Lecture 2 – Overview of Fortran 95/03

- **Fortran 95/03 remains the language of choice for many scientific programmers**
- **Latest extensions of Fortran allow for:**
  – Dynamic memory allocation
  – Object-oriented programming
  – Modules
  – Derived data types
  – Pointers
- **We will give a brief overview of Fortran 95/03 here to give you the basic information to create parallel programs**
  – structure of statements and programs, assignment statements, intrinsic functions, I/O, branches and loops, arrays, modules, data-types, pointers, memory allocation

Structure of a Fortran Statement

- **Fortran program consists of a series of *executable* and *non-executable* statements**
  - Executable statements describe actions (additions, subtractions, etc.)
  - Non-executable statements provide information necessary for the proper operation of the program
- **Fortran statements may be entered anywhere on a line and each line may be up to 132 characters long. Continuation lines are available by using an "&" between lines:**

$$100 \;\; output = input1 + input2 \qquad\qquad ! \; Sum \; the \; inputs$$

$$100 \;\; output = input1 \; \& \qquad\qquad ! \; Sum \; the \; inputs$$

$$\& \qquad + \; input2$$

These statements are identical

The number to the left is a unique statement label that can be used as a reference

The ! is used to comment

Structure of a Fortran Program

- **A Fortran program consists of a mixture of executable and non-executable statements that occur in a specific order**
- **Notice:**
  – Statements are case insensitive and can be in upper or lower case
  – Programming style varies. Some people like to capitalize key words, while others stay totally in upper or lower case.
  – Be consistent!! Readability is important!

Declaration section

```
Program    my_first_program

! Purpose :
!  To illustrate some of the basic features of a Fortran program
!

!  Declare the variables used in this program
INTEGER :: i, j, k                    ! All variables are integers
```

Execution section

```
! Get the variables to multiply together.
WRITE (*,*)' Enter the numbers to multiply :'
READ (*,*) i, j

! Multiply the numbers together
k = i * j

! Write out the results.
WRITE (*,*) ' Result = ' , k
```

Termination section

```
! Finish up.
STOP
END PROGRAM
```

Fortran Variables

- **A Fortran variable is a data object that can change value during execution. Each variable must have a unique name**
  - Up to 31 characters long
  - May contain any combination of alphabetic characters, digits and the underscore character
  - First character must be alphabetic
- **A Fortran constant is a data object that is defined before execution and does not change value**
- **Types of variables and constants:**
  - Numeric (INTEGER, REAL, and COMPLEX)
  - Logical (.TRUE. and .FALSE.)
  - Characters (CHARACTER)
  - Derived Data Types (which we will discuss later)

INTEGER, REAL, CHARACTERS

- **By default, variable names beginning with the letters I, J, K, L, M, and N are assumed to be INTEGER type <u>unless</u> declared otherwise**

```
INTEGER :: var1, var2, var3
```

- **By default, variable name beginning with any other letter are assumed to be REAL <u>unless</u> declared otherwise**

```
REAL :: var1, var2, var3
```

- **Character variables can be declared with lengths**

```
CHARACTER (len = < length >) :: var1, var2, var3   ! General form
CHARACTER :: var1                                  ! var1 has length = 1
CHARACTER (len = 10) :: var1                       ! var1 has length = 10
CHARACTER (10) :: var1                             ! var1 has length = 10
```

Parameters

- **Fortran constants may be assigned values with an executable statement or a PARAMETER statement in the declaration portion of the code**

```
type, PARAMETER :: name = value          ! where type is REAL, INTEGER, CHARACTER, LOGICAL
REAL, PARAMETER pi = 3.1415926
```

- **Real constants or hard coded real numbers should always have a decimal point!**
- **Integer constants or hard coded integers should not have a decimal point!**

Arithmetic Calculations

- **Arithmetic operations include**
  - \+ Addition
  - \- Subtraction
  - * Multiplication
  - / Division
  - ** Exponentiation

- **No two operators may occur side by side. Operators must be grouped:**

$$a * (-b)$$

$$a ** (-(b + 2)/3)$$

Integer Arithmetic

- **Integer arithmetic will result in the truncated integer**
  - Be careful when writing integer arithmetic code

