# Chapter 1: Fortran Programming:

# Detailed Notes

This chapter contains a brief introduction to the most commonly used FORTRAN commands, and should be read in conjunction with the example programs. See the 'what you need to know' handout for a summary.

First of all, how is a simple FORTRAN program structured?

## *FORTRAN Layout*

| | | |
|---|---|---|
| 1 | Program Tiger | TITLE |
| 2 | Implicit None | turns on spell-checking |
| 3 | Real:: energy, mass, velocity | declares variables |
| 4 | mass=1. | sets value for mass |
| 5 | velocity=1. | |
| 6 | energy = mass* (velocity**2)/ 2. | calculation $E = mv^2/2$ |
| 7 | ! mks units | comment |
| 8 | Print *, energy ! Prints to screen | print command and comment |
| 9 | End Program Tiger | ends execution |

### *Notes*

- Upper and lower case are interchangeable except for character strings (see below)
- Spaces are allowed except within the name of a variable or program
  Bad: Kinetic Energy
  Good: Kinetic_Energy, KineticEnergy

### *Names of Variables*

| Good | Bad / Not Allowed | Ugly |
|---|---|---|
| Mass | M+N | g3x2v |
| Kinetic_Energy | Kinetic Energy | x |
| MyName | 3X | real |
| Name3 | Name 3 | x3 |

# *Basic Data Types*

1 Real
2 Integer
3 Complex
4 Logical
5 Character
6 Character(len=12)

Variables are declared according to their type E.g.:

Integer:: height
Character(len=7)::name
(Note that Character is not the same as Character(len=1).)

If a variable is not declared, the compiler will guess its type, often getting it wrong!!!! The 'Implicit None' statement forces an error statement when this happens.
E.g. If the variable 'message' is misspelt as 'messaje' the compiler will give the following error message:

'ERROR: Implicit type for messaje.'

| TYPE | GOOD | BAD |
|---|---|---|
| Integer | 3214<br>-74 | 3,214 |
| Real | 3.15<br>-6.7<br>2.<br>2<br>3.15e1 (=$3.15\times10^1$) | 3.15.2<br><br><br><br>3.15e2.5 |
| Complex | $(2.7, -3.2)$ | $2.7 - 3.2I$ |
| Logical | .true.<br>.false. | true<br>false<br>t<br>f |
| Character Strings | "hello"<br>'hello'<br>"don't"<br>'she said "hi"' | hello (a variable name)<br>'hello (incomplete)<br>'don't do this' |

## *Arithmetic Expressions*

| FORTRAN | MATHS | |
|---|---|---|
| x=a*b | $x = ab$ | |
| x=a+b | Same | |
| x=a-b | Same | |
| x=a**b | $x = a^b$ | |
| x=a/b | $x = \frac{a}{b}$ | |
| x=a*b+c | $x = ab + c$ | (Multiplication done first) |
| x=(a*b)+c | $x = ab + c$ | |
| x=a/b*c/d+e | !!! | Who knows!!! Use brackets |
| x=(a/b)*(c/(d+e)) | $x = \frac{ac}{b(d+e)}$ | |

**The equals sign**: The = sign really means 'is assigned the value'. Therefore

x=x+1

is a VALID Fortran statement. The new value of x is assigned the value
< Old value of x>+1.

## *Integer Division*

Integer variables do not store fractions or remainders, so care is needed when performing operations with integers.

| | | |
|---|---|---|
| 8/3 | → | 2 |
| 4/5 | → | 0 |
| -8/3 | → | -2 |
| (5*8)/5 | → | 8 |
| 5*(8/5) | → | 5 |

If reals and integers are mixed, a product or division becomes real, e.g. 8.0/5 → 1.6.

**Example Program:**

```
Program CatsandDogs
Integer:: dog
Real:: cat
cat=8.0/5
dog=cat
print *,dog                 !→ prints 1
print *,cat                 !→ prints 1.6000000
End Program CatsandDogs
```

Note that:
dog=8.0/5
cat= dog
results in dog having value 1 and cat value 1.0000000.

# *Intrinsic Functions*

Fortran supplies numerous functions, e.g.,
Sqrt, Sin, Cos, Tan, Exp, Log.

