The Fortran Basics

Handout Two

September 26, 2013

A Fortran program consists of a sequential list of Fortran '**statements**' and '**constructs**'. A statement can be seen a continuous line of code, like '`b=a*a*a`' or '`INTEGER :: a,b`'. A construct is a group of statements that together form a specific task and are defined by the Fortran language. You will learn some Fortran constructs later in these notes. When defining the syntax of a Fortran statement or construct, in these notes, the following will be used

- Anything inside curly brackets, { }, will be a Fortran keyword.
- Anything inside angular brackets, <>, is the choice of the programmer
- Anything inside square brackets, [ ], will be optional to the statement.

1 Program Structure

When you are writing a Fortran program you must be careful to adhere to the required structure of a Fortran program. Each program 'unit' must have this structure. A program unit can be the main program, a subroutine, function or a module. So far you have only been introduced to the main program unit, so for now do not worry about the others. The required structure is explained with the help of your 'power3.f90' code,

```
 1.   PROGRAM power3
 2.   !*** program to calculate 3rd power of 5
 3.
 4.     IMPLICIT NONE
 5.
 6.     INTEGER :: a, b
 7.
 8.     a = 5
 9.     b = a*a*a
10.
11.    PRINT*, a,b
12.
13.  END PROGRAM power3
```

1. The program unit header statement **must** come first, for example

   `PROGRAM` <program name>

   Line 1 in the 'power3' code.

2. Next follows the 'specification region', in this region you put any external references, data type declarations, interface blocks and the '`IMPLICIT NONE`' statement. Don't worry about the items listed in the above sentence that you have not met yet they will be explained later when required.

   Lines 4 $\rightarrow$ **6** in the 'power3' code.

3. Next follows the execution statements that follow the logical order of whatever you are trying to do.

   Lines 8 → 11 in the 'power3' code.

4. Finally the '`END`' statement to tell the compiler that there are no more lines to compile. For example

   `END PROGRAM` <program name>

   Line 13 in the 'power3' code.

2 Intrinsic Data Types and '`IMPLICIT NONE`'

You have already seen the keyword `INTEGER` used in the 'power3' code you wrote earlier. It was used to declare variables of integer type. Fortran has five 'intrinsic' data types. The word intrinsic means that they are defined within the Fortran language standard. The five data types are;

'`INTEGER:`' For exact whole numbers.

'`REAL:`' For real numbers.

'`COMPLEX:`' For complex numbers of the form '`x+`iy'.

'`CHARACTER:`' For characters and strings of characters.

'`LOGICAL:`' For variables that have two possible values, '`.TRUE.`' or '`.FALSE.`' .

For this course **ALL** data types you use in your code must be declared before you use them. To insure this is done **ALL** of your codes must contain the '`IMPLICIT NONE`' statement at the beginning of each program you write and as you will see later in the course it must be defined in other sections of code where necessary.

2.1 Numeric declarations (`REAL`,`INTEGER` & `COMPLEX`)

The syntax of numeric declarations are of the form;

{*type*}[,{*attribute-list*}] [::] *variable*[=<*value*>][,*variable* [=<*value*>],...]

The 'type' is of course one of the data types '`INTEGER`', '`REAL`'... The 'attribute-list' is an optional set of keywords that can alter the nature of a variable being declared, there are many attributes that can be given and they will be dealt with as we need them. So for example the following are valid examples of numeric variable declarations. NOTE how the '`::`' is identified as being optional, however if there is a 'attribute-list' OR a variable is being assigned an initial value then the '`::`' is mandatory. I recommend that you use the '`::`' in all your declarations as it does no harm and improves the readability of your code.

```
REAL :: angle
INTEGER :: i,j,start=1
INTEGER :: a,b,c
COMPLEX :: plane
INTEGER, PARAMETER :: dozen=12
REAL, PARAMETER :: pi=3.1415927
```

NOTE: The last two declarations, in the above list, are an example of a variable being defined with an attribute. The '`PARAMETER`' attribute dictates that a **constant** is being defined. This means you can never change its value, later in your code, from the one it is given during its declaration. If you do try

to assign a value to a data type declared with the attribute '`PARAMETER`' the compiler will flag an error causing compilation to fail. This is useful for any of your own constants and also common constants for example π.

