

Fortran 95

Introductory course, 08-09 Oct 2009

Jarmo Pirhonen, Sami Saarinen, Tommi Bergman, Raimo Uusvuori

`firstname.lastname[at]csc.fi`

CSC - IT Center for Science Ltd.

CSC - Tieteen tietotekniikan keskus Oy

Espoo, Finland



Contents

- Overview
- Procedures
- File I/O and format descriptors
- Arrays
- Dynamic memory allocation and arrays as arguments
- Modules, Derived datatypes
- Transition to Fortran 95 (extra)
- New Features of Fortran 95/2003 (extra)

Fortran 90/95 overview

Contents

- Fortran 90/95
- Hints on programming style
- Variable declaration
- Control structures

Fortran 90/95

- Well suited for numerical computations
 - fast code (compiler!)
 - handy array data types
 - clarity of the code
 - portability of the code
 - availability of subroutine libraries
- Not recommended for
 - operating system programming (use C!)
 - development of user interfaces
 - Internet applications
 - data base applications

Example program

```
PROGRAM squarerootexample
  ! Program computes
  ! squarerrot of value
  ! plus one

  IMPLICIT NONE
  REAL :: x, y
  INTRINSIC SQRT

  WRITE (*,*) 'enter number x:'
  READ (*,*) x

  y = x**2 + 1

  WRITE (*,*) 'entered x:', x
  WRITE (*,*) 'value x**2 + 1:', y
  WRITE (*,*) &
    'value SQRT(x**2 + 1):', SQRT(y)
END PROGRAM squarerootexample
```

Fortran 90/95 new features

- new input format (*free-form source input*)
- transferable data types
- new control structures
- array syntax
- dynamic memory allocation
- derived data types
- modules (combine procedures and make them available to other program units)
- new standard procedures
- operator overloading, generic procedures
- recursion

Function declaration

```
PROGRAM function_example
  ! program reads in real number x and
  ! computes function value f(x).
  IMPLICIT NONE
  REAL :: x

  WRITE (*,*) 'Enter value x:'
  READ (*,*) x
  WRITE (*,*) 'Value x:', x
  WRITE (*,*) 'Result f(x):', f(x)
CONTAINS
  FUNCTION f(a) RESULT(f_out)
    IMPLICIT NONE
    REAL :: a, f_out
    f_out = a**2 + 1
  END FUNCTION f
END PROGRAM function_example
```

Function declaration ...

main program contains an *internal function* defined after **CONTAINS**-keyword:

```
CONTAINS
  FUNCTION f(a) RESULT(f_out)
    IMPLICIT NONE
    REAL :: a, f_out
    f_out = a**2 + 1
  END FUNCTION f
END PROGRAM function_example
```

which is called from the main program by $f(a)$.

Output:

```
Enter value x: 2.345
Value x: 2.345000
Result f(x): 6.499025
```



Arrays

Array declaration:

```
INTEGER, PARAMETER :: entries = 43
CHARACTER(LEN=30), &
    DIMENSION(entries) :: names
REAL, DIMENSION(entries) :: marks
```

Assigning values:

```
names(1) = 'George W.'
marks(1) = 10.0
names(2) = 'John A.'
marks(2) = 9.9
.
.
.
names(42) = 'Bill C.'
marks(42) = 4.1
names(43) = 'George W. B.'
marks(43) = 4.0
```

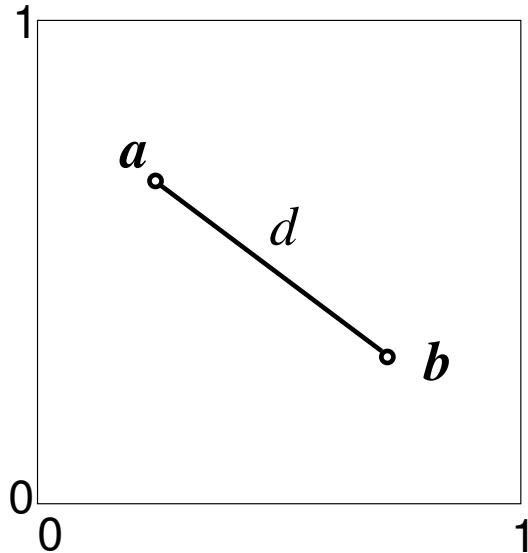
array functions and operations:

```
REAL :: average
average = SUM(marks)/entries
```



Example: a little simulation program

Random placement of N point pairs \mathbf{a} and \mathbf{b} , $0 \leq a_i \leq 1$ and $0 \leq b_i \leq 1$, $i = 1, 2$, inside a unit square



$$d = \sqrt{(\mathbf{a} - \mathbf{b})^2}$$

In search of the relative distance's d average value $E[d]$.

Mind:

`SQRT((a(1)-b(1))**2 + (a(2)-b(2))**2)`
is equivalent to `SQRT(SUM((a-b)**2))`

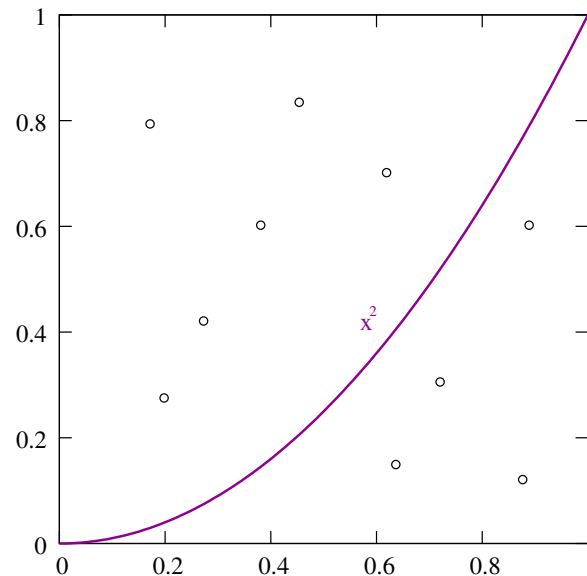
Example: a little simulation program

```
PROGRAM pointpairs
  ! Computes average distance between randomly
  ! placed point-pairs in unit square
  IMPLICIT NONE
  REAL, DIMENSION(2) :: a, b
  REAL :: d, s = 0.0
  INTEGER :: i, n
  WRITE (*,*) 'Enter amount of pairs:'
  READ (*,*) n
  IF (n > 0) THEN
    DO i = 1, n
      CALL RANDOM_NUMBER(a)
      CALL RANDOM_NUMBER(b)
      s = s + SQRT(SUM((a - b)**2))
    END DO
    d = s/n
    WRITE (*,*) 'Average distance (', n, ' pairs):', d
  ELSE
    WRITE (*,*) 'Negative number!'
  END IF
END PROGRAM pointpairs
```



