Simple File Input & Output

Handout Eight

Although we are looking at file I/O (Input/Output) rather late in this course, it is actually one of the most important features of any programming language. The ability of a program to communicate small or very large amounts of data to itself or another program at a known or unknown later date is of great importance. Fortran 90 has a wealth of commands for I/O and in this course we will only study a few of them. For example your matrix library module could be written so that it could write and read matrices to and from files saving you the trouble of having to type it in from the keyboard every time you run a code. We will look briefly at two types of files, **formatted** and **unformatted** files.

1 The '`OPEN`' statement

Before we can '`READ`' from or '`WRITE`' to a file the file must first be opened and a '`UNIT`' number assigned to it. The general form of the '`OPEN`' statement is

```
OPEN(specification_list)
```

the *specification_list* is a comma separated list of 'specifiers' that determine how the file should be opened. The list below contains the specifiers you need to know for this course but be aware that there are many more.

```
UNIT = unit_number
FILE = file_name
FORM = file_type
STATUS = file_status
ACTION = allowed_actions
```

The specifiers are explained below.

- *unit_number* : This **must** be present and is an integer. Note this 'number' identifies the file and must be unique so if you have more than one file open then you must specify a different *unit_number* for each file. Avoid using 0,5 or 6 as these `UNIT`s are typically picked to be used by Fortran as follows.
  - Standard Error = 0 : Used to print error messages to the screen.
  - Standard In = 5 : Used to read in data from the keyboard.
  - Standard Out = 6 : Used to print general output to the screen.
- *file_name* : A 'string' that holds the name of the file which we would like to open. There are situations in Fortran where you do not need to specify a file name, however this course will not require them so a file name will always be required.
- *file_type* : A string that holds the type of file that we would like to open, ie '`FORMATTED`' or '`UNFORMATTED`', both of these are explained later. If this specifier is not present then the '`FORM`' will default to '`FORMATTED`'.
- *file_status* : A string that holds the status of the file that we would like to open. Note : the default option is compiler dependent so it is recommended that you always specify one of these. We will consider three options;

  - ‘`NEW`’ : This indicates that the file should not yet exist and if it does exist the ‘`OPEN`’ statement will fail. If it does not exist then the ‘`OPEN`’ statement will create it.
  - ‘`OLD`’ : This indicates that the file should already exist and if it does not exist the ‘`OPEN`’ statement will fail.
  - ‘`REPLACE`’ : If the file already exists then it is deleted and a new file created with the same name. If the file does not exist then a new file is created.

- *allowed_actions* : If deemed necessary it is possible to specify the input/output actions that are permitted to act on the file. If no ‘`ACTION`’ specifier is present then the effect is determined by the particular Fortran implementation you are using. Generally, for most implementations, the file will be opened with ‘`READWRITE`’ as the default but as this is not guaranteed it is recommended that you always specify an action.
  - ‘`READ`’ : This states that the file is to be opened with read-only permissions. If you try to write to the file then an appropriate error message will be generated.
  - ‘`WRITE`’ : This states that the file is to be opened with write-only permissions. If you try to read information from the file then an appropriate error message will be generated.
  - ‘`READWRITE`’ : This states that the file is to be opened with both read and write permissions.

Example:

```
OPEN(UNIT=1,FILE="mat.txt",FORM="FORMATTED",STATUS="OLD",ACTION="READ")
```

This will ‘`OPEN`’ the file ‘`mat.txt`’ and assign it to ‘`UNIT`’ one. The file should already exist (‘`STATUS="OLD"`’). It will be a text file (‘`FORM="FORMATTED"`’) and is being opened with read permission only (‘`ACTION="READ"`’).

2 The ‘CLOSE’ statement

When a program has finished inputting and/or outputting from or to a file then it is prudent to close that file. This is done using the ‘`CLOSE`’ statement.

```
   CLOSE(UNIT=unit_number)
```

3 The ‘INQUIRE’ Statement

The ‘`INQUIRE`’ statement has many uses in Fortran, we will look at only one. In general the ‘`INQUIRE`’ statement can be used to investigate the various details of files. We will learn only how to use the ‘`INQUIRE`’ statement to test if a file actually exists. This is very useful, imagine if your were prompted to type in the name of a file by a Fortran code in order to ‘`READ`’ some data in from the file. If you typed in the wrong name and tried to ‘`OPEN`’ the file then the code would crash. If however you put the ‘`READ`’ filename process in a loop along with an ‘`INQUIRE`’ statement to test if the file actually exists before exiting the loop, you could allow for such a typing error. Consider the following where ‘`file_name`’ is a Fortran character string holding a file name and ‘`exist_file`’ is a variable of type ‘`LOGICAL`’.

```
INQUIRE(File=file_name, Exist=exist_file)
```

The above statement would check to see if the file ‘`file_name`’ existed. If ‘`file_name`’ does exist then the ‘`LOGICAL`’ variable ‘`exist_file`’ will be set to ‘`.TRUE.`’ else it will be set to ‘`.FALSE.`’ So if the above statement was included in a loop with the ‘`READ`’ statement to receive a filename the variable ‘`exist_file`’ could be used as a test to exit the loop. If ‘`exist_file`’ was ‘`.FALSE.`’ then an appropriate error message should be printed to the screen and another file name requested.

