Handout Five

October 22, 2013

1 Internal Functions & Subroutines

The main program unit can accommodate ('CONTAINS') two types of procedures (often called subprograms). These can be used to break up a large program and structure it to become more readable and efficient. Like the fortran intrinsic functions you have seen procedures can be repeatedly called to execute a series of statements that execute an often used task. This saves typing a sequential set of statements repeatedly in your code. Internal procedures 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 as shown below.

```
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`'. Program execution then moves into the subroutine '`calc`' which prompts the user to input values for the two variables '`a`' & '`b`' and then calculates $\sqrt{a^2 + b^2}$ and the result is assigned to the variable '`c`'. Program execution then moves back into the main program (following the '`CALL calc`') and continues on the next line where the result of the calculation is printed to the screen. If required '`calc`' could be called as many times as is necessary 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`' and edit it so that it appears as below. Compile and run the code.

```
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 argument declarations

    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. However, when using **internal** procedures it is recommended to always pass variables from the program unit through to the procedure declaring the dummy variables with the same names as those in the calling program unit (in this case the main program). This prevents the programmer 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 :** Variables are defined as **local** to a procedure if they have **NOT** been passed into that procedure via the argument list as dummy variables but they are declared locally and used inside the procedure. This means they are local to the procedure in which are initially created. Variables declared in the main program unit are called local variables to the main program.
- **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. The subroutine call;

```
CALL calc(a,b,c)       !** Call to subroutine "calc"
```

For the subroutine header

```
SUBROUTINE calc(d,e,f) !** Header line for subroutine "calc"
!** d is the dummy variable for a
!** e is the dummy variable for b
!** f is the dummy variable for c
```

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 data 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 by the dummy variable so it can 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 that 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 large 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 be 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 to protect them from being changed inadvertently by another part of the code. This protection can in part 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 variable declared in the main program unit are completely independent. Therefore changing the value of **local** variable called '`num1`' would not inadvertently change the value of a variable

called ‘num1’ in the main program unit.

Exercise Three : In a ‘handout5/exercise3’ directory copy the code from ‘exercise2’ edit it so that it appears as below. Compile and run the code.

```
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
```

In the above code both the main program and the internal subroutine ‘calc’ have declared ‘local’ variables with the same name ‘a’, ‘b’ & ‘c’. Changing the values of ‘a’, ‘b’ & ‘c’ in the internal procedure has no effect on the variables ‘a’, ‘b’ & ‘c’ in the main program.

.

Exercise Four : In a '`handout5/exercise4`' directory copy the code from '`exercise3`'edit it so that it appears as below. Compile and run the code.

```
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 in a similar fashion to subroutines except, as hinted at by their name, they return a value. This **return value** is declared inside the function, just as any other data type would be, and has the same name as that of the actual function. Inside the function the return value is then assigned the approprate result 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 used in a '`PRINT*,calc(d,e,f)`" 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.

```
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 can make use of the value but 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 prescribed value on entry into the procedure and that value may be used and 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
```

2 Second look at arrays.

So far you have learnt how to declare arrays and you have made use of '`REAL`' arrays to represent matrices in your matrix library module, where the first dimension indexes the 'rows' of the matrix and the second dimension indexes the 'columns'.

```
REAL, DIMENSION(4,5) :: mat1 !** 2D array to represent the matrix "mat1"
```

2.1 Array terminology

Here is a useful glossary of terms associated with arrays that will help you to understand later descriptions and information you may read in other texts.

- **size** - refers to the total number of elements in an array or if a 'dimension' is specified, then it refers to the number of elements in the specified dimension.
- **rank** - the total number of dimensions in the array ie. our class project deals with matrices that are represented by '`REAL`' arrays of rank two.
- **extent** - the number of elements in a particular dimension. For example, in the declaration above, '`mat1`' has an extent of four in its first dimension and an extent of five in its second dimension.
- **shape** - this refers collectively to both the rank and extents of an array. For arrays to have the same 'shape' they must have the same number of dimensions and the same 'exent' in each corresponding dimension.

  So to pass on the 'shape' of an array we need to pass on the number of dimensions and the number of elements in each dimension.
- **conformable** - Two, or more, arrays can be described as conformable if they have the same 'shape' (remember knowing the shape of an array means knowing the number of dimensions and also the extent in each dimension). Therefore to perform the standard fortran operations of addition, subtraction, multiply and divide the arrays must conform.

2.2 Array references

An individual array element can be referenced by supplying a valid index to each of the dimensions in the array. For example '`mat1(2,3)`' would return the value of the element in the second row and the third column of the matrix '`mat1`'. It is often useful to reference a sub-section of an array, for example consider the following bit of code.

