L11 – Fortran Programming - Part 3

1. Dealing with Characters

The key data type we have thus far ignored are characters. In general Fortran is not all that nice about handling characters, but does provide rudimentary tools for dealing with them. To start, we need to know how to declare characters. This is slightly different than our declarations of integers or real numbers. For example:

```
CHARACTER(LEN=1) :: A
CHARACTER(LEN=80) :: mystring
```

In the above example we have declared two variables: (1) **A** that will only store **1** character [as defined by **LEN=1**], and (2) **mystring** that will store **80** characters.

With character strings we have to define how many characters the variable can store, but we don't have to fill up all the characters. For example, I may want to use a variable **month** that stores the current month. I may want to give the variable a length of **LEN=9** so it will hold the name of the month with most number of characters (September).

Declaration of characters is done similarly as regular numbers. The following example shows how to read in a character input from the user:

```
PROGRAM readuser
IMPLICIT NONE
CHARACTER(LEN=80) :: window_function

!write a query to standard output
write(*,*) "What type of window function would you like to use?"
write(*,*) " [E.g., 'Boxcar or Blackman']"

!Now read in the users response
read(*,*) window_function

!Now write back out the user's response
write(*,*) " "
write(*,*) "You selected the: ", window_function, " function…"

END PROGRAM readuser
```

There are also **operators** and **intrinsic functions** that work on characters. The most important operator is the **concatenation operator** which is two forward slashes: //. That is, I may want to join two characters together. For example, to combine two characters into one:

```
CHARACTER(LEN=5) :: A, B
CHARACTER(LEN=10) :: C
A = 'one'
B = 'test'
C = A//B
```

Try concatenating operators in the above example. What happens?

As you may have noticed there are a number of spaces between the two characters. This may not be desirable, so there are the useful intrinsic functions **TRIM** and **ADJUSTL**. The **ADJUSTL** function shifts all the characters to be left justified, and the **TRIM** function trims off all the trailing blanks. Try the concatenation again with the following line:

```
C = TRIM(ADJUSTL(A))//TRIM(ADJUSTL(B))
```

It is often necessary to also convert integers and real numbers to characters. However, there is no intrinsic function in Fortran to do this. To do this we need to use a ***minor Fortran trick***. Let's assume we want to loop through an integer (say from 1 to 10) and append the number of the loop to a character string. The following example shows the way

```
PROGRAM aa
IMPLICIT NONE
INTEGER(KIND=4) :: J
CHARACTER(LEN=3) :: Jstr
CHARACTER(LEN=10) :: output

output = 'example_'

DO J=1,10

  ! Convert integer J into a character Jstr with 3 columns
  write(Jstr,"(I3.3)") J

  write(*,*) TRIM(ADJUSTL(output))//Jstr

ENDDO

END PROGRAM aa
```

Note that we have to write our variable **J** into the variable **Jstr**.

2. Intro to Arrays

Arrays are the last data type we will talk about in this class. Note, there are also **logical** and **complex** data types that may be important for some of your work, but **arrays** are definitely the most important.

Arrays allow us to store and manipulate many values with a single variable name. For example, we may have 4 measurements of seismic S-wave velocity and we want to store all of those measurements in the single variable called **Vs**. The following example shows us how we can hard wire these measurements into a single variable name:

```
PROGRAM arrays
IMPLICIT NONE
REAL(KIND=4), DIMENSION(4) :: Vs

Vs(1) = 7.26
Vs(2) = 3.48
Vs(3) = 2.50
Vs(4) = 2.48

write(*,*) Vs
write(*,*) Vs(3)

END PROGRAM arrays
```

The above example demonstrates the following points:

- When we declare an array variable we need to specify its **size**. In this case, we want to store **4** real numbers, so we declare the variable as being **REAL**, and that we will give it a **DIMENSION** of **4**.

- The position of each element in the array is given by a number inside parentheses. This is called the **array index**. Here, the **third element** of the variable **Vs** [denoted by **Vs(3)**] is **2.50**.

