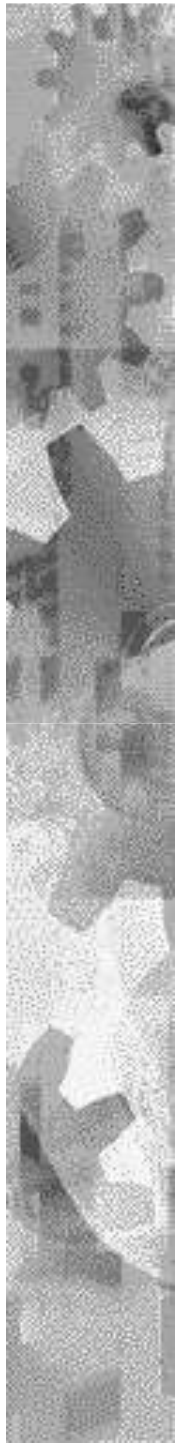
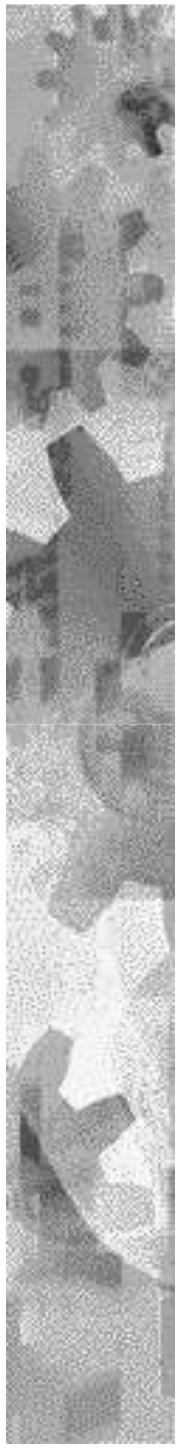


Computers in Engineering COMP 208

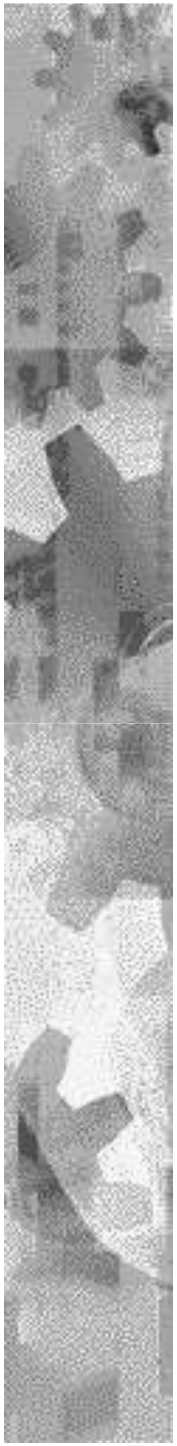
Subroutines

Michael A. Hawker



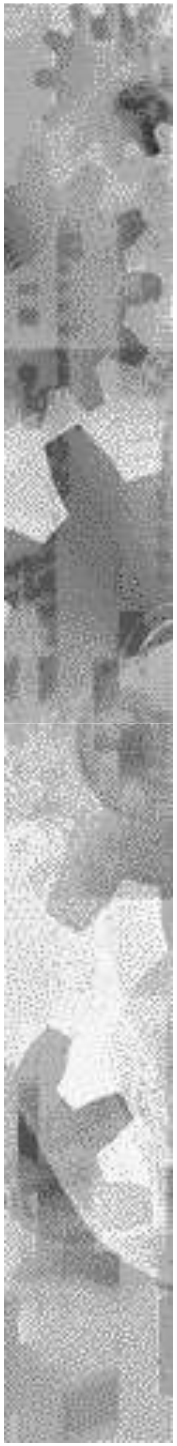
Subprograms

- Functions are one type of subprogram in FORTRAN
- Another type of subprogram FORTRAN allows is called a **subroutine**
- There are many similarities between them and we must be careful not to confuse the two types of subprograms



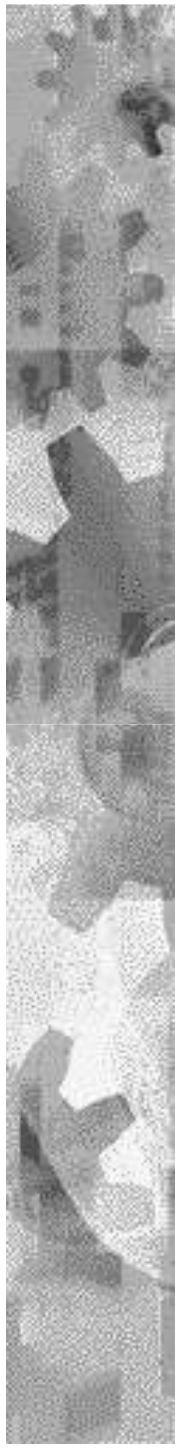
Subroutines

- Subroutines are used to define new actions
- Unlike functions, they do not return values
- They can modify the values of arguments or return values indirectly through the arguments
- For example a Sort subroutine may take an array as an argument and return the array with the values in sorted order



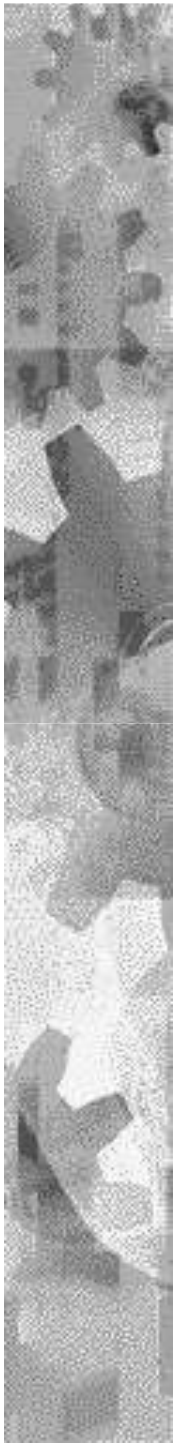
A Factorial Subroutine

- Previously we defined a function to compute factorials
- It acted as a new operator (like `sqrt`) to return a value directly
- We could also define a subroutine to compute factorials
- The result must be returned using an extra parameter