Another little simulation program

- Monte-Carlo integration of $\int_0^1 f(x) dx = \int_0^1 x^2 dx$
- Random placements of N points inside a unit square
 $0 \leq a_i \leq 1 \quad i = 1, 2$



- number of coordinate pairs satisfying $a_2 < f(a_1)$: $M \leq N$
- Area: $A \approx M/N$

Another little simulation program

```
PROGRAM mcint
  ! MC integral of x^2
  IMPLICIT NONE
  REAL, DIMENSION(2) :: a
  REAL :: area
  INTEGER :: i, n, ct = 0
  WRITE (*,*) 'Enter amount of points: '; READ (*,*) n
  IF (n > 0) THEN
    DO i = 1, n
      CALL RANDOM_NUMBER(a)
      IF (a(2) <= a(1)**2) THEN
        ct = ct + 1
      END IF
    END DO
    area = (1.0*ct)/(1.0*n)
    WRITE (*,*) 'Area (', n, ' points):', area
  ELSE
    WRITE (*,*) 'Negative number!'
  END IF
END PROGRAM mcint
```



Module declaration

```
MODULE array_stuff
CONTAINS
  FUNCTION sequence(a, b, n)
    IMPLICIT NONE
    REAL, INTENT(IN) :: a, b
    INTEGER, INTENT(IN) :: n
    REAL, DIMENSION(n) :: sequence
    REAL :: start, step
    INTEGER :: i

    IF (n == 1) THEN
      sequence = a
    ELSE
      step = ABS(a - b)/(n-1)
      start = MIN(a,b)
      sequence = (/ (start + i*step, &
                    i = 0, n-1) /)
    END IF
  END FUNCTION sequence
END MODULE array_stuff
```



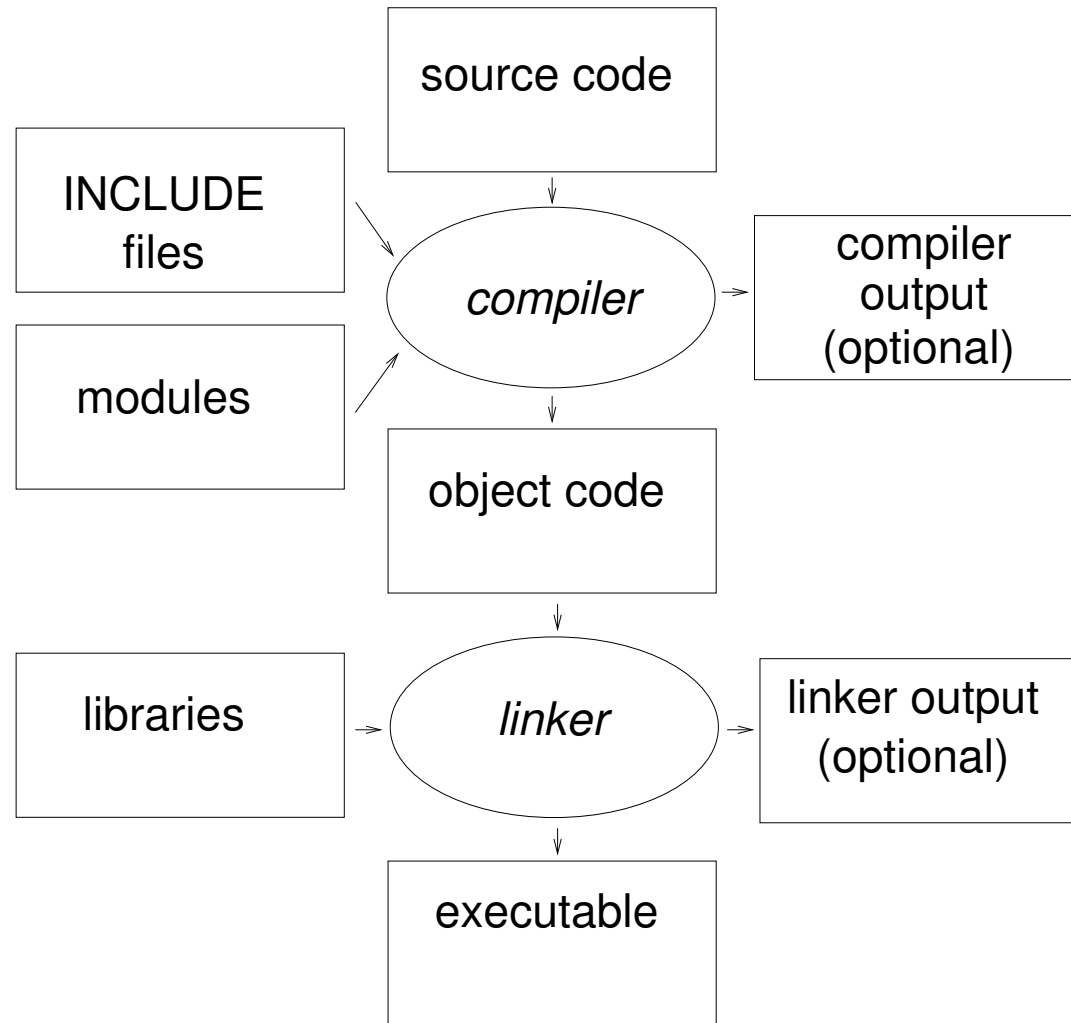
Example: module

```
PROGRAM array_test
  USE array_stuff
  INTEGER, PARAMETER :: n = 10
  REAL, DIMENSION(n) :: x, y
  CHARACTER(LEN=*), PARAMETER :: &
    form = '(A,/, (5F7.3))'
  x = sequence(0.0, 3.0, n)
  y = COS(x)
  WRITE (*,form) 'x: ', x
  WRITE (*,form) 'y: ', y
END PROGRAM array_test
```

