FORTRAN - PROCEDURES

A **procedure** is a group of statements that perform a well-defined task and can be invoked from your program. Information *ordata* is passed to the calling program, to the procedure as arguments.

There are two types of procedures:

- Functions
- Subroutines

Function

A function is a procedure that returns a single quantity. A function should not modify its arguments.

The returned quantity is known as **function value**, and it is denoted by the function name.

Syntax:

Syntax for a function is as follows:

```
function name(arg1, arg2, ....)
   [declarations, including those for the arguments]
   [executable statements]
end function [name]
```

The following example demonstrates a function named area_of_circle. It calculates the area of a circle with radius r.

```
program calling_func

   real :: a
   a = area_of_circle(2.0)

   Print *, "The area of a circle with radius 2.0 is"
   Print *, a

end program calling_func


! this function computes the area of a circle with radius r
function area_of_circle (r)

! function result
implicit none

   ! dummy arguments
   real :: area_of_circle

   ! local variables
   real :: r
   real :: pi

   pi = 4 * atan (1.0)
   area_of_circle = pi *  r**2

end function area_of_circle
```

When you compile and execute the above program, it produces the following result:

```
The area of a circle with radius 2.0 is
   12.5663710
```

Please note that:

- You must specify **implicit none** in both the main program as well as the procedure.
- The argument r in the called function is called **dummy argument**.

The result Option

If you want the returned value to be stored in some other name than the function name, you can use the **result** option.

You can specify the return variable name as:

```
function name(arg1, arg2, ....) result (return_var_name)
   [declarations, including those for the arguments]
   [executable statements]
end function [name]
```

Subroutine

A subroutine does not return a value, however it can modify its arguments.

Syntax

```
subroutine name(arg1, arg2, ....)
   [declarations, including those for the arguments]
   [executable statements]
end subroutine [name]
```

Calling a Subroutine

You need to invoke a subroutine using the **call** statement.

The following example demonstrates the definition and use of a subroutine swap, that changes the values of its arguments.

```
program calling_func
implicit none

   real :: a, b
   a = 2.0
   b = 3.0

   Print *, "Before calling swap"
   Print *, "a = ", a
   Print *, "b = ", b

   call swap(a, b)

   Print *, "After calling swap"
   Print *, "a = ", a
   Print *, "b = ", b

end program calling_func


subroutine swap(x, y)
implicit none

   real :: x, y, temp

   temp = x
   x = y
   y = temp

end subroutine swap
```

When you compile and execute the above program, it produces the following result:

```
Before calling swap
a = 2.00000000
b = 3.00000000
After calling swap
a = 3.00000000
b = 2.00000000
```

Specifying the Intent of the Arguments

The intent attribute allows you to specify the intention with which arguments are used in the procedure. The following table provides the values of the intent attribute:

Value	Used as	Explanation
in	intent*in*	Used as input values, not changed in the function
out	intent*out*	Used as output value, they are overwritten
inout	intent*inout*	Arguments are both used and overwritten

The following example demonstrates the concept:

```
program calling_func
implicit none

   real :: x, y, z, disc

   x= 1.0
   y = 5.0
   z = 2.0

   call intent_example(x, y, z, disc)

   Print *, "The value of the discriminant is"
   Print *, disc

end program calling_func


subroutine intent_example (a, b, c, d)
implicit none

   ! dummy arguments
   real, intent (in) :: a
   real, intent (in) :: b
   real, intent (in) :: c
   real, intent (out) :: d

   d = b * b - 4.0 * a * c

end subroutine intent_example
```

When you compile and execute the above program, it produces the following result:

```
The value of the discriminant is
   17.0000000
```

Recursive Procedures

Recursion occurs when a programming languages allows you to call a function inside the same function. It is called recursive call of the function.

When a procedure calls itself, directly or indirectly, is called a recursive procedure. You should declare this type of procedures by preceding the word **recursive** before its declaration.

When a function is used recursively, the **result** option has to be used.

Following is an example, which calculates factorial for a given number using a recursive procedure:

```
program calling_func
implicit none

   integer :: i, f
   i = 15

   Print *, "The value of factorial 15 is"
   f = myfactorial(15)
   Print *, f

end program calling_func

! computes the factorial of n (n!)
recursive function myfactorial (n) result (fac)
! function result
implicit none

   ! dummy arguments
   integer :: fac
   integer, intent (in) :: n

   select case (n)
      case (0:1)
         fac = 1
      case default
         fac = n * myfactorial (n-1)
   end select

end function myfactorial
```

Internal Procedures

When a procedure is contained within a program, it is called the internal procedure of the program. The syntax for containing an internal procedure is as follows:

```
program program_name
   implicit none
   ! type declaration statements
   ! executable statements
   . . .
   contains
   ! internal procedures
   . . .
end program program_name
```

The following example demonstrates the concept:

```
program mainprog
implicit none

   real :: a, b
   a = 2.0
   b = 3.0

   Print *, "Before calling swap"
   Print *, "a = ", a
   Print *, "b = ", b

   call swap(a, b)
```

```
   Print *, "After calling swap"
   Print *, "a = ", a
   Print *, "b = ", b

contains
   subroutine swap(x, y)
      real :: x, y, temp
      temp = x
      x = y
      y = temp
   end subroutine swap

end program mainprog
```

When you compile and execute the above program, it produces the following result:

```
Before calling swap
a = 2.00000000
b = 3.00000000
After calling swap
a = 3.00000000
b = 2.00000000
```