Computer Science 1510

Lecture 17

Lecture Outline

- Modules
- Recursion

Example 2: MODULE

```
MODULE Temperature
!-----------------------------------------------------------------------
! Module that contains the following subprograms for processing
! temperatures on various scales:
!    Fahr_to_Celsius:  a Fahrenheit-to-Celsius conversion function
!    Celsius_to_Fahr:  a Celsius-to-Fahrenheit conversion function
!-----------------------------------------------------------------------
   IMPLICIT NONE

CONTAINS
   !-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


   !-Celsius_to_Fahr----------------------------------------------------
   ! Function to convert a Celsius temperature to Fahrenheit.
   !   Accepts:  A temperature Temp in Celsius
   !   Returns:  The corresponding Fahrenheit temperature
   !-----------------------------------------------------------------------
   FUNCTION Celsius_to_Fahr(Temp)
      REAL:: Celsius_to_Fahr
      REAL, INTENT(IN) :: Temp
      Celsius_to_Fahr = 1.8 * Temp + 32.0
   END FUNCTION Celsius_to_Fahr

END MODULE Temperature
```

Example 2: MODULE

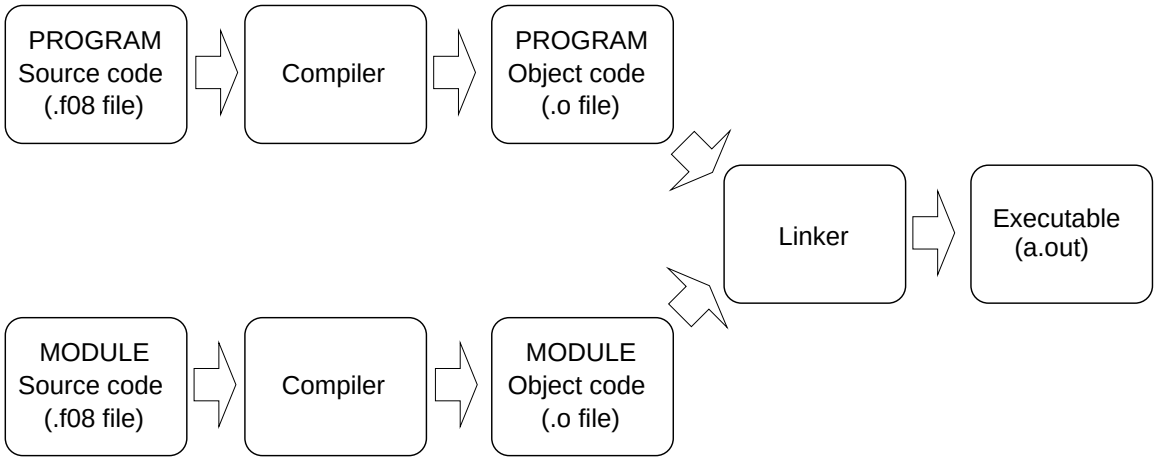
```
PROGRAM Temperature_Conversion_2
!-----------------------------------------------------------------
! The following program converts several Fahrenheit temperatures
! input by the user to Celsius.  Function Fahr_to_Celsius is used
! to compute the conversion.
!  INPUT:   far - temperature in Fahrenheit
!  OUTPUT:  cel - temperature in Celsius
!-----------------------------------------------------------------
   USE Temperature
   IMPLICIT NONE
   REAL::far,cel
   CHARACTER::next

   DO
      WRITE(*,*) 'Enter a temperature in Fahrenheit'
      READ(*,*) far
      ! Convert the temperature to Celsius
      cel=Fahr_to_Celsius(far)
      WRITE(*,*) far,'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_2
```

Compiling a MODULE

- In the case where a MODULE and a PROGRAM that uses the MODULE are in different source files, the compilation process can be viewed as follows.



- Compilation can be done in several ways:
  - A single step:

```
gfortran temp_mod.f08 temperature.f08
```

  - Produce object files and link together:

```
gfortran -c temp_mod.f08
gfortran -c temperature.f08
gfortran temp_mod.o temperature.o
```

Access

- Items within a module are either PUBLIC or PRIVATE.

- Items that are PRIVATE are known only within the module.

- Items that are PUBLIC can be accessed by any program or subprogram that uses the module.

