# Computers in Engineering COMP 208

Functions

Michael A. Hawker

## Functions in FORTRAN

- There are many useful operations which are not part of the basic instruction set of the computer
- FORTRAN provides many such functions for our use. We have seen some of these "intrinsic" or "predefined" functions such as

```
sqrt(x), exp(x), mod(x,y)
```

- FORTRAN also allows us to define our own new functions

## A Factorial Function

```
INTEGER FUNCTION Factorial(n)
  IMPLICIT NONE
  INTEGER :: n
  INTEGER :: i, Fact
  Fact = 1
  DO i = 1, n
    Fact = Fact * i
  END DO
  Factorial = Fact
END FUNCTION Factorial
```

## Function Definition Syntax

## Syntax of a function definition

```
type FUNCTION function-name
                   (arg1, arg2, ..., argn)
   IMPLICIT  NONE
   [declarations]
   [statements]
END FUNCTION function-name
```

# A Function Definition

```
INTEGER FUNCTION Factorial(n)
  IMPLICIT NONE
  INTEGER :: n
  INTEGER :: i, Fact
  ! **** body suppressed to save space ****
END FUNCTION Factorial
```

The keyword FUNCTION tells the compiler that we are defining a function

# A Function Definition

```
INTEGER FUNCTION Factorial(n)
  IMPLICIT NONE
  INTEGER :: n
  INTEGER :: i, Fact
  ! **** body suppressed to save space ****
END FUNCTION Factorial
```

The function computes a value to be used. We must specify the type of that value

This type specification precedes the word FUNCTION

## Naming the Function

```
INTEGER FUNCTION Factorial(n)
  IMPLICIT NONE
  INTEGER :: n
  INTEGER :: i, Fact
  ! **** body suppressed to save space ****
END FUNCTION Factorial
```

We must name any function we define so that we can refer to in when we use it

The name we give the function follows the keyword FUNCTION

## Function Parameters

```
INTEGER FUNCTION Factorial(n)
  IMPLICIT NONE
  INTEGER :: n
  INTEGER :: i, Fact
  ! **** body suppressed to save space ****
END FUNCTION Factorial
```

The variables inside parenthesis that follow the function name are called parameters or formal arguments

Some functions have no parameters. We still need parentheses but there are no variables

## Function Parameters

```
INTEGER FUNCTION Factorial(n)
  IMPLICIT NONE
  INTEGER :: n
  INTEGER :: i, Fact
  ! **** body suppressed to save space ****
END FUNCTION Factorial
```

The parameter types must be declared inside the function definition

## Local Variables

```
INTEGER FUNCTION Factorial(n)
  IMPLICIT NONE
  INTEGER :: n
  INTEGER :: i, Fact
  ! **** body suppressed to save space ****
END FUNCTION Factorial
```

Other variables used in the computation must also be declared.

These are called **local variables**

## Ending a Function Definition

```
INTEGER FUNCTION Factorial(n)
  IMPLICIT NONE
  INTEGER :: n
  INTEGER :: i, Fact
  ! **** body suppressed to save space ****
END FUNCTION Factorial
```

The definition terminates with END FUNCTION followed by the name of the function.

## Semantics – Function Body

```
INTEGER FUNCTION Factorial(n)
   ! *** declarations suppressed to save space
   ***
   Fact = 1
   DO i = 1, n
     Fact = Fact * i
   END DO
   Factorial = Fact
END FUNCTION Factorial
```

The body of a function is basically a FORTRAN program that tells the computer how to process the data values.

# Semantics – Function Body

```
INTEGER FUNCTION Factorial(n)
  ! *** declarations suppressed to save space
  ***
  Fact = 1
  DO i = 1, n
    Fact = Fact * i
  END DO
  Factorial = Fact
END FUNCTION Factorial
```

When using a function, we must provide a value for any parameters (in this example, n)

The statements of the function body are executed using these values for the parameters

# Semantics – Return Value

```
INTEGER FUNCTION Factorial(n)
  ! *** code suppressed to save space ***
  Factorial = Fact
END FUNCTION Factorial
```

A function computes a value.

