Handout Five

March 13, 2006

1 Internal Functions & Subroutines

The main program unit can 'contain' two types of, what can be described as, sub-programs. Sub-programs can be used to beak up a large program and structure it into a more 'readable' and less messy form. They can be repeatedly called to execute a series of statements that describe an often used task. This saves typing a sequential set of statements repeatedly in your code. Sub-programs, or procedures as they are often called, come in two different forms a 'SUBROUTINE' or a 'FUNCTION'. They are positioned inside the main program unit after the program execution statements and after a 'CONTAINS' statement. The following now details the syntax of the main program unit.

```
PROGRAM <name>

  < Specification section >
  < Executable Statements >

  CONTAINS
  !*** Internal Procedures Follow

  < Internal Procedure One >
  < Internal Procedure TWO >
     :         :          :
  < Internal Procedure n >

END PROGRAM <name>
```

1.1 Subroutines

A subroutine, just like the main program, must have a 'name' and this name is used to 'call' the subroutine during program execution. Inside the subroutine you can declare data types just as you have been doing in the specification section of the main program unit. You can pass information from the main program

to the subroutine through an 'argument list' to the subroutine.

Exercise One : An example of a simple subroutine without any arguments is given below. In 'handout5/exercise1' directory type in and compile the following code.

```
PROGRAM internal1
! **** Example of a Program with an internal subroutine

   IMPLICIT NONE

   REAL :: a,b,c

   CALL calc
   PRINT '("Answer = ",f10.4)',c

CONTAINS

   SUBROUTINE calc
     PRINT*,"Enter number one"
     READ*,a
     PRINT*,"Enter number two"
     READ*,b
     c=SQRT(a**2+b**2)
   END SUBROUTINE calc

END PROGRAM internal1
```

The program has one procedure in the form of a subroutine called 'calc'. The main program unit declares three 'REAL' data types `a,b & c` and then calls the subroutine 'calc' using the Fortran keyword 'CALL'. Then program execution moves into the subroutine 'calc' which prompts the user to input the two variables `a & b` and then calculates $\sqrt{a^2 + b^2}$. Execution then moves back into the main program (after the 'CALL calc') and continues on the next line where the result of the calculation is written to the screen. If we wanted to we could call 'calc' many times from the main program execution section. This would of course be useful if we needed to calculate 'SQRT(a**2+b**2)' several times.

1.2   Argument lists

The example code above 'internal1' can also be written so it uses an argument list to pass the variables `a, b & c` through to the subroutine calc. This is done by appending a pair of brackets to the subroutine name in the 'CALL' statement, and the arguments placed inside the brackets as a comma separated list. The arguments are received into the procedure declaration in the same way using parenthesis. The variables that are received into a procedure in this way are called **'dummy variables'**. See the example

below;

Exercise Two : In a 'handout5/exercise2' directory copy the code from 'exercise1' edit it to the following code and compile.

```
PROGRAM internal2
! **** Example of a Program with an internal subroutine

  IMPLICIT NONE

  REAL :: a,b,c

  CALL calc(a,b,c)
  PRINT '("Answer = ",f10.4)',c

CONTAINS

  SUBROUTINE calc(a,b,c)
    REAL :: a,b,c !*** Dummy variable declaration

    PRINT*,"Enter number one"
    READ*,a
    PRINT*,"Enter number two"
    READ*,b
    c=SQRT(a**2+b**2)
  END SUBROUTINE calc

END PROGRAM internal2
```

Notice that the dummy arguments in the subroutine have the same name as the actual arguments passed out of the main program. This does not need to be the case and they could have been given completely different names. If using internal procedures it is recommended to always pass variables from the program unit through to the procedure and declare the dummy variables with the same names as those in the calling program unit (in this case the main program). This prevents you being able to access the same variable via two different names, one name from the main program and also one from the dummy argument name passed through the argument list.

1.3 Local, Dummy Variables & Call by Reference

- **Local Variables :** are variables that have NOT been passed into a program unit are and are declared inside that program unit. This means they are local only to the program unit where they were initially created. So variables declared in the main program unit are called local variables in that unit.

- **Dummy Variables :** are variables that can only exist in procedures. They are accepted into the procedure via an argument list. The 'dummy argument list' in the procedure's header does not have to declare the variable with the same name as that in the calling argument list e.g.

```
CALL calc(a,b,c)

SUBROUTINE calc(d,e,f)
```

Dummy variables 'always' have an associated 'local variable' declared in another program unit elsewhere in the code. They can be seen as an alias to this local variable.

