Introduction to
Scientific & Engineering Computing
BIL 102FE (Fortran) Course
for
Week 8

Dr. Ali Can Takinacı

Assistant Professor

in

The Faculty of Naval Architecture and Ocean Engineering

80626

Maslak – Istanbul – Turkey

IMPROVED BUILDING BLOCKS

Procedures and modules were first introduced at week 4 as the fundamental building blocks in F programming. This chapter introduces several important extensions.

Recursion is an important mathematical concept, and both function and subroutines may be delayed to be recursive order to allow their use in appropriate recursive algorithms. Both recursive and non-recursive procedures in order to provide still more flexibility to a program.

F includes the capability for programmers to create their own data types to supplement the five intrinsic types, which are integer, real, character, logical and complex.

Since these data types must be derived from the intrinsic data types they are called **derived types**.

Recursive procedures

Since the concept is easier to understand, the recursive procedures shall be examined with functions, and then it will be extended to subroutines.

If a function is called recursively, either directly or indirectly, then the word `recursive` must be added before function in the initial statement:

`recursive function` *recursive_function_name*(...) `result`(*result*)

The calculation of factorials, which is a recursive algorithm, will be illustrated.

Example: Write a function to calculate n!.

Analysis: The factorial n is written by mathematicians as n! and is defined as follows:

$$0! = 1, \text{ and } n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 \text{ for } n \geq 1$$

Another, recursive, way of expressing this is:

for $n = 0$ => $n! = 1$

for $n \geq 1$ => $n! = n \times (n-1)$

Structure Plan

```
1  Select case on n
  1.1   n = 0
   1.1.1 factorial_n = 1
  1.2   n > 0
   1.2.1 factorial_n = n*factorial(n-1)
  1.3   n < 0
   1.3.1 Error – return factorial n = 0
```

```fortran
  recursive function factorial(n) result(factorial_n)
! dummy argument and result variable
  integer, intent(in) :: n
  real :: factorial_n
! determine whether further recursion is required
  select case (n)
  case(0)
! recursion has reached the end
  factorial_n = 1.0
  case (1:)
! more recursive calculation(s) requires
  factorial_n = n*factorial(n-1)
  case default
! n is negative - return zero as an error indicator
  factorial_n = 0.0
  end select
  end function factorial
```

A recursive subroutine operates in much the same way, and it is specified by including the word `recursive` before `subroutine` in the initial statement of the subroutine:

```
 recursive subroutine factorial(n, factorial_n)
!   dummy argument and result variable
    integer, intent(in) :: n
    real, intent(out)   :: factorial_n
!   determine whether further recursion is required
    select case (n)
    case(0)
!   recursion has reached the end
    factorial_n = 1.0
    case (1:)
!   recursive call(s) required to obtain (n-1)!
    call factorial(n-1, factorial_n)
    factorial_n = n*factorial_n
    case default
!   n is negative - return zero as an error indicator
    factorial_n = 0.0
    end select
    end subroutine factorial
```

Passing procedures as arguments

It is possible to have a procedure as a dummy argument, in which case the dummy argument is called a dummy procedure.

However, the declaration of a dummy procedure takes a quite different form from that of any other type of dummy argument.

For the purpose of the declaration is to provide information about the procedure’s **interface** in contrast to a dummy variable where only the type and certain other attributes (real, integer, character or logical) are required.

```
interface
   interface_body
end interface
```

where the syntax of the interface_body is the same as that of a procedure, but without any declarations of local variables and without any executable statements. For example, the interface block for a function, which takes two real arguments and delivers a real result might be

```
interface
    function dummy_fun(a,b) result(r)
      real, intent(in) :: a, b
      real :: r
    end function dummy_fun
end interface
```

If there are several dummy procedures then all the interface bodies may be included in a single interface block:

```
interface
      subroutine one_arg(x)
            real, intent(inout) :: x
      end subroutine one_arg

      recursive subroutine two_args(x,y)
            real, intent(inout) :: x, y
      end subroutine two_args
end interface
```

The interface of the actual procedure argument corresponding to a dummy procedure must agree with that of the dummy procedure except that its name. The name of any dummy arguments or result variable may be different.