Some of these operate only on particular data types, e.g., Sqrt should be given a real rather than an integer.

| FORTRAN | MATHS |
|---|---|
| x=Sqrt(4.0) | $x = \sqrt{4} = 2.0$ |
| pi=4.*atan(1.) | pi= $4 * \tan^{-1}(1) = \pi$ |
| y=sin(pi/2.) | $y = \sin(\frac{\pi}{2}) = 1.0$ |

# *Basic Input and Output*

## *To print to screen*

Print *, OR Write(*,*)

The first * means STANDARD OUTPUT.
The second * means AUTOMATIC FORMATTING.

**Example:**

Write(*,*) “Mass=”, answer,”grams”

Here ‘answer’ is a real variable storing 27.2. The output is

Mass=27.2000000grams

## *To read from the keyboard*

Read *, OR Read (*,*)

## *To read and write to and from files*

To open a file called ‘foot.bal’ insert the command

OPEN(17, FILE=’foot.bal’)

(Note: instead of 17, any small(ish) integer other than 5 (=the keyboard) and 6 (=the screen) may be used.)

After this command can read from ‘foot.bal’ using Read (17,*), e.g.

```
Integer :: Score
Open (17, File='foot.bal')
Read (17,*) score
```

and (optional), when all reading is finished

```
Close (17)
```

Each call to Read goes to a new line by default, so if foot.bal contains

3
1
0
4

the first Read gets 3, the second 1 etc. etc.

If the file ‘lion.txt’ does not exist, the command

OPEN(17, FILE=’lion.txt’)

will create it. This is important if you are writing to a new file. Note that a file should usually be opened only once (do not place an OPEN command inside a DO loop.)

# *DO loops*

DO loops are used when the same operation needs to be repeated 'many' times. Below are three different types of loop used in different situations. The type of DO loop to use depends on whether the number of times you want the loop to run is known beforehand (type 2) or depends on the results of operations carried out in the loop (Type 1a or b).

## *Basic Syntax for the Three Types*

**Type 1a**

```
DO
    statement
      .
      .
      .
    IF (condition) EXIT
    more statements
      .
      .
      .
END DO
```

**Type 1a**

```
DO WHILE (condition)
    statement
      .
      .
      .
      .
      .
      .
      .
      .
END DO
```

**Type 2**

```
DO count=1,N
    statement
      .
      .
      .
      .
      .
      .
      .
      .
END DO
```

When the END DO statement is reached the computer returns to the DO line and starts over again.

If in Type 1a the EXIT statement cannot be reached, or in Type 1b (condition) is never met, the program runs forever, an INFINITE LOOP!!

To exit an infinite loop....hit Ctrl-C

It is customary to indent statements within DO loops, (by e.g. 3 spaces). It is also common practise to place the EXIT as near the top or bottom of the loop as possible.

Note, however, that all three types can be easily used to execute some statements exactly 10 times.

**Type 1a**

```
count=1
DO
    IF (count>10) EXIT
    statements
      .
      .
    count=count+1
END DO
```

**Type 1a**

```
count=1
DO WHILE (count <=10)
    statement
      .
      .
      .
    count=count+1
END DO
```

**Type 2**

```
DO count=1,10
    statement
      .
      .
      .
      .
END DO
```

Type 2 is clearly the most succinct!!

*General Form for Method 2*

```
Integer :: count
      DO count = a,b,c
        PRINT *, count
      END DO
```

This prints out....

$$a$$
$$a + c$$
$$a + 2c$$
$$.$$
$$.$$
$$a + jc$$

where $j$ is an integer such that $a + jc \leq b$ and $a + (j + 1)c > b$.

**Example**

```
      DO count=3,16,4
```

gives

3
7
11
15

```
      DO count=16,3,-4
```

gives

16
12
8
4

**Example**

Calculate $\sin x$ for $x = 0, 0.2\pi, 0.4\pi, ..., \pi$.

**GOOD LOOP**

```
pi=4.* atan (1)
DO count=0,10,2
    x=sin(count*0.1*pi)
    PRINT *,x
END DO
```

**BAD LOOP**

```
pi=4.* atan (1)
DO count=0,1,0.2
    x=sin(count*pi)
    PRINT *,x
END DO
```

The second loop fails because $5 * 0.2 \neq 1$ in computer arithmetic, as 0.2 cannot be stored exactly with a finite number of bits in binary!