3 / 4 will result in 0

5 / 4 will result in 1 compared to the following for REAL arithmetic

3./4. will result in 0.75

5./4. will result in 1.25

Order of Arithmetic Operations

- **Operations will be performed using the following hierarchy:**
  - Operations delineated between ( ) will be performed first with the inner-most done first
  - All exponentials are evaluated next, working from right to left
  - All multiplications and divisions are evaluated next working from left to right
  - All additions and subtractions are evaluated next, working from left to right

Mixed-Mode Arithmetic

- **When a mixed-mode operation is encountered, Fortran converts the integer into a real number and then performs the operation**
  - The order of operation is therefore important

$$\text{answer} = 1.25 + 9/4 \qquad !\,\text{results in answer} = 3.25$$

$$\text{answer} = 1.25 + 9./4 \qquad !\,\text{results in answer} = 3.5$$

  - Don’t program mixed-mode operations! Bad Form!
  - I usually take points off for mixed-mode operations!
- **For exponentiation, only use a real power when absolutely necessary. Integer powers are performed must faster without evaluating natural logs and exponential functions.**

Relational Operators

- **Logic operators are typically used as part of IF statements. They include:**

==	Equal to (.eq. Fortran 77)
/=	Not equal to (.ne. Fortran 77)
>	Greater than (.gt. Fortran 77)
>=	Greater than or equal to (.ge.Fortran 77)
<	Less than (.lt.Fortran 77)
<=	Less than or equal to (.le. Fortran 77)

- **Combinational logic operators include:**

I1.AND.I2	Result is TRUE if both I1 and I2 are TRUE, otherwise FALSE
I1.OR.I2	Result is TRUE if either I1 or I2 are TRUE, otherwise FALSE
I1.EQV.I2	Result is TRUE if I1 is the same as I2, otherwise FALSE
I1.NEQV.I2	Result is TRUE if one of I1 and I2 is , TRUE, otherwise FALSE
.NOT.I1	Result is TRUE if I1 is FALSE and FALSE if I1 is TRUE

Hierarchy of Logic Operators

- **If multiple logic operators are used in a single statement without ( ) to delineate groupings, then the operators are performed in the order:**
  - Relational operators, ==, /=, <, <=, >, >= are evaluated from left to right
  - All .NOT. operators are evaluated
  - All .AND. operators are evaluated from left to right
  - All .OR. operators are evaluated from left to right
  - All .EQV and .NEQV. operators are evaluated from left to right

Intrinsic Functions

- **Fortran has many built in (intrinsic) functions to deal with trigonometric, log, exponential, etc. operations. These functions are very fast, computationally. They include:**

TABLE 2–6
Some common intrinsic functions

Function name and arguments	Function value	Argument type	Result type	Comments
`SQRT(X)`	$\sqrt{x}$	R	R	Square root of x for $x \geq 0$.
`ABS(X)`	$\lvert x \rvert$	R/I	*	Absolute value of x.
`ACHAR(I)`		I	CHAR(1)	Returns the character at position `I` in the ASCII collating sequence.
`SIN(X)`	$\sin(x)$	R	R	Sine of x (x must be in *radians*).
`COS(X)`	$\cos(x)$	R	R	Cosine of x (x must be in *radians*).
`TAN(X)`	$\tan(x)$	R	R	Tangent of x (x must be in *radians*).
`EXP(X)`	e^x	R	R	e raised to the xth power.
`LOG(X)`	$\log_e(x)$	R	R	Natural logarithm of x for $x > 0$.
`LOG10(X)`	$\log_{10}(x)$	R	R	Base 10 logarithm of x for $x > 0$.
`IACHAR(C)`		CHAR(1)	I	Returns the position of the character `C` in the ASCII collating sequence.
`INT(X)`		R	I	Integer part of x (x is truncated).
`NINT(X)`		R	I	Nearest integer to x (x is rounded).
`REAL(I)`		I	R	Converts integer value to real.
`MOD(A,B)`		R/I	*	Remainder or modulo function.
`MAX(A,B)`		R/I	*	Picks the larger of `a` and `b`.
`MIN(A,B)`		R/I	*	Picks the smaller of `a` and `b`.
`ASIN(X)`	$\sin^{-1}(x)$	R	R	Inverse sine of x (results in *radians*).
`ACOS(X)`	$\cos^{-1}(x)$	R	R	Inverse cosine of x (results in *radians*).
`ATAN(X)`	$\tan^{-1}(x)$	R	R	Inverse tangent of x (results in *radians*).