2.2 Numeric Operators

All the standard numeric operators exist in Fortran. 'Operators' act on 'operands'. Some have one operand and are termed **monadic** and some have two and are termed **dyadic**. Here is a table of the common Fortran numeric operators, they are numbered in order of precedence [1] being highest and at the top.

```
[1]   **    : Exponentiation operator, dyadic
[2]   /     : Division operator, dyadic
[2]   *     : Multiplicative operator, dyadic
[3]   -     : minus operator monadic for negation
[4]   +     : additive operator, dyadic
[4]   -     : minus operator, dyadic
```

If you are building a complicated expression you can use parentheses to change the default order that sub-expressions are evaluated making sure that they are being evaluated in the order you require.

Example

Consider the following expression, `8.0/4.0*3.0=6.0` : There is some confusion here as '`*`' and '`/`' have the same order of precedence, so which will Fortran evaluate first? When this occurs Fortran will operate a 'left to right' rule so the left most expression in the conflict will be evaluated first. So `8.0/4.0` is calculated and then the result is multiplied by `3.0`. To have Fortran calculate first `4.0*3.0` and divide the result into `8.0` you could make use of parenthesis and write `8.0/(4.0*3.0)=0.666` . This left to right rule however does not apply to exponentiation. For example a**b**c, in this case a right to left rule is used to dictate the order of evaluation.

2.3 Character declarations (`CHARACTER`)

The syntax of character declarations are of the form;

`CHARACTER` [(`LEN=`<*length*>)][,{*attribute-list*}] [::] *variable*[=<*value*>][,*variable*[=<*value*>],...]

This is similar to the numerical declaration, however, there is now an option '`(LEN=`<*length*>)' specifier. This length specifier allows the declaration of a character variable of more than one character. Consider the following character declarations.

```
CHARACTER :: char1,char2,letter="I"
CHARACTER, PARAMETER :: a="a",b="b",c="c"
CHARACTER (LEN=6) :: class1="MT3501", class2="MT3502",class3="MT3503"
CHARACTER (LEN=*), PARAMETER :: news1="Man United will win the league!"
CHARACTER (LEN=*), PARAMETER :: news2="Scotland will win the next world cup!"
```

NOTE: The last declaration has an '`*`' as its length specifier. This is because '`PARAMETER`' (**only**) declarations can have their length assumed by the compiler, so you do not have to state it for this case.

2.4 Logical declarations (LOGICAL)

The syntax of numeric declarations are of the form;

`LOGICAL`[,{*attribute-list*}] [::] *variable*[=<*value*>][,*variable* [=<*value*>],...]

Some examples of logical type declarations.

```
LOGICAL :: switch, gate
LOGICAL :: ans1=.TRUE. ans2=.FALSE.
```

NOTE: The logical type can have only two values, '`.TRUE.`' or '`.FALSE.`'.

2.5 Valid variable names

Notice that 'power3.f90' uses the variable names `a` and `b`. This raises the question, what are valid variable names? A Fortran 90 name can have up to 31 characters. This can be any combination of letters (upper case or lower case), the ten digits 0-9 and the underscore character '_'; the first character **must** be a letter. Unlike Linux, which is case sensitive, Fortran does not differentiate between upper and lower case characters. However, I recommend that you always use the convention that variables you define are always lower case while Fortran key words (such as 'PROGRAM' and 'REAL') are in upper case. This makes it easier for people, including you, to read your programs.

3 Programing Style

- Always comment your code so you can understand it later and so others can make sense of it if they have to read it (including those who have to mark your work). Remember a comment is any text on a line following a '`!`'. Remember that comments are ignored by the compiler.
- Any Fortran 90 keywords and intrinsic functions such as `SIN, COS` and `TAN` should be in 'UPPER CASE' letters and user created entities such as variable names should be in lower case.
- 'Indentation', this is essential for program readability. An indentation of two spaces should be applied to program unit bodies and any statements inside fortran constructs should be indented a further two spaces. This will become clearer as you see example codes in the notes.
- **Always** use '`IMPLICIT NONE`'
- You should always choose the names of your 'data types' (`REAL, INTEGER` etc.) so that they reflect their purpose in the code. For example you could call a constant π in your code '`pi`'.
- You need to write your codes so they are efficient. That is they use a little of the computers resources as possible. Computer memory (RAM), CPU clock cycles and disk space) are all system resources to be conserved.

4 Decision Making :: The '`IF`' Statement and Construct

Decision making is an essential and powerful component of programing.