Output:

```
x:
0.000 0.333 0.667 1.000 1.333
1.667 2.000 2.333 2.667 3.000
y:
1.000 0.945 0.786 0.540 0.235
-0.096 -0.416 -0.691 -0.889 -0.990
```

Compiling and linking



Hints on programming style

- Alternative ways of input:

- old fixed form input format (Fortran 77)

```
123456
```

```
|C This is a comment  
|      DO 9 J= 1, 10  
|      DO 9 K= 1, 10  
|9      L= J + K
```

- free-form source input

```
| ! This is a comment  
| DO J= 1, 10  
|   DO K= 1, 10  
|     L= J + K  
|   END DO  
| END DO
```

- do not mix input format styles in same program unit!

Free-form source input

- not column based
- maximum row length may be 132 characters
- names of variables may contain 31 characters (alpha-numerical and underscore)
- comments start with !
command separation ;
continued lines end with &
- max. 39 breaks per input line
- no distinction between lower and uppercase character

style: standard procedure calls and global variables in uppercase
local variables and self declared procedures in lowercase



A good programming style

- use free-form input!
- use indents; e.g. continued lines and nested loops

```
| WRITE(outputstring,'(a, i2)') &  
|     'Number of loop counts will be', M  
| DO J= 1, M  
|     DO K= 1, M  
|         L= J + K  
|     END DO  
| END DO
```

- write keywords in uppercase: `CALL my_subroutine(a,b)`
- variable identifiers should express their purpose:

| | |
|---|--|
| <pre>DO i=1,N a = a + x(i) END DO</pre> | <pre>DO i=1,NumberOfSamples totalsum = totalsum + sample(i) END DO</pre> |
|---|--|

Variable declaration

- Variables can be initialized at their declaration:

```
IMPLICIT NONE
```

```
INTEGER :: n = 0
```

```
INTEGER, DIMENSION(3) :: &  
  idx = (/ 1, 2, 3 /)
```

```
REAL, DIMENSION(-1:1) :: &  
  x = (/ 0.0, 1.0, 2.0 /)  
REAL, DIMENSION(100) :: y = 0.0
```

```
COMPLEX :: imag_unit = (0.0, 1.0)
```

```
CHARACTER(LEN=80) :: &  
  myname = 'James Bond'
```

```
LOGICAL :: iLikeFortran = .TRUE.
```

KIND-attribute/statement for default types

- the variable representation method (precision) may be declared using the `KIND`-statement
- the `KIND`-attribute is a compiler-dependent unit
- the corresponding values can be inquired by these standard functions

```
SELECTED_INT_KIND(r)
```

```
SELECTED_REAL_KIND(p)
```

```
SELECTED_REAL_KIND(p, r)
```

- integers between $-10^r < n < 10^r$
- real numbers range from $10^{-r} \dots 10^r$ and precision at least p decimals



Double precision

- old Fortran 77 code:

```
DOUBLE PRECISSION X, Y  
X = 1.D0  
Y = 2.D0*ACOS(X)
```

- how we do it these days:

```
INTEGER, PARAMETER :: &  
    dp = SELECTED_REAL_KIND(12,100)  
REAL (KIND=dp) :: x, y  
x = 1.E0_dp  
y = 2.E0_dp*ACOS(x)
```

Numerical precision

● Intrinsic functions related to numerical precision

| | |
|---------------------------------------|---|
| <code>KIND(a)</code> | returns kind parameter of argument |
| <code>SELECTED_REAL_KIND(n, m)</code> | select kind parameter for given precision of real |
| <code>SELECTED_INT_KIND(n)</code> | select kind parameter for given precision of integer |
| <code>TINY(a)</code> | The smallest positive number |
| <code>HUGE(a)</code> | The largest positive number |
| <code>EPSILON(a)</code> | The least positive number that added to 1 returns a number that is greater than 1 |
| <code>PRECISION(a)</code> | The decimal precision |
| <code>DIGITS(a)</code> | The number of significant digits |
| <code>RANGE(a)</code> | The decimal exponent |
| <code>MAXEXPONENT(a)</code> | The largest exponent |
| <code>MINEXPONENT(a)</code> | The smallest exponent |

Numerical precision ...

```
PROGRAM prectest
  IMPLICIT NONE
  INTEGER, PARAMETER :: &
    sp = SELECTED_REAL_KIND(6,30), &
    dp = SELECTED_REAL_KIND(10,200)
  REAL(KIND=sp) :: a
  REAL(KIND=dp) :: b
  PRINT *, HUGE(a), TINY(a), &
    RANGE(a), PRECISION(a)
  PRINT *, HUGE(b), TINY(b), &
    RANGE(b), PRECISION(b)
END PROGRAM prectest
```

Output:

```
3.4028235E+38 1.1754944E-38 37 6
1.797693134862316E+308 2.225073858507201E-308 307 15
```

Logical operators

- Fortran has a special type `LOGICAL`

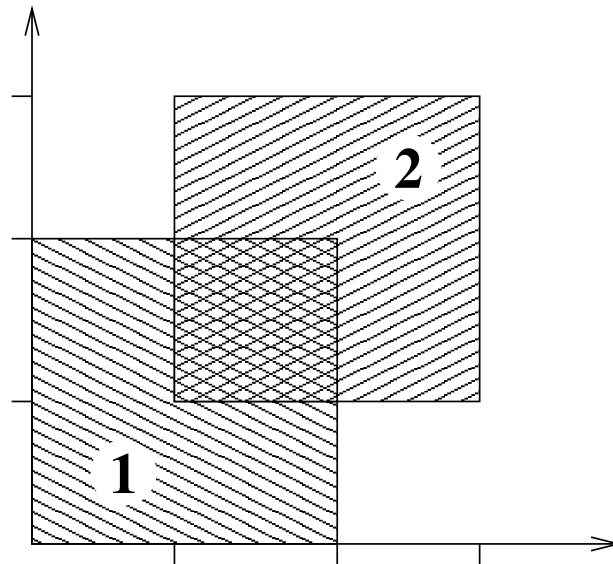
- Relational operators:

`<` or `.LT.`, `>` or `.GT.`, `==` or `.EQ.`, `/=` or `.NE.`, `<=` or `.LE.`, `>=` or `.GE.`

- Logical expressions:

`.AND.`, `.OR.`, `.NOT.`, `.EQV.`, `.NEQV.`

- Exercise: placement of points into 2 overlapping squares:



Logical operators ...

```
PROGRAM placetest
  IMPLICIT NONE
  LOGICAL :: square1, square2
  REAL :: x, y
  WRITE (*,*) 'Enter point &
    &coordinates x and y'
  READ (*,*) x, y

  square1 = (x >= 0 .AND. x <=2 &
    .AND. y >= 0 .AND. y <= 2)
  square2 = (x >= 1 .AND. x <=3 &
    .AND. y >= 1 .AND. y <= 3)

  IF (square1 .AND. square2) THEN
    WRITE (*,*) 'Point within &
      &both squares'
  ELSE IF (square1) THEN
    WRITE (*,*) 'Point in square 1'
  ELSE IF (square2) THEN
    WRITE (*,*) 'Point in square 2'
  ELSE
    WRITE (*,*) 'Point outside'
  END IF
END PROGRAM placetest
```

Assignment statements

- Automatic change of representation (important feature of Fortran!)

```
PROGRAM numbers
  IMPLICIT NONE
  INTEGER :: i
  REAL :: r
  COMPLEX :: c, cc
  i = 7.3
  r = (1.618034, 0.618034)
  c = 2.7182818
  cc = r*(1,1)
  WRITE (*,*) i, r, c, cc
END PROGRAM numbers
```

Output: `7 1.6180340 (2.7182817 , 0.0000000) (1.6180340 , 1.6180340)`

Control structures

- IF THEN ELSE
- SELECT CASE
- DO loops

Referenced IF-construct

- Optional name-reference in order to have check at compilation and to get better readability

```
PROGRAM referenced_if
IMPLICIT NONE
  REAL :: x, y, eps, t

  WRITE (*,*) &
    'Enter values x and y'
  READ (*,*) x, y
  eps = EPSILON(x)

  outside: IF (ABS(x) > eps) THEN
    inside: IF (y > eps) THEN
      t = y/x
    ELSE IF (ABS(y) < eps) THEN inside
      t = 0.0
    ELSE inside
      t = -y/x
    END IF inside
    WRITE (*,*) 'result:', t
  END IF outside

END PROGRAM referenced_if
```

CASE selection

- SELECT CASE statement matches a entries of a list against the case index
- usually arguments are strings, characters or integers
- DEFAULT-branch if no match found

```
...  
INTEGER :: i  
LOGICAL :: isprimenumber  
...  
SELECT CASE(i)  
  CASE (1,3,5,7)  
    isprimenumber = .TRUE.  
  CASE (2,4,6,8:10)  
    isprimenumber = .FALSE.  
  CASE DEFAULT  
    isprimenumber = testprimenumber(i)  
END SELECT  
...
```

DO loops

1. DO-loop with an integer counter

```
DO i=1,numberOfPoints,stepsize
  xcoordinate(i) = i * stepsize * 0.05
END DO
```

2. DO WHILE-construct

```
READ (*,*) x
DO WHILE (x > 0)
  totalsum = totalsum + x; READ (*,*) x
END DO
```

3. DO without loop-control

```
READ (*,*) x
DO
  if (x<0 ) EXIT; totalsum = totalsum + x; READ (*,*) x
END DO
```

Inside-loop controls: EXIT and CYCLE

- EXIT terminates the loop
- CYCLE skips to the next loop-instance
- example: conditional branches in loop

```
DO
  READ (*,*) x
  IF (x < 0.0) THEN
    EXIT
  ELSE IF (x > upperlimit) THEN
    CYCLE
  END IF
  totalsum = totalsum + 1.0/SQRT(x)
END DO
```

Euclidean algorithm

```
PROGRAM GCD
  IMPLICIT NONE
  INTEGER, PARAMETER :: &
    long = SELECTED_INT_KIND(9)
  INTEGER (KIND=long) :: m, n, t
  WRITE (*,*) 'Enter positive integers m and n:'
  READ (*,*) m, n
  WRITE (*,*) 'm:', m, ' n:', n
  positivecheck: IF (m > 0 .AND. n > 0) THEN
    DO WHILE (n /= 0)
      t = MOD(m,n)
      m = n
      n = t
    END DO
    WRITE (*,*) 'greatest common divisor:', m
  ELSE
    WRITE (*,*) 'Negative value entered'
  END IF positivecheck
END PROGRAM GCD
```

Mind: $G.C.D.(m, n) = G.C.D.(n, m \bmod n)$, if $m \bmod n \neq 0$



Procedures

Contents

- Procedures
- Internal procedures
- Recursive procedures
- External procedures
- Arguments of procedures

Procedures

Advances of **structured programming** based on modules, functions and subroutines

- testing and debugging separately from the rest of the program
- recycling of code in other programs (modules)
- improved readability of code
- re-occurring tasks can be easily handled with procedures

In Fortran 90/95 built-in procedures as well as declared procedures can be used

Subroutines vs. functions

- procedures = subroutines and functions
- data between **subroutines** is exchanged by means of arguments only
- a **function** returns values according to its declared type
- in a good programming style, a function should not change values of its arguments
- Fortran by default uses **call by reference** (modifies value of arguments)
- variables declared inside procedures are (usually) **local**

Declaration of procedures

- **subroutines:**

```
SUBROUTINE name(variables_parameters)
[declarations]
[statements]
END SUBROUTINE name
```

- **functions:**

```
[type] FUNCTION name(variables_parameters)
[declarations]
[statements]
END FUNCTION name
```

