McGill

COMP 208 – Computers in Engineering

COMP 208
Computers in Engineering

Lecture 11

Jun Wang
School of Computer Science
McGill University
Fall 2007

Review

- READ and WRITE both take 2 parameters
- 2nd parameter: format
  - * (default; format determined by compiler)
  - label
  - format string

```
WRITE (*, 100)
WRITE (*, "(A15, F7.2)")
```

- FORMAT statement

```
label FORMAT (format-descriptors)
```

- 1st parameter: file descriptor
  - * (default; screen for WRITE; keyboard for READ)
  - an integer file descriptor

Review

- File input/output: 3 steps
  - Step 1: open the file

```
OPEN(UNIT=n, FILE="filename")
OPEN(n, FILE="filename")
```

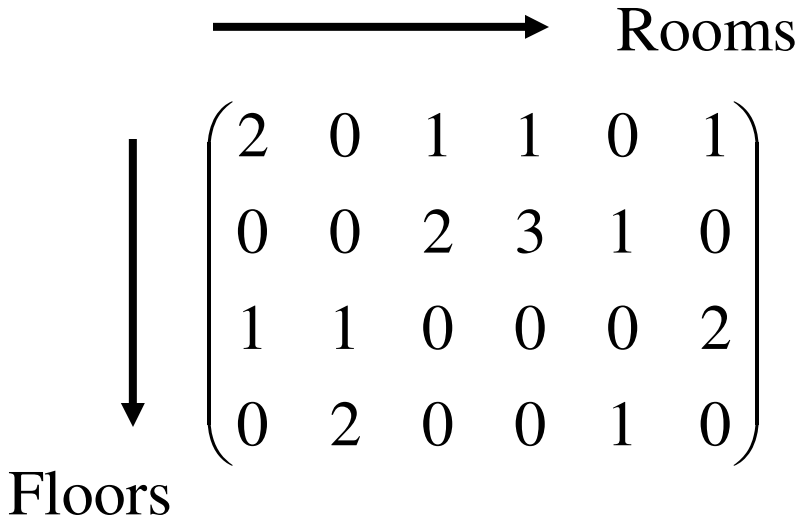
```
OPEN (UNIT=10, FILE="data.txt")
OPEN (10, FILE="data.txt")
```

  - Step 2: input/output with READ/WRITE
  - Step 3: close the file

```
CLOSE(UNIT=n)
CLOSE(n)
```

A Two Dimensional Array

A small hotel with 4 floors and 6 rooms on each floor could use a table with 4 columns and 6 rows to represent the number of occupants in each room:



A 2-dimensional array

- 2 dimensions: Rooms and Floors
- We can use 4 1-D arrays instead, each representing the data for one floor
- The reason we need 2-D arrays is exactly the same reason why we need 1-D array: to refer to data in a generic way, and thus to process data with algorithms.

A 2-dimensional array

- Find out which floor has the most guests:

```
INTEGER :: Floor1(6), Floor2(6)
INTEGER :: Floor3(6), Floor4(6)
...
max = -1

n = 0
DO j=1, 6
  n = n + Floor1(j)
END DO
IF (n > max) THEN
  max = n
  maxfloor = 1
END IF

n = 0
DO j=1, 6
  n = n + Floor2(j)
END DO
IF (n > max) THEN
  max = n
  maxfloor = 2
END IF

! AND so on for Floor3 and Floor4
```

```
INTEGER :: Rooms(4, 6)
...

max = -1
DO i=1, 4
   ! find total on floor i
   n = 0
   DO j=1, 6
      n = n + Rooms(i, j)
   END DO

   IF (n > max) THEN
      max = n
      maxfloor = i
   END IF
END DO
```

A 2-dimensional array

- Assuming on each floor, room 5 is the only double room, and we want to find out how many empty double rooms there are:

Use 4 1-D arrays

```
INTEGER :: Floor1(6)
INTEGER :: Floor2(6)
INTEGER :: Floor3(6)
INTEGER :: Floor4(6)
...

! Find out how many empty
! double rooms in the hotel

n = 0
IF (Floor1(5) == 0) n = n+1
IF (Floor2(5) == 0) n = n+1
IF (Floor3(5) == 0) n = n+1
IF (Floor4(5) == 0) n = n+1
```

Use 1 2-D arrays

```
INTEGER :: Hotel(4, 6)
...

! Find out how many empty
! double rooms in the hotel

n = 0
DO i=1, 4
  IF (Hotel(i,5) == 0) n=n+1
END DO
```

Back to 1-dimensional array

- Find out the total number of the 50 different types of cards

Using 50 simple variables

```
INTEGER :: Card1, Card2
...
INTEGER :: Card49, Card50
...

total = 0
total = total + Card1
total = total + Card2
...
total = total + Card50
```

Using a 1-D array of 50 elements

```
INTEGER :: Cards(50)
...

total = 0
DO i=1, 50
  total = total + Cards(i)
END DO
```

What if there are 500 types of cards?

Declaring Multidimensional Arrays

The declaration is similar to that of one dimensional arrays

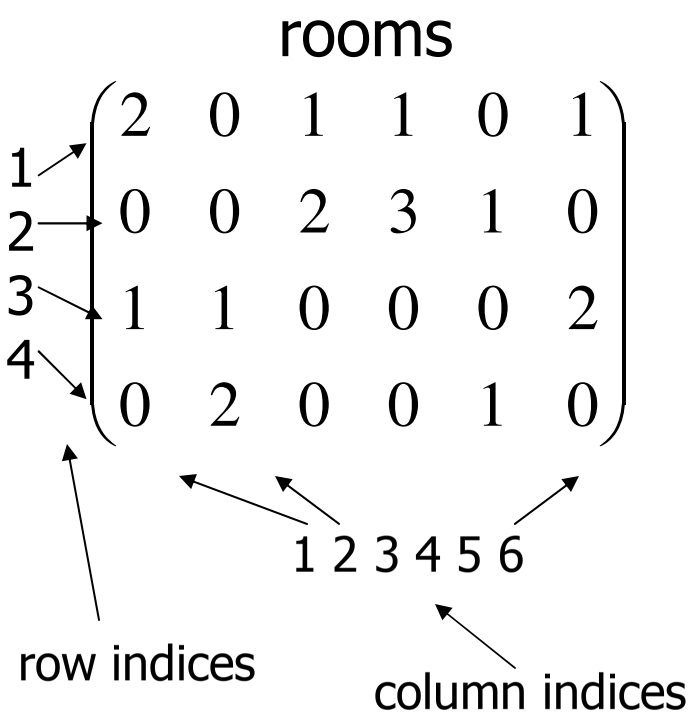
We specify a size for each dimension

```
Type :: name(bound1, bound2)
```

```
INTEGER :: Rooms(4,6)
REAL :: Matrix(25,25)
```

Using Tables (2-D arrays)

- To refer to an element, we must specify both the row index and the column index
- For example the number of occupants in the 5th room on the 2nd floor could be accessed by indexing as `Rooms(2,5)`



Occupancy and Capacity

A Hotel could be represented by two tables

- One table might contain room occupancies
- A second table might represent maximum room capacities

$$\begin{pmatrix} 2 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 2 & 3 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 2 \\ 0 & 2 & 0 & 0 & 1 & 0 \end{pmatrix}$$

occupancy

$$\begin{pmatrix} 2 & 2 & 2 & 3 & 3 & 2 \\ 2 & 2 & 2 & 3 & 3 & 3 \\ 2 & 2 & 2 & 3 & 3 & 2 \\ 2 & 2 & 3 & 3 & 3 & 3 \end{pmatrix}$$

capacity

Three Dimensional Arrays

We cam combine the two tables into a single structure that represents the Hotel

This would form a three dimensional object

3 dimensions: **Occupancy-Capacity**, **Floor**, **Room**

$$\begin{pmatrix} \begin{pmatrix} 2 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 2 & 3 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 2 \\ 0 & 2 & 0 & 0 & 1 & 0 \end{pmatrix}_{3} & \begin{matrix} 2 \\ 3 \end{matrix} \\ \begin{pmatrix} 2 & 2 & 2 & 3 & 3 & 2 \\ 2 & 2 & 3 & 3 & 3 & 3 \end{pmatrix}_{3} & \end{pmatrix}$$