# *IF and IF blocks*

## *The one-line IF statement*

**Examples:**

```
IF (count==5) EXIT
IF (x/ =0) y=1/x
IF (hadenough.OR.count>3) EXIT
```

(hadenough is of logical type in this case).

## *IF blocks*

If too much information must be placed after the IF (condition) then we can go to multiple lines

```
IF (x/=0) THEN
y=1/x
z=y+2
END IF
```

Possible relations going into an IF statement are
>, <, >=, <=, ==, / =, representing $>$, $<$, $\geq$, $\leq$, $=$, $\neq$ respectively

IMPORTANT:: Note that == really means 'equals' whereas = means 'is assigned the value'.

Other possibilities include logical operations acting on logical variables
.and., .or., .not.
Note that the computer treats entire expression $(x > 3)$ as a logical variable that is '.true.' if $x > 3$ and '.false.' if $x \leq 3$.

## *ELSE statement*

**Example**

```
IF (x/=0) THEN
y=sin(x)/x
ELSE
y=1
END IF
```

Note that the logical equivalence symbol is .eqv.
Not equivalence is .neqv.

**Example:**

```
logical :: giraffe
giraffe = (2**2 == 4)
```

1. IF (giraffe) THEN ....
2. IF (giraffe==.true.) THEN .... !WRONG
3. IF (giraffe .eqv. .true.) THEN .... !GOOD BUT SAME AS (1)

## *Arrays*

```
Integer :: matrix1(3,4)
Real :: position(3)
Character(len=20) :: namelist(100)
```

Matrix1 contains 12 integers, position 3 reals, and namelist 100 × 20 character strings

### *Filling arrays*

- **Method 1: One by one**
  ```
  Real :: x(3)
  x(1)=2.47
  x(2)=3.e10
  x(3)=-6.3
  ```
- **Method 2: as a vector**
  ```
  Real :: x(3)
  x=(/ 2.47, 3.e10,-6.3 /)
  ```
- **Method 3: Using a DO loop**
  ```
  Real :: x(3)
  DO i=1,3
        x(i)=7.5+2*i
  END DO
  ! this gives x=(9.5, 11.5, 13.5)
  ```
- **Method 4: Using assignment statements**
  ```
  Real :: x(3), y(3), z(3)
  y= (/ 5,6,7 /)
  z=2             ! z=(2,2,2)
  x=y+z           ! x=(7,8,9)
  ```

### *Parameter Declaration*

Parameters are variables that are 'hardwired' at the start of the program. Array sizes can be declared using a parameter variable, but not a normal integer.

**Example**

```
Integer, Parameter :: maxsize=100
Character(len=20):: names(maxsize)
```

- 'maxsize' cannot be changed during the program
- 'maxsize' is available for setting the size of arrays

In the example above, 'names' contains 100 strings of 20 characters.

## *Modifying Index Ranges*

**Example**

Integer :: bear(0:4), fox(2,-2:1)

In this example, 'bear' is a one dimensional array of length 5, with index range from 0 to 4.

$$\begin{array}{rcl} \text{bear} & = & (/\,.\,,\,.\,,\,.\,,\,.\,,\,./) \\ & & \quad 0\ \ 1\ \ 2\ \ 3\ \ 4 \end{array}$$

'fox' is a two dimensional array $2 \times 4$ with

$$\begin{array}{rcl} \text{fox} & = & \left(\begin{array}{cccc} .\,, & .\,, & .\,, & .\,, \\ .\,, & .\,, & .\,, & .\,, \end{array}\right) \begin{array}{c} 1 \\ 2 \end{array} \\ & & \quad -2\ -1\ \ 0\ \ 1 \end{array}$$

## *Accessing Subarrays*

**Example**

```
Integer :: x(3), wolf(3,0:4),row,col
DO row=1,3
  DO col=0,4
    wolf(row,col)=row*col
  END DO
END DO
```

gives

$$\text{wolf} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 2 & 4 & 6 & 8 \\ 0 & 3 & 6 & 9 & 12 \end{pmatrix}$$

then the following statements, processed in order give

```
x=wolf(:,2)              ! gives x = (2,4,6)    i.e. wolf(:,2) is the column with index 2.
x=wolf(:,4)              ! gives x = (4,8,12)
x=wolf(3,0:2)            ! gives x = (0,3,6)
x=wolf(:,2)              ! gives x = (2,4,6)
x(2:3)=wolf(1:2,4)       ! gives x = (2,4,8), as x(1) is unchanged.
x=0                      ! gives x = (0,0,0)...sets all components to zero
x(1:2)=wolf(3,3:4)       ! gives x = (9,12,0)
```