- **Call by Reference :** In fortran when you pass a variable name through an argument list from the calling unit to a procedure you are actually passing the address in memory (memory location) where the computer stores the value of the variable. This means that fortran does **NOT** make a new copy of the variable for the procedure being called instead the procedure accepts the address of the local variable passed in from the calling program unit. The address is then used to create a dummy variable which in turn can directly access the value or make changes to the original local variable it is associated with. So, whenever you use a dummy variable it is identical to using the local variable associated with it from the calling program unit. Therefore, **any** changes you make to the dummy variable will also be made to the associated variable from the original calling program unit.

- **Call by Value :** Fortran does **not** use call by value but uses instead 'call by reference' explained above. There are other languages that use call by value "c" and "c++" are two which spring to mind and use 'Call by Value' as their default method for passing arguments to 'sub-programs'. In call by value a new copy of every variable in the argument list is made for the procedure being called and any changes made to the variables are **not** passed back into the calling procedure. Some 'Call by Value' implementations allow the programmer to force the variable to be copied back afterwards but this is a example of poor programming methodology!

- **Advantages of Call by Reference :** Why does fortran use 'Call by Reference'? Remember fortran is designed for numerical computation! This demands speed and efficiency. 'Call by Reference' is very much faster than 'Call by Value' especially when using massive multi dimensional arrays. 'Call by Reference' also uses far less memory, this is because the code is not creating a new copy of every variable being passed in through the argument list.

1.4   Scope

In the example code 'internal2' above the variables `a,b & c` are declared in the main program. They are however 'visible' in any internal procedures of the main program unit unless some variables of exactly the same name are declared locally in the internal procedure. By 'visible' we mean the variable's value can used and/or changed. This range of visibility of the variables a,b & c is referred to as their scope. It is often desirable to limit the scope of data objects and this can be done using Fortran 'modules' which we will cover later in the course. Until then when using internal procedures it is a good idea to pass any data objects that are needed in the procedure through as arguments to that procedure. This makes the procedure more 'independent' and 'self contained' and could be transferred to other codes without needing modification. Also note that any variables declared locally in an INTERNAL procedure that have the same name as a declared variable in the main program unit ('NOTE' Not dummy variables) will supersede the data in the main program unit. By superseding we mean that the variable in the internal procedure has absolutely nothing to do with the variable of the same name in the calling program unit.

Consider the example below

```fortran
PROGRAM internal3

  IMPLICIT NONE

  REAL :: a,b,c ! ! *** Declare Local a,b,c to main program unit

  PRINT*,"Enter number one"
  READ*,a
  PRINT*,"Enter number two"
  READ*,b

  CALL calc(a,b,c)

  PRINT '(/,"IN MAIN PROGRAM"," a =",f5.2," b =",f5.2," c=",f5.2)',a,b,c

CONTAINS

  SUBROUTINE calc(d,e,f)

    REAL :: d,e,f ! *** Dummy variables
    REAL :: a,b,c !  *** Declare a,b,c local to subroutine

    f=SQRT(d**2+e**2)
    a=d/2 ; b=e/2 ; c=f/2
    PRINT '(/,"IN SUBROUTINE  "," a =",f5.2," b =",f5.2," c=",f5.2)',a,b,c

  END SUBROUTINE calc

END PROGRAM internal3
```

The variables `a,b` & `c` are declared as local in both the main program unit and the internal subroutine `calc`. The variables are now read from the keyboard into the main program, however changing the value of `a,b` & `c` in the subroutine has no effect on the value in the main program. This is because `a,b` & `c` in the subroutine supersede `a,b` & `c` in the main program and are separate local objects.

.

Exercise Four : In a 'handout5/exercise4' directory copy the code from 'exercise3' edit it so the subroutine 'calc' looks like it does below.

```
SUBROUTINE calc(d,e,f)

  REAL :: d,e,f ! *** Dummy variables

  f=SQRT(d**2+e**2)
  PRINT '(/,"[1] IN SUBROUTINE "," a =",f5.2," b =",f5.2," c=",f5.2)',a,b,c
  a=d/2 ; b=e/2 ; c=f/2

  PRINT '(/,"[2] IN SUBROUTINE "," a =",f5.2," b =",f5.2," c=",f5.2)',a,b,c

END SUBROUTINE calc
```

Do you notice any difference in the results. If so why?

1.5 Functions

Functions behave identically to subroutines except, as suggested by their name, they return a value. This value is returned to the calling program unit using the functions name. The function name is declared as a data-type inside the body of the function just as you would any other variable. Inside the function the function name is then assigned the value that needs to be returned to the calling program unit. As a function returns a value it could be used in a mathematical expression, just like an intrinsic function ie SIN(x), or in a 'PRINT' statement. The simple program used earlier, to demonstrate how a subroutine works, could be written as a function in the following way.