Notes:
* = Result is of the same type as the input argument(s).
R = REAL, I = INTEGER, CHAR(1) = CHARACTER(len = 1)

From Chapman

Input and Output

- **Input and Output statements are written as:**

  READ (unit, format) input_list

  where "unit" is defined device and "format" is a statement number that defines the lay-out of the data to be read

  <u>Example-1</u>: READ (10,30) var, ivar
  30 FORMAT(f10.,5x,I5)

  reads from unit 10 (could be from a file or keyboard) and using format statement 30 that says var will be read as a floating-point number in columns of 10 and ivar will be read as an integer 5 spaces to the right in columns of 5. (Note: integers are assumed to be read as right-adjusted)

  <u>Example-2</u>: READ (*,*) var, ivar

  reads from the default device (keyboard) in a free format. (Note that numbers must be separated by a space or a comma, or be on separate lines in free format)

Input and Output

Example-3: WRITE (10,30) var, ivar

```
      WRITE (10,30) var, ivar
   30 FORMAT('real data =' f10.5, 2x, 'integer data = ',i5)
```

writes to unit 10 (could be file or terminal) using format statement 30 to define the lay-out of the data that is written.

Example-4:

```
      WRITE(*,*) 'real data = ',var,2x,'integer data = ',ivar
```

writes out var and ivar with the format included as part of the write statement. Note that var is written out as a floating point and ivar is written out as an integer since they are defined as real and integers, respectively

- **We will discuss this more in the next lecture**

Initialization of Variables

- **It is good practice to**
  - declare all variables used in a program
  - initialize all variables introduced in a program
  - at a minimum, you should use implicit declaration of variables

$$implicit\ \ real(kind=8)\,(a-h,o-z)$$ Double precision real

- **A good way to force yourself to declare all variables is through the use of the**

$$implicit\ none$$

statement. This forces every variable to be declared, INTEGER, REAL, COMPLEX, CHARACTER, OR LOGICAL, otherwise the program will not compile correctly. This also helps to pick up typo’s in your code.

Program Design

- **Designing the program is a very important first step before coding of the program even starts**
- **Use of “top-down” layout and flow charts is very useful in structuring the program**
  - Clearly state the problem that you are trying to solve
  - Define the inputs required by the program and the outputs produced by the program
  - Design the algorithm that you intend to implement in the program
  - Turn the algorithm into Fortran statements
  - Test the resulting program
- **Indentations are used to delineate “blocks” of code associated with loops and logic (shown below)**
- **Object-oriented design (OOD) is a modern method of code design (this will be discussed in a later lecture)**

Branches

- **Branches are Fortran statements that allow us to select and execute specific sections of code while skipping other sections of code.**
- **The most common form of a branch is the IF statement:**

```
IF(logical_expression)THEN
    statement 1
    statement 2
  .
  .
END IF
```

- **Example:**

```
IF(a**2 <= b) THEN
    write(*,*) a,b
END IF
```

writes the a and b variables when a^2 is less than or equal to b, otherwise it skips the write statement

- **Example:**

```
IF(a**2 <=b) write(*,*) a,b
```

ELSE and ELSE IF Blocks

- **Sometimes we may want to execute one block of statements if some condition is true and a different set of statements if other conditions are true. This can be done with *ELSE* and *ELSEIF* blocks:**

- **Example:**

```
IF(a**2 <= b) THEN
   write(*,*) ‘a**2 is less than or equal to b’, a,b
ELSE
   write(*,*) ‘a**2 is greater than b’
END IF
```

ELSE and ELSE IF statements must be on separate lines

- **Example:**

```
IF(a**2 < b) THEN
   write(*,*) ‘a**2 = b’, a,b
ELSE IF(a**2 = b) THEN
   write(*,*) ‘a**2 is equal to b’, a,b
ELSE
   write(*,*) ‘a**2 is greater than b’, a,b
END IF
```

Any number of ELSE IF statements may appear. An ELSE IF will be tested IFF all other IF tests above it fail so order is important.