A Factorial Subroutine

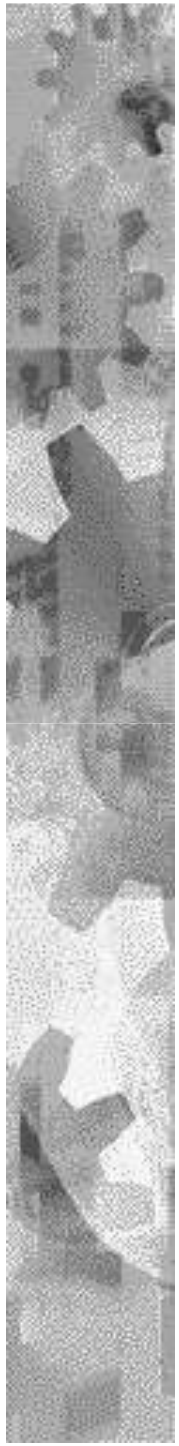
```
SUBROUTINE Factorial(n,Fact)
  IMPLICIT NONE
  INTEGER :: n, Fact
  INTEGER :: i

  Fact = 1
  DO i = 1, n
    Fact = Fact * i
  END DO
END SUBROUTINE Factorial
```



A Factorial Subroutine

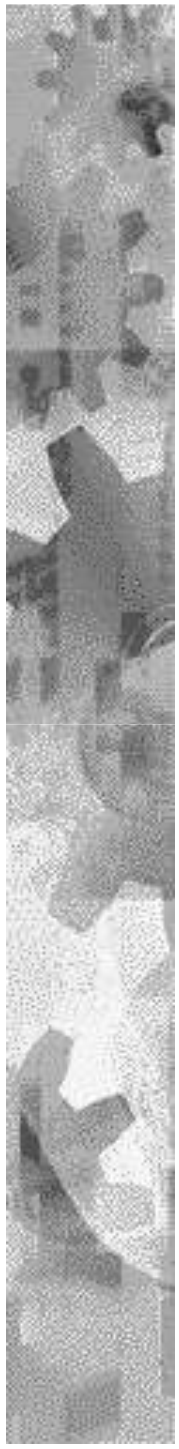
```
SUBROUTINE Factorial(n,Fact)
  IMPLICIT NONE
  INTEGER :: n, Fact
  INTEGER :: i

  Fact = 1
  DO i = 1, n
    Fact = Fact * i
  END DO
END SUBROUTINE Factorial
```

- The subroutine does not return a value directly
- The parameter Fact is used to hold the value that is being computed



Computing Statistics

```
! ------------------------------------------------------------------
! Read an indeterminate number of real values and compute their mean,
! variance and standard deviation.
! ------------------------------------------------------------------
PROGRAM  MeanVariance
   IMPLICIT   NONE
   INTEGER :: Number, IOstatus
   REAL    :: Data, Sum, Sum2
   REAL    :: Mean, Var, Std
   Number = 0
   Sum    = 0.0
   Sum2   = 0.0
   DO
      READ(*,*,IOSTAT=IOstatus)  Data
      IF (IOstatus < 0)  EXIT
      Number = Number + 1
      CALL  Sums(Data, Sum, Sum2)
   END DO
   CALL  Results(Sum, Sum2, Number, Mean, Var, Std)
   CALL  PrintResult(Number, Mean, Var, Std)
END PROGRAM MeanVariance
```



IOSTAT

We can write:

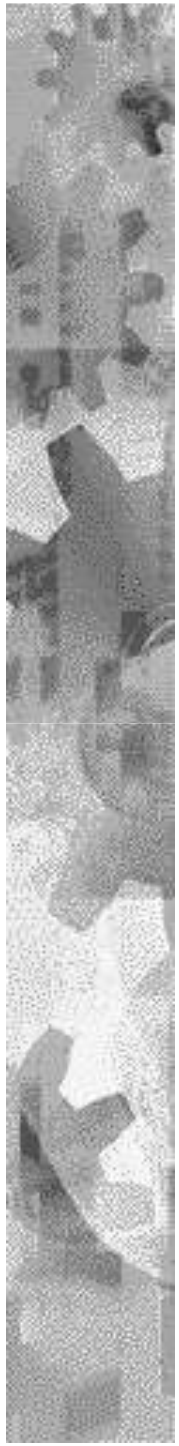
```
INTEGER :: IOstatus
READ(*,*,IOSTAT=IOstatus) v1, v2, . . ., vn
```

The variable following " IOSTAT = " can be any variable of type INTEGER

This variable is assigned a value after the READ is executed

1. If the value is zero, the read was successful
2. If the value is negative, the end of file was reached
3. If the value is positive, there was an error in the input

An end of file signal is sent from the keyboard when you press Ctrl-Z (in windows)

