L10 – Fortran Programming - Part 2

1. Control Structures – DO Loops

Just like any computing language Fortran would be pretty useless without some control structures. Fortunately they aren't much different from our C Shell cases, and are trivial to implement. As always looping is essential; In Fortran we use a **DO Loop**. There are 3 primary ways we can do this:

DO Loop Example 1:

The most common way to do this looks as follows:

```
DO Variable = Start, End, Increment

       your code…

ENDDO
```

Here is an example code:

```
PROGRAM doloopexample
IMPLICIT NONE
INTEGER(KIND=4) :: J     ! Looping Variable

DO J=1,10

  write(*,*) "J = ", J

ENDDO

END PROGRAM doloopexample
```

All we have done is:

- Define a **loop variable** (in this case the variable **J** which is an **integer**).
- Let **J** = **1** at the start.
- Execute all commands that are in between the **DO** and **ENDDO** statements
- Increase **J** by **1** (i.e., **J** now equals **2**) and again execute all statements between the **DO** and **ENDDO** statements but this time with value of **J** being **2**.
- Keep doing this until **J** = **10**.

Note, that in Fortran (as opposed to the C Shell) we do not explicitly need to state that we let **J = J+1**. This is assumed by Fortran. But, what if we wanted **J** to increase by more than **1** at a time? Then we just need to add an increment. E.g., change the above example code to have the following line:

```
DO J=1,101,10
```

Note that the Fortran 95 standard states the inclusion of: ***DO statements using REAL variables***. However, I haven't seen this implemented in any compilers yet, and we are thus still stuck using **integers** as **loop variables**.

DO Loop Example 2:

This example is akin to the while statement in the C Shell. The basic syntax looks like:

```
DO WHILE ( Some logical expressions )

          your code…

ENDDO
```

Here is a simple example:

```
PROGRAM domore
IMPLICIT NONE
REAL(KIND=4) :: X

X = 1.0

DO WHILE (X <= 10.0)

     write(*,*) "X =", X

     X = X + 0.25

ENDDO

END PROGRAM domore
```

This example let's us keep executing the commands inside our **DO** loop until the value of **X** has increased above **10.0**.

DO Loop Example 3:

The final way that we can perform a DO loop is as follows:

```
DO

          your code…

          IF ( some logical expressions ) EXIT

ENDDO
```

This final form is very similar to that in Example #2. We could rewrite that example as:

```
PROGRAM domore
IMPLICIT NONE
REAL(KIND=4) :: X

X = 1.0

DO
     write(*,*) “X =”, X
     X = X + 0.25

     IF (X > 10.0) EXIT

ENDDO

END PROGRAM domore
```

2. Control Structures – IF THEN ELSE

If you have the C Shell scripting down, then these will look extremely familiar to you. The syntax for **IF THEN** statements in Fortran looks like:

```
!Basic form of the If statement
IF ( some logical expressions ) THEN
          your code…
ENDIF

! Or, with some other options…
IF ( some logical expressions ) THEN
     your code …
ELSEIF ( some other logical expressions ) THEN
     more code…
ELSE
     even more code…
ENDIF
```

The key here is that in Fortran we use the following operators in our logical expressions:

Operator	**Meaning**
= =	Equal to
/=	Not equal to
>=	Greater than or equal to
<=	Less than or equal to
<	Less than
>	Greater than
.AND.	Logical AND
.OR.	Logical OR
.NOT.	Logical NOT

As an example of how we use these, let's just do a simple test of an angle measured in a Cartesian space and see which quadrant it lies in:

```
PROGRAM noname
IMPLICIT NONE
REAL(KIND=8) :: Theta, x, y

!Define x,y coordinates
x = 0.5
y = 0.25

!Determine angle in degrees
Theta = (ATAN2(x,y))*(180.0/3.141592654)

! Let's check and see which quadrant theta lies in
!  based on the angle Theta
IF (  Theta >= 0.0 .AND. Theta < 90.0 ) THEN
     write(*,*) "Theta is in upper right quadrant…"

ELSEIF ( Theta >= 90.0 .AND. Theta <= 180.0) THEN
     write(*,*) "Theta is in lower right quadrant…"

ELSEIF (Theta < -90.0 .AND. Theta >= -180.0) THEN
     write(*,*) "Theta is in lower left quadrant…"

ELSEIF (Theta < 0.0 .AND. Theta >= -90.0) THEN
     write(*,*) "Theta is in upper left quadrant…"

ELSE
     write(*,*) "Error:  Theta has a nonrealistic value"

ENDIF


END PROGRAM noname
```

Similar to C Shells Fortran also provides a **CASE** statement. The syntax can be looked up on any of the Fortran related web sites.