Example: Write a program which uses a procedure to print the values of a function for a sequence of values between two specified limits, and test the procedure with the following functions and values of x:

1.) $x^3 - 3x^2 - 4x + 12 = 0$
2.) $2e^x - e^{-x} = 0$
3.) $\sin(2x) - 2\cos(x) = 0$

Analysis: This program requires a module containing the three functions,

a second module containing the print procedure, and a main program to set things going.

It will be proceeded directly to the solution, using two modules.

1. First module: It contains the definitions of three functions.

```
module functions
public :: f1,f2,f3
contains
function f1(x) result(fx)
real, intent(in) :: x
real :: fx
fx = x**3 - 3.0-x*x - 4.0*x + 12.0
end function f1
function f2(x) result(fx)
real, intent(in) :: x
real :: fx
fx = 2.0*exp(x) - exp(-x)
end function f2
function f3(x)  result(fx)
real, intent(in) :: x
real :: fx
fx = sin(2.0*x) - 2.0*cos(x)
end function f3
end module functions
```

2. Second module: It contains the print and interface procedures.

```
    module use_functions
    public :: list_function
    contains
    subroutine list_function(f,x1,x2,xinc)
    !     dummy arguments
    interface
    function f(x) result(fx)
    real, intent(in) :: x
    real :: fx
    end function f
    end interface
    real, intent(in)::x1,x2,xinc
    !     local variable
    real :: x
    !     loop to print values of f(x) for specified values of x
    x = x1
    do
    print *,"x=",x,"f(x)=",f(x)
    x = x + xinc
    if (x > x2) then
    exit
    end if
    end do
    end subroutine list_function
end module use_functions
```

3. Main program.

```
program test_functions
use functions
use use_functions
real, parameter :: pi=3.1415927, twopi=2.0*pi,
piby4=0.25*pi
print *,"f(x) = x**3 - 3.0*x*x - 4.0*x + 12"
call list_function(f1, -4.0, 4.0, 0.5)
print *,"f(x) = 2.0*exp(x) - exp(-x)"
call list_function(f2, -10.0, 10.0, 1.0)
print *,"f(x) = sin(2.0*x) - 2.0*cos(x)"
call list_function(f3, -twopi, twopi, piby4)
end program test_functions
```

Creation of special data types

F includes the capability for programmers to create their own data types to supplement the five intrinsic types, which are integer, real, character, logical and complex. Since these data types must be derived from the intrinsic data types they are called **derived types**.

A derived data is defined by a special sequence of statements, which in their simplest form are as follows:

```
   type, public:: new_type
        component_definition
        .
        .
        .
end type new_type
```

There may be as many component definitions as required, and each takes the same form as a variable declaration. Unlike the declaration of variables, however, derived type definitions may *only* appear in a module. It gains an access to the new data type with a **public** attribute. It is also permissible to declare derived types to be **private**, but then the type is only available within the module.

As an example a new data type called person, which would contain all information, could be defined as:

```
type, public:: person
character (len=12) :: first_name
character (len=1)  :: middle_initial
character (len=12) :: last_name
integer :: age
character (len=1) :: sex    ! Male or Female
character (len=11) :: social_security
end type person
```

Once a new type has been defined the variables may be declared in a similar way to that used for intrinsic types:

```
type(person) :: jack, jill
```

Such declarations will need access to the type a definition, which is why such definitions must always be placed in a module.

A constant value of a derived type is written as a sequence of constants corresponding to the components of the derived type, enclosed in parentheses and preceded by the type name:

```
jack = person(“Jack”,”R”,”Hagenbach”,47,”M”,”123-45-6789”)
jill = person(“Jill”,”M”,”Smith”,39,”F”,”987-45-6789”)
```

This form of defining a constant value for derived type is called a **structure constructor**.