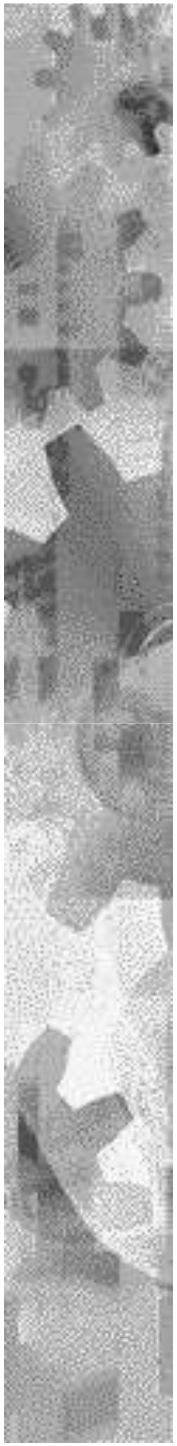


Computing Statistics

```
! ------------------------------------------------------------------
! Read an indeterminate number of real values and compute their mean,
! variance and standard deviation.
! ------------------------------------------------------------------
PROGRAM  MeanVariance
   IMPLICIT   NONE
   INTEGER :: Number, IOstatus
   REAL    :: Data, Sum, Sum2
   REAL    :: Mean, Var, Std
   Number = 0
   Sum    = 0.0
   Sum2   = 0.0
   DO
      READ(*,*,IOSTAT=IOstatus)  Data
      IF (IOstatus < 0)  EXIT
      Number = Number + 1
      CALL  Sums(Data, Sum, Sum2)
   END DO
   CALL  Results(Sum, Sum2, Number, Mean, Var, Std)
   CALL  PrintResult(Number, Mean, Var, Std)
END PROGRAM MeanVariance
```

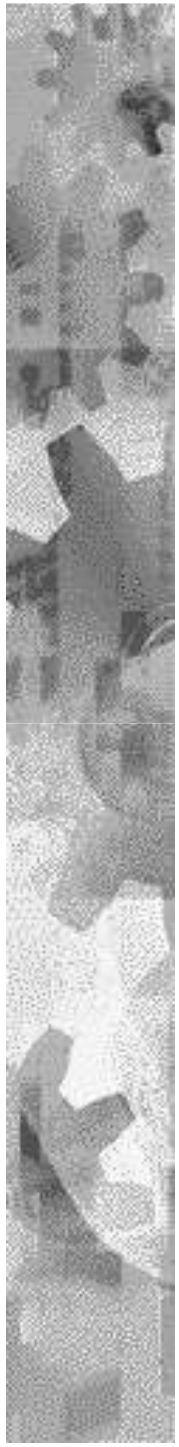
Compute Sum and Sum of Squares

```
! ------------------------------------------------------------------
!    This subroutine takes three REAL values:
!    (1)  x      - the input value
!    (2)  Sum    - x will be added to this sum-of-input
!    (3)  SumSQR - x*x is added to this sum-of-squares
! ------------------------------------------------------------------
   SUBROUTINE  Sums(x, Sum, SumSQR)
      IMPLICIT  NONE
      REAL :: x                ! Input Parameter
      REAL :: Sum, SumSQR      ! Input and Output Parameters
      Sum    = Sum + x
      SumSQR = SumSQR + x*x
   END SUBROUTINE  Sums
```



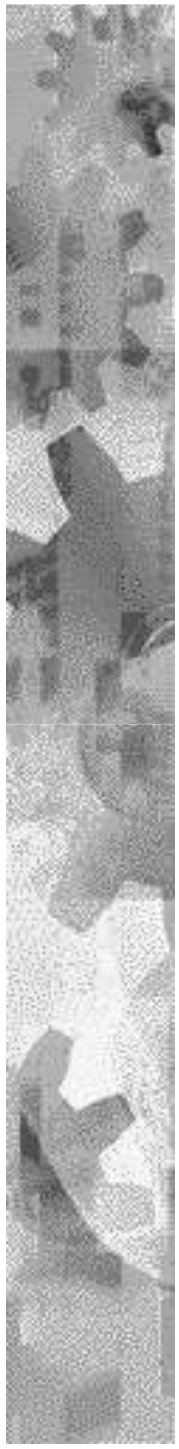
Computing the Statistics

```
! ------------------------------------------------------------------
! Compute the mean, variance and standard deviation
!
!    (1) Sum       - sum of input values
!    (2) SumSQR    - sun-of-squares
!    (3) n         - number of input data items
!    (4) Mean      - computed mean value
!    (5) Variance  - computed variance
!    (6) StdDev    - computed standard deviation
! ------------------------------------------------------------------
   SUBROUTINE  Results(Sum, SumSQR, n, Mean, Variance, StdDev)
      IMPLICIT  NONE
      INTEGER :: n
      REAL :: Sum, SumSQR              ! Input Parameters
      REAL :: Mean, Variance, StdDev   ! Output Parameters
      Mean = Sum / n
      Variance = (SumSQR - Sum*Sum/n)/(n-1)
      StdDev   = SQRT(Variance)
   END SUBROUTINE
```



Output the Results

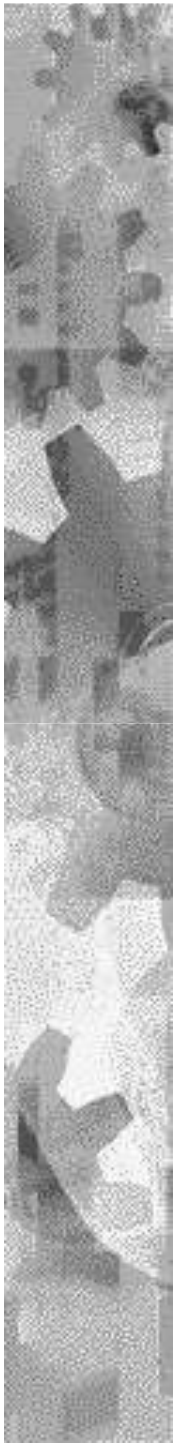
```
! ------------------------------------------------
! Display the computed results.
! ------------------------------------------------
   SUBROUTINE  PrintResult(n, Mean, Variance, StdDev)
      IMPLICIT  NONE
      INTEGER :: n
      REAL :: Mean, Variance, StdDev
      WRITE(*,*)
      WRITE(*,*) "No. of data items  = ", n
      WRITE(*,*) "Mean               = ", Mean
      WRITE(*,*) "Variance           = ", Variance
      WRITE(*,*) "Standard Deviation = ", StdDev
   END SUBROUTINE  PrintResult
```



Advantages of Subprograms

Why use subprograms?