It must provide this value to the expression that used the function

In Fortran, it returns the result by assigning the value to be returned to a variable that has the same name as the function name

## Semantics – Return Value

```
INTEGER FUNCTION Factorial(n)
  ! *** code suppressed to save space ***
  Factorial = Fact
END FUNCTION Factorial
```

To return the value, the function definition must have one or more assignments of the form:

**`function-name = expression`**

The type of the expression must be the same as the type of the function

## Semantics – Return Value

```
 INTEGER FUNCTION Factorial(n)
   ! *** code suppressed to save space ***
   Factorial = Fact
END FUNCTION Factorial
```

The function name is called a dummy or pseudo variable

It is not a true variable because it does not have a memory cell allocated to it

Therefore the name of the function should not appear as a variable in any other expression

# Example – Function Definition

```
INTEGER FUNCTION Minimum (x, y, z)
  IMPLICIT NONE
  INTEGER :: x, y, z

  IF (x <= y .AND. x <= z) THEN
    Minimum = x
  ELSE IF (y <= x .AND. y <= z) THEN
    Minimum = y
  ELSE
    Minimum = z
  END IF

END FUNCTION Mimimum
```

## Defining vs. Using Functions

- As mentioned some functions (e.g. sqrt) have been predefined for us
- We have seen how to define new functions that are not intrinsic
- We only have to define a function once and then we can use it as often as we wish
- How do we use the function we defined?

## Using Functions

- User-defined function are used in the same way as Fortran intrinsic functions.
- They can appear as part of any expression
- When we use them, we must provide values for the arguments
- The function returns a value and that value is used in evaluating the rest of the expression

# The Semantics of Using Functions

- If **`func`** is the name of a function with parameters $p_1$, $p_2$ and $p_3$, we can write **`func(`**$e_1$**`,`** $e_2$**`,`** $e_3$**`)`** in an expression
- This uses the function to compute a value
- The value is then used in the expression
- Think of how we have used intrinsic functions like mod, sqrt and others

## The Semantics of Using Functions

What happens when the program evaluates
**`func(`$e_1$`, `$e_2$`, `$e_3$`)`**?

- The arguments $e_1$, $e_2$ and $e_3$ are evaluated
- It uses the values of these arguments to initialize the parameters
- The computer evaluates the function, using the function definition provided
- When it reaches the end of the function evaluation, it returns the value obtained

## Examples of Using Functions

```
! compute the square root of discriminant d

  d = SQRT(b*b - 4.0*a*c)

if ((mod(year,4) == 0 .and. mod(year,100) /= 0) .or.
            mod(year,400) == 0) then
      lastday = 29
```

## Prime Numbers

- Applications that use cryptography, random number generators, simulation, hashing and others require prime numbers
- What is a prime number?
- A positive integer is prime if it has no proper divisors (i.e. the only divisors are 1 and the number itself)
- For example 2, 3, 5, 29, 67, 83, 97, $2^{1257787}$-1

## Problem

- Find all prime numbers less than a given value, n
- Some facts:
  - 2 is a prime number
  - All primes greater than 2 are odd
  - If we could determine whether a given number is prime or not, we could write a program that tests each odd number between 2 and n
  - Of course, n must be positive and 2 or greater

## High Level Solution

1. We begin by assuming we are able to determine whether a given number is prime or not
2. We input n and test to see whether it is within a valid range (input validation). We assume we are able to validate the input.
3. We then test each odd number between 2 and n using for primality. We use a loop structure to process each of these numbers.

