Karlsruher Institut für Technologie

Fortran 95/2003 Course

Procedures and Modules by Hartmut Häfner
March 25, 2015

STEINBUCH CENTRE FOR COMPUTING - SCC

Main Program

- Form:

```
PROGRAM [name]
[specification statements]
[executable statements]
...
END [PROGRAM [name]]
```

Procedures: Functions and Subroutines

- Structurally, procedures may be:
  - External - self contained (not necessarily Fortran, but good luck with type conversion for arguments)
  - Internal - inside a program unit (new to Fortran 90)
  - Module - member of a module (new to Fortran 90)
- Fortran 77 has only external procedures

External Procedures

```
SUBROUTINE name (dummy-argument-list)
 [specification-statements]
 [executable-statements]
 ...
 END [SUBROUTINE [name]]
```

or

```
FUNCTION name (dummy-argument-list)
 [specification-statements]
 [executable-statements]
 ...
 END [FUNCTION [name]]
```

- Note: RETURN statement no longer needed.

Internal Procedures

- Each program unit can contain internal procedures
- Internal procedures are collected together at the end of a program unit and preceded by a `CONTAINS` statement
- Same form as external procedures except that the word `SUBROUTINE/FUNCTION` must be present on the `END` statement
- Variables defined in the program unit remain defined in the internal procedures, unless redefined there
  - New way of making "global" variables
- Nesting of internal procedures is not permitted

Example for Internal Procedure

```
PROGRAM main
  IMPLICIT NONE
  REAL :: a=6.0, b=30.34, c=98.98
  REAL :: mainsum
  mainsum = add()
CONTAINS
  FUNCTION add()
    REAL :: add
    ! a,b,c defined in 'main'
    add = a + b + c
  END FUNCTION add
END PROGRAM main
```

`INTERFACE` Blocks

- If an 'explicit' interface for a procedure is provided, compiler can check argument inconsistency
- Module and internal procedures have an 'explicit' interface by default
- External procedures have an 'implicit' interface by default
- An `INTERFACE` block can be used to specify an 'explicit' interface for external procedures
- Always use an `INTERFACE` block in the calling program unit for external procedures
- Similar in form and justification for to C function prototypes

Syntax of `INTERFACE` Blocks

- General form:

```
INTERFACE
  interface_body
END INTERFACE
```

where interface_body is an exact copy of the subprogram specification, its dummy argument specifications and its END statement

- Example:

```
INTERFACE
  REAL FUNCTION func(x)
    REAL, INTENT(IN) :: x
  END FUNCTION func
END INTERFACE
```

INTENT Attribute

- Argument intent - can specify whether an argument is for:

```
input (IN)
output (OUT)
or both (INOUT)
```

- Examples:

```
INTEGER, INTENT(IN) :: in_only
REAL, INTENT(OUT) :: out_only
INTEGER, INTENT(INOUT) :: both_in_out
```

Sample Program: NO INTERFACE

```
PROGRAM test
  INTEGER :: i=3,j=25
  PRINT *,'The ratio is ',ratio(i,j)
END PROGRAM test

REAL FUNCTION ratio(x,y)
  REAL,INTENT(IN):: x,y
  ratio=x/y
END FUNCTION ratio
```

Output

```
  Floating point exception
  Beginning of Traceback:
      Started from address 453c in routine 'RATIO'.
      Called from line 3 (address 421b) in routine ‘TEST'.
      Called from line 316 (address 21531b) in routine '$START$'.
  End of Traceback.
  Floating exception (core dumped)
```

Sample Program: WITH INTERFACE

```
PROGRAM test
  INTERFACE
    REAL FUNCTION ratio(x,y)
      REAL, INTENT(IN)::x,y
    END FUNCTION ratio
  END INTERFACE
  INTEGER :: i=3,j=25
  PRINT *,'The ratio is ',ratio(i,j)
END PROGRAM test

REAL FUNCTION ratio(x,y)
  REAL,INTENT(IN):: x,y
  ratio=x/y
END FUNCTION ratio
```

