14

Files

"It is a capital mistake to theorise before one has data."

Sir Arthur Conan Doyle

Aims

The aims of this chapter are:

- To review the process of file creation at a terminal.
- To introduce more formally the idea of the file as a fundamental entity.
- To show how files can be declared explicitly by the OPEN and CLOSE statements.
- To introduce the arguments for the OPEN and CLOSE statements.
- To demonstrate the interaction between the READ/WRITE statements and the OPEN/CLOSE statements.

14 Files

When you work interactively on a terminal, you are working with files, files that contain programs, files that contain data, and perhaps files that are libraries. The file is fundamental to most modern operating systems, and almost all operations are carried out on files.

In this chapter we are going to extend some of your ideas about files. Let us consider what kinds of files you have met so far:

1. Text files. These are the source of your programs, compilation listings, etc. They can be examined by printing them. They can also be transmitted around a computer system fairly easily. A file sent to a printer is a text file. Mail messages are generally plain text files. Note that when mail messages arrive in your mail box they will then typically contain additional nonprintable information.

2. Data files. These exist in two main forms: firstly those prepared by using an editor, (hence a text file) and those prepared using a package or program, in a computer readable form, but not directly readable by a human.

3. Binary, object or relocatable files, e.g., output from the compiler, satellite data. They cannot be printed. To examine files like these you need to use special utilities, provided by most operating systems.

The above categories account for the majority of files that you have met so far.

If you use a word processor then you will also have met files that are textual with additional nonprintable information.

Let us now consider how we can manipulate files using Fortran. They will generally be data files, and will thus be text files. They can therefore be listed, etc., using standard operating system commands.

14.1 Data files in Fortran

These allow us to associate a logical unit number with any arbitrary file name during the running of the program; e.g.,

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

would associate the name DATA and the logical unit 1, so that

```
READ(UNIT=1,FMT=100) X
```

would read from DATA. Note that for this to work on some operating systems the file DATA must be *local* to the session; we specify the name as a character variable. If we then wanted to use a subsequent data file, we could have another OPEN

statement, but if we want to use the same logical unit number, we must first CLOSE the file

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

before we

```
OPEN(UNIT=1,FILE='DATA2')
```

In this way we can keep referring to logical unit 1, but change the file associated with it. This can be useful in interactive programs where we wish to analyse different sets of data, e.g.:

```
PROGRAM ch1401
IMPLICIT NONE
REAL :: X
CHARACTER (7) :: WHICH
  OPEN(UNIT=5,FILE='INPUT')
  DO
    WRITE(UNIT=6,FMT='('' DATA SET NAME, OR END'')')
    READ(UNIT=5,FMT='(A)') WHICH
    IF(WHICH == 'END') EXIT
    OPEN(UNIT=1,FILE=WHICH)
    READ(UNIT=1,FMT=100) X
    ...
    CLOSE(UNIT=1,FILE=WHICH)
  END DO
END PROGRAM ch1401
```

One useful feature of the OPEN statement is that there are other parameters. What would happen, for example, if the file is not there? To take care of this you can use the IOSTAT and STATUS keywords, e.g.,

```
OPEN(UNIT=1,FILE='DATA',IOSTAT=FileStat,STATUS='OLD')
```

STATUS can be equated to one of four values:

```
STATUS='OLD'
STATUS='NEW'
STATUS='SCRATCH'
STATUS='UNKNOWN'
```

If we say STATUS='NEW', we are creating a new file and it should not matter whether a file of the same name is present; 'SCRATCH' does not concern us, while 'UNKNOWN' implies that if a file of the correct name is present use it, but if not

create a 'NEW' one. If you omit the STATUS= keyword altogether, the value 'UNKNOWN' will be assumed. If we use STATUS='OLD' and the file is not present, this will cause an error which will be reflected in the value associated with the variable `Open_File_Status`. Consider the following example:

```
...
OPEN(UNIT=1,FILE='DATA',IOSTAT=FileStat,STATUS='OLD')
IF (FileStat > 0) THEN
   PRINT *,' Error opening file, please check'
   STOP
END IF
READ(UNIT=1,FMT=100) X
...
```

The program will terminate after printing an appropriate error message. The standard defines that if an error occurs then IOSTAT will return a positive integer value. A value of zero is returned if there is no error.

14.2 Summary of options on OPEN

UNIT: The unit number of the file to be opened.

IOSTAT: The I/O status specifier designates a variable to store a value indicating the status of a data transfer operation. It takes the following form:

IOSTAT=*i-var*

i-var

is a scalar integer variable. When a data transfer statement is executed, *i-var* is set to one of the following values:

- A positive integer indicating that an error condition occurred.
- A negative integer indicating that an end-of-file or end-of-record condition occurred. The actual values vary between compilers.
- Zero indicating no error, end-of-file, or end-of-record condition occurred.

Execution continues with the statement following the data transfer statement or the statement identified by a branch specifier (if any).

An end-of-file condition occurs only during execution of a sequential READ statement; an end-of-record condition occurs only during execution of a nonadvancing READ statement.

FILE: Character expression specifying the file name.

STATUS: Character expression specifying the file status. It can be one of 'OLD', *'NEW', 'SCRATCH' or 'UNKNOWN'*.

ACCESS: Character expression specifying whether the file is to be used in a sequential or random fashion. Valid values are *SEQUENTIAL* (the default) or *DIRECT*.

The two most common access mechanisms for files are sequential and direct. Consider a file with 1000 records. To get at record 789 in a sequential file means reading or processing the first 788 records. To get at record 789 in a direct access file means using a record number to immediately locate record 789.