- Supports top down program design to simplify developing complex programs
- Allows for independent testing of subtasks
- Allows us to develop reusable code
- Isolates the program from side effects that may be caused by the subprogram



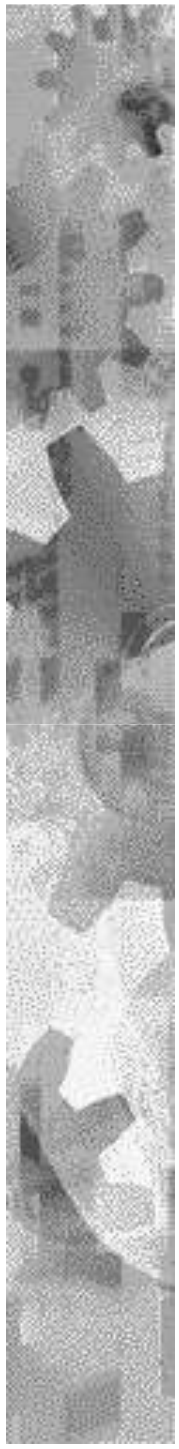
Subroutine Definitions

Syntax

```
SUBROUTINE  subroutine-name
  (arg1,arg2,...,argn)
   IMPLICIT  NONE
   [declarations]
   [code]
END SUBROUTINE  subroutine-name
```

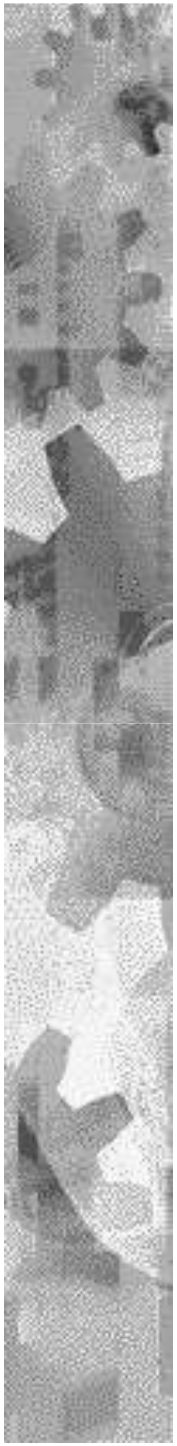
The definition starts with **SUBROUTINE**, followed by the subroutine's name.

Following the name, the list of parameters is specified



Function or Subroutine?

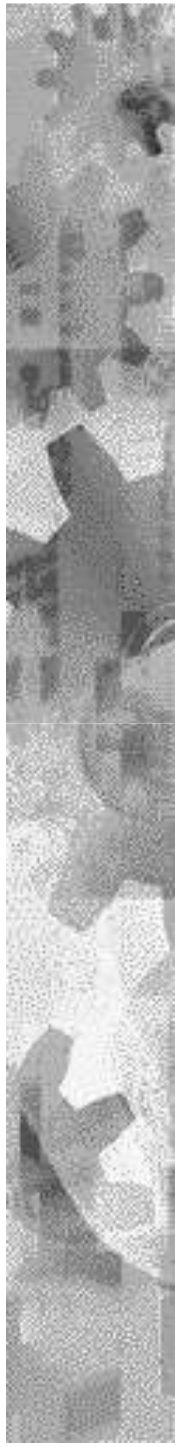
- Factorial takes a single argument and returns a single value
- Defining it as a function seems more natural
- Defining it as a subroutine is more forced
- Sometimes we don't have a choice



Factorial
Function vs. Subroutine

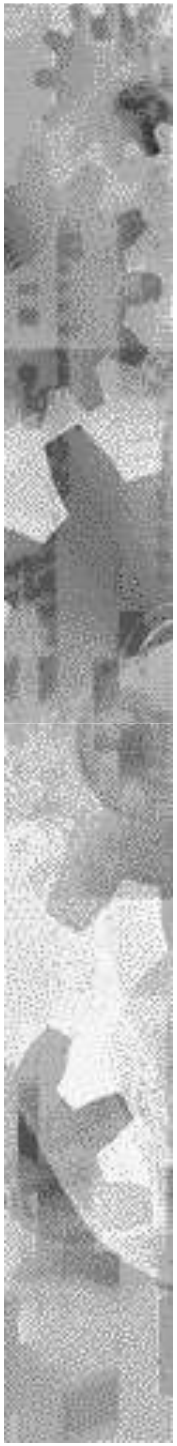
```
INTEGER FUNCTION Factorial(n)
  IMPLICIT NONE
  INTEGER :: n
  INTEGER :: i, Fact
  Fact = 1
  DO i = 1, n
    Fact = Fact * i
  END DO
  Factorial = Fact
END FUNCTION Factorial
```

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



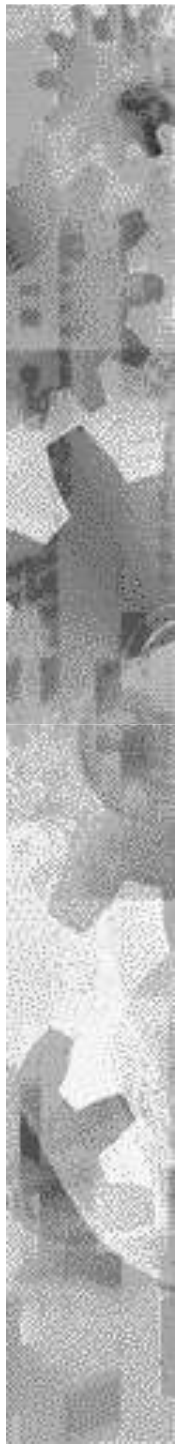
Functions, Subroutines What's the Difference?