A component of a derived type variable is referred directly by following the name of variable by a percentage sign and the name of the component.

The following statement changes the last name of `Jill` to that of `Jack`, for example if she had married with `Jack`.

```
jill%last_name = Jack%Last_name
```

A derived type can be used in the definition of another derived type:

```
    type, public:: employee
    type(person) :: employee
          character (len=20) :: department
    real :: salary
end type employee
```

However, operations between two objects of the same derived type are more difficult because although it would be meaningful to write

```
Pat%salary - Tom%salary
```

to establish the difference between the salaries of `Pat` and `Tom`, since both are real values the expression

```
Pat%department - Tom%department
```

is meaningless because both components are character strings.

Example 1: Define two data types, one to represent a point by means of its coordinates (in two-dimensional space only) and the other to represent a line (also in two-dimensional space) by the coefficients of its defining equation. Write a program which reads the coordinates of two points and which then calculates the line joining them, printing the equation of the line.

Analysis: First the two derived types – `point` and `line` must be established. The `point` consists of two real components, representing the x and y coordinates, respectively.

A straight line is defined by an equation of the form $ax + by + c = 0$. From simple analytical geometry knowledge, the coefficients can be defined with

$a = y_2 - y_1$; $b = x_1 - x_2$; $c = y_1x_2 - y_2x_1$

where (x_1, y_1) and (x_2, y_2) are the two coordinate points on the line.

The structure plan would be:

```
Module geometry
(defines derived types for points and lines)

function distinct_points(p1,p2) result(distinct)
1. Set distinct true if x_coordinates or y_coordinates differ

function line_from_points(p1,p2) result(joint_line)
2. Calculate and return the coefficients of the line joining
   the points p1 and p2.
```

```
program geometry_example
(uses geometry module)

function distinct_points(p1,p2) result(distinct)
1. Reads coordinates of two points

2. If the points are distinct
  2.1. Calculate the coefficients of the line joining
       the points
  2.2. Print equation of line
otherwise
  2.3. Print an error message
```

Solution: The module and program would be:

```
module geometry
public :: distinct_points,line_from_points

!     type definitions
      type, public :: point
!     cartesian coordinates of the point
      real :: x,y
      end type point

      type, public :: line
!     coefficients of defining equation
      real :: a,b,c
      end type line

!     constant declaration
      real, parameter, public :: small = 1.0e-5

      contains
```

```
function distinct_points(p1,p2) result(distinct)
!   returns true if the two points supplied as arguments
!   are not efficiently coincident
!   dummy arguments and result variable declaration
    type(point), intent(in) :: p1,p2
    logical :: distinct
!   set result true if either pair of corresponding
!   coordinates are different
distinct= abs(p1%x-p2%x)>small .or. abs(p1%y-p2%y)>small
end function distinct_points
function line_from_points(p1,p2) result(join_line)
!    returns the line joining the two points supplied as
!    arguments
!    dummy arguments and result variable declaration
     type(point), intent(in) ::p1,p2
     type(line) :: join_line
!    calculate coefficients of line
     join_line%a = p2%y - p1%y
     join_line%b = p1%x - p2%x
     join_line%c = p1%y*p2%x - p2%y*p1%x
end function line_from_points
end module geometry
```

```
     program geometry_example
    !   A program to use derived types for two-dimensional
    !   geometric calculations
        use geometry
    !   contains point and line type definitions
    !   constant small definition and functions
    !   distinct_points and line_from_points
    !   variable and constants declarations
        type(point) :: p1,p2
        type(line) :: p1_to_p2
    !   read data
        print *, "enter coordinates of first point (x,y)"
        read *, p1
        print *, "enter coordinates of second point(x,y)"
        read *, p2
```

```
    !   test for coincident points
        if (distinct_points(p1,p2)) then
    !   calculate coefficients of equation representing the line
        p1_to_p2 = line_from_points(p1,p2)
    !   print result
        print *,"the equation of the line joining these two"
        print *,"points is ax + by + c = 0"
        print *,"where a =",p1_to_p2%a
        print *,"      b =",p1_to_p2%b
        print *,"      c =",p1_to_p2%c
        else
        print *,"error: the two points supplied are coincident!"
        end if
       end program geometry_example
```

Another example will make this concept clearer

Example 2: Define a data type which can be used to represent complex numbers, and then use it in a program which reads two complex numbers and calculates and prints their sum, difference and product.