3. Outputting Results – Writing Files

Writing variables to files in Fortran involves the following steps:

1) **OPEN** up a file to write with an associated **UNIT** number.

2) **WRITE** the data to that file.

3) **CLOSE** the file.

The following examples will show the basic procedure which varies slightly if you want to write out ASCII formatted files (normal situation) or if you want to save time and space and write out binary files.

Writing ASCII Files

In our previous examples of writing our output to the screen (standard out) we used a **write(*,*)** statement. You may have been asking what are the ***'s** for? These are essentially short cuts for the **UNIT** and **FORMAT** statements, which we will describe below.

Our first step is to **OPEN** up files: The simplest way to do this is:

```
OPEN(UNIT=1,FILE='myfile.dat')
```

Here I have associated a **UNIT Number = 1**, with the file I just created called **myfile.dat**.

I can open up more files at the same time if I want to, but then I need to use a different **UNIT Number**. E.g., to open up another file:

```
OPEN(UNIT=2,FILE='anotherfile.xyz')
```

Now, whenever we refer to either of the above two files we refer to them by their **Unit Number**. For example, now if I want to write out the variable **X** into the file **unit 1**, I can do:

```
write(UNIT=1,*) X
```

I don't actually have to type out **UNIT** every time and just say:

```
write(1,*) X
```

I could similarly write out the variable **Y** to file **unit 2**:

```
write(2,*) Y
```

Once we are done writing to our files we need to **CLOSE** off the files:

```
CLOSE(1)
CLOSE(2)
```

A full code example showing how to write an ASCII file is shown here:

```
PROGRAM asciiexample
IMPLICIT NONE
REAL(KIND=4) :: X
INTEGER(KIND=4) :: J

X = 10.0      !Initialize a variable X
```

```
!Open up a new file called test.data
OPEN(UNIT=1,FILE=’test.data’)
DO J=1,10                        !Loop 10 times

     write(1,*) J, X             !write out J and X to unit 1

     X = X/2.0

ENDDO
CLOSE(1)      !We are done writing so close off Unit 1

END PROGRAM asciiexample
```

In the above example we have the **Unit=1**, and **Format=***. In Section 5 we will discuss the Format statement further.

Writing Binary Files

Writing binary files can be accomplished with the open statement as follows:

```
OPEN(UNIT=1,FILE='filename',FORM='unformatted')
```

No format statement can be included when writing, so writing is done as follows:

```
WRITE(1) “what ever you want to write”, variables
```

Hence, writing binary files is easier than writing formatted files; however, special care must be taken when reading back in unformatted data. In particular, the exact kind type used to write out the file must be used when reading back in the data, or else you will read in pure garbage. The following code example shows how to write out data in binary format:

```
PROGRAM bin
IMPLICIT NONE
INTEGER(KIND=4) :: I, nr

OPEN(UNIT=1,FILE='bin_test',FORM='unformatted')
nr = 10

WRITE(1) nr
DO I = 1,10

 WRITE(1) I

ENDDO
CLOSE(1)

END PROGRAM bin
```

Options with the OPEN statement

Our above examples are quite simplistic, but encompass 99% of what you will want to do with writing output files. However, sometimes you may want to do something a little more advanced such as append to a file that already exists. There are some additional actions that may be done with the **OPEN** statement. For example:

To only write to a file if it doesn't already exist:

```
OPEN(UNIT=1,FILE='myfile.dat',STATUS='new',IOSTAT=ios)
```

To check and see if a file already exists and append to it:

```
OPEN(UNIT=1,FILE='myfile.dat',STATUS='old',POSITION='append')
```

Notice we have used the **IOSTAT** (Input/Output status) variable **ios**. Here we need to declare **ios** at the beginning of our code:

```
INTEGER(KIND=4) :: ios
```

This can help in error detection. Imagine the first situation, where we only want to open the file if it doesn't already exist.

Create a file called **testing.dat** and then try the next code example:

```
PROGRAM testio
IMPLICIT NONE
INTEGER(KIND=4) :: ios

OPEN(UNIT=1,FILE='testing.dat',STATUS='new',IOSTAT=ios)

write(*,*) ios

END PROGRAM testio
```

A common problem is that not all Fortran compilers return the same value for **IOSTAT** depending on whether a file exists or not. But, if you know what value your compiler returns you can then do something useful such as give the user a warning that the file isn't being opened because it already exists. But, beware, the code may not be portable to different machines.

