Technical University of Denmark

DANISH CENTER FOR APPLIED MATHEMATICS AND MECHANICS

High Performance Computing

FORTRAN, OpenMP and MPI

41391

Content

- Day 2:
  - Program units and procedures
    - Main program.
    - External subprograms.
    - Modules.
    - Internal subprograms.
    - Arguments of procedures.
    - Functions.
    - Explicit and implicit interfaces.
    - Recursion.
    - Overloading.

The main program

- A complete program must have ONE main program:

[PROGRAM *program-name*]

    [*specification-stmts*]

    [*executable –stmts*]

[CONTAINS

    internal-subprograms]

END [PROGRAM [*program-name*]]

- The END terminates the program.

The main program

Example:

```
PROGRAM test
CHARACTER(LEN=*) :: string = ‘hello world!’
PRINT*,string
END PROGRAM test
```

- Another way to stop the program is with the STOP statement:

```
STOP
STOP ‘Incomplete data. Program terminated.’
STOP 12345
```

The stop statement

Example:

```
program stop
implicit none
character(len=20), parameter :: msg = 'error message'
integer, parameter :: fatal = 444
integer :: info

info = 0
info = fatal
if (info.eq.fatal) then
   stop fatal
else
   stop msg
endif
end program stop
```

External subprograms

- Subroutine is declared by:

[prefix] SUBROUTINE *subroutine-name [(dummy-argument list)]*
    [*specification-stmts*]
    [*executable-stmts*]
    [CONTAINS
        internal-subprograms]
END SUBROUTINE [*subroutine-name*]

Prefix:

- RECURSIVE
- PURE
- ELEMENTAL

External subprograms

- Example:

```
SUBROUTINE example(x)
REAL :: x ! Dummy argument
PRINT*,'x = ',x
END SUBROUTINE example

PROGRAM main
REAL :: y
y = 12.0
CALL example(y) ! Actual argument passed to subroutine
CALL example(-4.0) ! Value pass to subroutine
END
```

External subprograms

- Internal procedure:
  - Only host program can use an internal procedure
  - An internal procedure has access to host entities by *host association*, that is, names declared in the host program unit are usable within the internal procedure.
  - Exercise: what is printed ?

```
program main
implicit none
real      :: x
integer :: info,i

x      = 1.0
info = 2
i      = 3
call sub(info)

contains
   subroutine sub(i)
   implicit none
   integer :: i
   print*,'i = ',i
   print*,'x = ',x
   end subroutine sub
end program main
```

External subprograms

- Function is declared by:

[prefix] FUNCTION *function-name [(dummy-argument list)]*
    [*specification-stmts*]
    [*executable-stmts*]
    [CONTAINS
        internal-subprograms]
END FUNCTION [*function-name*]

Prefix:
- RECURSIVE
- PURE
- ELEMENTAL

External subprograms

- Function returns a value:
  - The value returned by the function may be a *scalar* or an *array of intrinsic* or *user define types*.
  - An INTERFACE is required when the function value is an array or user defined type.
  - Excessive use of functions can be computationally costly – unless the compile can (is told to) *inline* the function (copy-in during the compilation).

External subprograms

- A FUNCTION is similar to a SUBROUTINE – but a FUNCTION returns a value and may enter expressions.

  Example:

```
IF (distance(a,b).GT.distance(b,c)) THEN
…
FUNCTION distance(a,b)
REAL :: distance
REAL, INTENT(IN) :: a,b
distance = …  ! The function MUST be given a value
END FUNCTION distance
```

Alternative declaration:
REAL FUNCTION distance(a,b)

External subprograms

- FUNCTIONs may change the input arguments, values in modules, rely on saved local data, or perform input-output operations.
- However, these are known as ***SIDE-EFFECTS*** and conflicts with good programming practice.
- Where these effects are needed, use SUBROUTINES !

Example:

```
FUNCTION distance(a,b)
REAL :: distance
REAL :: a,b ! Removed the INTENT(IN)
distance = sqrt(a**2 + b**2)
IF (distance.GT.0.2) a = 0.0 ! Legal but BAD programming style
END FUNCTION distance
```

The RETURN statement

- The control may be returned to the calling program before the END statement by the RETURN statement.