Analysis: The rules for addition, subtraction and multiplication are simply derived as:

$$(x_1,y_1) + (x_2,y_2) = (x_1+ x_2, y_1+ y_2)$$
$$(x_1,y_1) - (x_2,y_2) = (x_1- x_2, y_1- y_2)$$
$$(x_1,y_1) * (x_2,y_2) = (x_{1*}x_2 - y_{1*}y_2 , x_{1*}y_2 + x_{2*}y_1)$$

The structure plan would be:

1. Define a data type for complex numbers
2. Read two complex numbers
3. Calculate their sum, difference and product
4. Print result

The module and program would be:

```
    module complex_arithmetic
!   this module contains a derived type
!     definition
!       for complex numbers
    type, public :: complex_number
   real :: real_part, imaginary_part
   end type complex_number
   end module complex_arithmetic
```

```
program complex_example
!  A program to illustrate the use of a derived type to
perform
!  complex arithmetic
use complex_arithmetic
!     variable definitions
type(complex_number) ::c1,c2,csum,cdif,cprod
!     read data
print *,"enter first complex number in the form of (x,y)"
read *, c1
print *,"enter second complex number in the form of
(x,y)"
read *, c2
!     calculate sum, difference and product
csum%real_part=c1%real_part+c2%real_part
csum%imaginary_part=c1%imaginary_part+c2%imaginary_part
cdif%real_part = c1%real_part-c2%real_part
cdif%imaginary_part=c1%imaginary_part - c2%imaginary_part
```

```
    cprod%real_part = c1%real_part*c2%real_part-
                c1%imaginary_part * c2%imaginary_part
    cprod%imaginary_part = c1%real_part * c2%imaginary_part +
                     c1%imaginary_part * c2%real_part
    !  print results
    print *,"the sum of the two numbers is", csum
    print *,"the difference between the two numbers is",cdif
    print *,"the product the two numbers is",cprod
   end program complex_example
```

Controlling access to entities within a module

Derived types provide good programming practice to group related variables together in a derived type definition in a module in order that the type may then be easily used throughout a program. The access control within a module can be supplied by using **private** attribute. This restricts the use of the derived component in a module.

```
type, public :: complex_number
   private
   real :: a, phi
  end type complex_number
```

The privacy only applies *outside* the module in which the type definition appears. Within the module, including all its module procedures, the components are fully accessible.

Example: In the previous example a data type to represent complex numbers was defined and used this to carry out addition, subtraction and multiplication. Write a module, which contains a similar data type, whose components are hidden from the user of the module, and which also contains four procedures to carry out addition, subtraction, multiplication and division between two complex entities.

Analysis: It has been already carried out most of the work for this module, other than complex division and the four new procedures to carry out input/output and conversion. Complex division was not discussed in that example, but is included here for completeness. The formula required is as follows:

$$\frac{(x_1, y_1)}{(x_2, y_2)} = \left(\frac{x_1 * x_2 + y_1 * y_2}{x_2^2 + y_2^2}, \frac{x_2 * y_2 - x_1 * y_2}{x_2^2 + y_2^2} \right)$$

Input and output procedures are needed because derived type input and output takes place component by component, and the components will not be accessible outside the module. Two conversion procedures are required in order to allow access to the real and imaginary parts. They are all quite straightforward, however, and it can be proceeded straight to the solution.

The principle of data hiding or, more generally, of only allowing access to a restricted set of the entities in a module is extremely important for secure programming.