Exercise Five : In a 'handout5/exercise5' Copy the code from exercise two and edit it to use a function instead of a subroutine.

```fortran
PROGRAM internal5
! *** Program to demonstrate a function
  IMPLICIT NONE

  REAL :: a,b,c ! *** Declare a,b,c

  PRINT*,"Enter number one"
  READ*,a
  PRINT*,"Enter number two"
  READ*,b

  c=calc(a,b)

  PRINT '("Answer = ",f10.4)',c

CONTAINS

  FUNCTION calc(d,e)

    REAL :: d,e   ! *** Dummy variables
    REAL :: calc  ! *** Local Variables

    calc=SQRT(d**2+e**2)

  END FUNCTION calc

END PROGRAM internal5
```

1.6   The 'INTENT' attribute

As mentioned in 'handout2' variable declarations can have 'attributes.' Dummy variables in a function or subroutine can be given an 'INTENT' attribute in their declaration statement. This is useful as it allows us to provide more information to the compiler with respect to what the dummy variable is intended for in the procedure. This allows the compiler to do more checking and maybe in some situations produce faster code. From now on you should use these attributes in ALL your programs that have procedures!! There are three possibilities for the 'INTENT' attribute.

1. 'INTENT(IN)' : This means that the variable must have been assigned a value before the subroutine or function is called. Also the subroutine or function is not allowed to change that value.

2. 'INTENT(OUT)' : The dummy argument must not be used inside the procedure until it has been assigned a value inside the procedure.

3. 'INTENT(INOUT)' : The dummy argument must hold a value on entry into the procedure and that value may be changed by the procedure.

So for example the `internal5` example program above could use the `INTENT` attribute inside its function as follows;

```
REAL, INTENT(IN) ::  d,e !  *** Dummy variables
```

1.7 Assumed shape arrays

Arrays can be passed to procedures just as any other data object. One of the neatest ways to do this is using an 'assumed shape' in the dummy argument declaration. This is done in the '`DIMENSION`' attribute part of the declaration. For each rank in the '`DIMENSION`' attribute, instead of explicitly giving a size for that dimension a colon is explicitly written instead. Please do **exercise six** on the next page before reading the next section.

1.8 Automatic arrays

Arrays, inside a procedure, can also be declared where their extent in any dimension can depend on dummy arguments. These are called automatic arrays ie.

```
FUNCTION example(mat1,mat2,m,n)
  REAL, DIMENSION(:,:), INTENT(IN)  :: mat1 !** Dummy
  REAL, DIMENSION(:,:), INTENT(IN)  :: mat2 !** Dummy
  INTEGER :: m,n !** Dummy

  REAL, DIMENSION(SIZE(mat1,1),SIZE(mat2,2))  :: mat3
  REAL, DIMENSION(m,n)  :: mat4
```

So the local array '`mat3`' is declared such that its extent in the 1st dimension is the same as the extent of the 1st dimension of the dummy variable '`mat1`' and its extent in the 2nd dimension is the same as the extent of the 2nd dimension of the dummy variable '`mat2`'

The '`SIZE`' function

The '`int=SIZE(arr,dim)`' function returns an integer that holds the extent in the dimension `dim` of the array `arr`. This is very useful as it can be used in local declarations to base the extents of local arrays on the size of dummy variables.

Exercise Six : In a 'handout5/exercise6' Type in the following code to demonstrate the use of assumed shape arrays.

```
PROGRAM assume
! *** Example Program to demonstrate assumed shape arrays

  IMPLICIT NONE

  INTEGER :: i,j
  INTEGER, PARAMETER :: m=5,n=6
  REAL, DIMENSION(m,n) :: array ! Create an array
  REAL :: ans

  DO i=1,m ! *** Assign the array elements a value
    DO j=1,n
      array(i,j)=i+j !** Value equals sum of indices.
    ENDDO
  ENDDO

  ans=sumarr(array,m,n)

  PRINT '("Summation of all elements = ",f10.4)',ans

CONTAINS

  FUNCTION sumarr(aa,k,p)

    REAL, INTENT(IN), DIMENSION(:,:) :: aa ! *** Dummy variable
    INTEGER, INTENT(IN) :: k,p             ! *** Dummy variables
    REAL :: sumarr                         !** Function declaration

    sumarr=0  ! *** Set initially to be zero
    DO i=1,k
      DO j=1,p
        sumarr=sumarr+aa(i,j)
      ENDDO
    ENDDO

  END FUNCTION sumarr

END PROGRAM assume
```

.

Class Project :: Part Two : In a new directory 'class_project/part2' copy your code from 'class_project/part1' and edit it so that the array input, output and array multiplication are all done in appropriate separate procedures. Call the three required procedures MULMAT, GETMAT & OUTMAT.

Hints :

1. You should make use of automatic arrays and the SIZE function.

2. Build the new code up bit by bit. ie. write the GETMAT (function to read in a matrix) first and make sure it works before going on to the MULMAT function