4.1 The If Statement

One of the most common and simplest form of decision commands in Fortran is the `IF` statement. It takes the general form,

IF (<*logical expression*>) <*statement*>

A logical expression **always** reduces to either '`.TRUE.`' or '`.FALSE.`'. It is rather useless but the most basic form of a logical expression would be simply the two logical constants '`.TRUE.`' and '`.FALSE.`', for example

```
IF (.TRUE.) PRINT*,"Hello From Me"
```

This statement will always of course print 'Hello From Me' because '`.TRUE.`' can only ever reduce to itself '`.TRUE.`' . *

4.2 Relational Operators, Logical Operators & Logical Expressions

Relational Operators

Relational operators are used to test the relationship between two numeric operands for that specific operator. The relational operators are,

```
<  or .LT. : less than
>  or .GT. : greater than
== or .EQ. : identically equal to
/= or .NE. : not equal to
<= or .LE. : less than or equal to
>= or .GE. : greater than or equal to
```

as mentioned above any expression involving these operators will reduce down to either '`.TRUE.`' or '`.FALSE.`'.

Logical Operators

Logical operators act on logical expressions to create a larger logical expression. The Fortran logical operators are listed in the table below and are in order of precedence, given in square brackets, from the top to the bottom.

```
[1] .NOT.   : Negate the following operand
[2] .AND.   : .TRUE. if both operands are .TRUE.
[3] .OR.    : .TRUE. if at least one operand is .TRUE.
[4] .EQV.   : .TRUE. if both operands reduce to the same
[4] .NEQV.  : .TRUE. if both operands are different
```

The 'order of precedence' is important when dealing with logical expressions, however you can change

the order in which sub-expressions are evaluated by making use of parenthesis. Consider,

---

Exercise One (I): Work out the values of `ans1, ans2` and `ans3`. Check them by writing a small code and printing them out to screen using the fortran '`PRINT`' command. Create a '`handout2`' directory and in it create a '`exercise1`' directory. Write your code in the '`exercise1`' directory and call it '`logical_test.f90`'. Remember to compile use the command '`f90 -o logical_test logical_test.f90 `' .

```
LOGICAL :: tst1=.TRUE., tst2=.TRUE., tst3=.FALSE.
LOGICAL :: ans1,ans2,ans3

ans1=tst1 .OR. tst2 .AND. tst3
ans2=(tst1 .OR. tst2) .AND. tst3
ans3=tst1 .OR. (tst2 .AND. tst3)

PRINT*, ans1,ans2,ans3
```

NOTE : In the '`ans1`' expression the '`tst2 .AND. tst3`' sub-expression is evaluated first!

Now try the following exercise that uses logical expressions that involve both relational operators and logical operators.

Exercise One (II): Work out the values of `test1, test2` and `test3`. Check them by writing a small code and printing them out to screen. Do this in the '`exercise1`' directory but write the code in a new file called '`logical_test2.f90`' .

```
REAL     :: height=1.85, weight=95.0
INTEGER  :: age=55
LOGICAL  :: drinker=.TRUE., employed=.TRUE., smoker=.TRUE.
LOGICAL  :: test1,test2,test3

If (age>50) Print*,"Over fifty"
test1 = employed .AND. (age<45)
test2 = smoker .AND. drinker .AND. (height<2.00)
test3 = (.NOT. smoker .OR. (age<=55) .AND. (weight>50)) .AND. (height<=1.84)

PRINT*,test1,test2,test3
```

4.3 The If Construct

The 'IF' construct is more flexible than the single 'IF' statement. The 'IF' construct can be seen as having three forms

Form One

The IF' construct in its most simple form is

```
IF (<condition>) THEN
  < statement 1>
  < statement 2>
  <    etc     >
ENDIF
```

This form simply performs a single test and if the test result is **true** then a series (or a single) Fortran statements that lie in the 'body' of the construct are executed.

Form Two

```
IF (<condition>) THEN
  <statements if condition is true>
ELSE
  <statements if condition is false>
ENDIF
```

This second form of the construct allows the program to take action on a series of Fortran statements if the the condition is **false** as well as if it is **true**. This is done by the addition of the 'ELSE' keyword.

