Computer Science 1510

Lecture 10

January 27, 2016

Lecture Outline

- Functions

Subprograms

- Algorithms to solve larger problems should be broken down into several sub-algorithms, each of which solves a smaller sub-problem.

- This way, each sub-algorithm may be tested separately to ensure that it works correctly before being incorporated into the larger algorithm.

- We can write sub-programs to implement such sub-algorithms.

- There are two types of sub-programs in Fortran, one called a `FUNCTION`, and a second called a `SUBROUTINE`.

- The general distinction between the two is that a `SUBROUTINE` is used to perform a section of an algorithm where many variable values may change, while a `FUNCTION` is provided with values and returns a single value.

Functions

- We have seen some of the built-in functions available in Fortran (`sin`, `cos`, `exp`, etc.).

- We can also write our own (ie. programmer-defined) functions and *call* them as we would for built-in functions.

- Syntax:

```
Function heading
   variable-declarations
   body-of-function
   RETURN
END FUNCTION function-name
```

- `Function heading` is one of:

```
FUNCTION function-name(formal-argument-list)
                    or
type-identifier FUNCTION function-name(formal-argument-list)
```

  where `function-name` is any legal Fortran identifier, `formal-argument-list` is a comma separated list of identifiers, and `type-identifier` is the type of the data returned by the function.

Functions

- In the first form of the function heading, the type of the data returned by the function must be declared in the variable declarations section inside the function.

- The `formal-argument-list` contains a list of variables that are *passed into* the function.

- Variable declarations within a `FUNCTION` consist of:

  1. Arguments - Variables passed into the `FUNCTION`.
  2. Variables local to the function. These variables are seen only by the `FUNCTION` and not by the main `PROGRAM`.
  3. If the first form of the function heading is used, a variable with the same name as `function-name` must be declared.

- The variable with the same name as `function-name` is assigned a value which is returned to the main `PROGRAM` when the `RETURN` or `END FUNCTION` statement is reached.

Functions

- Arguments passed into the function should have an `INTENT` specifier added to their declarations, which specifies how the arguments are to transfer information.

- For example,

  ```
  REAL,INTENT(IN)::average
  ```

  specifies that the value of `average` is not to be changed within the function.

- The body of a function has the same form as the body of a main program, with the additional requirement that a value be assigned to the variable `function-name`.

- That is, an assignment statement like the following must be included:

  ```
  function-name=expression
  ```

Example: Temperature conversion function

```
FUNCTION Fahr_to_Celsius(temp)
   REAL::Fahr_to_Celsius
   REAL,INTENT(IN)::temp
   Fahr_to_Celsius = (temp-32.0)/1.8
   RETURN
END FUNCTION Fahr_to_Celsius
```

A subprogram can be made accessible to a program in three ways:

1. Placed inside a main program (ie. before the `END PROGRAM` statement) – *Internal subprograms*.

2. Placed inside a module which is imported into a program (see later) – *Module subprograms*.

3. Placed after the `END PROGRAM` statement – *External subprograms*.

Example: Temperature conversion

```
PROGRAM Temperature_conversion
!-----------------------------------------------------------------
! The following program converts several Fahrenheit temperatures
! input by the user to Celsius.  Function Fahr_to_Celsius is used
! to compute the conversion.
!  INPUT:   fahr - temperature in Fahrenheit
!  OUTPUT:  cel - temperature in Celsius
!-----------------------------------------------------------------
   IMPLICIT NONE
   REAL::fahr,cel
   CHARACTER::next

   DO
      WRITE(*,*) 'Enter a temperature in Fahrenheit'
      READ(*,*) fahr
      ! Convert the temperature to Celsius
      cel=Fahr_to_Celsius(fahr)
      WRITE(*,*) fahr,'in Fahrenheit is',cel,'in Celsius'
      WRITE(*,*) 'Are there more temperatures to convert? [y/n]'
      READ(*,*) next
      IF (next/='y') EXIT
   END DO

CONTAINS
   FUNCTION Fahr_to_Celsius(temp)
   !-----------------------------------------------------------
   ! This function converts a temperature from Fahrenheit to Celsius
   !  ACCEPTS:  temp - a temperature in Fahrenheit
   !  RETURNS:  the corresponding temperature in Celsius
   !-----------------------------------------------------------
      REAL::Fahr_to_Celsius
      REAL,INTENT(IN)::temp
      Fahr_to_Celsius = (temp-32.0)/1.8
      RETURN
   END FUNCTION Fahr_to_Celsius
END PROGRAM Temperature_conversion
```

Argument Association

- A *call* or *reference* to a function has the form:

  `function-name(actual-argument-list)`

- Arguments in the `actual-argument-list` do not have to have the same names as the arguments in the `formal-argument-list`.