  Example:

```
SUBROUTINE example(x)
REAL :: x
IF (x.LT.0.0) RETURN ! If x<0 exit the SUBROUTINE
…
END SUBROUTINE example
```

Example – main with subroutines

```
PROGRAM game
INTEGER :: info
CALL shuffle(info) ! Shuffle the cards.
CALL deal(info)    ! Deal the cards.
CALL play(info)    ! Play the game.
CALL display(info) ! Display the result
END PROGRAM game
```

Example – main with functions

```
PROGRAM game
INTEGER :: info
info = shuffle ! Shuffle the cards.
info = deal     ! Deal the cards.
info = play     ! Play the game.
info = display ! Display the result
END PROGRAM game
```

Both examples (using subroutines or functions) require that we can pass and access the variables between the different procedures. There are two possibilities:
(1) through global data (MODULE/COMMON) or
(2) argument list.

Modules

- Packaging of:
  - Global data.
  - Derived types.
  - Operations (internal subprograms).
  - Interface blocks.
- Encapsulation of data and associated operations.

ADVICE: use modules:

1) To encapsulate EVERY routine in its own single module.
2) To carry GLOBAL data.
3) For implicit interface blocks.
4) For local arrays.

Modules

- Module is declared by:

    MODULE *module-name*
    [*specification-stmts*]
    [CONTAINS
        module-subprograms]
END [MODULE [*module-name*]]

- The module is accessed by:

        USE *module-name*

Modules

Example: the module ‘state’ is declared in the FORTRAN file: m_state.f:

```
MODULE state
    INTEGER, DIMENSION(:), POINTER :: cards
    INTEGER :: ncards
END MODULE state
```

and accessed in the main program and the subprograms by:

```
USE state
```

The module is a FORTRAN file (not a header file). It MUST be compiled before the compilation of the subprograms using the module. ALSO the subprograms MUST be recompiled if the module file is modified (and hence recompiled).

Modules

- Module may use other modules – but NOT itself – directly or indirectly!
- Example:

```
SUBROUTINE shuffle
 USE state
 IMPLICIT NONE ! Comes after the USE statement
 INTEGER :: info
 ncards = 52
 ALLOCATE(cards(ncards)) !cards is now allocated and
 … shuffle the cards          ! stored in the module
 END SUBROUTINE shuffle
```

COMMON BLOCKS

- In FORTRAN 77, global data is contained in COMMON BLOCKS:
  COMMON /*common-name/ common-list*
  declaration-stmts
- The common block name has global scope.

  Example:
  COMMON /hand/ cards,ncard
  INTEGER cards(52)
  INTEGER ncards

COMMON BLOCKS

- In FORTRAN 77 all arrays are fixed in size at compile time!
- To access the data, INCLUDE the common block in the subprogram.
- Example:

```
    SUBROUTINE shuffle
    IMPLICIT NONE
    INCLUDE 'common.inc'
...
```

Formally, each common block NOT present in the main program is required to have the: SAVE /common-name/

Arguments to procedures

- An alternative to MODULEs is to pass *arguments* to procedures:
  - Declare the variable in the subprogram that calls the subroutine/function.
  - Pass the *actual* arguments to the subroutine/function.
  - The subroutine/function receives the variables as *dummy* arguments.

Arguments to procedures

- Example:

```
PROGRAM main
REAL :: x
CALL SUB(x)          ! Actual argument (pass by reference)
CALL SUB(1.5)        ! Actual argument (pass by value)
CALL SUB(x+1.5)      ! Actual argument (pass by value)
…
SUBROUTINE (y)
REAL :: y ! Dummy argument
…
```

Dummy arguments

- Dummy arguments must match
  – The type.
  – The type parameters.
  – The shape.
- The name does not have to be the same – from that the name: "dummy" argument.
- Actual arguments may be:
  – a variable.
  – an expression.
  – a procedure name.

Pointer arguments

- If the actual arguments is a POINTER, the dummy argument *may* also be a POINTER:
  - When the subprogram is called the rank must match, and the pointer association status is copied to the dummy argument.
- A POINTER actual argument is also permitted to correspond to a non-POINTER dummy argument.
  - In that case, the pointer must have a target and the target is associated with the dummy argument.