Form Three

```
IF (<condition1>) THEN
  <statements if condition1 is true>
ELSEIF (<condition2>) THEN
  <statements if condition2 is true>
ELSEIF (<condition3>) THEN
  <statements if condition3 is true>
ELSEIF (<condition4>) THEN
  <statements if condition4 is true>
ELSE
  <statements all conditions are false>
ENDIF
```

The final form of the 'IF' construct allows the programmer to produce a 'mutually exclusive' list of options. NOTE there can be any number of 'ELSEIF' branches in the construct. Also NOTE that the

'ELSE' is optional.

Exercise Two: Create an 'exercise2' directory and in it write a program to calculate a the roots of a quadratic equation $y = ax^2 + bx + c$. Make use of the 'IF' construct to categorise the quadratic into two real roots, one repeated real root, complex roots or no roots at all (ie. it is not a valid quadratic). To categorise the quadratic you will have to calculate the value of the discriminent 'b**2-4*a*c'. Output the root(s) to the screen. Also calculate where the the quadratic has a turning point and the nature of this turning point and print these out to the screen. This is your first, in this course, real attempt at a meaningful Fortran code so it may seem a little daunting but spend some time thinking about how you are going to approach the problem. It can help to write down your ideas on paper first! Remember the roots of a quadratic are given by,

$$\frac{-b+\sqrt{b^2-4ac}}{2a} \quad \& \quad \frac{-b-\sqrt{b^2-4ac}}{2a}.$$

NOTE : For this exercise do NOT use the data type 'COMPLEX' . You can solve the quadratic using only the 'REAL' data type for all three cases of roots.
As this is your first real attempt at a piece of code much of the code is given below. You are advised to make use of this and fill in the bits of missing code indicated by '****'.

```
PROGRAM quad
!## Program to investigate a quadratic equation (y=a*x*x+b*x+c)

  IMPLICIT NONE      !## Force explicit declaration of all variables

  !## Declare required variables
  ****
  ****
  IF (check) THEN  ! check must be .TRUE. if "a" does not equal zero

  !*** Calculate  the roots of the quadratic. Note there can be
  !*** three separate cases depending on the discriminent.
    ****
    ****
  ELSE !*** a==0 so not a quadratic
    PRINT*,"This is not a valid quadratic"
  ENDIF

  IF (check) THEN
    !## Calculate the quadratics turning point
    ****
    ****
    IF (a .LT. 0)  THEN
      PRINT*,"Turning point is a maximum"
    ELSE
      PRINT*,"Turning point is a minimum"
    ENDIF
  ENDIF
```

---

5 Looping :: The 'DO' Construct

Computers are very powerful when dealing with repetitive tasks. To allow the programmer to implement such a feature Fortran includes a construct called the 'DO' loop. It has a few different forms;

5.1 The 'Indexed DO' Construct

One form of looping in Fortran is the 'Indexed DO' construct which has the form.

```
DO <loop variable>=<expr1>,<expr2>[,<expr3>]
  <body statements>
END DO
```

The loop 'variable' runs from 'expression one' in increments of 'expression three' until it reaches the value 'expression two' and then the loop terminates. Note that the three expressions could all be simply integer numbers. If the optional 'stride' of the loop (<expr3>) is omitted then a stride of one is assumed. For example you should now calculate the 3rd power of all integers up to 8. This can be achieved by using a 'DO' loop.

Exercise Three In an 'exercise3' directory type the following code and save it as 'power3_loop.f90' compile and run it.

```
PROGRAM power3_loop

!*** 3rd power of a for 1 <= a <= 8

  IMPLICIT NONE

  INTEGER :: a, b

  DO a = 1, 8
    b = a*a*a
    PRINT*,a,"to the power of three = ",b
  END DO

END PROGRAM power3_loop
```

The line 'DO a = 1, 8' tells the computer to repeat the section of the program between this line and the line 'END DO' . The part 'a = 1, 8' means that this section of code is to be repeated for a = 1, a = 2, a = 3 ... a = 8. Run the program and make sure that it works. NOTE that the indentation used inside the 'DO' loop makes it easy to see where it starts and ends.