Solution: The module and program would be:

```
  module complex_procedures
      public :: c_add, c_subt, c_mult, c_divs, print_complex
      public :: read_complex, create_complex, extract_complex

!     complex data derived type definition
      type, public :: complex_number
      private
      real :: real_part, imag_part
      end type complex_number

      contains

      function c_add(z1,z2) result(c_sum)
      type(complex_number), intent(in) :: z1, z2
      type(complex_number) :: c_sum
      c_sum%real_part = z1%real_part + z2%real_part
      c_sum%imag_part = z1%imag_part + z2%imag_part
      end function c_add
```

```
        function c_subt(z1,z2) result(c_sub)
        type(complex_number), intent(in) :: z1, z2
        type(complex_number) :: c_sub
        c_sub%real_part = z1%real_part - z2%real_part
        c_sub%imag_part = z1%imag_part - z2%imag_part
        end function c_subt

        function c_mult(z1,z2) result(c_mul)
        type(complex_number), intent(in) :: z1, z2
        type(complex_number) :: c_mul
!       local variable to avoid writing more data
        real ::temp_1, temp_2
        temp_1 = z1%real_part * z2%real_part
        temp_2 = z1%imag_part * z2%imag_part
        c_mul%real_part = temp_1 - temp_2
        temp_1 = z1%real_part * z2%imag_part
        temp_2 = z1%imag_part * z2%real_part
        c_mul%imag_part = temp_1 + temp_2
        end function c_mult
```

```fortran
      function c_divs(z1,z2) result(c_div)
      type(complex_number), intent(in) :: z1, z2
      type(complex_number) :: c_div
!     local variable to avoid writing and calculating more data
      real ::temp_1, temp_2, denom
      denom = z2%real_part**2 + z2%imag_part**2
      temp_1 = z1%real_part * z2%real_part
      temp_2 = z1%imag_part * z2%imag_part
      c_div%real_part = (temp_1 + temp_2) / denom
      temp_1 = z2%real_part * z1%imag_part
      temp_2 = z1%real_part * z2%imag_part
      c_div%imag_part = (temp_1 - temp_2) / denom
      end function c_divs

      subroutine print_complex(z)
      type(complex_number), intent(in) :: z
!     can not be done outside module
      print *,z
      end subroutine print_complex
```

```
        subroutine read_complex(z)
        type(complex_number), intent(out) :: z
!       can not be done outside module
        read *,z
        end subroutine read_complex

        subroutine create_complex(real_part, imag_part,z)
        real, intent(in) :: real_part, imag_part
        type(complex_number), intent(out) :: z
        z%real_part = real_part
        z%imag_part = imag_part
        end subroutine create_complex

        subroutine extract_complex(real_part, imag_part,z)
        type(complex_number), intent(in) :: z
        real, intent(out) :: real_part, imag_part
        real_part = z%real_part
        imag_part = z%imag_part
        end subroutine extract_complex

       end module complex_procedures
```

```
        program test_complex
        use complex_procedures

!       variabvle declaration
        type(complex_number) ::z1, z2, z3
        real :: re, im

!       read a complex number
        print *, "enter two complex numbers (z1,z2)"
        call read_complex(z1)

!       read two reals and form a complex number
        read *, re,im
        call create_complex(re,im,z2)

!       multiply the two complex numbers and print their product
        z3 = c_mult(z1,z2)
        print *, "the product of these two numbers (z1*z2) is"
        call print_complex(z3)
```

```
!     add the two complex numbers asnd print the real
!     and imaginary parts of their sum
      z3 = c_add(z1,z2)
      call extract_complex(re,im,z3)
      print *,"the sum of these two numbers (z1+z2) is ", re,im

!     subtract the two complex numbers and print their result
      z3 = c_subt(z1,z2)
     print*,"the difference between the two numbers (z1-z2) is"
      call print_complex(z3)

!     divide the two complex numbers and print their result
      z3 = c_divs(z1,z2)
      print *, "the division of the two numbers (z1/z2) is"
      call print_complex(z3)

  end program test_complex
```