- A function defines a new operation
- It takes some "input" values (arguments) and returns a result
  - For example, sqrt, mod, factorial
- A subroutine defines a new action analogous to a statement.
- It might modify its arguments but doesn't return a result directly
  - For example, sort



Functions, Subroutines What's the Difference?

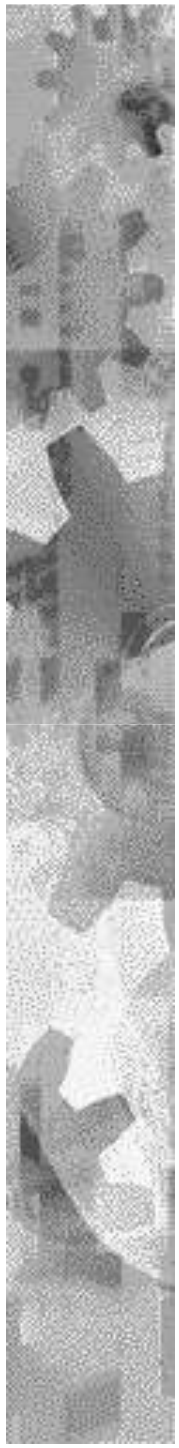
- A function must assign a value to the dummy variable which is the name of the function
- The name of the subroutine is not a dummy variable and is not assigned a value



Functions, Subroutines What's the Difference?

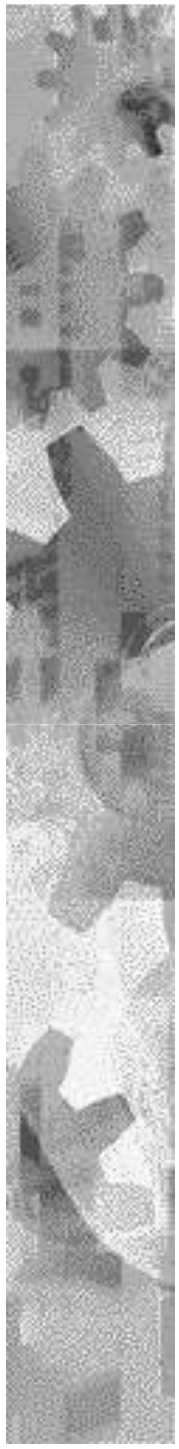
- A function is invoked implicitly by using it in an expression.
- After executing, it returns a value to be used in evaluating the expression

- A subroutine is called explicitly. It appears in the program where a statement can appear
- After executing it just returns
- The argument values may have changed



Definition vs. Usage

- We have discussed how to define subprograms and the difference between function and subroutine definition
- Definitions are a one time thing
- Once defined, the subprograms can be used throughout the program
- The way functions and subroutines are used differs



Using Subroutines

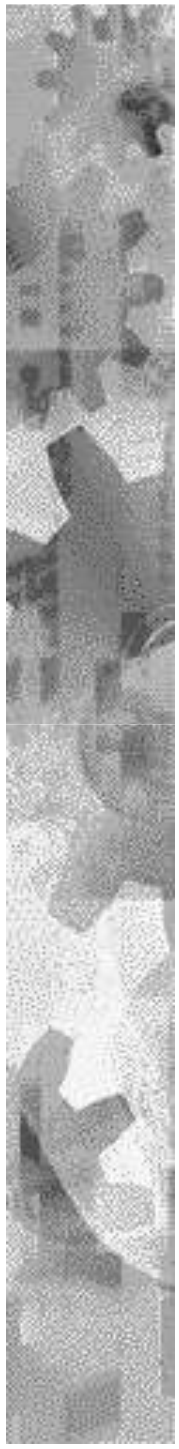
Subroutines are invoked with a CALL statement.

Syntax of CALL

```
CALL subroutine-name (e1, e2, ..., en)
CALL subroutine-name ()
CALL subroutine-name
```

In the first form, the subroutine has n parameters.

If a subroutine does not have any arguments, it can be called with or without parentheses



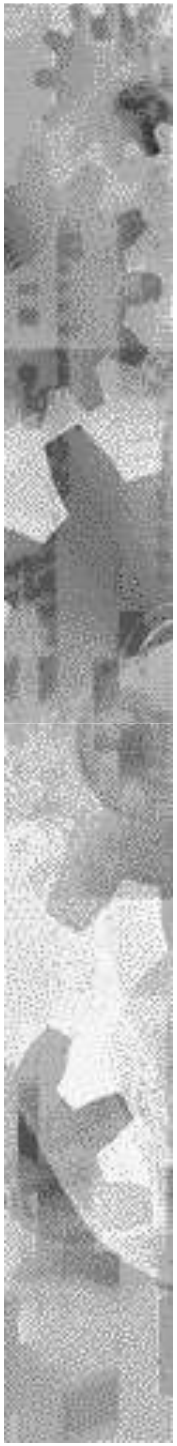
Example

To use the Factorial **subroutine,** use the statement

```
CALL Factorial(7,result)
WRITE (*,*) result+9
```

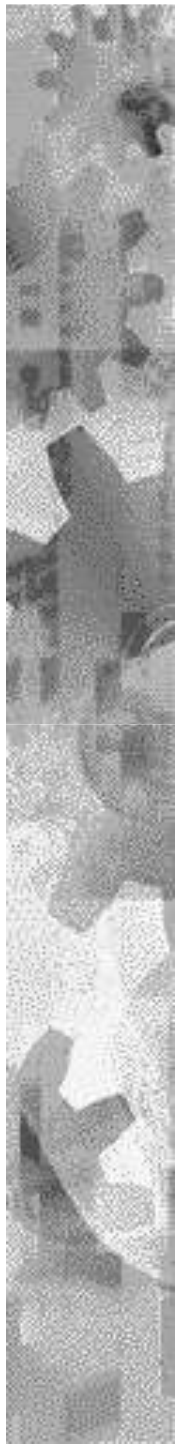
To use the Factorial **function**, reference it directly in an expression

```
WRITE (*,*) Factorial(7)+9
```