- That is, arguments passed into a subprogram are not required to have the same names as the corresponding arguments inside the function.

- When using a function, the intent is that the values of the arguments passed to the function do not change inside the function.

- The `INTENT(IN)` specification ensures that this is the case. When a variable is declared as `INTENT(IN)`, the value of the actual argument is passed to the corresponding formal argument, and the value of the formal argument cannot change within the function.

Argument Association

- If `INTENT(IN)` is not specified for a formal argument, then the value of that formal argument can be changed inside the function, with the same change to the actual argument.

- For example, if we had not included the `INTENT(IN)` specification for `temp` in the program above, then we could assign a different value to `temp` within the function (for example, `temp=0.0`). When the function returned, the value of `fahr` in the main program would be 0.

- The number and type of actual arguments must agree with the number and type of formal arguments.

Functions of several variables

- In the above example, the function `Fahr_to_Celsius` has only a single argument. However, functions can have many arguments.

- For example, the following function computes the value of

$$f(x,y,n) = \begin{cases} x^n + y^n & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$$

```
FUNCTION F(x,y,n)
   REAL::F
   REAL,INTENT(IN)::x,y
   INTEGER,INTENT(IN)::n
   IF (x>=y) THEN
      F=x**n + y**n
   ELSE
      F=0.0
   END IF
END FUNCTION F
```

Using FUNCTIONS

- As for built-in functions, programmer-defined functions can be used in any place where a single value can be used.

- For example, assignment statements, arithmetic expressions, logical expressions, or loop conditions.

- Examples:

```
X=AVERAGE(1,2,3)
Y=X+AVERAGE(A,B,C)
IF (X > AVERAGE(A,B,3))
DO WHILE (AVERAGE(A,B,C) > D)
```

- In each case, when `AVERAGE` is reached, control is transferred to the beginning of the `FUNCTION`, and the body of the function is executed. When `RETURN` or `END PROGRAM` is reached, the value of `AVERAGE` is returned and used in place of the function call.

Example: Multiple functions

```
PROGRAM Poisson_Probability
!-------------------------------------------------------------------------
! Program to calculate the Poisson probability function using the
! function subprogram Poisson.  Identifiers used are:
!   AveOccurs   : average # of occurrences of phenomenon per
!                 time period
!   NumOccurs   : number of occurrences in a time period
!   Probability : Poisson probability
!   NumProbs    : number of probabilities to calculate
!   I           : DO-loop control variable
!   Poisson     : internal function to calculate Poisson probability
!   Factorial   : internal function to calculate factorials
! Input:   NumProbs and values for AveOccurs and NumOccurs
! Output:  Poisson probabilities
!-------------------------------------------------------------------------
  IMPLICIT NONE
  REAL :: AveOccurs, Probability
  INTEGER :: NumProbs, I, NumOccurs

  WRITE(*,*) 'This program calculates Poisson probabilities.'
  WRITE(*, '(1X, A)', ADVANCE = "NO") &
         'How many probabilities do you wish to calculate? '
  READ *, NumProbs

  DO I = 1, NumProbs
     WRITE(*, '(1X, A)', ADVANCE = "NO") &
           'Enter average # of occurrences per time period: '
     READ(*,*) AveOccurs
     WRITE(*, '(1X, A)', ADVANCE = "NO") &
           'Enter # of occurrences for which to find probability: '
     READ(*,*) NumOccurs
     Probability = Poisson(AveOccurs, NumOccurs)
     WRITE(*,'(1X, "Poisson probability = ", F6.4 /)') Probability
  END DO
```

```
CONTAINS
  !-Poisson ------------------------------------------------------------
  ! Function to calculate the Poisson probability
  !                              N   -Lambda
  !                       Lambda * e
  !           Poisson(N) =  ------------------
  !                              N!
  ! Function Factorial is called to calculate N!
  !
  ! Accepts:  Lambda - average number of occurrences per time period
  !           N - number of occurrences in that time period
  ! Returns:  The Poisson probability given by the formula above
  !---------------------------------------------------------------------
  FUNCTION Poisson(Lambda, N)
    REAL :: Poisson
    REAL, INTENT(IN) :: Lambda
    INTEGER, INTENT(IN) :: N
    Poisson = (Lambda ** N * EXP(-Lambda)) / REAL(Factorial(N))
  END FUNCTION Poisson

  !- Factorial ----------------------------------------------------------
  ! Function to calculate the factorial N! of N which is 1 if N = 0,
  ! 1 * 2 * . . . * N if N > 0.
  !
  ! Accepts:  Integer N
  ! Returns:  The integer N!
  !---------------------------------------------------------------------
  FUNCTION Factorial(N)

     !--------------------
     !  See Assignment 2
     !--------------------

  END FUNCTION Factorial

END PROGRAM Poisson_Probability
```

Scope of variables

- *Scope* refers to where a variable is accessible and can be used.