- **subroutine call:**

```
CALL name(argument)
```

- **function call:**

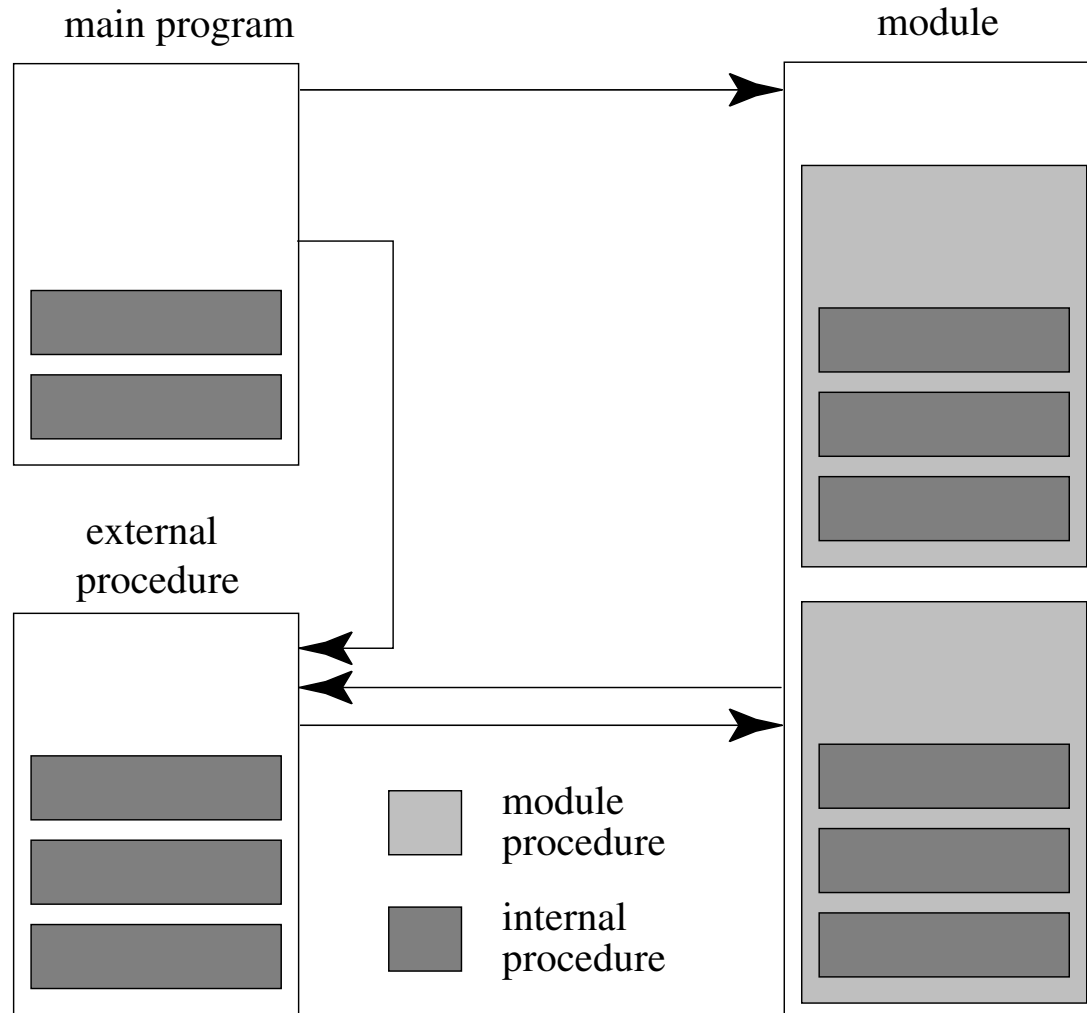
```
result = name(argument)
WRITE (*,*) 'Result:', name(argument)
```

User defined procedures

- three ways to define procedures:
 1. **internal procedures**: declared within the program structure
 2. **external procedures**: independently declared, not necessarily written in Fortran
 3. **modules**: taken into use via USE-command
- internal procedures and procedures in modules provide a **defined interface**, such that compatibility of arguments can be checked at compilation

In the following we mainly focus on internal procedures; modules will be explained in a following lecture

Program units



Internal procedures

- each program unit may contain internal procedures
- internal procedures are declared at the end of the program unit, after the `CONTAINS`-keyword
- variables and objects declared within the program unit can be used in internal procedures (provided they are not re-declared)
- BUT: a good programming style uses arguments to pass data
- no nested declaration of internal procedures

Internal function declaration

```
PROGRAM internal_function_test
  IMPLICIT NONE
  INTEGER, PARAMETER :: &
    dp=SELECTED_REAL_KIND(12,50)
  REAL(KIND=dp) :: x=1.0_dp, y=2.0_dp, z
  z = add(x,y)
  WRITE(*,*) 'Result:', z, add(x,y)
CONTAINS
  REAL(KIND=dp) FUNCTION add(a,b)
    REAL(KIND=dp) :: a, b
    add = a + b
  END FUNCTION add
END PROGRAM internal_function_test
```

or alternatively

```
FUNCTION add(a,b) RESULT(theresult)
  REAL(KIND=dp) :: a, b, theresult
  theresult = a + b
END FUNCTION add
```



Internal subroutine declaration

```
PROGRAM internal_subroutine_test
  IMPLICIT NONE
  REAL :: x = 1.0, y = -1.0
  WRITE (*,*) 'in x,y: ', x, y
  IF (x > y) THEN
    CALL swap(x,y)
  END IF
  WRITE (*,*) 'out x,y: ', x, y
CONTAINS
SUBROUTINE swap(a,b)
  IMPLICIT NONE
  REAL :: a, b
  REAL :: aux
  aux = a
  a = b
  b = aux
END SUBROUTINE swap
END PROGRAM internal_subroutine_test
```

Arguments to procedures

- **call by reference**: changes of an argument's value inside a procedure changes the argument itself
- the compiler can check compatibility of the arguments if the interface of a procedure is known at compilation time (internal procedure + module)
- the utilization of arguments can be passed to the compiler using the `INTENT`-keyword. Use for better performance!