FORM: Character expression specifying

FORMATTED if the file is opened for formatted I/O

or

UNFORMATTED if the file is opened for unformatted I/O

The default is formatted for sequential access files and unformatted for direct access files. If the file exists, FORM must be consistent with its present characteristics.

As noted earlier data are maintained internally in a binary format, not immediately comprehensible by humans. When we wish to look at the data we must write it in a formatted fashion, i.e., as a sequence of printable ASCII characters — text, or the written word. This formatting will carry with it an overhead in terms of the time required to do it. It will also carry with it the penalty of conversion from one number base (internally binary) to another and also loss of significance due to rounding with whatever edit descriptors are used, e.g., writing out as F7.4.

If we are interested in reusing data on the same system and compiler then we can use the unformatted option and avoid both the time overhead (as there is no conversion between the internal and external formats) and the loss of significance associated with formatted data.

Please note that unformatted files are rarely portable between different computer systems, and sometimes even between different compilers on the same system.

We will look again at the use of unformatted files in Chapter 28 when we deal with efficiency and the space-time trade-off.

RECL: Integer variable or constant specifying the record length for a direct access file. It is specified in characters for a formatted file and words for an unformatted file.

BLANK: Character expression having one of the following values:

'NULL' if blanks are to be ignored on reading. Note that a field of all blanks is treated as 0!

'ZERO' if blanks are to be treated as zeros.

14.3 More foolproof I/O

Fortran provides a way of writing more foolproof programs involving I/O. This is done via the IOSTAT keyword on the READ statement. Consider the following:

```
PROGRAM ch1402
IMPLICIT NONE
INTEGER :: IO_Stat_Number=-1
INTEGER :: I
  DO
    READ (UNIT=*,FMT=10,&
     IOSTAT=IO_Stat_Number) I
    10 FORMAT(I3)
    PRINT *,' iostat=',IO_Stat_Number
    PRINT *,I
    IF (IO_Stat_Number==0) EXIT
  END DO
END PROGRAM ch1402
```

The following data input should be tried and the values of IO_Stat_Number should be examined

- A valid three-digit number + [RETURN] key
- A three-digit number with an embedded blank, e.g., 1 2 + [RETURN] key
- [RETURN] key only
- [CTRL] + Z
- Any other nonnumeric character on the keyboard
- 100200300 + [RETURN] key
- [CTRL] + C

This will then enable you to write programs that handle common I/O errors.

Consider the following:

```
PROGRAM ch1403
INTEGER , DIMENSION(10) :: A =&
  (/-1,-1,-1,-1,-1,-1,-1,-1,-1,-1/)
INTEGER :: IO_Stat_Number=0
INTEGER :: I
  OPEN(UNIT=1,FILE='DATA.DAT',STATUS='OLD')
```

```
   DO I=1,10
      READ (UNIT=1,FMT=10,IOSTAT=IO_Stat_Number) A(I)
      10 FORMAT(I3)
      IF (IO_Stat_Number == 0) THEN
          CYCLE
      ELSEIF (IO_Stat_Number == -1) THEN
         PRINT *,' End of file detected at line ',I
         PRINT *,' Please check data file'
         EXIT
      ELSEIF (IO_Stat_Number  > 0 ) THEN
         PRINT *,' Non numeric data at line ',I
         PRINT *,' Please correct data file'
         EXIT
      ENDIF
   END DO
   DO I=1,10
      PRINT * , ' I = ',I,' A(I) = ',A(I)
   ENDDO
END PROGRAM ch1403
```

The above program is system specific but interestingly the following compilers return the same value for end of file. They return different values for nonnumeric data:

- NAG/Salford compiler.
- DEC Alpha OPENVMS compiler.
- NagAce Fortran 90 under Solaris.
- Sun F90 compiler (release 2.x).
- Nag F95 compiler under Solaris.
- Compaq/Dec F95, 6.01A.

What happens with a completely blank line?

Note that in the above example the testing for the various conditions only exits the DO loop for reading data from the file. This means that execution would continue with the statement immediately after the END DO statement. This may not be what we want in all cases, and the EXIT may be replaced with a STOP statement to terminate execution immediately.

14.4 Summary

The file is a fundamental entity within the operating system.

A file may be manipulated in Fortran by associating its name with a unit number. All subsequent communication within the program is through the unit number.

When a file is opened there are a large number of equatable keywords which may be employed to establish its characteristics.

The default file type used in Fortran is *sequential formatted,* but several other esoteric types may be used.

14.5 Problems

1. Write a program to write the first 500 integers to a file using formatted I/O. Put 10 values on a line, with a blank as the first character of the line, and eight columns allowed for each integer, with two spaces between integer fields.

Now write a program to read this file into an array, and write the numbers in reverse order over the original data, i.e., the data file now contains the first 500 numbers in descending order.

Now modify the first program to add the next 500 integers to the same file, so that the file now comprises the first 500 numbers in descending order, and the next 500 numbers in ascending order.

2. To write and maintain a crude database of student details, we might do the following: create separate files for each year — CLAS1, CLAS2, CLAS3, or COF84, COF85, COF86, and so on. In either case there is an unchanging prefix, CLAS or COF, and a variable suffix, which identifies membership within the overall group. In each of the files we may wish to record details like name, date of birth, address, courses taken, etc. Such files will require updating as details change or as errors are noted. Write (or sketch out) a program which would select and maintain such records and would allow corrected files to be printed out. While you might feel that the most appropriate tool for this job is an editor, you might find it too powerful a tool. An editor can leave files in a sorry state. Naturally, any program like this should be helpful (so called '*user friendly*'). Is this sort of information sensitive enough to require security checks and passwords?