## *Manual Formatting*

Difficulty: Write(2,*) or Print *, both rely on computer to decide formatting.

```
Program Format
Integer :: x=34
Real :: y=2.7, z=-120
Character(len=30) :: myformat
myformat="(A,I6,A,2F12.4)"
write(6,myformat) "x=",x," and y and z =",y,z
End Program Format
```

The output from this is: (blank spaces not in strings are denoted by '*')

x=****34 and y and z =******2.7000***-120.0000

**Comments**

The write statement has five things to write out:

| | OUTPUT | FORMAT DESCRIPTION |
|---|---|---|
| 1. | "x=" | A (Character String) |
| 2. | x | I6 (Integer with 6 spaces) |
| 3. | " and y and z =" | A (Character String) |
| 4. | y | F12.4 (Real with 12 spaces, four digits after decimal pt.) |
| 4. | z | F12.4 (Real with 12 spaces, four digits after decimal pt.) |

I6 means 6 spaces allocated, with spaces filling in front

$$\begin{aligned} 34 &\rightarrow \ ****34 \\ -1207 &\rightarrow \ *-1207 \\ &\text{etc} \end{aligned}$$

F8.2 is a format for real nos. and means 8 spaces in total (including decimal pt., '-' sign and spaces, with exactly two numbers after the decimal pt.

$$\begin{aligned} -102 &\rightarrow \ *-102.00 \\ 80.738 &\rightarrow \ ***80.74 \\ &\text{etc} \end{aligned}$$

## *Other format descriptions*

Consider the number 0.0217.

| FORMAT | OUTPUT | EQUIVALENT TO |
|---|---|---|
| F8.4 | ***0.0217 | |
| ES9.2 | **2.17E-02 | $2.17 \times 10^{-2}$ |
| E9.2 | **0.22E-02 | $0.22 \times 10^{-1}$ (rounded) |
| E9.3 | *0.217E-02 | $0.217 \times 10^{-1}$ |

**ES format:** The digit in front of the point is always 1-9.
**E format:** The digit in front of the point is always 0.

**Integer formatting**

For $x = 4$, can force initial zeros, e.g.,

| | |
|---|---|
| I5.3 | **004 |
| I5.2 | ***04 |

**Including Spaces**

E.g. to insert 7 spaces: Use 7x,
(I4,7x,I4) inserts 7 spaces between the $\leq$4 digit integers.

**Note:** That the format can be inserted directly into the write statement, for example,

write(6,"(3f12.4)") y,z,w

gives a format for three reals y,z,w.

# *Subroutines and Functions*

## *Subroutines*

Subroutines are self-contained 'mini'-programs written to perform specific tasks. They are usually found after the main program (or in separate files which are 'linked' during compilation.

```
Program Middle
Integer :: lion,tiger,bear
        .
        .
Call Findmidpoint(lion,tiger,bear)
        .
Contains
        .
Subroutine Findmidpoint(a,b,midpoint)
Integer, Intent(in) :: a, b
Integer, Intent(out) :: midpoint
Integer :: silly
silly = a + b
midpoint = silly/2
End Subroutine Findmidpoint

End Program Middle
```

- In this subroutine 'a' takes the value of 'lion', 'b' takes the value of 'tiger' and 'bear' will be assigned the calculated value of 'midpoint'.
- The types of all corresponding variables must match (e.g. 'lion' with 'a', etc.)
- As a,b have been declared Intent(in), they cannot be changed in the subroutine.
- If it is desired to change the value of the input variables, they can be declared Intent(inout)

**Example**

```
Program Intswap
Integer :: x=-10, y=17
Print *,x,y
Call Swap(x,y)
Print *,x,y

Contains

Subroutine Swap(a,b)
Integer, intent(inout) :: a,b
Integer :: temp
   temp=a
   a=b
   b=temp
End Subroutine Swap

End Program Intswap
```