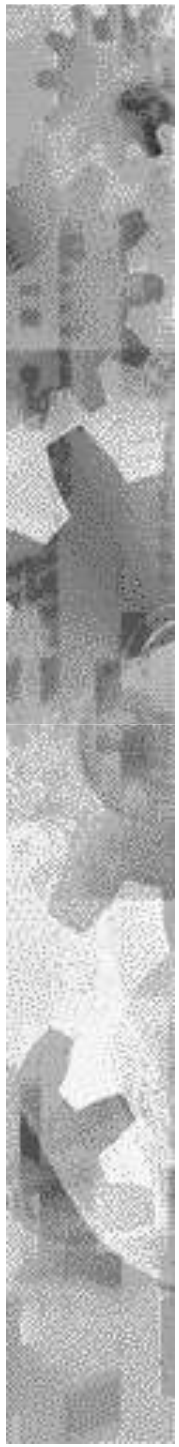
The Semantics of Call

1. When a CALL statement is executed,
   - The values of the arguments are passed to the parameters
   - The number of actual arguments in the CALL statement must match the number of formal parameters
   - The type of each argument must match the type of the corresponding formal parameter
2. The body of the called subroutine is executed.



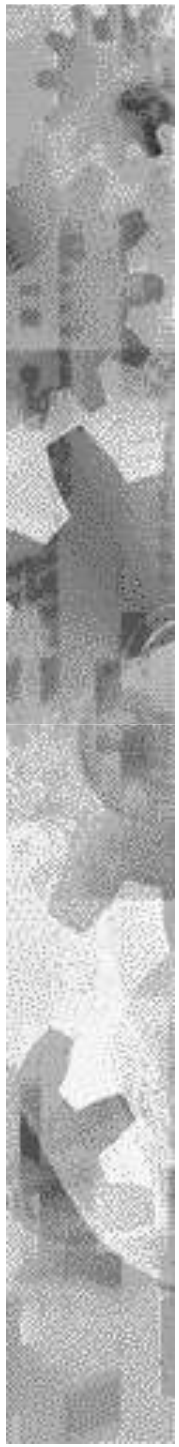
The Semantics of Call

3. When END SUBROUTINE is reached,
   - Execution of the subprogram ends
   - The next statement following the CALL statement is executed.
4. If a variable was passed as an argument, any changes made to it remain



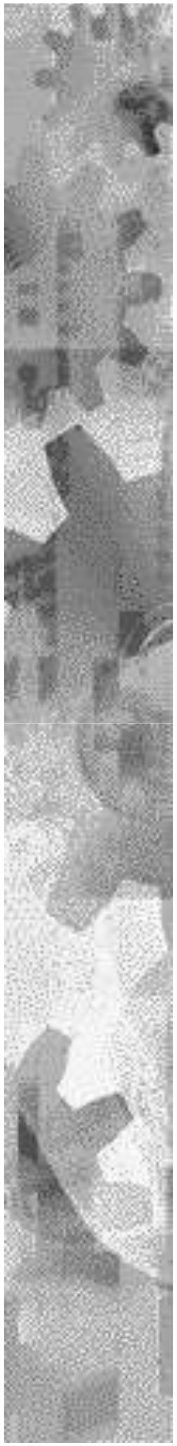
Minimum Function

```
INTEGER FUNCTION MinimumF (x, y, z)
  IMPLICIT NONE
  INTEGER :: x, y, z
  IF (x <= y .AND. x <= z) THEN
    MinimumF = x
  ELSE IF (y <= x .AND. y <= z) THEN
    MinimumF = y
  ELSE
    MinimumF = z
  END IF
END FUNCTION MimimumF
```



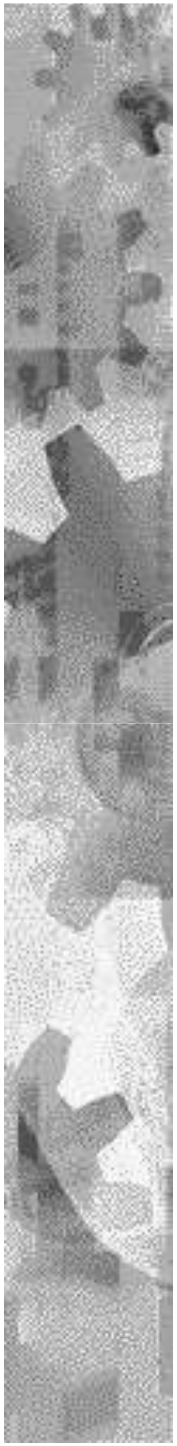
Minimum Subroutine

```
SUBROUTINE MinimumS (x, y, z, m)
  IMPLICIT NONE
  INTEGER :: x, y, z, m
  IF (x <= y .AND. x <= z) THEN
    m = x
  ELSE IF (y <= x .AND. y <= z) THEN
    m = y
  ELSE
    m = z
  END IF
END SUBROUTINE MimimumS
```



