces and lines

The X format is also useful for input. There may be fields in your data which you do not wish to read. These are easily omitted by the X format:

```
READ(UNIT=*,FMT=100) A,B
100 FORMAT(F10.4,10X,F8.3)
```

Similarly, you can *jump over* or ignore entire records by using the oblique. Do note, however, that

```
READ(UNIT=*,FMT=100) A,B
100 FORMAT(F10.4/F10.4)
```

would read A from one line (or record) and B from the next. To omit a record between A and B, the format would need to be

```
100 FORMAT(F10.4//F10.4)
```

Another way to skip over a record is

```
READ(UNIT=*,FMT=100)
100 FORMAT()
```

with no variable name at all.

13.6 Reading

As you have already seen, reading, or the input of information, is accomplished through the READ statement. We have used

```
READ *,X,Y
```

for list directed input from the terminal, and

```
READ(UNIT=*,FMT=100) X,Y
```

for formatted input from the terminal. These forms may be expanded to

```
READ(UNIT=*,FMT=*) X,Y
```

or

```
READ(UNIT=*,FMT=100) X,Y
```

for input from the terminal, or to

```
READ(UNIT=5,FMT=*) X,Y
```

or

```
READ(UNIT=5,FMT=100) X,Y
```

when we wish to associate the READ statement with a particular unit number (or format label, for formatted input). As with the WRITE statement, these last two READ statements may be abbreviated to

```
READ(5,*) X,Y
```

and

```
READ(5,100) X,Y
```

13.7 File manipulation again

The OPEN and CLOSE statements are also relevant to files which are used as input, and they may be used in the same ways. Besides introducing the notion of manipulating lots of files, the OPEN statement allows you to change the default for the treatment of blanks. The default is to treat blanks as null, but the statement BLANK='ZERO' changes the default to treat blanks as zeros. There are other parameters on the OPEN, which are considered elsewhere.

Once you have OPENed a file, you may not issue another OPEN for the same file until it has been CLOSEd, except in the case of the BLANK= parameter. You may change the default back again with

```
OPEN(UNIT=10,FILE='Example.dat')
READ(UNIT=10,FMT=100) A,B
...
OPEN(UNIT=10,FILE='Example.dat',BLANK='ZERO')
READ(UNIT=10,FMT=100) A,B
```

This implies that, within the same input file, you may treat some records as blank for null, and some as blank for zero. This sounds very dangerous, and is better done by manipulating individual formats if it has to be done at all.

Given that you may write a file, you may also *rewind* it, in order to get back to the beginning. The syntax is similar to the other commands:

```
REWIND(UNIT=1)
```

This often comes in useful as a way of providing backing storage, where intermediate data can be stored on file and then used later in the processing.

The notion of records in Fortran input and output has been introduced. If you are confident in your understanding of this ambiguous and nebulous concept, you can *backspace* through a file, using the statement

```
BACKSPACE(UNIT=1)
```

which moves back over a single record on the designated file. There is no point in trying to BACKSPACE or REWIND if the input is from the keyboard or terminal.

13.8 Reading using array sections

Consider the following output:

```
 50.0  47.0  70.0  89.0  30.0  46.0  =  55.33
 37.0  67.0  85.0  65.0  68.0  98.0  =  70.00
 25.0  45.0  65.0  48.0  10.0  36.0  =  38.17
 89.0  56.0  82.5  45.0  30.0  65.0  =  61.25
 68.0  78.0  95.0  76.0  98.0  65.0  =  80.00
 ====  ====  ====  ====  ====  ====
 53.8  58.6  79.5  64.6  47.2  62.0
```

A program to read this file using array sections is as follows:

```
program ch1305
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 100,(exam_results(r,1:ncol)),people_average(r)
    100 format(1x,6(1x,f5.1),4x,f6.2)
  end do
  read *
  read 110, subject_average(1:ncol)
  110 format(1x,6(1x,f5.1))
  do r=1,nrow
    print
200,(exam_results(r,c),c=1,ncol),people_average(r)
```

```
      200 format(1x,6(1x,f5.1),'   = ',f6.2)
   end do
   print *,'   ====   ====   ====   ====   ====   ====='
   print 210, subject_average(1:ncol)
   210 format(1x,6(1x,f5.1))
end program ch1305
```

Note also the use of

read *

to skip a line.

If you are on a UNIX or Linux system use diff to compare the input and output files. They should be the same.