## Top Down Approach

- This is an example of "Top-Down Programming"
- We design a solution that assumes we are able to obtain the solution to various subproblems
- This high level solution makes two assumptions
  - We know how to obtain and validate the input
  - We know how to determine that a number is prime

```
PROGRAM Primes
 IMPLICIT NONE
 INTEGER :: Range, Number, Count
 integer :: GetNumber
 LOGICAL :: Prime

 Range = GetNumber()
 Count = 1
 WRITE(*,*) "Prime number #", Count, ": ", 2
 DO Number = 3, Range, 2
   IF (Prime(Number)) THEN
     Count = Count + 1
     WRITE(*,*) "Prime number #", Count, ": ", NUMBER
   END IF
 END DO
 WRITE(*,*) "There are ", Count, " primes between 2 and
   ", Range
END PROGRAM Primes
```

## The Missing Pieces

- This solution made two assumptions
  - We know how to obtain and validate the input
  - We know how to determine that a number is prime
- We used functions to write the program based on these assumptions
- We have to define the functions we used

## Obtain and Validate the Input

```
INTEGER FUNCTION GetNumber()
  IMPLICIT NONE
  INTEGER :: N
  WRITE(*,*) "What is the range ? "
  DO
    READ(*,*) N
    IF (N >= 2) EXIT
    WRITE(*,*) "The range value must be >= 2. "&
               "Your input is ", N
    WRITE(*,*) "Please try again:"
  END DO
  GetNumber = N
END FUNCTION GetNumber
```

## Is a number, M, Prime?

- Look for divisors less than M, where M>2
- We need a loop that checks goes the potential divisors
  - Potential divisors are odd numbers, 3, 5, 9, 11, …
  - For each one check whether it divides M evenly
- A clever observation: We only have to check for divisors up to $\sqrt{M}$
- That is divisor*divisor must be less than M

## Testing for Primality

```
LOGICAL FUNCTION Prime(Number)
  IMPLICIT NONE
  INTEGER :: Number
  INTEGER :: Div
  IF (Number == 2) THEN
    Prime = .TRUE.
  ELSE IF (MOD(Number,2) == 0) THEN
    Prime = .FALSE.
  ELSE
    Div = 3
    DO
     IF (Div*Div>Number .OR. MOD(Number,Div)==0) EXIT
     Div = Div + 2
    END DO
    Prime = Div*Div > Number
  END IF
END FUNCTION Prime
```

## Complete Program (1)

```
PROGRAM Primes
  IMPLICIT NONE
  INTEGER :: Range, Number, Count
  INTEGER :: GetNumber
  LOGICAL :: Prime
  Range = GetNumber()
  Count = 1
  WRITE(*,*) "Prime number #", Count, ": ", 2
  DO Number = 3, Range, 2
    IF (Prime(Number)) THEN
      Count = Count + 1
      WRITE(*,*) "Prime number #", Count, ": ", Number
    END IF
  END DO
  WRITE(*,*) "There are ", Count, " primes between 2 and ", &
              Range
END PROGRAM Primes
```

## Complete Program (2)

```
!------------------------------------------------------
! This function does not require any formal argument.
!------------------------------------------------------
 INTEGER FUNCTION GetNumber()
  IMPLICIT NONE
  INTEGER :: N
  WRITE(*,*) "What is the range ? "
  DO
    READ(*,*) N
    IF (N >= 2) EXIT
    WRITE(*,*) "The range value must be >= 2. "&
               "Your input is ", N
    WRITE(*,*) "Please try again:"
  END DO
  GetNumber = N
END FUNCTION GetNumber
```

# Complete Program (3)

```
LOGICAL FUNCTION Prime(Number)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: Number
  INTEGER :: Div
  IF (Number == 2) THEN
    Prime = .TRUE.
  ELSE IF (MOD(Number,2) == 0) THEN
    Prime = .FALSE.
  ELSE
    Div = 3
    DO
      IF (Div*Div>Number .OR. MOD(Number,Div)==0) EXIT
      Div = Div + 2
    END DO
    Prime = Div*Div > Number
  END IF
END FUNCTION Prime
```

## Some Caveats

There are a couple of requirements for the evaluation of `func(e`$_1$`, e`$_2$`, e`$_3$`)` to make sense

- The number of parameters and arguments must be equal.
- The type of the corresponding arguments and parameters must be the same
- The arguments can be constants, variables or more general expressions.