INTENT attribute

- the `INTENT` attribute declares whether a formal argument is
 - into a procedure (`IN`)
 - intended for transferring a value out of it (`OUT`)
 - in both directions (`INOUT`)
- this is valuable information for the compiler (code optimization!)
- example:

```
SUBROUTINE sumup(totalsum,part)
  IMPLICIT NONE
  REAL, INTENT(INOUT) :: totalsum
  REAL, INTENT(IN) :: part
  totalsum = totalsum + part
END SUBROUTINE sumup
```

Keyword arguments

- provided the procedure's interface is known, arguments can be passed by keywords (the name of the variable)
- example:

```
REAL FUNCTION f(lower,upper,tol)
  IMPLICIT NONE
  REAL, INTENT(IN) :: lower, upper, tol
  ...
END FUNCTION f
```

methods of calling the function:

```
v = f(0.0, 1.0, 0.1)
x = f(lower=1.0, upper=10.0, tol=0.01)
y = f(upper=1.0, lower=0.0, tol=0.001)
z = f(0.0, tol=0.001, upper=1.0)
```

Optional arguments

- declaration of optional arguments using the `OPTIONAL`-attribute (function's interface must be known)
- example:

```
REAL FUNCTION f(lower,upper,tol)
  IMPLICIT NONE
  REAL, INTENT(IN) :: lower, upper
  REAL, INTENT(IN), OPTIONAL :: tol
  REAL :: local_tol
  ...
  IF (PRESENT(tol)) THEN
    local_tol = tol
  ELSE
    local_tol = EPSILON(local_tol)
  END IF
END FUNCTION f
```

- presence of optional arguments may be checked with `PRESENT`

Optional arguments ...

methods of calling the function:

```
x = f(0.0, 1.0, 0.01)
```

```
y = f(0.0, 1.0)
```

```
z = f(upper = 1.0, lower = 0.001)
```

Saving local variables

- by default objects in a subroutine or function are dynamically allocated whenever the procedure is invoked
- only saved objects retain their value and definition, association, and allocation status
- they are stored in static memory by the `SAVE` statement and attribute

```
REAL, SAVE :: a
```

```
REAL :: b, c, d  
SAVE b, d
```

```
SAVE
```

- if the save-list (statement without argument) is omitted, all variables are saved

Recursive procedures

- Recursion = calling a procedure from within itself
- F77 does not support recursion, F90/95 does
- recursive function example:

```
RECURSIVE FUNCTION fibonacci(n) RESULT(fibo)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n
  INTEGER :: fibo
  IF (n <= 2) THEN
    fibo = 1
  ELSE
    fibo = fibonacci(n-1) + fibonacci(n-2)
  END IF
END FUNCTION fibonacci
```

Recursive procedures

● recursive subroutine example:

```
RECURSIVE SUBROUTINE play_it_again(song, instance, enough)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: song, instance, enough
  INTEGER :: local_song, local_instance
  ...
  IF (instance < enough) THEN
    CALL play_it_again(song, instance + 1, enough)
  END IF
  ...
END IF
END SUBROUTINE play_it_again
```



External procedures - 1

- external procedures are declared in separate program units
- they are kind of a relict from F77-times
- nowadays modules should be preferably used!
- BUT: library routines as well as routines written in other programming languages are external procedures
- linked to the program (statically or as a run-time object)

External procedures - 2

- the interface of an external procedure can be declared using an `INTERFACE`-block (checked at compilation)

```
INTERFACE
  REAL FUNCTION my_external_function(x,y,z)
  REAL, INTENT(IN) :: x,y,z
  END FUNCTION my_external_function
  SUBROUTINE my_external_subroutine(x,z)
  REAL, INTENT(IN) :: x
  REAL, INTENT(OUT) :: z
  END SUBROUTINE my_external_subroutine
END INTERFACE
```

External procedures - 3

- also operators:

```
INTERFACE OPERATOR ( + )  
    FUNCTION b_plus(a, b)  
        LOGICAL, INTENT(IN) :: b_plus, a, b  
    END FUNCTION b_plus  
END INTERFACE
```

- example (see earlier):

```
PROGRAM external_subroutine_test  
    IMPLICIT NONE  
    INTERFACE exswap  
        SUBROUTINE swap(a,b)  
            REAL :: a, b  
        END SUBROUTINE swap  
    END INTERFACE  
    REAL :: x = 1.0, y = -1.0  
    WRITE (*,*) 'in x,y: ', x, y  
    CALL exswap(x,y)  
    WRITE (*,*) 'out x,y: ', x, y  
END PROGRAM external_subroutine_test
```

Arguments to procedures - 1

- a procedure's argument-list may contain another procedure, except from standard-subroutines or internally declared procedures
- that leaves us with externally declared procedures, standard-functions (such as SIN) or module procedures
- standard-functions being passed as argument have to be declared using the `INTRINSIC`-statement (in order that the compiler can distinguish it from a variable name)

Arguments to procedures - 2

- Example: Gaussian (two-point formula) integration of a scalar-function $f(x)$ over the interval $[a, b]$:

$$\int_a^b f(x) dx \approx \frac{b-a}{2} \left[f\left(\frac{a+b}{2} - \frac{b-a}{2\sqrt{3}}\right) + f\left(\frac{a+b}{2} + \frac{b-a}{2\sqrt{3}}\right) \right]$$

- integrate $f(x) = \sin(x)$ over interval $[1, 2]$

```
PROGRAM integrate
  IMPLICIT NONE
  REAL :: a=1.0, b=2.0
  INTRINSIC SIN
  WRITE (*,*) 'Numerical result:', gauss(SIN, a, b)
  WRITE (*,*) 'Analytical result:', COS(a) - COS(b)
```