13.9 Timing of reading formatted files

A program to read a formatted file is shown below:

```
program ch1306
implicit none
integer , parameter :: n=1000000
integer , dimension(1:n) :: x
real    , dimension(1:n) :: y
integer :: i
real :: t,t1,t2,t3,t4,t5
character*10 :: comment
   open(unit=10,file='ch1306.txt')
   call cpu_time(t)
   t1=t
   comment=' Intial '
   print 100,comment,t1
   do i=1,n
      read(10,200) x(i)
      200 format(1x,i10)
   end do
   call cpu_time(t)
   t2=t-t1
   comment = ' i read '
   print 100,comment,t2
   do i=1,n
      read(10,300) y(i)
      300 format(1x,f10.0)
   end do
```

```
   call cpu_time(t)
   t3=t-t1-t2
   comment = ' r read '
   print 100,comment,t3
   100 format(1x,a,2x,f7.3)
   do i=1,10
     print *,x(i), ' ' , y(i)
   end do
end program ch1306
```

Some timing data from the Intel compiler follows:

```
   Intial         0.063
   i read         1.922
   r read         1.828
              1      1.000000
              2      2.000000
              3      3.000000
              4      4.000000
              5      5.000000
              6      6.000000
              7      7.000000
              8      8.000000
              9      9.000000
             10      10.00000
```

13.10 Timing of reading unformatted files

The following is a program to read from an unformatted file:

```
program ch1307
implicit none
integer , parameter :: n=1000000
integer , dimension(1:n) :: x
real    , dimension(1:n) :: y
integer :: i
real :: t,t1,t2,t3,t4,t5
character*10 :: comment
   open(unit=10,file='ch1307.txt',form='unformatted')
   call cpu_time(t)
   t1=t
   comment=' Intial '
   print 100,comment,t1
   read(10) x
```

```
   call cpu_time(t)
   t2=t-t1
   comment = ' i read '
   print 100,comment,t2
   read (10) y
   call cpu_time(t)
   t3=t-t1-t2
   comment = ' r read '
   print 100,comment,t3
   100 format(1x,a,2x,f7.3)
   do i=1,10
     print *,x(i), ' ' , y(i)
   end do
end program ch1307
```

Some timing data from the Intel compiler follows.

```
   Intial         0.047
   i read         0.063
   r read         0.063
              1      1.000000
              2      2.000000
              3      3.000000
              4      4.000000
              5      5.000000
              6      6.000000
              7      7.000000
              8      8.000000
              9      9.000000
             10      10.00000
```

13.11 Errors when reading

In discussing some aspects of input, it has been pointed out that errors may be made. Where such errors are noticed, in the sense that something illegal is being attempted, there are two options:

- Print a diagnostic message, and allow correction of the mistake.
- Print a diagnostic message, and terminate the program.

The only time that the first makes sense is when you are interacting with a program at a terminal. Some Fortran implementations provide correction facilities in a case like this, but most do not.

Chapter 21 looks at how we handle errors in input data, together with a more in-depth coverage of file I/O.

13.12 Summary

Values may be read in from the keyboard, terminal or from another file through fixed formats.

Much of the structure of input format statements is very similar to that of the output formats. Broadly speaking, data written out in a particular format may be read in by the same format. However, there is greater flexibility, and quite a variety of forms can be accepted on input.

A key distinction to make is the interpretation of blanks, as either nulls or zeros; alternative interpretations can radically alter the structure of the input data.

Fortran allows file names to be associated with unit numbers through the OPEN statement. This statement allows control of the interpretation of blanks, although this can also be done through the BN and BZ formats.

Files can also be manipulated through REWIND and BACKSPACE.

13.13 Problems

1. Write a program that will read in two reals and one integer, using

```
FORMAT(F7.3,I4,F4.1)
```

and that, in one instance treats blanks as zeros and in the second treats them as nulls. Use PRINT * to print the numbers out immediately after reading them in. What do you notice? Can you think of instances where it is necessary to use one rather than the other?

2. Write a program to read in and write out a real number using

```
FORMAT(F7.2)
```

What is the largest number that you can read in and write out with this format? What is the largest negative number that you can read in and write out with this format? What is the smallest number, other than zero, that can be read in and written out?

3. Rewrite two of the earlier programs that used READ,* and PRINT,* to use FORMAT statements.

4. Write a program to read the file created by either the temperature conversion program or the litres and pints conversion program. Make sure that the programs ignore the line printer control characters and any header and title information. This

kind of problem is very common in programming (writing a program to read and possibly manipulate data created by another program).

5. Use the OPEN, REWIND, READ and WRITE statements to input a value (or values) as a character string, write this to a file, rewind the file, read in the values again, this time as real variables with blanks treated as null, and then repeat with blanks as zeros.

6. Demonstrate that input and output formats are not symmetric — i.e., what goes in does not necessarily come out.

7. Can you suggest why Fortran treats blanks as null rather than zero?

8. What happens at your terminal when you enter faulty data, inappropriate for the formats specified? We will look at how we address this problem in Chapter 21.