## Argument-Parameter Association

- When a function is used, the argument values are used to initialize the parameters
- The way this is done is not as simple as it might seem
- It varies from language to language and is very different in Fortran than in C
- We present the three formal rules that Fortran uses

## Warning!

- The next few slides are very technical in nature
- The material might be a bit tedious but it is important to understand
- It will become even more important later in the course when we study C

## Argument-Parameter Association

- The first rule tells us what happens when a constant or expression is used as an argument
  - This is the most intuitive case
- The second rule deals with arguments that are variables
  - The association is a bit more complex here

# Rule 1 -- Expressions

If an actual argument is an expression or a constant, it is evaluated and the result is saved into a temporary location. Then, the parameter becomes a reference to this temporary cell

```
INTEGER :: a = 10, b = 3, c = 37
WRITE(*,*) Minimum(18,c-a,a+b)
```

When the function is invoked, new temporary variables named x, y and z are created. The value of x is initialized to 18, y to 27 and z to 13.

The function returns 13.

## Rule 2 -- Variables

If an actual argument is a variable, the corresponding formal argument is made to refer to the same memory cell.

```
INTEGER :: a = 10, b = 3, c = 37
WRITE(*,*) Minimum(a,b,c)
```

When the function is invoked, there are no new variables created. The parameter x refers to a, y to b and z to c. We say x is an **alias** for a. There are two names for the same memory cell.

The function returns 3.

## What Does This Function Do?

```
REAL FUNCTION DoSomething (a, b)
  IMPLICIT NONE
  INTEGER :: a, b
  a = a - b
  b = a + b
  a = b - a
  DoSomething = a
END FUNCTION DoSomething
```

## What Happens?

```
INTEGER :: x = 12, y = 5
WRITE (*,*) x, y
WRITE (*,*) DoSomething (12, 5)
WRITE (*,*) x, y
```

Output:

```
12 5
5
12 5
```

## What Happens?

```
INTEGER :: x = 12, y = 5
WRITE (*,*) x, y
WRITE (*,*) DoSomething (x, y)
WRITE (*,*) x, y
```

Output:

```
12   5
5
5   12
```

## What Happens?

More problematically

```
INTEGER :: x = 12, y = 7
WRITE (*,*) x, y, DoSomething (x, y)
```

# Function Execution (summary)

When a function is invoked

1. arguments are evaluated
2. parameter-argument associations are made
3. function body executes until it reaches the END FUNCTION statement
4. value assigned to the function name is returned
5. all temporary storage is released
6. this value is used where the function was invoked

## Where Do Function Definitions Go?

The structure of a program is:

```
PROGRAM  program-name
IMPLICIT  NONE
[declarations]
[statements]
END PROGRAM  program-name
[function definitions]
```

Function definitions are placed in the same file as the program

They can be anywhere in the file but usually follow the program

## Average of Three Numbers

```
PROGRAM  Avg
  IMPLICIT  NONE
  REAL :: a, b, c, Mean
  READ(*,*)  a, b, c
  Mean = Average (a, b, c)
  WRITE(*,*) a, b, c, Mean
END PROGRAM  Avg

REAL FUNCTION  Average(a, b, c)
  IMPLICIT  NONE
  REAL :: a, b, c
  Average = (a + b + c) / 3.0
END FUNCTION  Average
```

## Two Functions (part 1)

```
PROGRAM  TwoFunctions
   IMPLICIT  NONE
   INTEGER :: a, b, BiggerOne
   REAL    :: GeometricMean

   READ(*,*)  a, b
   BiggerOne = Maximum(a,b)
   GeometricMean = GeoMean(a,b)
   WRITE(*,*)  "Input = ", a, b
   WRITE(*,*)  "Larger one = ", BiggerOne
   WRITE(*,*)  "Geometric Mean = ", GeometricMean
END PROGRAM  TwoFunctions
```

## Two Functions (part 2)

```
INTEGER FUNCTION  Maximum(a, b)
   IMPLICIT  NONE
   INTEGER :: a, b
   IF (a >= b) THEN
      Maximum = a
   ELSE
      Maximum = b
   END IF
END FUNCTION  Maximum

REAL FUNCTION  GeoMean(a, b)
   IMPLICIT  NONE
   INTEGER :: a, b
   GeoMean = SQRT(REAL(a*b))
END FUNCTION  GeoMean
```