The above example shows one way to declare the elements of our array. Here is an equivalent example where we use the **array index notation** to state that we will be declaring the values of array elements 1 through 4:

```
PROGRAM arrays
IMPLICIT NONE
REAL(KIND=4), DIMENSION(4) :: Vs

Vs(1:4) = (/ 7.26, 3.48, 2.50, 2.48 /)
write(*,*) Vs(3)

END PROGRAM arrays
```

These examples have thus far been confined to vector data (i.e., one column of data) however, our arrays do not have to be confined in such a manner. We can specify an array to have several rows and columns of data. For example if I have **8-rows** of data, and **4-columns** of data:

Then I can specify this as:

```
REAL(KIND=4), DIMENSION(4,8) :: myvariable
```

In general we prescribe dimensions as **DIMENSION(number_columns, number_rows)**.

Note that this is backwards from Matrix notation (as is used in Matlab) which usually specifies array indices as (row,column).

Let's look at an example in detail here. Let's assume we have **4 rows** and **2 columns** of data. This could be depth in the Earth and Seismic wave velocity for example:

	column 1	column 2
row 1	0.0	1.45
row 2	20.0	6.80
row 3	40.0	8.10
row 4	60.0	8.08

Using our index notation we can declare the variable **Vp** for P-wave velocity in the Earth and store both our depth (first column) and velocity (2nd column).

```
PROGRAM arrays
IMPLICIT NONE
REAL(KIND=4), DIMENSION(2,4) :: Vp
INTEGER(KIND=4) :: J

! Put data values into column 1, rows 1:4
Vp(1,1:4) = (/0.0, 20.0, 40.0, 60.0/)

! Put data values into column 2, rows 1:4
Vp(2,1:4) = (/1.45, 6.80, 8.10, 8.08/)

! Write out data one row at a time
DO J=1,4
    write(*,*) Vp(:,J)
ENDDO

END PROGRAM arrays
```

3. Reading in array data

A key feature of Fortran 90 that wasn't available in Fortran 77 or earlier versions is the addition of **ALLOCATABLE** arrays. With f77 you always needed to declare the size of the array at the onset of the program. However, with the new syntax you can wait until later. All you have to do is declare the shape. Then at some later point you can decide how many elements will go into the array. The following example shows how to read data from a file into a Fortran array variable. The data from the input file may have differing numbers of lines.

```
PROGRAM aa
IMPLICIT NONE
REAL(KIND=4), DIMENSION(:), ALLOCATABLE :: mydata
INTEGER(KIND=4) :: nr, J
CHARACTER(LEN=100) :: infile

! Ask the user for some information about the data to be read in
write(*,*) "Enter the name of the data file to read..."
read(*,*) infile
write(*,*) "Enter the number of lines in the input data file..."
read(*,*) nr
```

```
! Allocate the memory required in variable mydata
ALLOCATE(mydata(nr))

! Open up the file to read
OPEN(UNIT=1,FILE=infile)

! Now read the file into variable mydata
DO J=1,nr
  read(1,*) mydata(J)
ENDDO

! We are done with the file so now close it out
CLOSE(1)

! For fun, let's write back out to standard out
DO J=1,nr
  write(*,*) mydata(J)
ENDDO

END PROGRAM aa
```

The important points to note are:

- We stated that we don't know how many elements will be in our array with the **DIMENSION(:), ALLOCATABLE** statement.
- We read how many elements to expect from the user into the integer variable **nr**.
- Once we knew how many elements to expect we reserved space in our memory with the **ALLOCATE(mydata(nr))** statement. This just reserved **nr** elements into our array **mydata**.
- Now that the memory is allocated we can read our data into the array, as is done with the **read** statement. Note, we loop through the file (after opening it) with a standard **DO** loop letting the **loop variable J** act as the **array index**.

Note that sometimes we may want to use the same variable again, but with a different number of elements. In this case we can use the **DEALLOCATE** statement to clear the memory. Then we can **ALLOCATE** the variable again with a new size.

IMPORTANT: Often times when we run one of our programs we will get an error. A common error to see is the much dreaded **Segmentation Fault**. If you see this error it is generally because you tried to write a value to an array index that doesn't exist. For example, I may have an array **mydata** that has **DIMENSION(100)**. If I try something like **mydata(101) = something**, then I will get an error because I am asking the program to place a value into a memory location that doesn't exist. OK, you have been warned!