Output

```
cf90-1108 f90: ERROR TEST, File = ratio_int.f90, Line = 3, Column = 33
  The type of the actual argument, "INTEGER", does not match "REAL",
  the type of the dummy argument.
```

Sample Program: INTERNAL PROCEDURE

```
PROGRAM test
 INTEGER :: i=3,j=25
 PRINT *,'The ratio is ',ratio(i,j)

 CONTAINS

   REAL FUNCTION ratio(x,y)
     REAL,INTENT(IN):: x,y
       ratio=x/y
   END FUNCTION ratio

END PROGRAM test
```

Output

```
cf90-1108 f90: ERROR TEST, File = ratio_int.f90, Line = 3, Column = 33
  The type of the actual argument, "INTEGER", does not match "REAL",
  the type of the dummy argument
```

Using Keywords with Arguments

- With intrinsic Fortran functions, have always been able to use a keyword with arguments

```
READ(UNIT=10,FMT=67,END=789) x,y,z
```

- If interface provided, programmer can use keywords with their own f90 procedure arguments.

- ADVANTAGES: Readability and override order of arguments

Using Keywords with Arguments (2)

- Example Interface:

```
REAL FUNCTION area (start, finish, tol)
  REAL, INTENT(IN) :: start, finish, tol
  ...
END FUNCTION area
```

- Call with:

```
a = area(0.0, 100.0, 0.01)
b = area(start = 0.0, tol = 0.01, finish = 100.0)
c = area(0.0, finish = 100.0, tol = 0.01)
```

- Once a keyword is used, all the rest must use keywords

Optional Arguments

- If interface provided, programmer can specify some arguments to be optional
- Coder's responsibility to ensure a default value if necessary (use `PRESENT` function)
- The `PRESENT` intrinsic function will test to see if an actual argument has been provided.

Example

```
REAL FUNCTION area (start, finish, tol)
 IMPLICIT NONEREAL, INTENT(IN), OPTIONAL :: start, &
                                            finish, tol
 REAL :: ttol
 ...
 IF (PRESENT(tol)) THEN
   ttol = tol
 ELSE
   ttol = 0.01
 END IF
 ...
END FUNCTION area
```

- Need to use local variable ttol because dummy argument tol cannot be changed because of its `INTENT` attribute

Using Structures as Arguments

- Procedure arguments can be of derived type if:
  - the procedure is internal to the program unit in which the derived type is defined
  - or the derived type is defined in a module which is accessible from the procedure

Array-valued Functions

- In Fortran 90, functions may have an array-valued result:

```
FUNCTION add_vec (a, b, n)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n
  REAL, DIMENSION (n), INTENT(IN) :: a, b
  REAL, DIMENSION (n) :: add_vec
  INTEGER :: i
  DO i = 1, n
    add_vec(i) = a(i) + b(i)
  END DO
END FUNCTION add_vec
```

Recursive Procedures

- In Fortran 90, procedures may be called recursively:
  - either A calls B calls A, or
  - A calls A directly (`RESULT` clause required).
- The `RESULT` clause specifies an alternate name (instead of the function name) to contain the value the function returns.
- `RESULT` clause **required** for recursion.

Recursive Procedures (2)

- Must be defined as recursive procedure:

```
RECURSIVE FUNCTION fact(n) RESULT(res)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n
  INTEGER :: res

  IF (n == 1) THEN
    res = 1
  ELSE
    res = n * fact(n - 1)
  END IF
END FUNCTION fact
```

Generic Procedures

- In Fortran 90, you can define your own generic procedures
- Need distinct procedures for specific type of arguments and a „generic interface“:

```
INTERFACE generic_name
  specific_interface_body
  specific_interface_body
  ...
END INTERFACE
```

- Each distinct procedure can be invoked **using the generic name only**
- The actual procedure used will depend on the type of arguments