Pointer arguments

- Example:

```
REAL, POINTER, DIMENSION(:,:) :: array
ALLOCATE(array(80,80))
CALL find(array)
...
SUBROUTINE find(c)
REAL, DIMENSION(:,:) :: c        ! Assume shape array
```

Actual argument

Dummy argument

Assume shape arrays require an INTERFACE block in the calling procedure (more tomorrow)

Pointer arguments

- Example:

```
USE m_alloc
REAL, DIMENSION(:,:), POINTER :: array
INTEGER :: n
n = 1000
CALL alloc(array,n)
…
MODULE m_alloc ! We place alloc() in a module to create
    CONTAINS        ! the interface block automatically
    SUBROUTINE alloc(c,n)
    REAL, DIMENSION(:,:), POINTER :: c
    INTEGER, INTENT(IN) :: n
    IF (.NOT.ASSOCIATED(c)) THEN
        ALLOCATE(c(n,n))
```

Allocatable arguments

- Example (**F95, F0x – not F90**):

```
USE m_alloc
REAL, DIMENSION(:,:) , ALLOCATABLE :: array
INTEGER :: n
n = 1000
CALL alloc(array,n)
…
MODULE m_alloc ! We place alloc() in a module to create
    CONTAINS      ! the interface block automatically
    SUBROUTINE alloc(c,n)
    REAL, DIMENSION(:,:), ALLOCATABLE :: c
    INTEGER, INTENT(IN) :: n
    IF (.NOT.ALLOCATED(c)) THEN
        ALLOCATE(c(n,n))
```

Argument INTENT

- Example:

```
SUBROUTINE shuffle(ncards,cards)
INTEGER :: ncards,icard
INTEGER, DIMENSION(ncards) :: cards
DO icard=1,ncards
      cards(icard) = …
ENDDO
END SUBROUTINE shuffle
```

It is not immediately clear if the variable ncards is modified by the routine!
The user will have to go through the code to see if ncards appear on the LHS.

Argument INTENT

- We can specify the intent on the declaration statement: example:

```
SUBROUTINE shuffle(ncards,cards)
INTEGER, INTENT(IN) :: ncards
INTEGER :: icard
INTEGER, DIMENSION(ncards), INTENT(OUT) :: cards
```

- If a dummy argument has no explicit intent, the actual argument may be a variable or an expression, but the actual arguments MUST be a variable if the dummy argument is redefined.

```
CALL sub(i+4)
...
SUBROUTINE sub(k)
INTEGER, INTENT(OUT) :: k ! will not work;
                          ! the compiler will complain
```

Argument INTENT

- Traditionally, FORTRAN compilers do not check this! (routines are compiled independently) .
- Breaking the rule results in program errors at run time – and are hard to find !
- Thus, INTENT(IN,OUT,INOUT) is recommended.
- INTENT is not allowed (in F90/95) if the dummy argument is a POINTER: ambiguity about INTENT: target or pointer association ?

```
program main
call sub(10)
end program main

subroutine sub(i)
integer :: i
i = 23
end subroutine sub
```

Interfaces

- The *interface* to a subprogram describes:
  - The type of subprogram (SUBROUTINE/FUNCTION).
  - The name and type of the arguments.
  - The properties of the result if the subprogram is a FUNCTION.
- The interface is **required** for:
  - assumed shape arrays: “(:)-type” arrays.
  - allocation of dummy arrays with the POINTER/ALLOCATABLE attribute in subprograms.
  - optional arguments.

Interfaces

- Interfaces to internal subprograms are not required as they are contained within the same programming unit.
- Interfaces to external subprograms are obtained through either:
  - the USE statement (the external subprogram has to be in a MODULE; see example on p. 27) – OR:
  - explicit declaration

```
INTERFACE
    interface-body
END INTERFACE
```

Interface-body is an exact copy of the header of the subprogram.