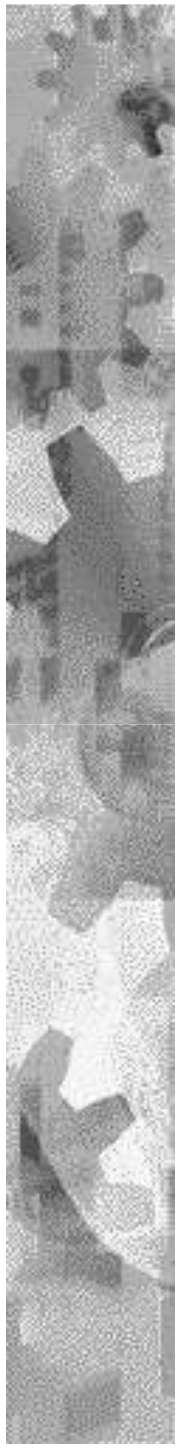
Examples of Use

```
INTEGER :: a, b, c, result
READ (*,*) a, b, c

CALL MinimumS(a,b,c,result)
WRITE (*,*) "The minimum of ", a, b, c, " is: ",&
            result

WRITE (*,*) "The minimum of ", a, b, c, " is: ",&
            MinimumF(a,b,c)
```



Rules for Argument Association

- The rules for associating arguments with formal parameters are the same as the rules we described for functions

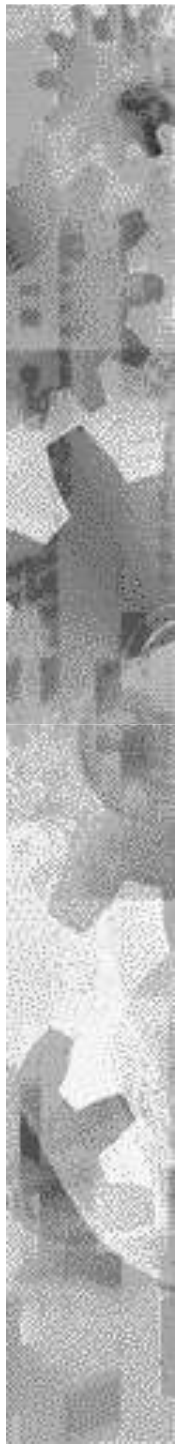


Multiple Results

- Sometimes we want operators that return multiple results
- FORTRAN functions cannot be used because they can only return a single value
- In such cases (as in the next example) we must use subroutines

Date Conversion

```
! ----------------------------------------
! Take an integer input Number in the form of
! YYYYMMDD and convert it to Year, Month and Day
! ----------------------------------------
SUBROUTINE  Conversion (Number, Year, Month, Day)
  IMPLICIT  NONE
  INTEGER :: Number
  INTEGER :: Year, Month, Day
  Year  = Number / 10000
  Month = MOD(Number, 10000) / 100
  Day   = MOD(Number, 100)
END SUBROUTINE  Conversion
```



“Means” Example

Problem:

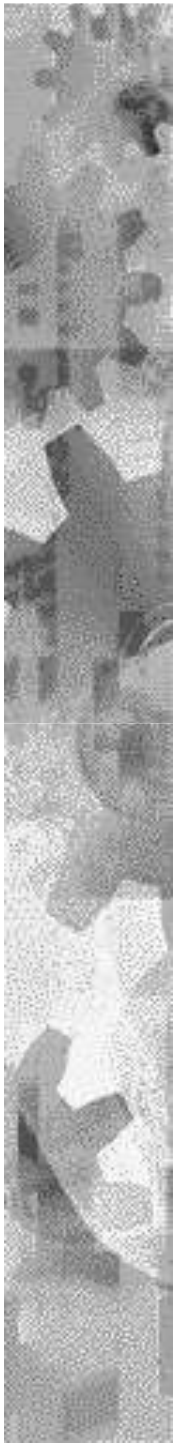
- Compute the arithemetic, geometric and harmic means of three real values

Question:

- Do we use a function or subroutine?

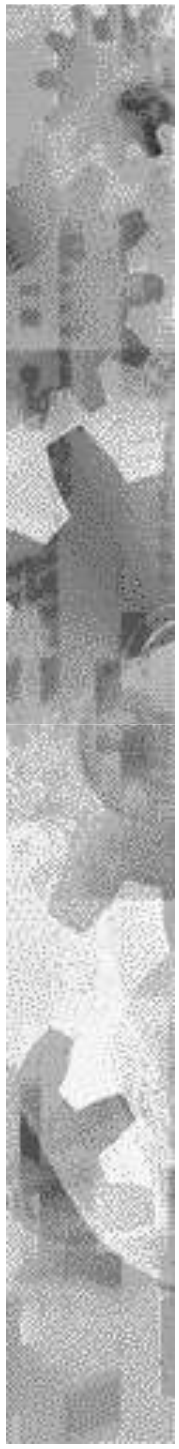
A function only returns a single result.

To compute three different values, we have to use a subroutine



“Means” Example

```
! ---------------------------------------------
! Subroutine to take three REAL values and compute
! their arithmetic, geometric, and harmonic means.
! ---------------------------------------------
SUBROUTINE  Means(a, b, c, Am, Gm, Hm)
  IMPLICIT  NONE
  REAL :: a, b, c
  REAL :: Am, Gm, Hm
  Am = (a + b + c)/3.0
  Gm = (a * b * c)**(1.0/3.0)
  Hm = 3.0/(1.0/a + 1.0/b + 1.0/c)
END SUBROUTINE  Means
```



Swap

Problem:

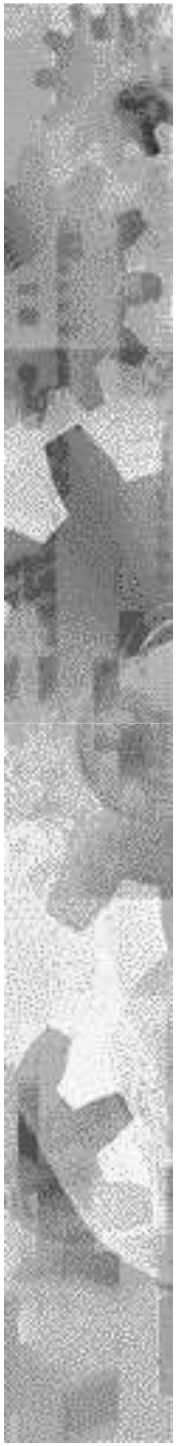
Interchange the values stored in two integer variables

Question:

Do we use a function or subroutine?