Naming Block IFs

- **You can give a block IF a unique name in order to make your code more readable**

- **Example**

```
ASQUARED: IF(a**2 < b) THEN
   write(*,*) ‘a**2 = b’, a,b
ELSE IF(a**2 = b) THEN
   write(*,*) ‘a**2 is equal to b’, a,b
ELSE
   write(*,*) ‘a**2 is greater than b’, a,b
END IF ASQUARED
```

Branching with CASE

- **Another form of branching involves the CASE construct. It allows a particular code block to execute based upon a single integer, character, or logical expression:**

- **Example:**

```
PICKTYPE: SELECT CASE (itype_current)
CASE (itype1)
  statements….
CASE (itype2)
  statements…
CASE DEFAULT
  statements…
END SELECT PICKTYPE
```

IF itype_current is in the range of values of itype1, then the first block will execute

Examples of itype1, itype2 might be 1:20 or -5, or -10:55, or even a single integer

Optional name

Loops

- **We very often want to do a set of operations over elements of an array. In engineering problems, we often want to do a series of operations over the points (or cells) in a computational grid. We can do this using loops. There are different types of loops in Fortran**
  - Logical DO Loop
  - Iterative or counting DO Loop

Logical Loops

- **Logical loops can be written with IF's inside a loop:**

- **Example:**

```
DO
   statement 1
   statement 2…
   IF (logical_expression) EXIT
   statement n
   statement n+1
END DO
```

The statements are executed indefinitely until the logical_expression becomes true at which point control leaves the loop and executes the first statement after the END DO

Logical Loops

- **Logical loops can also be written with the WHILE construct:**

- **Pseudo Code**

  Example:

```
WHILE
    ….
    IF (logical_expression) EXIT
    ….
END of WHILE
```

- **Example:**

```
DO WHILE (logical_expression)
     …
END DO
```

Iterative or Counting Loops

- **We can also execute a block of statements N times by simply executing a DO loop with a counter**

- **Example:**

```
DO icounter = istart, iend, increment
    statement 1
    statement 2….
END DO
```

- **The statements between the DO and END DO are executed "(iend-start+1)/increment" times.**
- **The variable, icounter, may be referenced inside the loop. The values of istart, iend, and increment must be specified and <u>must be integers or defined integer variables</u>.**
  - I will take points off if non-integers are used as do loop indices!
- **"increment" is optional with a default value of 1**

Iterative or Counting Loops

- **Example:**

```
DO I = 1,101
   X =  REAL(I-1)
   Y = Y + X*100.
END DO
```

Note the use of decimals in the numbers when performing real arithmetic

- **Example:**

```
DO N = 10,-10,-2
   F = 2.*REAL(N)**2
   Z = 55.*F**2 – 25.
END DO
```

Note the use of integer 2 in power when possible…much faster

Iterative or Counting Loops

- **Iterative loops can be nested.**

- **Example:**

```
DO I = 1,20
   DO J = 1,20
      A = REAL(I)*REAL(J)
      X = 10.*REAL(I)
      B = A*X
   END DO
END DO
```

Cycle and Exit

- **Iterative loops can be controlled using the CYCLE and EXIT statements.**
  - CYCLE in conjunction with an IF statement can be used to return operation back to the beginning of the DO loop.
  - EXIT in conjunction with an IF statement can be used to terminate a DO loop

- **Examples:**

```
DO I = 1,5
   IF(I == 3) CYCLE
   WRITE(*,*) ‘TEST ‘, I
END DO
Results in:     TEST 1
                TEST 2
                TEST 4
                TEST 5
```

```
DO I = 1,5
   IF(I == 3) EXIT
   WRITE(*,*) ‘TEST ‘, I
END DO
Results in:     TEST 1
                TEST 2
```

Naming Loops

- **DO loops can be named in order to improve organization and "readability" of your code**

- **Example:**

```
MULTIPLY: DO I = 1,20
   DO J = 1,20
      A = REAL(I)*REAL(J)
      X = 10.*REAL(I)
      B = A*X
   END DO
END DO MULTIPLY
```

- **Example:**

```
TEST: DO I = 1,5
   IF(I == 2) CYCLE TEST
   IF(I == 4) EXIT TEST
   WRITE(*,*) 'TEST CASE ', I
END DO TEST
```

Combining IF Blocks and DO Loops

- **When combining DO-loops with logical IF-blocks, it is generally much more computationally efficient to put the DO-loop inside of the IF-block**
  - Logical IF-blocks are computationally expensive
  - By placing an IF-block inside of a DO-loop, the expense of the IF statement is greatly multiplied

- **EXAMPLE:**

```
IF(TEST<=1.0) THEN
  DO I = 1,32,2
    TEMP = REAL(I)*100.
  END DO
END IF
```