- A module with a PRIVATE statement indicates that all items that are not declared PUBLIC will be PRIVATE.

```
MODULE name
   IMPLICIT NONE
   PRIVATE
   REAL,PUBLIC,PARAMETER :: pi=3.14159
   !...
END MODULE name
```

- The PRIVATE attribute is used to hide implementation details that do not need to be known outside of the module.

Access

- To make the subprograms within a PRIVATE module accessible, we list them as in,

```
MODULE name
   IMPLICIT NONE
   PRIVATE
   !...
   PUBLIC :: my_func1, my_func2
CONTAINS
   FUNCTION my_func1()
      ! ...
   END FUNCTION my_func1
   FUNCTION my_func2()
      ! ...
   END FUNCTION my_func2
END MODULE name
```

Access

- A `PUBLIC` variable within a module can have its access partially restricted by using the `PROTECTED` attribute. For example,

```
INTEGER, PUBLIC, PROTECTED :: my_int=3
```

- The `my_int` variable can be accessed by a program that uses the module, but its value cannot be changed.

- When a program uses a module, there can be issues with conflicting variable names, or in some cases, a different variable name than in the module is preferred. This can be indicated within the `USE` statement:

```
USE math_module, logarithm_base => e
```

  Here, the variable `e` within the module called `math_module` will be called `logarithm_base` within the program that uses the module.

Access

- It is also possible to specify exactly which variables to be used from a module by using the `ONLY` attribute.

- For example,

```
USE math_module, ONLY : pi
```

Recursion

- Up to now, all of our references to functions have involved a main program calling a function, or a function calling another function.
- It is possible to have a function call itself.
- This is known as *recursion*.
- Consider computing $n!$.
- Once we have computed a factorial, $5!$ for example, it is easy to calculate $6!$ since $6! = 6 \cdot 5!$.
- In general $n! = n(n-1)!$ (Note that $0! = 1$).
- This leads to the recursive definition,

$$\begin{aligned} 0! &= 1 \\ \text{For } n > 0, \quad n! &= n(n-1)! \end{aligned}$$

Recursion

- In general, a function is defined recursively if it includes:

  1. A *base case*, in which the value of the function is specified for one or more values of the argument.
  2. A *recursive step*, in which the value of the function for the current value of the argument is defined in terms of previously defined function values and/or argument values.

- In the case of the factorial function, the base case is $0! = 1$, while the recursive step is $n! = n(n-1)!$.

- To define a recursive subprogram in Fortran, we attach the keyword RECURSIVE to the subprogram heading, as well as a RESULT clause.

- For example,

```
RECURSIVE FUNCTION Factorial(n) RESULT (Fact)
```

Example 1: Recursion

```
PROGRAM Test_recursion
    IMPLICIT NONE
    INTEGER :: n
    WRITE(*,*) 'Enter a value for n'
    READ(*,*) n
    WRITE(*,*) 'Factorial = ',Factorial(n)

CONTAINS

    RECURSIVE FUNCTION Factorial(n) RESULT (Fact)
        INTEGER :: Fact
        INTEGER, INTENT(IN) :: n

        IF (n==0) THEN
            Fact = 1
        ELSE
            Fact = n * Factorial(n-1)
        END IF
    END FUNCTION Factorial

END PROGRAM Test_recursion
```

Example 2: Recursion

```
PROGRAM Euclid
   IMPLICIT NONE
   INTEGER::m,n
   WRITE(*,*) 'Enter two integers m and n, where m>n'
   READ(*,*) m,n
   ! Call recursive function to compute the GCD
   WRITE(*,*) 'The greatest common divisor is ',GCD(m,n)

CONTAINS

    RECURSIVE FUNCTION GCD(a,b) RESULT (factor)
        INTEGER,INTENT(IN)::a,b
        INTEGER::factor,q,r
        q = a/b
        r = a-q*b
        IF (r==0) THEN
            factor = b
        ELSE
            factor = GCD(b,r)
        END IF
    END FUNCTION GCD

END PROGRAM Euclid
```

Recursion

- Note that the name of the RESULT variable is declared instead of the name of the function. This can be done for any function.

- In many cases, a nonrecursive algorithm may be easier to implement than a recursive algorithm.

- This is the case with the factorial function.

- In other cases, a nonrecursive algorithm may not be obvious or easy.