FORTRAN LESSON 7

Sometimes it is convenient in a Fortran program to use *files* for accessing or storing data - especially when large amounts of data are involved. Too much keyboard input during the run of a program leads to mistakes and tedium, while too much screen output has similar consequences. Putting data into files - both for input and output - is a more leisurely and less error-prone approach.

Open

The *open* command is used to open files - that is, it makes files available so that Fortran can read or write to them. The simplest form of the command is

open (unit = *number*, file = "*name*") .

In place of *number* you insert a positive integer (but not 6) to be used to refer to the file, and instead of *name* you insert the name of the file. Here are examples of *open* commands:

```
open (unit = 2, file = "scores")
open (unit = 7, file = "a:scores.txt")
open (unit = 5, file = "h:\\results\\primes")
open (unit = 24, file = "c:\\fortran\\data\\divisors.dat")   .
```

Fortran uses the unit number to access the file with later *read* and *write* statements. Several files can be open at once, but each must have a different number. There is one thing to remember about numbering a file - you cannot use the number 6, as GNU Fortran reserves that number to refer to the screen.

Note that quotes enclose the filename. Also, in specifying a directory path for a file, you must use double backslashes instead of single ones. Do not put a space on either side of the colon after the drive letter. If you do not specify a drive or directory path for a file, or if you specify the same drive upon which GNU Fortran is installed but without a path, GNU Fortran will by default assume the file is located on the same drive and in the same directory from where Fortran is running.

If the named file does not already exist, Fortran will create it; if it does exist, Fortran will replace it. (So don't mistakenly give the file the same name as another important file!)

Close

The *close* command is used to close one or more files - examples are

close (5) , close (1, 3, 8) .

The first of these commands closes the file numbered 5, while the second closes the three files numbered 1, 3, and 8. It is not necessary to close files; all files will automatically be closed when an *end* or *stop* statement is executed. However, in programs handling large amounts of data it can be prudent to close files before the end of the program in order to avoid possible memory problems and to increase efficiency.

Write (to Files)

The *write* command is used to write data to a file. For example, the command

```
write (7,*)
```

works like a *print* * command, except that data is written to the file numbered 7 instead of to the screen. The two statements

```
print *, "The solutions to the equation are : ", x1, x2
write (7,*) "The solutions to the equation are : ", x1, x2
```

produce exactly the same output, except that the first writes to the screen and the second to file number 7. The command "write (7,*)" on a line by itself serves as a line feed, skipping a line in the file numbered 7 before the next writing to that file.

You can also use *write* statements in conjunction with format statements to write to a file; this gives you better control of formatting. In the following, the first number in "write (7,5)" refers to the file number and the second to the label of the format statement:

```
      write (7,5) "The solutions are ", x1, " and ", x2
5     format (a,f16.10,a,f16.10)
```

The "write (7,5)" command works exactly like the similar command "write (*,5)", except that in the former output is directed to file number 7, and in the latter to the screen.

Each execution of a *write* command writes to a single line in a file. The next *write* command will write to a new line.

Here is a program that finds and prints to a file the divisors of an integer n :

```
      program divisors
c     This program finds the divisors of an integer input by the user.
c     The divisors are printed to a file.
      integer n, k, d(10)
      open (unit = 1, file = "divisors")
      print *, "Enter a positive integer :"
      read *, n
      write (1,*) "Here are the divisors of ", n, " :"
      k = 0
      do i = 1, n
          if (mod(n,i) .eq. 0) then
              k = k + 1
              d(k) = i
          end if
          if (k .eq. 10) then
```

```
                write (1,5) (d(j), j = 1, 10)
                k = 0
            end if
        end do
        write (1,5) (d(j), j = 1, k)
5       format (10i7)
        close (1)
        print *, "The divisors are listed in the file 'divisors'. Bye."
        end
```

Note that the program counts the divisors, storing them in an array d, until 10 are accumulated; then it prints these 10 on a single line, reserving 7 places for each divisor. It then begins a new count and repeats the procedure until all divisors are found. The last write statement prints whatever divisors are left over after the last full line of 10. The *close* statement, included here for demonstration only, is unnecessary, as the program is all but finished at that point and the *end* statement will automatically close the file anyway.

Read (from Files)

The *read* statement is used to read data from a file. Generally data is read from a file in the standard way, line-by-line and from left to right. But you must remember that each *read* statement begins reading a new line, whether or not the preceding *read* statement used all the data in the preceding line.

Suppose for example that a file is numbered 7, and that the first two lines of the file contain the data (separated by commas)

```
1.23 , 4.56 , 7.89
11, 13 , "Sally"
```

If the first two *read* statements in the program are

```
read (7,*) x, y, z
read (7,*) m, n, first   ,
```

then the program assigns x = 1.23, y = 4.56, z = 7.89, m = 11, n = 13, first = "Sally". The variables will have to be declared in the program to correspond with the data assigned them by the read statements. For instance, in the above situation x, y, and z will have been declared real variables, m and n integers, and "first" a character variable. Failure to match variable types with data types will most likely lead to error messages.

It is possible that a program does not know beforehand the length of a file. If data is being read from a loop, there is a way to exit the loop when all data in the file has been read, thereby avoiding a program hang-up. One simply modifies the *read* statement to something like

```
read (7,*,end=10)   .
```

This command instructs Fortran to read the next line in the file numbered 7, but to jump to the statement labelled 10 in the program in the event that the last line in that file has already been read.

You can also use format specifiers in *read* statements, but this can be somewhat tedious and we will not go into the details. As but one example, suppose you want to make the assignments

```
n = 77    ,    x = 123.45    ,    y = 67.8   ,
```

where n is an integer and x and y are real variables. Then you may use the read and format statements

```
      read (7,5) n, x, y
5     format (i2,f5.2,f3.1)    ,
```

and in file number 7 place the corresponding line of data

```
7712345678    .
```

Fortran will read and separate the data, insert appropriate decimal points, and assign it to the variables. But as you can see the method is confusing and perhaps not worth the effort.