A better option is to use the **INQUIRE** statement. This is a logical function that returns a true or false answer as to whether your file already exists.

```
LOGICAL :: file_exists
INQUIRE(FILE='testing.dat',EXIST=file_exists)
write(*,*) file_exists
```

4. Outputting Results – the Format Statement

So, far we have only specified the output format with an asterisk (*****) which doesn't provide any formatting information at all. Generally this is all you need to do. However, sometimes you may want the output to look fancy or you need it to be in a very specific format to be read in by another computer program. Fortran has a simple method to format output:

1) Somewhere in the code put a **FORMAT** statement with a reference number. This might look like:

   ```
   100 FORMAT(I3)
   ```

   Where the **100** before the **FORMAT** statement is the **reference number**. We will discuss what goes inside the **FORMAT** statement later, but suffice it for now to say that it includes all of the directions on how the output should look.

2) Use the reference number in place of the asterisk (***)** in your write statements. E.g.,

   ```
   write(UNIT=1, FMT=100)
   ```

Integer Format

The integer format (**I**) is the easiest to specify. If I say **I2** then I want to use two columns to display my integer. Similary **I4** would mean to use four columns. Try the following example:

```
PROGRAM formatexample
IMPLICIT NONE
INTEGER(KIND=4) :: J

J = 1

100 FORMAT(I2)
write(UNIT=*,FMT=100) J

101 FORMAT(I2.2)
write(UNIT=*,FMT=101) J

J = 100
write(UNIT=*,FMT=100) J

102 FORMAT(I4)
write(UNIT=*,FMT=102) J

END PROGRAM formatexample
```

F – Format for Reals

With real numbers we need to concern ourselves with the decimal point. Basically we define how many total columns we want to use, and then how many of those columns should be numbers after the decimal point. Imagine the example where we want to write out longitudes with 2 decimal points. A typical longitude may be a number like lon = -179.50. So, including the negative sign and the decimal point, we need **7** columns to display this number, and **2** columns after the decimal point. So, we specify our format as **F7.2**.

Displaying the number -179.50 with F7.2 Format:

column	**1**	**2**	**3**	**4**	**5**	**6**	**7**
value	**-**	**1**	**7**	**9**	**.**	**5**	**0**

In our code we just write:

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

A – Alphanumeric Format

We haven’t talked about character strings yet, but the easiest way to specify that we are writing out characters is:

```
100 FORMAT(A)
```

i.e., just use the letter **A**.

More complicated Output.

Of course all of the above types may be combined into a single format statement. Try the following example where we use an **X** to represent the number of spaces in between numbers:

```
PROGRAM example
IMPLICIT NONE
INTEGER(KIND=4) :: J
REAL(KIND=4) :: x, y

J = 1
x = 1000.5
y = 200.1564

write(*,100) “J, x, y = “, J, x, y

100 FORMAT(A,5X,I1,2X,F7.2,2X,F7.2)

END PROGRAM example
```

5. Homework

1) One of the easiest ways to determine how long it takes a code to run is to imbed the executable between two **date** commands in a C Shell. For example, suppose I have a code named **mycode.x** that I want to determine how long it took to run. I could make a C Shell script like:

```
#!/bin/csh

date
./mycode.x
date
```

The only problem with this is that the output might look like:

```
Tue Aug 3 02:59:27 MDT  2010
Thu Aug 5 03:55:35 MDT  2010
```

In most circumstances this is easy enough to decipher, but sometimes the code may run for days and you can not quickly determine how much time it took to run. If you are doing a lot of benchmarking you might just want to know really quick how many minutes did the code take to run.

Hence, the solution is to write a code: **difdate.f90**. This code will read in the two lines from the **date** command. [Note that reading formatted input is the same as writing formatted output, i.e., use **read(*,FMT=100)** instead of **write(*,FMT=100)**].

As output this code will report the difference in time between the two date commands. It will report the difference in two ways: (1) Total number of decimal minutes – e.g., **50.23 m**, and (2) Total days, hours, minutes, and seconds: e.g, **2 d 13 h 4 m 13.2 s**.

The code should work for any combination of dates and times, even if there are years in between the date commands output.

Hint: The easiest solution may involve converting your year, month and day into an integer value based on Julian day.