Example: Generic Subroutine swap

- Consider the following two external subroutines for swapping the values `REAL` and `INTEGER` variables:

```
SUBROUTINE swapreal (a, b)
 REAL,INTENT(INOUT):: a,b
 REAL :: temp
 temp = a
 a = b
 b = temp
END SUBROUTINE swapreal
```

```
SUBROUTINE swapint (a, b)
 INTEGER,INTENT(INOUT):: a,b
 INTEGER :: temp
 temp = a
 a = b
 b = temp
END SUBROUTINE swapint
```

Example: Generic Subroutine swap (2)

- The extremely similar routines can be used to make a generic swap routine with the following interface:

```
INTERFACE swap    ! generic name

  SUBROUTINE swapreal (a, b)
    REAL, INTENT(INOUT) :: a, b
  END SUBROUTINE swapreal

  SUBROUTINE swapint (a, b)
    INTEGER, INTENT(INOUT) :: a, b
  END SUBROUTINE swapint
END INTERFACE
```

In the main program, only the generic name is used

```
 INTEGER :: m,n
REAL :: x,y
 ...
CALL swap(m,n)
CALL swap(x,y)
```

Modules

- Very powerful facility with many applications in terms of program structure
- Method of sharing data and/or procedures to different units within a single program
- Allows data/procedures to be reused in many programs
  - Library of useful routines
  - "Global" data
  - Combination of both (like C++ class)
- Very useful role for definitions of types and associated operators

Modules (2)

- Form:

```
MODULE module-name
  [specification-stmts]
  [executable-stmts]
[CONTAINS
  module procedures]
END [MODULE [module-name]]
```

- Accessed via the `USE` statement

Modules: Global Data

- Can put global data in a module and each program unit that needs access to the data can simply `USE` the module
  - Replaces the old common block trick

```
MODULE globals
  REAL, SAVE :: a, b, c
  INTEGER, SAVE :: i, j, k
END MODULE globals
```

- Note the new variable attribute `SAVE`

Modules: Global Data (2)

- Examples of the USE statement:

```
USE globals                      ! allows all variables in the
                                 ! module to be accessed
USE globals, ONLY: a, c          ! Allows only variables
                                 ! a and c to be accessed
USE globals, r => a, s => b ! allows a,b to be accessed
                                  ! with local variables r,s
```

- `USE` statement must appear at very beginning of the program unit (right after `PROGRAM` statement)

Module Procedures

- Modules may contain procedures that can be accessed by other program units

- Same form as external procedure except:
  - Procedures must follow a `CONTAINS` statement
  - The `END` statement must have `SUBROUTINE` or `FUNCTION` specified.

- Particularly useful for a collection of derived types and associated functions that employ them.

Module Example: Adding Structures

```
MODULE point_module
  TYPE point
    REAL :: x, y
  END TYPE point
  CONTAINS
    FUNCTION addpoints (p, q)
      TYPE (point), INTENT(IN) :: p, q
      TYPE (point) :: addpoints
      addpoints%x = p%x + q%x
      addpoints%y = p%y + q%y
    END FUNCTION addpoints
END MODULE point_module
```

A program unit would contain:

```
USE point_module
TYPE (point) :: px, py, pz
 ...
pz = addpoints (px, py)
```

Modules: Generic Procedures

- A common use of modules is to define generic procedures, especially those involving derived types.

```
MODULE genswap
  TYPE point
    REAL :: x, y
  END TYPE point
  INTERFACE swap    ! generic interface
    MODULE PROCEDURE swapreal, swapint, swaplog, swappoint
  END INTERFACE
CONTAINS
  SUBROUTINE swappoint (a, b)
    TYPE (point), INTENT(INOUT) :: a, b
    TYPE (point) :: temp
    temp = a; a=b; b=temp
  END SUBROUTINE swappoint
  ...       ! swapint, swapreal, swaplog procedures are defined here
END MODULE genswap
```

Overloading Operators

- Can extend the meaning of an intrinsic operator to apply to additional data types - operator overloading

- Need an `INTERFACE` block with the form:

```
INTERFACE OPERATOR (intrinsic_operator)
  interface_body
END INTERFACE
```

Overloading Operators (2)