As a final note: Check out the Fortran tricks notes that I have put together. In these notes I show how to read data into an array where you never need to specify how many lines of data exist.

4. Whole Array Operations

Now we can start to look at the beauty of storing arrays in Fortran90: **whole array operations!** Doing operations with arrays is similar to doing operations with regular variables. For example, we can add each element of two arrays just by:

```
C = A + B
```

Or, we can take the square root of each element in an array:

```
C = SQRT(B)
```

Or element by element multiplication:

```
C = A*B
```

Note, that the above operation multiplies each element of the array by the corresponding element in the other array.

E.g.,

1	2		1	2		1×1	2×2
3	4	×	3	4	=	3×3	4×4

If you want to do matrix multiplication instead you can use the **MATMUL** intrinsic function.

E.g.,

```
C = MATMUL(A,B)
```

If your arrays are composed of vectors, then there is also an intrinsic function for the dot product:

```
C = DOT_PRODUCT(Vector_1, Vector_2)
```

These kinds of features reperesent a huge improvement over f77 in which one would have to write **DO** loops and loop over all of the elements in the arrays, performing the operations element by element.

The following example shows the use of a couple of the most useful array intrinsics:

```
PROGRAM minmax
IMPLICIT NONE
REAL(KIND=4), DIMENSION(2,8) :: mydata
REAL(KIND=4), DIMENSION(2) :: maxcolumns
REAL(KIND=4) :: mydata_max, mydata_min

! Load in some data values to play with
! ------------------------------------------------------------!
! Put data values into column 1, rows 1:8
mydata(1,1:8) = (/0.0, 20.0, 40.0, 60.0, 40.0, 20.0, 0.0, 15.0/)

! Put data values into column 2, rows 1:8
mydata(2,1:8) =(/1.45, 6.80, 8.10, 8.08, 7.20, 7.00, 9.34, 2.65/)
! ------------------------------------------------------------!


!  Play around with the min/maxval functions
! ------------------------------------------------------------!
! Find the Maximum value in mydata
mydata_max = MAXVAL(mydata)
write(*,*) "Maximum Value anywhere in array:  ",  mydata_max

! Find the Minimum value in mydata
mydata_min = MINVAL(mydata)
write(*,*) "Minimum Value anywhere in array:  ",  mydata_min

! Using a MASK, find the largest value in mydata less than 10.0
mydata_max = MAXVAL(mydata,MASK=mydata < 10.0)
write(*,*) "Maximum Value < 10.0:  ",  mydata_max

! Using DIM, find the maximum values in each column
maxcolumns = MAXVAL(mydata,DIM=2)
write(*,*) "Maximum Value of column 1: ", maxcolumns(1)
write(*,*) "Maximum Value of column 2: ", maxcolumns(2)
! ------------------------------------------------------------!

END PROGRAM minmax
```

The above example demonstrates two really useful features of many of the whole array intrinsics: (1) The ability to limit the operation based on logical expressions using the **MASK** feature, and (2) The ability to perform the same operation distinctly on columns or rows of the array using the **DIM** feature. Both of these features would have previously required the use of looping but can now be done in a single line of code. Very sexy indeed.