This gives output

$$\begin{array}{cc} -10 & 17 \\ 17 & -10 \end{array}$$

## *Functions*

Functions are similar to subroutines, but are designed to return an output of a given form from a specified input. Notice that they are called implicitly (not using CALL).

```
x=2
y=3
z=sillyfun(x,y)
    .
    .
    .
Function sillyfun(a,b)
Integer, intent(in)::a,b
Real:: sillyfun
  sillyfun=sin(2.4*a*b)
End Function Sillyfun
```

## *SIZE and SUM Commands*

Sometimes when array sizes have been been defined by a series of operations, it is useful to be able to recover their dimensions. The SIZE command does this by returning the number of elements in an array.

**Example**

For a 1-dimensional list
size(list) gives the number of elements in the list. For

$$m = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

size(m,1)=2 gives the number of elements of first dimension (=2).
size(m,2)=3 gives the number of elements of second dimension (=3).

Sometimes it is also useful to sum the elements in arrays. The SUM command gives a quick way of doing this.

**Example**

```
Integer:: V(3)= (/ 1,2,3 /), W(3)=(/ 5,6,7 /), Z
Z=SUM(V*W)                          ! Evaluates the scalar product = 5+12+21=38.
```

# *Modules: Program Layout*

Useful functions, subroutines, variable declarations and constants (e.g. pi=3.14159) and data types (see later) can be collected in MODULES.

```
Module Useful
Implicit None
Real :: pi=3.14159
Integer:: ......

Contains

Function Nice(....)
    .
    .
Subroutine Cheerful(....)
    .
    .
End Module Useful
```

The module can then be used as follows:

```
Program Example
Use Useful
Implicit None
    .
```

**Notes**

- The module can have no executable statements, e.g. x=2*pi, just declarations functions and subroutines.
- The program using the module has a 'Use' statement.
- A program can use more than one module
- If modules and program are stored in one file modules should come first

# *Recursion*

A function or subroutine cannot call itself unless it is declared *recursive*. By allowing subroutines and functions to call themselves, we create the possibility of implementing *recursive algorithms*.

A recursive algorithm solves a (difficult) problem of order $N$ indirectly, by breaking it down into problems of order $N-1$ or less, and then calling itself to reduce these constituent problems still further. In this manner the problem is eventually reduced to problems of low order (e.g. order 1) that may be solved directly.

**Example 1: The factorial function (see example program)**

**Example 2: Towers of Hanoi (see example program)**

**Objective**: Transfer rings to the third stick.
**Restriction**: Rings may only be placed on top of larger rings.

**A Recursive Algorithm**:

The problem of moving $N$ rings from stick a to stick c may be broken down as follows:

1. Move N-1 rings from stick a to stick b (to see how, go to comment 4).
2. Move ring N to c.
3. Move N-1 rings from b to c (to see how, go to comment 4).
4. Break down the $N-1$ problems using (1-3). E.g. To move $N-1$ rings from a to b, Move $N-2$ rings from a to c, move ring $N-1$ from a to b, and finally move $N-2$ rings from c to b. The problem of moving $N-2$ rings is broken down in exactly the same way.

**Plan to Implement in FORTRAN:**

To implement in FORTRAN we need the following ingredients: **Data**: represent the position of the rings as a 3 integer arrays, a(N), b(N), c(N). N is number of rings. a is first stick, b second stick, c third stick.
a(1) is lowest ring on a
a(2) is second lowest ring on a
etc.
Initially, we have

a=( 4,3,2,1)
b=(0,0,0,0)
c=(0,0,0,0)

'4' represents the largest ring, '1' the smallest, '0' means no ring.

**Routines**:

1. A function which moves the top ring from one stick to another
   e.g. Subroutine movetop (from, to)
   (i.e. movetop (b,c) moves top ring from b to c.)
2. Something to print out positions
3. Calculate number of rings on a stick.

See the example program for details.