Accessing Values

To access an individual cell, we need three indices

- We select which table we want
  - the first – representing room occupancy or
  - the second – representing room capacity
- We specify the floor (the row index) and
- We specify the room (the column index)

Accessing Values

For example the number of occupants in the 5th room on the 2nd floor could be accessed by indexing as

```
Hotel(1,2,5)
```

The capacity of the same room would be

```
Hotel(2,2,5)
```

Another 3-D array example is: using 7 tables to represent the occupancy data on each day of the week.

Instead of 7 separate tables, we use a 3-D array: the 3 dimensions are, day-of-week, floor, room

Processing Tables

In going through all cells in a table, there are two methods

- Cells can be accessed row by row
- This is called row-major order
- Cells can be accessed column by column
- This is called column-major order

Fortran stores tables internally in column major order

Using Implied Loops

Read the array values row-by-row with an implied DO loop

```
INTEGER :: X(2,4)
. . .
DO I = 1, 2
  READ(*,*) (X(I,J), J = 1, 4)
END DO
```

Row Major (again)

Assume the input is

```
4 8 3 9
7 5 2 6
```

The resulting matrix, X, is

$$\begin{pmatrix} 4 & 8 & 3 & 9 \\ 7 & 5 & 2 & 6 \end{pmatrix}$$

Implied Do Loops (again)

Read the array values column-by-column with an implied DO loop

```
INTEGER :: X(2,4)
. . .
DO J = 1, 4
  READ(*,*) (X(I,J), I = 1, 2)
END DO
```

Each time this READ is executed, it accepts 2 numbers

What happened?

If the input was the same as the last example

```
4 8 3 9
7 5 2 6
```

3,9 on first line and 2,6 on second line are ignored

The resulting matrix, X, would be

$$\begin{pmatrix} 4 & 7 & ? & ? \\ 8 & 5 & ? & ? \end{pmatrix}$$

Column Major (again)

Assume the input is

```
4  8
3  9
7  5
2  6
```

The resulting matrix, X, is

$$\begin{pmatrix} 4 & 3 & 7 & 2 \\ 8 & 9 & 5 & 6 \end{pmatrix}$$

Nested Implied Loops

We can use nested implied DO loops to read the values

The following code reads values into an array in row major order

The values can appear on one or more lines with no restrictions

```
INTEGER :: X(2,4)
. . .
READ(*,*) ((X(I,J), J = 1, 4), I = 1, 2)
```

inner loop

Outer loop

Column Major Order

We can also use nested implied loops to read in column major order

```
INTEGER :: X(2,4)
. . .
READ(*,*) ((X(I,J), I = 1, 2), J = 1, 4)
```

Just plain read

What about trying this?

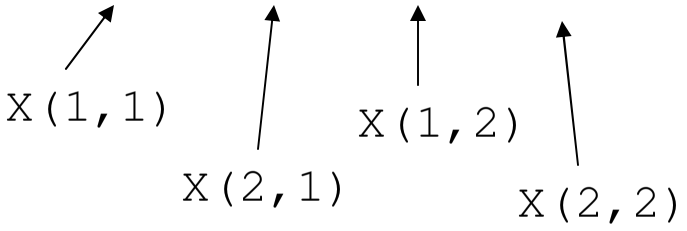
```
INTEGER :: X(2,4)
. . .
READ(*,*) X
```

How are the input values placed?