- The scope of a variable is the program or subprogram in which it is declared.

- For example, a subprogram may require variables in addition to the formal arguments. Such *local variables* declared within a subprogram can only be accessed and used within that subprogram, and are said to have *local scope*.

- When control is passed from a subprogram back to a main program all local variables are lost.

- Variables declared in a main program are global to the entire program, including any internal subprograms (except within subprograms that have local variables with the same name, in which case the local variable has precedence).

Scope of variables

- Since programs and subprograms are compiled independently and recombined at the linking stage, variable names and labels can be reused without causing a conflict.

- Each program or subprogram has its own private memory assigned to it, called a *stack*.

- If two subroutines have a local variable with the same name, these variables do not conflict since each is located within the subroutines own stack.

- What about when we want subprograms to share information?

Saving local variables

- The values of local variables in a subprogram are not retained from one execution to the next unless,

  1. they are initialized in their declarations, or
  2. they are declared using the SAVE attribute.

- Example: the following will result in `Count` having the same value on each call to function `F`.

```
INTEGER FUNCTION F(...)
    INTEGER :: Count
    ...
    Count = Count + 1
    ...
END FUNCTION F
```

- Changing the declaration to either,
  `INTEGER, SAVE :: Count` or
  `INTEGER :: Count = 0` will ensure that the value of `Count` will be saved from one call to the next.

External subprograms

- As noted previously, function subprograms can be either internal, module, or external subprograms.

- External subprograms can be placed in the same source file as a main program, after the END PROGRAM statement.

- An advantage to external subprograms is that they can be used by other programs that need the particular functionality.

- In the case of internal subprograms, a program is aware of what arguments are required, and can thus check that the subprogram is being used properly.

- However, in the case of an external subprogram, the main program contains no definition of what the subprogram looks like, and can therefore not check whether it is being used properly.

- To overcome this, an explicit interface should be included in the main program.

Fortran: INTERFACE

- In the main program we include an `INTERFACE` block at the beginning of the program.

- The `INTERFACE` block tells the main program what function is present, what type of arguments it takes, and what the type of the output variable is.

- Syntax:

```
INTERFACE
   interface-body
END INTERFACE
```

  The `interface-body` contains:

  1. The subprogram heading (with possible different names for the formal arguments).
  2. Declarations of the arguments and the result type in the case of a function.
  3. An END statement.

- Note that there are no executable statements in an `INTERFACE` block.

Example: Explicit interface

```
PROGRAM Temperature_Conversion_4
!-----------------------------------------------------------------
! The following program converts several Fahrenheit temperatures
! input by the user to Celsius.  Function Fahr_to_Celsius is used
! to compute the conversion.
!  INPUT:   fahr - temperature in Fahrenheit
!  OUTPUT:  cel - temperature in Celsius
!-----------------------------------------------------------------
   IMPLICIT NONE

   INTERFACE
      FUNCTION Fahr_to_Celsius(Temp)
         REAL:: Fahr_to_Celsius
         REAL, INTENT(IN) :: Temp
      END FUNCTION Fahr_to_Celsius
   END INTERFACE

   REAL::fahr,cel
   CHARACTER::next

   DO
      WRITE(*,*) 'Enter a temperature in Fahrenheit'
      READ(*,*) fahr
      ! Convert the temperature to Celsius
      cel=Fahr_to_Celsius(fahr)
      WRITE(*,*) fahr,'in Fahrenheit is',cel,'in Celsius'
      WRITE(*,*) 'Are there more temperatures to convert? [y/n]'
      READ(*,*) next
      IF (next/='y') EXIT
   END DO

END PROGRAM Temperature_Conversion_4
```

```
!-Fahr_to_Celsius ----------------------------------------------------
! Function to convert a Fahrenheit temperature to Celsius.
!   Accepts:  A temperature Temp in Fahrenheit
!   Returns:  The corresponding Celsius temperature
!---------------------------------------------------------------------

FUNCTION Fahr_to_Celsius(Temp)
   REAL:: Fahr_to_Celsius
   REAL, INTENT(IN) :: Temp
   Fahr_to_Celsius = (Temp - 32.0) / 1.8
END FUNCTION Fahr_to_Celsius
```