Introduction to
Scientific & Engineering Computing
BIL 102FE (Fortran) Course
for
Week 4

Dr. Ali Can Takinacı

Assistant Professor

in

The Faculty of Naval Architecture and Ocean Engineering

80626

Maslak – Istanbul – Turkey

BASIC BUILDING BLOCKS

Two types of procedures (subroutine and function) and module in F

Programs and modules

A module is primarily a means of collecting together a set of related objects (procedures, variables and data types)

A Module does not contain any executable statement

Procedures

A special section of programs

It is known as procedures

Modules normally collect procedures together

In F, all procedures must be defined within modules

Intrinsic Functions

The function takes one or more values (called arguments)
and
create single result

There are 97 intrinsic functions defined in F
e.g.
sin(x) calculates the value of *sin*x where x is in radians
log(x) calculates the value of $\log_e x$
sqrt(x) calculates the square root of x

```
real :: x, y
y = abs(x)
```

will produce the absolute value of the real variable x.

Functions

The function statement, which is one of a sub-program of F, takes the form

`function` *`name`*`(`*`d1, d2`*`, ...) result(`*`result_name`*`)`

where d1, d2, ... are dummy argument which are used when the function is executed.
The *result_name* is the final result of the function after execution.

The final statement of the function is the end function statement,

taking the form,

`end function` *`name`*

As an example, a function, which calculates the cube root of a positive real numbers and a main program is given

```
        module cube_root_calc_1
        public :: cube_root
        contains
        function cube_root(x) result(root)
!       Dummy argument declaration
        real::x
!       Result variable declaration
        real::root
!       Local variable declaration
        real::log_x
!       Calculate cube root by using logs
        log_x = log(x)
        root = exp(log_x/3.0)
        end function cube_root
        end module cube_root_calc_1
! *****************************************************************
        program test_cube_root
        use cube_root_calc_1
        real :: x
        print *," type real positive number"
        read *, x
        print *,"the cube root of x=",x," is",cube_root(x)
        end program test_cube_root
```

There are three important points to notice about the function.

1. The variable *log_x* and *root* are not accessible from outside the function. They are called internal variables or local variables, and have no existence outside the function. Therefore the names *log_x* and *root* can be used in the main program or another procedure without any difficulty since they are isolated inside the function.

2. Every function must have exactly one such result variable whose name must appear in the result clause of the function statement.

3. The dummy argument, x, must be defined in the phrase intent(in) after the type declaration. This informs the compiler that the dummy argument can not be changed

When the end function statement is obeyed it causes execution of the program to return to the point program at which the function was referenced. For example the statement

a = b*cube_root(c) + d

will cause the cube root of c to be calculated by the function , multiplied by b, added to the current value of d, then the sum is assigned to the variable a.

It is possible to write a function, which has no argument such as

```
function name() result name(result_name)
```

Although this kind of use is not very good, the function is referenced such as

a = b*name() + c.

As a result, writing its name, followed by any arguments it may have enclosed in parenthesis references a function.

The execution of a function yields to a single value.

<u>**Subroutines**</u>

A subroutine is accessed by means of a <u>call statement</u>.
This gives the name of the subroutine and list of arguments, which will be used to transmit information between the calling program unit and the subroutine:

```
call name(arg1, arg2, ....)
```

The call statement interrupts the execution of main program and the program flow goes to the statement contained within the subroutine *name*.

When the subroutine has completed its task it returns to the place where it was called.

A program sample (main program)

```
     program various_roots
!    A program to show the use of the subroutine roots
     real :: x,root_2,root_3,root_4,root_5
     print *," type real positive number"
     read *, x
!    obtain roots
     call roots(x,root_2,root_3,root_4,root_5)
     print *,"the square root of x=",x," is",root_2
     print *,"the cube   root of x=",x," is",root_3
     print *,"the fourth root of x=",x," is",root_4
     print *,"the fifth root of  x=",x," is",root_5
     end program various_roots
```

A program sample (sub program)

```
    module various_roots_1
    public :: roots
    contains
    subroutine roots(x,square_root,cube_root,fourth_root,fifth_root)
!   subroutine  to calculatebvarious rots of a positive real numbers
!   supplied as the first argument, and returned them in the second to !
!   fifth arguments
!   Dummy argument declaration
    real, intent(in)::x
!   Result variable declaration
    real, intent(out)::square_root,cube_root,fourth_root,fifth_root
!   Local variable declaration
    real::log_x
!   Calculate square root using intrinsic sqrt()
    square_root = sqrt(x)
!   Calculate other root by using logs
    log_x = log(x)
    cube_root = exp(log_x/3.0)
    fourth_root = exp(log_x/4.0)
    fifth_root = exp(log_x/5.0)
    end subroutine roots
    end module various_roots_1
```

some important points

The subroutine calculates the square root, the cube root, the fourth root and the fifth root of a positive real number

The code is very similar to that written for the corresponding function.