Fortran stores arrays in column major order

For example, if the input is:

```
1   2   3   4   5   6   7   8
```



The table would be:

```
1  3  5  7
2  4  6  8
```

More Matrix Processing

So far we have just focused on reading values into a matrix

Here are some more examples of applications where we process the cells of a two dimensional array

Initialize a Matrix to be the Identity Matrix

```
INTEGER :: SIZE = 10
INTEGER :: Ident (10, 10)
INTEGER :: i, j
DO i = 1, SIZE
  DO j = 1, SIZE
    IF (i == j) THEN
      Ident(i,i) = 1
    ELSE
      Ident(i,j) = 0
    END IF
  END DO
END DO
```

```
1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
```

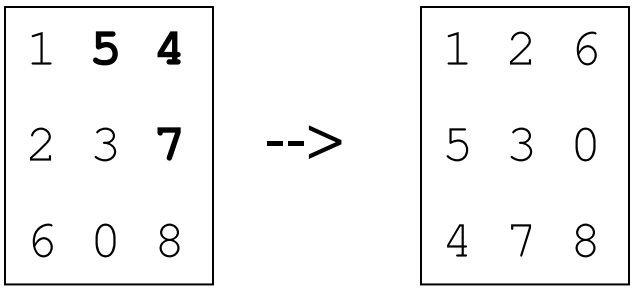
Sum Two Matrices

```
INTEGER :: SIZE = 20
REAL :: A(20,20), B(20,20), C(20,20)
INTEGER :: i, j

DO i = 1, SIZE
  DO j = 1, SIZE
    C(i,j) = A(i,j) + B(i,j)
  END DO
END DO
```

$$\begin{bmatrix} 1 & 2 & 4 \\ 2 & 0 & 7 \end{bmatrix} + \begin{bmatrix} 3 & 1 & 5 \\ 2 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 3 & 9 \\ 4 & 2 & 8 \end{bmatrix}$$

Transpose a Square Matrix

```
INTEGER :: SIZE = 20
REAL :: A(20,20), B(20,20), C(20,20)
INTEGER :: i, j
REAL :: Temp

DO i = 1, SIZE
  DO j = i+1, SIZE
    Temp = A(i,j)
    A(i,j) = A(j,i)
    A(j,i) = Temp
  END DO
END DO
```



Exercise: What if j's initial value was 1 instead of i+1?

Multiply Square Matrices

```
INTEGER :: SIZE = 20
REAL :: A(20,20), B(20,20), C(20,20)
INTEGER :: i, j, k

DO i = 1, SIZE
  DO j = 1, SIZE
    C(i,j) = 0
    DO k = 1, SIZE
      C(i,j) = C(i,j) + A(i,k)*B(k,j)
    END DO
  END DO
END DO
```

C(i,j) is the product of the i-th row of A and the j-th column of B

An Application

- A power generating station has 4 generators
- To determine productivity of each of the generators we sample the power supplied at 6 different time periods
- How do we represent the data?

Data Representation

Use a two dimensional array with each column representing the power supplied by a generator