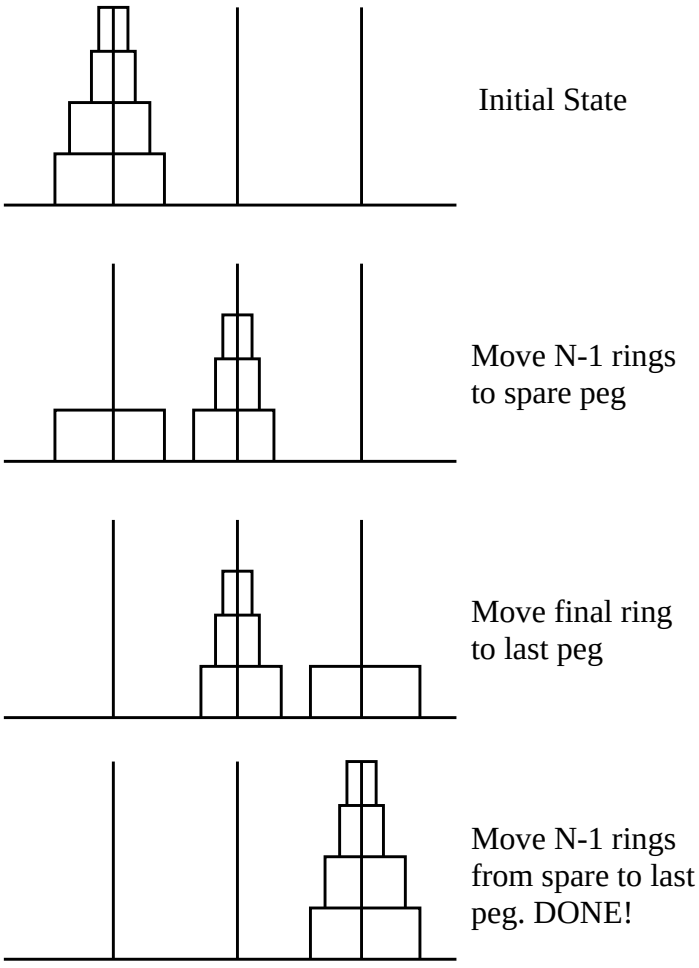


Figure: An illustration of the algorithm for the towers of hanoi.

# *Object-Orientated Programming*

**Purpose**: to write re-usable, error-free programs

**Jargon**:

1. An **abstract data structure** is some form of data (e.g. a set of numbers 1-150, a digitalised photograph, etc.) together with functions and subroutines used to manipulate the data (e.g. union / intersection of sets, cropping / removing red-eye, for photograph).
2. A **class** is a method of implementing the Abstract data structure. e.g. a user-defined data type, and functions which use the type.
3. An **object** is a variable declared using the user-defined type. e.g. S,T,V,W in Example 13.

# *User Defined Data Types*

## *General Structure*

**Declaration**

```
type mydata
  variabletype1 :: label1
  variabletype2 :: label2
    .
    .
end type mydata
```

('variabletype1::' and 'variabletype2::' refer to declaration of intrinsic variables e.g. 'real::', 'logical::' or 'character(len=3)::'.)

Note that some of the data inside mydata can be in the form of arrays.

**Example:**

```
type league
  character(len=20)::teamnames(20)
end type league
```

**Usage**

Declaring 'premiership' to be a league

```
type(league) :: premiership
```

**Filling in Values**

Values of the teamnames in premiership can be filled in as follows

```
premiership%teamnames(1)='Liverpool'
premiership%teamnames(2)='Arsenal'
premiership%teamnames(3)='Cambridge United'
```

etc.

or, alternatively

premiership=league( (/ 'Liverpool','Arsenal','Cambridge United',..../) )

i.e. with a vector of character strings of length 20.

# *Operator Overloading*

**Purpose**: Extend meaning of '$*$', '+', '-' etc. so that they work on user defined types.

It can be convenient to represent complex operations acting on user defined types or arrays (e.g. union or intersection of sets - see example 13) by a simple operator such as '$*$' or '+'.

FORTRAN allows the meaning of operators such as '$*$', '+' etc. to be EXTENDED, so that when the operator is applied to the objects in question (for example a set of character strings, or a data type) for which the operator WOULD NOT NORMALLY HAVE A MEANING, fortran knows to look in a (supplied) subroutine or function to apply the operator.

**Syntax**

to overload '$*$' with the function intersection:

```
interface operator(*)
   module procedure intersection
end interface
```

note that 'intersection' can be either a subroutine or a function that acts on variables of type(set). If V and W are of type(set) then V$*$W will be their intersection.