4 Formatted Files (text files)

Formatted files are simply text files that we can look at in an editor like nano, there is nothing unusual about them, in fact the Fortran 90 codes that you write are saved as text files. To declare a file as being formatted, the '`FORM`' specifier is set to be the string `"FORMATTED"` in the '`OPEN`' statement.

4.1 Writing text files

The '`WRITE`' command is used to write data to a file.

```
WRITE(UNIT=<unit_number>,FMT=<format_string>) <data_list>
```

- *unit_number* : Set to be the unit number of the opened file.
- *format_string* : You can specify a string of formatting characters that describe how the data is to be written to the file or use the default '`FMT=*`'. ie

  ```
  WRITE(UNIT=14, FMT="(3(F10.7,1x))") a,b,c
  ```

  ```
  WRITE(UNIT=14, FMT=*) a,b,c
  ```

  The first example would write three real numbers that occupy ten columns and have seven decimal places with one blank space separating the numbers. The second example uses the default Fortran format that is specified with '`FMT=*`'.
- *data_list* : The data that is to be written.

4.2 Reading in text files

The '`READ`' command is used to read in data from a file.

```
READ(UNIT=<unit_number>,FMT=<format_string>) <data_list>
```

- *unit_number* : Set to be the unit number of the opened file.
- *format_string* : You can specify a string of formatting characters that describe how the data is arranged in the file or use the default '`FMT=*`'. ie

  ```
  READ(UNIT=14, FMT="(3(F10.7,1x))") a,b,c
  ```

  ```
  READ(UNIT=14, FMT=*) a,b,c
  ```

  The first example would read in three real numbers that occupy ten columns and have seven decimal places with one blank space separating the numbers. The second example uses the default Fortran format specified with '`FMT=*`'.
- *data_list* : The declared data types that the data is to be read into.

The following example code is a function that reads in from a file any real matrix. The arguments to the function are the number of rows '`m`', the number of columns '`n`' and a string '`filename`' containing the name of the text file that the matrix is stored in. NOTE the text file should be in the same directory as the fortran source code.

```
FUNCTION getmatf(m,n,filename)
!**** Function to input a matrix from a text file. The number of rows
!**** (m) and the number of columns (n) are input arguments to the
!**** function along with the name of the file

  INTEGER, INTENT(IN) :: m,n               !**** Dummy declaration
  CHARACTER(LEN=*), INTENT(IN) :: filename !**** Dummy declaration

  REAL, DIMENSION(m,n) :: getmatf   !**** Local Declaration
  INTEGER :: i

  !***** Open the file passed in as the string "filename" on unit one
  < put the correct OPEN statement here >

  DO i=1,m !**** Do for each row
    < Put the correct READ statement here > !*** Read in row at a time
  ENDDO

  CLOSE(UNIT=1) !***** Close the file

END FUNCTION getmatf
```

Note that the simplest method of formatting it is to use the default fortran format 'FMT=*'. This means that we do not have to be strict about the number of decimal places or how many blank spaces separate each element from its neighbouring element.

Class Project Six (part I) : Copy the above function into your matrix library and fill in the two missing lines. To test the function type in the matrix below into a text file (exactly how you see it below) using 'nano' and read in the matrix using the above function. Print out the matrix using your 'outmat' procedure.

```
11.0    2.0    3.0     4.0
21.43   6.0    7.30    8.0
19.0    10.0   11.0   12.0
13.0    14.0   15.230 16.0
```

now write a complimentary 'SUBROUTINE outmatf(mat,filename)' to 'WRITE' a matrix to a text file. Test this by outputting the transpose of the above matrix to a file called 'mat2.txt' and then reading it back in again using the 'getmatf' function and then print it to the screen.

5 Unformatted Files (binary files)

Text files are used simply for convenience, this is because being text files they are easy for us to open in an editor or print out to paper. The computer, however, stores data in a much more efficient way. It stores information as raw binary data files. These are far more efficient as they use less disk space and are quicker to 'READ' in and 'WRITE' out. In fortran binary files are written as 'unformatted' files. We use 'unformatted' files in exactly the same way except in the 'OPEN' statement use 'FORM="UNFORMATTED"' instead of 'FORM="FORMATTED"' and there is **NO** 'FMT' specifier.

Class Project Six (Part II) : Write another function called 'getmatf_u' and a subroutine called 'outmatf_u' to write out and read in matrices using unformatted I/O. Test them with the matrix below.

```
11.0    2.0    3.0      4.0
21.43   6.0    7.30     8.0
19.0    10.0   11.0   12.0
13.0    14.0   15.230 16.0
```

NOTE : Give all your 'unformatted' files the dot extension '.dat' ie files like 'mat1.dat' and 'mat2.dat' and DO NOT try to open them in 'nano'. They are binary files not text files so would make no sense at all, but they can mess up your terminal window if you try to look at them. Also, because you are writing 'UNFORMATTED' files and will not be looking at then on screen or paper then you do **not** need to output them row by row, to the file, so can simply 'WRITE' the whole array.