Example:

```
INTERFACE
    SUBROUTINE sub(x,y)
    REAL :: x,y
    END SUBROUTINE sub
END INTERFACE
```

Procedures as arguments

- Procedures may be passed as arguments to subprograms

  Example: function

```
REAL FUNCTION minimum(a,b,fun) ! fun is a dummy arg.
REAL, INTENT(IN) :: a,b
INTERFACE ! Must appear before any executable statements !
    REAL FUNCTION fun(x)
    REAL, INTENT(IN) :: x
    END FUNCTION fun
END INTERFACE
REAL :: f,x
…
f = fun(x)
…
END FUNCTION minimum
```

Procedures as arguments

Example: main program

```
PROGRAM main
REAL :: a,b,f
REAL, EXTERNAL :: minimum,myfun

f = minimum(1.0,2.0,myfun)
END PROGRAM main

! Now define the specific function here:
REAL FUNCTION myfun(x)
REAL :: x
myfun = COS(x) ! example
END FUNCTION myfun
```

Keyword and optional arguments

- Dummy argument names may be declared OPTIONAL.

  Example:

```
MODULE m_opt
    CONTAINS
    SUBROUTINE opt(a,b,c,d)
    REAL, INTENT(IN) :: a,b
    REAL, OPTIONAL :: c,d
    IF (PRESENT(c)) THEN
    …
    ENDIF
    END SUBROUTINE opt
END MODULE m_opt
```

- Note, positional arguments (non-optional) may not follow optional arguments (so optional arguments consists of a list of trailing arguments).

Keyword and optional arguments

- Optional arguments are addressed with the keywords defined by the name of the dummy argument.
- Optional arguments require an INTERFACE.

  Example:

```
PROGRAM main
USE m_opt ! Access to implicit interface
REAL :: a1,a2,a3,a4
CALL opt(a1,a2)
CALL opt(a1,a2,a3)
CALL opt(a1,a2,d=a4)
END SUBROUTINE opt
```

Scope of labels and names

- *Scope*:
  - range of validity of an object label, name.
  - Important: knowing when you can safely reuse a variable name.
- *Scoping unit:*
  - A derived-type definition.
  - A procedure interface body, excl. any derived-type definitions, and interface bodies contained within it, or
  - A program unit or subprogram excl. derived-type definitions, interface bodies contained within it.

Scope of labels and names

- Example:

```
Module scope1
…
CONTAINS
    SUBROUTINE scope2
    TYPE scope 3
        …
    END TYPE scope3
    INTERFACE
        …           ! scope4
    END INTERFACE
    CONTAINS
    FUNCTION scope5()
    END FUNCTION scope5
    END SUBROUTINE scope2
END MODULE scope1
```

Recursion

- A subprogram may NOT call itself unless it has the prefix: RECURSION.
- Recursive *functions* are required to have a separate name for the result.
- A copy of the local data is created for each recursion.

Recursion

Example:

```
RECURSIVE FUNCTION fun(top) RESULT(s)
INTEGER :: s,top
IF (top.LT.1) THEN
      s = 0
      RETURN
ENDIF
s = top + fun(top-1)
END FUNCTION fun
```

What is fun(9) ?

Overloading and generic interfaces

- Explicit interface blocks may be used to perform *overloading*:
  - Overloading enables calls to several procedures through the same generic name.
  - Overloading can extend intrinsic operations or assignments.
- To overload a module procedure (where an explicit interface already exist) use MODULE PROCEDUREs.

Overloading and generic interfaces

```
INTERFACE gamma
    FUNCTION sgamma(x)
        INTEGER, PARAMETER :: MK = KIND(1.0E0)
        REAL(MK) :: x,sgamma
    END FUNCTION sgamma
    FUNCTION dgamma(x)
        INTEGER, PARAMETER :: MK = KIND(1.0D0)
        REAL(MK) :: x,dgamma
    END FUNCTION dgamma
END INTERFACE gamma
```

Overloading and generic interfaces

- If the sgamma and dgamma are already in the module:

```
INTERFACE gamma
    MODULE PROCEDURE sgamma, dgamma
END INTERFACE gamma
```

Overloading and generic interfaces

```
MODULE addition
    INTERFACE OPERATOR (+)
        MODULE PROCEDURE addchar
    END INTERFACE OPERATOR(+)
    CONTAINS
    FUNCTION addchar(a,b) RESULT(sum)
    CHARACTER(LEN=*), INTENT(IN) :: a,b
    CHARACTER(LEN=LEN(a)) :: sum
    INTEGER :: ilena,ilenb
    ilena = LEN_TRIM(a)
    ilenb = LEN_TRIM(b)
    sum = a(1:ilena)//b(1:ilenb) ! // is standard FORTRAN concatenation
    END FUNCTION addchar
END MODULE addition
```