Arguments to procedures - 3

CONTAINS

```
REAL FUNCTION gauss(f,a,b)
  IMPLICIT NONE
  INTERFACE
    REAL FUNCTION f(x)
      REAL :: x
    END FUNCTION f
  END INTERFACE
  REAL, INTENT(IN) :: a, b
  REAL :: p1, p2
  p1 = 0.5*(a+b) - (b-a)/(2.0*SQRT(3.0))
  p2 = 0.5*(a+b) + (b-a)/(2.0*SQRT(3.0))
  gauss = 0.5*(b-a)*(f(p1) + f(p2))
END FUNCTION gauss
END PROGRAM integrate
```



Output:

| | |
|--------------------|------------|
| Numerical result: | 0.95622051 |
| Analytical result: | 0.95644909 |

File input and output

Contents

- Basics
- Opening and closing files
- Writing files
- File format descriptors
- Reading files
- Formatted vs. unformatted files
- Direct access
- File properties
- NAMELIST-files
- Internal I/O

File I/O: basics

- Writing to and reading from a file is basically similar to writing onto a screen or reading a keyboard
- Differences
 - File must be opened first:
`open(unit=iu, file='foo')`,
where `n` is the unit or file number (integer) and `foo` file name.
 - Subsequent writes and reads need to refer to right unit number:
`write(n, ...)`
 - File must finally be closed: `close(unit=iu)`

Opening and closing a file - 1

Format:

- `open([unit=]iu, file='name' [,options])`
- `close([unit=]iu [, options])`

For example

- `open(10,file='data.dat',status='new')`
- `close(10,status='delete')`



Opening and closing a file - 2

Notes

- First parameter is the unit number. Keyword “unit=” can be left out.
- Units 5, 6, and 0 are predefined:
 - 5 is standard input, 6 is standard output, and on several Unix systems, 0 is standard error.
 - These are always open, and assumed default if unit number is not given: `read(*, ...)`, `write(*, ...)`
 - On some systems these predefined units are automatically opened and you may not be able to re-open them yourself by using `open-statement`

Opening a file

- `file='name'` specifies the file to be opened. Keyword `file` is obligatory, and the name must be quoted. A string variable can be used:

```
iu = 10
fdata = 'data.dat'
open(iu,file=fdata) ...
```

- Only unit number and file name are necessary
 - `open([unit=]iu, file='name' [,options])`
- but other options are possible:
 - `status, position, form, access, iostat, err, recl, action`

Opening file: options - 1

- `status`: (existence of the file) 'old', 'new', 'replace', 'scratch', 'unknown'
- `position`: (where to write) 'append'
- `action`: (mode) 'write', 'read', 'readwrite'
- `form`: (text or binary file) 'formatted', 'unformatted'
- `access`: (direct or sequential access) 'direct', 'sequential'
- `iostat`: (error value, output) integer, 0 = OK, others = error
- `err`: (error exit) line number where to go on error
- `recl`: (record length, input) integer

Opening file: options - 2

Examples:



```
iu = 10
open(iu,file='data.dat', status='old', iostat=iovar)
if (iovar /= 0) then
  write(*,*) 'Cannot open file, error code: ',iovar
else
  read(iu,...) ...
endif
```



```
iu = 11
open(iu,file='data.bin', status='new', form='unformatted')
```

Opening file: options - 3

Examples:

```
● iu = 12
  open(iu,file='data.db', status='new',
        iostat=iovar, access='direct', err=999)
  write(iu,rec=1) ...
  goto 1000
999 continue
  if (iovar /= 0) then
    write(*,*) 'Cannot open db-file, error code: ',iovar
  endif
1000 continue
```

Writing to a (text) file

- Writing into a file is done in the same way as any write, but the corresponding unit number is now given as a parameter:
 - `write(unit=iu,fmt=*) str`
 - `write(iu,*) str`
- Formats and options can be used as usually. If keyword 'unit' is used, also 'fmt' must be used.
- If one writes to unit number `iu`, with no file associated to it, usually a file "fort.<iu>" is created as a result.
- The unit number `iu` can only have positive values usually up to a few hundreds, the maximum value being an implementation dependent constant that may or may not be possible to increase during the runtime.

Format descriptors

- Prettify your output by using format descriptors rather than the free format descriptor (*), either by

```
write(iu,fmt='(1x,a," = ",i5,2x,g15.7)') str,int(12.3),457e-5
```

- or by

```
1234 format(1x,a," = ",i5,2x,g15.7)
      write(iu,1234) str,int(12.3),457e-5
```

- Format descriptors can also be used when reading in (text) files

Reading from a (text) file - 1

- Reading is done as usual
 - `read(unit=iu,ftm=*) str`
 - `read(iu,*) str`
- Option `iostat` can be used to check errors (>0) and end of file conditions (<0):

```
iu = 10
read(iu,*, iostat=rerr) str
if (rerr < 0 ) then
  write(*,*) 'End of file'
else if (rerr > 0) then
  write(*,*) 'Read error'
else
  write(*,*) 'Read OK: ', str
endif
```

Reading from a (text) file - 2

- Alternatively `err` and `end` options can be used
- They specify which label number to jump in case of error or end of file condition
- As result less readable code than in the previous page :

```
iu = 10
read(iu,*, end=998, err=999) str
goto 1000 ! All ok
998 continue ! EOF
write(*,*) 'End of file'
goto 1100
999 continue ! Error
write(*,*) 'Read error'
goto 1100
1000 continue
write(*,*) 'Read OK: ', str
1100 continue
```

Formatted vs. unformatted files

- By default all the files are *text files*, or *formatted files*
 - Human readable
 - Portable i.e. machine independent
- Fortran can also use *binary files*, or *unformatted files*
 - Machine readable only
 - *Much* faster to use than *formatted files*
 - Suitable for large amount of data (reduces file size)
 - Internal format used for numbers: no number conversion, no rounding errors
 - Not necessarily portable (need to have the same internal data representation between machines that are writing and reading such files)

Formatted vs. unformatted files

- Example of unformatted write

```
open(10, file='foo.dat', form='unformatted')
write(10) rval
write(10) string
close(10)
```

- Notice:

no format (FMT-keyword) in write: `write(unit=iu) ...`

- Reading is done similarly:

```
read(10) rval
read(10) string
```

- These Fortran binary files can not directly be read via (f.ex) C-program, since each `write` prepends and appends an implicit record length to the record being written