- Example: Define '+' for character variables

```
MODULE over
  INTERFACE OPERATOR (+)
    MODULE PROCEDURE concat
  END INTERFACE

  CONTAINS

  FUNCTION concat(cha, chb)
   CHARACTER (LEN=*), INTENT(IN) :: cha, chb
   CHARACTER (LEN=LEN_TRIM(cha) + LEN_TRIM(chb)) :: concat
     concat = TRIM(cha) // TRIM(chb)
   END FUNCTION concat

END MODULE over
```

Overloading Operators (2)

```
PROGRAM testadd
  USE over
  CHARACTER (LEN=23) :: name
  CHARACTER (LEN=13) :: word
  name='Balder'
  word='convoluted'
  PRINT *,name // word
  PRINT *,name + word
END PROGRAM testadd
```

Output

```
Balder       convoluted
Balderconvoluted
```

Defining Operators

- Can define new operators - especially useful for user defined types
- Operator name must have a '.' at the beginning and end
- Need to define the operation via a function which has one or two non-optional arguments with `INTENT(IN)`

Defining Operators (2)

- Example - find the straight line distance between two derived type „points“

```
PROGRAM main
  USE distance_module
  TYPE (point) :: p1, p2
  REAL :: distance
  ...
  distance = p1 .dist. p2
  ...
END PROGRAM main
```

Defining Operators (3)

- Where the "distance_module" looks like this:

```
MODULE distance_module
  TYPE point
    REAL :: x, y
  END TYPE point
  INTERFACE OPERATOR (.dist.)
    MODULE PROCEDURE calcdist
  END INTERFACE
CONTAINS
  REAL FUNCTION calcdist (px, py)
    TYPE (point), INTENT(IN) :: px, py
    calcdist = SQRT ((px%x-py%x)**2 + px%y-py%y)**2 )
  END FUNCTION calcdist
END MODULE distance_module
```

Assignment Overloading

- When using derived data types, may need to extend the meaning of assignment (=) to new data types:

```
REAL :: ax
TYPE (point) :: px
 ...
ax = px ! type point assigned to type real
        ! not valid until defined
```

- Need to define this assignment via a subroutine with two non-optional arguments, the first having INTENT(OUT) or INTENT(INOUT), the second having INTENT(IN) and create an interface assignment block:

```
INTERFACE ASSIGNMENT (=)
  subroutine interface body
END INTERFACE
```

Assignment Overloading (2)

- In the following example we will define the above assignment to give ax the maximum of the two components of `px`

```
MODULE assignoverload_module
  TYPE point
    REAL :: x,y
  END TYPE point
  INTERFACE ASSIGNMENT (=)
    MODULE PROCEDURE assign_point
  END INTERFACE
CONTAINS
  SUBROUTINE assign_point(ax,px)
    REAL, INTENT(OUT)::ax
    TYPE (point), INTENT(IN)::px
    ax = MAX(px%x,px%y)
  END SUBROUTINE assign_point
END MODULE assignoverload_module
```

Program Structure: Using Interface Blocks

- When a module/external procedure is called:
  - which defines or overloads an operator, or the assignment
  - using a generic name

- Additionally, when an external procedure:
  - is called with keyword/optional argument
  - is an array-valued/pointer function or a character function which is neither a constant nor assumed length
  - has a dummy argument which is an assumed-shape array, a pointer/target
  - is a dummy or actual argument (not mandatory but recommended)

Program Structure: Summary

- Fortran 77 style:
  - Main program with external procedures, possibly in a library
  - No explicit interfaces, so argument inconsistencies are not checked by compiler

    .
- Simple Fortran 90:
  - Main program with internal procedures
  - Interfaces are 'explicit', so argument inconsistencies are trapped by compiler.

- Fortran 90 with modules:
  - Main program and module(s) containing interfaces and possibly specifications, and external procedures (possibly precompiled libraries).

Program Structure: Summary (2)

- A Fortran 90 version of the Fortran 77 style - with interfaces to permit compiler checking of argument inconsistencies.

- Main program and module(s) containing specifications, interfaces and procedures. No external procedures.
  - Expected for sophisticated Fortran 90 programs.