```
REAL, DIMENSION(6,6) :: mat !** 2D array
REAL, DIMENSION(2,3) :: submat !** 2D array

!** Assume some code here to read values into our array "mat"

submat=mat(2:3,3:5) ! ** Set submat to be a subsection of the matrix "mat"
```

The array '`submat`' is declared to be a rank two array with an extent of two in the first dimension and three in the second dimension. The array '`mat`' is declared to be a rank two array with an extent of six in the first dimension and six in the second dimension. Now assume we have assigned values to all the

elements of the array '`mat`'. Next the array '`submat`' is assigned the values of a sub-section of the array '`mat`', specifically '`submat`' is set to be equal to the elements referenced by the rows two and three, and the columns three four and five of the matrix stored in the array '`mat`'. NOTE how the 'shape' of both sides of the assignment 'conform'. That is the extents in both dimensions of '`submat`' equal the extents referenced in both dimensions of '`mat`'.

2.3 Array construction

Arrays like other data objects can be constructed in your code at run-time. For example the following bit of code declares a one dimensional array '`aa`' with an extent of ten and then assigns to it the values one through to ten, one in index one, two in index two, three in index three and so on.

```
INTEGER, DIMENSION(10) :: aa !** 1D array

aa=(/1,2,3,4,5,6,7,8,9,10/)
PRINT '("The array aa = ",10i3)',aa
```

NOTE the use of a formatted '`PRINT`' statement used to print out the array in a more appropriate fashion.
The main restriction to this method of array construction is that it can only be used for arrays of rank one. For arrays of higher dimensions the '`RESHAPE`' intrinsic function must be used in conjuction with the above.

2.4 Array Syntax and Expressions

The standard Fortran arithmetic operators $(+, -, *, /)$ etc can be used on two arrays of identical **shape** (i.e. arrays are **conformable** see section (2.1)) to produce a result array of identical shape. This array arithmetic is straightforward and works on an element by element basis. So if array '`array1`' is to be added to '`array2`', given they are conformable, then each element of the result array will be the sum of the two corresponding elements of '`array1`' and '`array2`'. See the two examples below where in both expressions the two array operands are conformable.

$(1, 2, 4, 3) + (2, 5, 2, 1) = (3, 7, 6, 4)$

$(2, 1) * (4, 2) = (8, 2)$

If an array operation involves a scalar then the scalar operates on every element in the array eg.

$(1, 2, 3) * 3 = (3, 6, 9)$.

Subsections of arrays can also be used in expressions as long as each operand in the array expression conforms to the same shape. For example,

```
aa=bb(2:4,6:8)-cc
```

is a valid operation as long as '`cc`' and '`aa`' are arrays of shape '`(/3,3/)`' and '`bb(2:4,6:8)`' is a valid reference for the array '`bb`'.

2.5 Array and Procedures

Arrays can be passed to procedures/sub-programs just as any other data type and this can be done in a few different ways. Also '`local`' arrays can be declared inside procedures where their extents depend on information contained in dummy arguments being passed into the procedure.

2.5.1 Explicit shape dummy arrays

These arrays appear in the dummy argument list for a procedure. The extents of their dimensions are explicitly declared by using dummy arguments passed through the argument list. For the purpose of this course we will not be using dummy arrays of this type instead we will be making use of the assumed shape dummy arrays explained in the next section.

2.5.2 Assumed shape dummy arrays

A neat method of passing dummy arrays is by using an **assumed shape** array 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 specifying a size for that dimension, a colon is used its place. The **rank** of the array being passed through must match that of the dummy argument in the procedure. Please do **exercise six** on the next page before reading the next section.

2.5.3 Automatic arrays

An automatic array is a **local** array in a procedure. It has its **shape** set inside the procedure using information passed through the procedure's argument list. The dimension bounds may be integer expressions involving any variables accessible at that point: normally this means other dummy variables. They are local to a procedure and they cannot be given the '`SAVE`' attribute (Explained later in the course).

2.5.4 Example demonstrating arrays and procedures

See the example code below showing the declaration part of a function using assumed shape and automatic arrays.

```
FUNCTION example(mat1,mat2,m,n)

  REAL, DIMENSION(:,:), INTENT(IN)  :: mat1 !** Dummy (assumed shape)
  REAL, DIMENSION(:,:), INTENT(IN)  :: mat2 !** Dummy (assumed shape)
  INTEGER :: m,n !** Dummy

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

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'

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
  REAL :: ans

  DO i=1,m ! *** Assign the array elements a value
    DO j=1,n
      array(i,j)=i+j
    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

    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
```

2.6 The 'SIZE' function

The 'SIZE' function returns an 'INTEGER' type. The return value is the extent (number of elements) in a given dimension or the **total** number of elements in the array. The 'SIZE' function can take two arguments. The first argument is an **array** type whose 'SIZE' is to be queried. The second argument is optional, if present it is an 'INTEGER' type denoting the dimension in the array that is to be queried.

```
num1=SIZE(array,dim)}
num2=SIZE(array}
```

The first example above returns the extent in the dimension 'dim' of the array 'array' and assigns that value to 'num1'. The second example returns the total number of elements in the array 'array' and assigns that value to 'num2'. This function is often used in automatic local array declarations to declare the extents of local arrays based the size of some dummy array variables.

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

---