The dummy arguments have given an *intent(out)* attribute, to indicate that they are to be used to transfer information from the subroutine back to the calling program.

The name of the subroutine is simply a means of identification and the interface of for a subroutine is the name of subroutine, together with the number and type of any dummy argument

Finally
A subroutine may call other subroutines but it must not call itself.

Actual and dummy arguments

When a function or subroutine is referenced
the information between the calling program unit and the subroutine or the function
is passed through its arguments

The relationship between the actual arguments in the calling program unit

and

the dummy arguments in the subroutine or function is of vital importance

The order and types of the actual arguments
must correspond exactly with the order

and

types of the corresponding dummy arguments.

Example

Write a subroutine which will take two character arguments as input arguments, containing two names (a “first name” and a “family name”, respectively) and which will return a string containing the two names with exactly one space separating them.

Analysis & Solution

The major difficulty is in the declaration of the length of the two names in the subroutine. This difficulty can be handled by using an assumed-length character declaration in the F. This can only be used for declaring a dummy argument, and involves replacing the length specifier by an asterisk:

```
character (len=*) :: character_len_argument
```

This assumes that the length from the corresponding actual argument in the calling program is the same as the length of the dummy argument in the subroutine.

The other difficulty is the redundant spaces at the beginning or the end of the two names and then inserting exactly one character between them. The intrinsic function, **trim**, removes any trailing blanks from argument and **adjustl** moves argument enough spaces to the left to remove any leading blank

```
      program make_full_name
!     Variable declaration
      character (len=80)::name_1,name_2,name
!     read the first name
      print *,"enter the first name"
      read *,name_1
!     read the surname
      print *,"enter the surname"
      read *,name_2
!     combine two names by using the subroutine named
           get_full_name
      call get_full_name(name_1,name_2,name)
      print *,"the combination of the two names is => ",name
      end program make_full_name
      subroutine get_full_name(first_name,last_name,full_name)
!     Subroutine to join two names to form a full name with
      a single space between the first and last names
!     Dummy argument declaration
      character (len=*), intent(in) ::first_name,last_name
      character (len=*), intent(out) ::full_name
!     use adjustl to remove redundant leading blanks, and trim
!     to remove redundant blanks atr the end of first name
      full_name = trim(adjustl(first_name)) // " " //
     c            adjustl(last_name)
      end subroutine get_full_name
```

some important points

Although the dummy arguments are declared to be of assumed length, the corresponding actual arguments are declared with specific lengths in the calling program (here main program) unit.

The result of the **adjustl** function has been used as the argument to **trim**.

The first function **adjustl** moves its argument, *first_name*, to the left to eliminate any leading blanks

the second **trim** takes the result and removes any trailing blanks.

Local and global objects

A module may be used to make variables and constants available to several procedures.
This can be performed using a **public** attribute in the declaration statement.
Any **public** statements must appear before any **declaration** statements.

For example if one wished to use the values of π, g (acceleration of gravity) and e in a number of different procedures, a simple module performing this would be,

```
module Natural_Constants
real,parameter,public::pi=3.1415927,g=9.81,e=2.7182818
end module Natural_Constants
```

In order to obtain access to the constants defined in the module Natural_Constants the **use** statement must appear immediately after the initial statement, which may be a module, a procedure or a main program as shown in *the module_example_1* example.

```
Program module_example_1
Use Natural_Constants
print *,g,pi
end program module_example_1
```

In this example, the values, g and π, are printed out without any assignment to them but the module, Natural_Constants, must be compiled before Program module_example_1.

Giving procedure variables an initial name

If there is a situation when a variable in a procedure may be required to have an **initial value** on the first reference to the procedure the **save** attribute can be used.

```
real, save :: b=1.23
integer, save :: count=0
```

In this program part example, the initial values are assigned to the variable *b* as the real and the variable *count* as the integer.

Procedures as an aid to program structure

The modular program development, which is the key concept of the software engineering, enables us to break the design of a program into several smaller, more manageable sections known as modules, and procedures.

But this requires much more experience and effort on programming.