Sometimes what you want to know is not what the maximum or minimum values are, but where they occur in an array. For example, suppose we have a seismogram and we are not concerned with what the maximum amplitude is, but at what time the maximum amplitude occurs at. Consider the following example:

```
PROGRAM minmax2
IMPLICIT NONE
REAL(KIND=4), DIMENSION(2,8) :: seismogram
REAL(KIND=4) :: time_max, amp_max
INTEGER(KIND=4), DIMENSION(2) :: time_max_index

! Make a fake very, very short seismogram
! ------------------------------------------------------------!
! Put timing information in column #1
seismogram(1,1:8) = (/0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0/)

! Put amplitude information in column #2
seismogram(2,1:8)=(/1.45,1.53,1.62,1.73,1.41,1.38,1.33,1.20/)
! ------------------------------------------------------------!

! Now let's play with the MAXLOC intrinsic
! ------------------------------------------------------------!
! Find the array indices of the max in each column
time_max_index = MAXLOC(seismogram,DIM=2)

time_max = seismogram(1,time_max_index(2))
amp_max = seismogram(2,time_max_index(2))

write(*,*) "Time to maximum value: ", time_max, " (sec)"
write(*,*) "Amplitude of maximum value: ", amp_max
! ------------------------------------------------------------!

! Now let's do it again using the MASK and say we want to find
!   the next highest amplitude
! ------------------------------------------------------------!
time_max_index=MAXLOC(seismogram,DIM=2,MASK=seismogram < amp_max)

time_max = seismogram(1,time_max_index(2))
amp_max = seismogram(2,time_max_index(2))

write(*,*) "Time to next largest value: ", time_max, " (sec)"
write(*,*) "Amplitude of next largest value: ", amp_max
! ------------------------------------------------------------!

END PROGRAM minmax2
```

The final array function I wish to discuss is the incredibly versatile **WHERE** control structure. This is similar to **IF THEN** statements only it applies to entire arrays. The basic syntax looks like:

```
WHERE ( some logical statements)
   your code…
ELSEWHERE ( more logical statements)
   your code…
ENDWHERE
```

To make it's operation clear let's look at a simple example:

```
PROGRAM wherestatement
IMPLICIT NONE
REAL(KIND=4), DIMENSION(2,8) :: seismogram
INTEGER(KIND=4) :: J

! Make a fake very, very short seismogram
! ------------------------------------------------------------!
! Put timing information in column #1
seismogram(1,1:8) = (/0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0/)

! Put amplitude information in column #2
seismogram(2,1:8) = (/1.45,1.53,1.62,1.73,1.41,1.38,1.33,1.20/)
! ------------------------------------------------------------!

! write out the initial seismogram (time, amplitude)
! ------------------------------------------------------------!
write(*,*) " "
write(*,*) "Initial seismogram"
write(*,*) "--------------------------------------"
DO J=1,8
  write(*,*) seismogram(1,J), seismogram(2,J)
ENDDO
write(*,*) "--------------------------------------"
write(*,*) " "

! Manipulate the amplitude values based on the
!  time values w/o a Loop!
! ------------------------------------------------------------!
WHERE (seismogram(1,:) >= 6.0)
  seismogram(2,:) = 0.0
ELSEWHERE (seismogram(1,:) >= 4.0)
  seismogram(2,:) = 1.0
ELSEWHERE
  seismogram(2,:) = 2.0
ENDWHERE

! write out the final seismogram (time, amplitude)
! ------------------------------------------------------------!
write(*,*) " "
write(*,*) "Final seismogram"
write(*,*) "--------------------------------------"
DO J=1,8
  write(*,*) seismogram(1,J), seismogram(2,J)
ENDDO
write(*,*) "--------------------------------------"
write(*,*) " "

END PROGRAM wherestatement
```

Note that in the above example we can base our logical statements on **subarray sections**! In this case we based our logic just on the values in the first column of data.

5. Optimization Flags

Thus far we haven't talked too much about compiling our codes. But, at this point it may be prudent to point out that there are tons of flags that we can use during compile. The compiler man page describes these and you should look around to see what is available. But, as a primary note all compilers have a standard **O**ptimization flags. If I want to compile my code and make it run a little faster I might type:

```
>> g95 mycode.f90 -O2 -o mycode.x
```

where the –O2 flag says to optimize this code to run a bit faster. You might get even better output from the –O4 flag:

```
>> g95 mycode.f90 -O4 -o mycode.x
```

But you need to be a little bit careful to make sure the results still make sense. Sometimes the compiler will perform tricks that will make the code run faster, but at the result of the numerical accuracy of the calculations.

In addition to the **–O** flags most compilers have special flags that let you optimize the code for the specific cpu your computer is using.

6. Homework

1) Write a program that will read in a 1-column data set of arbitrary size from a file, and will output the average, standard deviation, and variance of the data set. Recall the definition of standard deviation (σ) is:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

Where N is the total number of samples in the data set, and x_i is the ith data sample. Recall that the variance is just = σ^2.