‘ ’

Exercise Four In an '`exercise4`' directory copy your '`power3_loop.f90`' from **exercise three** changing its name to '`factorial.f90`' in the copying process. Edit it to create a code that will calculate the factorial value of the first eight integers and print them out to screen. The copy command you use could be similar to the example below.

```
fortran1 exercise4> cp ../exercise3/power3_loop.f90 factorial.f90
```

5.2 The Conditional '`DO`' Loop Construct

It is useful to have a looping construct that allows termination of the loop only when a certain condition has been met. For this situation Fortran has the '`DO - END DO`' looping construct.

```
DO
  <statement body>
  IF (<logical expression>) EXIT
  <statement body>
END DO
```

The logic of this style is that the <statement body> is iterated through many times until the <*logical expression*> is true.

5.3 The '`DO WHILE`' Construct

Fortran has the '`DO WHILE`' looping construct that does a very similar job to the conditional '`DO`' loop construct above. You should NOT use this '`DO WHILE`' form as the above conditional '`DO`' loop construct is more general and it is clear where the loop exits. The '`DO WHILE`' loop always exits at the top of the loop construct as this is where the test takes place. It is presented in these notes as you may well see it in some older codes.

```
DO WHILE <condition>
  <body statements>
END DO
```

The logic of this style is that the <statement body> is repeatedly executed '`WHILE`' the <*condition*> is true.

5.4 Nesting 'Do' Loops

It is often useful to 'nest' 'Do' loops.

```
DO i=1,10
  DO j=1,50
    <body statements>
  END DO
END DO
```

Exercise Five In an 'exercise5' directory write a code that does the summation

$$\sum_{i=10}^{19} \left(i * \sum_{j=21}^{30} j \right).$$

Use two nested 'DO' loops for the calculation that iterate from 10 → 19 and 21 → 30!

****

5.5 The 'CYCLE' Statement

By using the'CYCLE' statement you can force the natural flow of program execution to ignore the remaining statements in the body of the 'DO' loop that follow the 'CYCLE' statement. Hence, program execution returns to the 'DO' loop header to continue with the next iteration of the loop. Consider the following code.

```
PROGRAM test_cycle

  IMPLICIT NONE

  INTEGER i

  DO i=1,5
    IF (i==3) CYCLE
    PRINT*,i
  ENDDO

END PROGRAM test_cycle
```

Compiling and then running the code would print out the numbers 1,2,4 and 5. The number three would not be output because the program flow would be caught by the 'CYCLE' statement within the 'IF' statement.

6 Numeric Arrays : Part One

Arrays can be declared in Fortran and are in fact one of the most powerful features of the Fortran 90 language. A one dimensional numeric array is simply a list of numbers and could be used, for example,

to represent a vector. A two dimensional array could be used to represent a matrix. Arrays for a given numeric data type are declared using the attribute '`DIMENSION(<array size>)`' in the attribute list of a numeric declaration statement. To declare an '`INTEGER`' array of one dimension, called '`intlist`' to hold thirty values use,

```
INTEGER, DIMENSION(30) :: intlist !** 1D array
```

to create a two dimensional array of real numbers with four elements in each extent of the array use,

```
REAL, DIMENSION(4,4) :: mat1 !** 2D array
```

For example the factorial program could save the answers into an array.

```
PROGRAM factorial
  IMPLICIT NONE
  REAL, DIMENSION(20) :: fact
  INTEGER :: n, nmax = 20
  !** set the first value
  fact(1) = 1.0
  !** calculate factorials
  DO n = 2, nmax
    fact(n) = fact(n-1) * n
  END DO
  PRINT*,fact
END PROGRAM factorial
```

6.1 Printing Arrays to the screen

You have seen how to print integers, reals etc. to the screen using the '`PRINT`' statement. If '`mat1`' was a two dimensional array of real numbers representing a matrix then the command,

```
PRINT*,mat1
```

would just print out all the elements of the matrix in a messy list to the screen. To print out the matrix in a structured form use the print statements

```
PRINT*,mat1(1,:) !** Row one
PRINT*,mat1(2,:) !** Row two
PRINT*,mat1(3,:) !** Row three
PRINT*,mat1(4,:) !** Row four
```

This works because, in the code '`mat1(1,:)`', there are two arguments inside the brackets. The 1^{st} is the number '1' and means reference only row one (the first dimension of the array), the 2^{nd} argument is a colon (:). The colon means reference EVERY element in the column (the second dimension of the array).

Exercise Six : Write a program to create a two dimensional array with four elements in each dimension to represent a matrix. Set each of the matrix elements to be equal to the sum of its row index plus twice its column index. Hint: use a nested '`DO`' loop. Print out the matrix to the screen