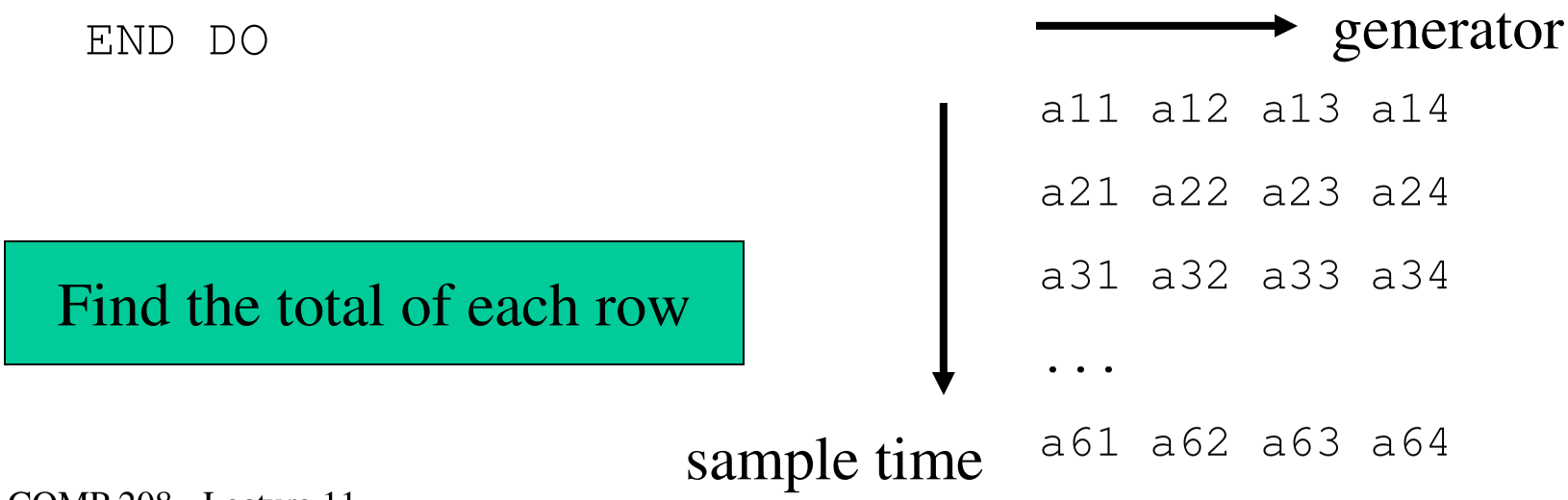
Each row represents a time of measurement

```
INTEGER :: gens = 4
INTEGER :: samples = 6
REAL :: power(6, 4)
```

Power Output

We can calculate the power output by the entire plant at each sample time

```
REAL :: power_output(samples)
. . .
DO time = 1, samples
  power_output(time) = 0
  DO gen = 1, gens
    power_output(time) =
           power_output(time)+ power(time,gen)
  END DO
END DO
```



Generator Output

We can calculate the average output of each generator

```
REAL :: gen_sum(gens), gen_ave(gens)
. . .
DO gen = 1, gens
  gen_sum(gen) = 0
  DO time = 1, samples
    gen_sum(gen) =
            gen_sum(gen) + power(time,gen)
  END DO
  gen_ave(gen) = gen_sum(gen)/samples
END DO
```

Find the average of each column

McGill

COMP 208 – Computers in Engineering

Delegating Tasks

Subprograms

Nathan Friedman

Fall, 2007

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

High Level Solution: divide-and-conquer

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 for primality. We use a loop structure to process each of these numbers.

pseudo-code

```
get valid number n
DO i=2, n
  IF (i is prime) THEN
    print i
  END IF
END DO
```

Note the 2 sub-problems:
- "get valid number n"
- "i is prime"

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

1. We know how to obtain and validate the input
2. We know how to determine that a number is prime

```
PROGRAM Primes
 IMPLICIT NONE
 INTEGER :: Range, Number, Count
 integer :: GetNumber
 LOGICAL :: Prime
 ! Get a valid upper limit
 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

1. We know how to obtain and validate the input
2. 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

Assume M has a divisor x > SQRT(M) , then
M = x * d, d < SQRT(M)
But d must have already been checked

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
```

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 as we saw in the example

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
  Fact = 1
  DO i = 1, n
    Fact = Fact * i
  END DO
  Factorial = Fact
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 **formal 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

The parameters are variables local to the function.

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

- 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

```
PROGRAM fun
   IMPLICIT NONE
   INTEGER :: square, a, b, c

   a = square(3) ! a is 9
   b = square(a) ! b is 81
   c = square(a+1) ! c is 100

   WRITE (*,*), "Square of 6 is: ", square(6)
END PROGRAM fun


INTEGER FUNCTION square(x)
   IMPLICIT NONE
   INTEGER :: x

   square = x * x
END FUNCTION square
```

pass constant for function parameter

pass variable

pass expression