A function returns a result.

To perform an action that modifies variables, we have to use a subroutine

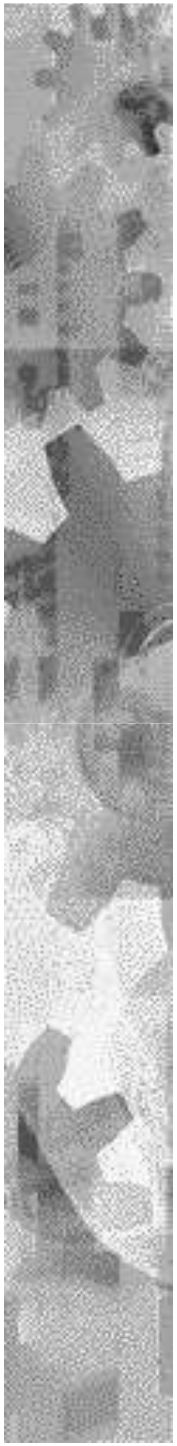


Swap

```
SUBROUTINE  Swap(a, b)
  IMPLICIT  NONE
  INTEGER :: a, b
  INTEGER :: temp
  temp = a
  a = b
  b = temp
END SUBROUTINE  Swap
```

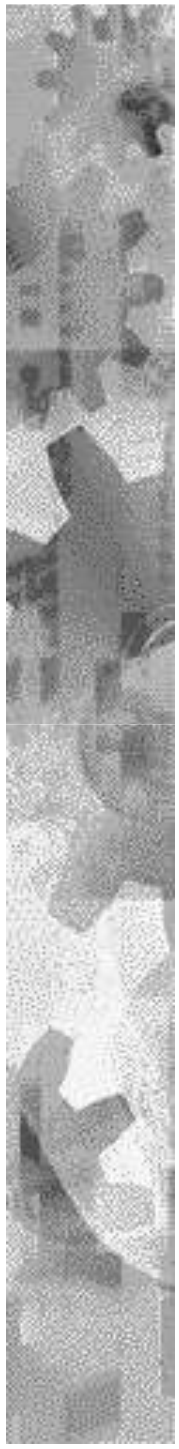
Example:

```
i = 4
j = 9
CALL Swap(i, j)
WRITE (*,*) i, j
```



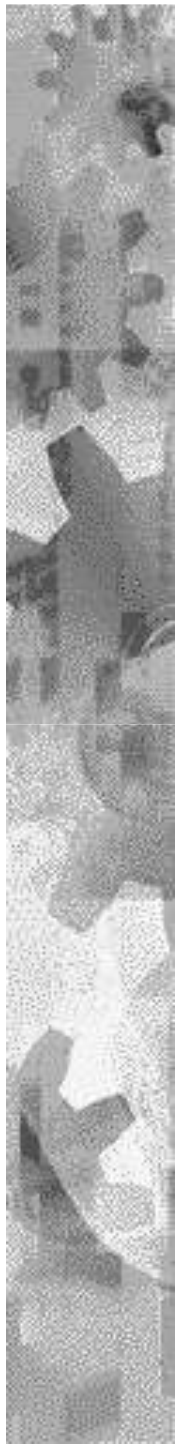
Array Parameters

- When we declare an array in a program, we must specify its size
- The compiler allocates storage for the specified number of memory cells
- When we write a subprogram definition to process an array, we want it to be generic
- That is, we want to be able to use it with different arrays, possibly of different sizes



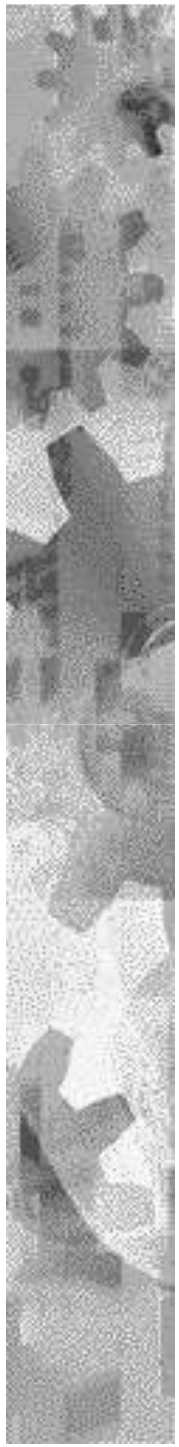
Minimum Value in an Array

```
REAL FUNCTION Min (A, n)
  IMPLICIT NONE
  INTEGER :: n
  INTEGER :: I
  REAL :: A(n)
  Min = A(1)
  DO I = 2, n
    IF (A(I) < Min) Min = A(I)
  END DO
  RETURN Min
END FUNCTION Min
```



Array Parameters

- Why can we declare an array when we don't know its size?
- The compiler does not allocate storage when we define a subprogram
- When we invoke a subprogram the compiler just makes the name an alias for an existing block of storage



Sorting Values in an Array

```
SUBROUTINE Sort (A, n)
  IMPLICIT NONE
  INTEGER :: n
  REAL :: A(n)
  INTEGER :: j, k, minptr
  DO j = 1,n-1
    minptr = j
    DO k = j+1,n
      IF (A(k)<A(minptr)) minptr = k
    END DO
    IF ( J /= minptr) THEN
      CALL SwapReals (A(j), A(minptr))
    END IF
  END DO
END SUBROUTINE Sort
```