Direct access files

- By default files are *sequential*: lines are written/read one after another, but *direct access* files can be written/read in random order.
 - They can be practical e.g. in database access.
 - They are binary (unformatted), but can also be formatted.
 - Length of a record is given using `recl` in `OPEN`. It is the no. of characters for formatted, the no. of bytes for unformatted files (on some machines the no. of words).
 - The record number is given using `rec`-keyword in read/write.
- For example

```
integer, dimension(10) :: r
iu = 20
open(iu, file='data.db', access='direct', recl=40)
write(iu,rec=13) r
```

Getting file properties

- Properties of a file or a unit can be requested by `inquire`
 - `inquire(file='xxx', list...)`
 - `inquire(unit=x, list...)`
- `list` is a list of options or inquiries:
 - `exists` (file exists; logical)
 - `opened` (file or unit is opened; logical)
 - `form` ('formatted'/'unformatted')
 - `access` ('sequential'/'direct')
 - `action` ('read', 'write', 'readwrite')
 - ... etc.

Getting file properties

● Example:

```
logical :: isfile
inquire (file='foo.dat', exists=isfile)
if (.not. isfile) then
  write(*,*) 'File does not exist'
else
  ...
endif
```

NAMELIST

- NAMELIST is a user friendly mechanism to input data values using *variable name – value* -pairs in a NAMELIST-group:

```
program namehunt
  implicit none
  integer :: deer=2, rabbit=4, moose=1, duck=0
  namelist /game/ deer,rabbit,moose,duck
  write(*,*) 'Captured game:'
  read(*,NML=game)
  write(*,NML=game)
end program namehunt
```

- Input begins with & and group name, ends with /:

```
&game rabbit=1, deer=3, horse=1 /
```

- Output:

```
&GAME
  DEER=   3, RABBIT=   1, MOOSE=   1, DUCK=   0, /
```

Internal I/O

- Sometimes it is handy to read/write data from/to Fortran character strings to filter out or to format data
- This can be accomplished by replacing the `unit`-number in `read/write` statement with a character string
- No actual (external) files are used
- The simplest forms correspond to use of `TRANSFER`-function
- An example:

```
character(len=10), parameter :: input = '12345.7890'  
integer :: n7890  
character(len=22) :: output  
read(input, fmt='(6x,i4)') n7890  
print *, 'n7890=', n7890  
write(output, fmt='(a14,i8)') 'Hello world! ', n7890
```

Arrays

Contents

- Declaration statements
- About array terminology
- Array syntax and element ordering
- Array sections
- Initialization
- Array intrinsic functions

Declaration statements

- Explicit shape arrays' declaration

```
REAL, DIMENSION(0:100,101,10:110) :: x  
REAL :: y(101,101,101)
```

- An array can have up to seven dimensions with the fastest growing dimension being the left-most one

- Tip: use PARAMETER constants to define a shape

```
INTEGER, PARAMETER :: m=100, n=50  
REAL, DIMENSION(n,m) :: coeffmat  
REAL, DIMENSION(m) :: vector
```

- Also possible

```
REAL, DIMENSION(200) :: x  
INTEGER, DIMENSION(size(x)) :: ix
```

About array terminology

- Arrays can be of any intrinsic type or derived type
- The *rank* of an array is the number of dimensions
- *Array extent* is the number of elements in any dimension
- The *shape* of an array is determined by its rank and its extent in each dimension
- An *array element* is one of the individual elements in an array
- An *array section* is a subset of elements of an array

Array syntax

- Array syntax can reference to an element:

- Element-by-element processing

```
INTEGER, PARAMETER :: n=100
REAL, DIMENSION(n,n) :: array
INTEGER :: i, j
```

```
DO j = 1, n
  DO i = 1, n
    array(i, j) = 1.0
  END DO
END DO
```

Array syntax (cont'd)

- Array syntax can reference to whole arrays (and array sections):

- Whole array assignment

```
INTEGER, PARAMETER :: n=100  
REAL, DIMENSION(n,n) :: array  
array = 1.0
```

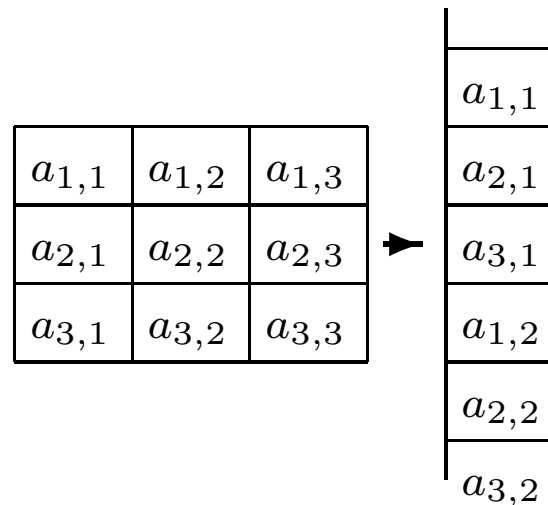
- ... or to signify use of array sections

```
array(:, :) = 1.0
```

- Please note that intrinsic functions are often elemental, for example: numeric, mathematical, logical and bit functions apply to a single element, not a section of an array

Array element ordering

- The standard does not specify how arrays should be located in memory
- Conceptual ordering: column major form
- 2D array example:



Array sections

- Element-by-element loop:

```
DO j = 3, 5
  DO i = 2, 8, 2
    array(i, j) = 1.0
  END DO
END DO
```

- Using array syntax (the same result as in the above loop)

```
array(2:8:2, 3:5) = 1.0
```

Array sections (cont'd)

- Right hand side is computed first:

```
x(1:10) = x(10:1:-1)
```

- This does not give the same result as

```
DO i = 1, 10  
    x(i) = x(11-i)  
END DO
```

- Correct explicit coding (assume: x-array's type is REAL)

```
REAL :: tmp(size(x))  
DO i = 1, 10 ; tmp(i) = x(11-i) ; END DO  
DO i = 1, 10 ; x(i) = tmp(i) ; END DO
```

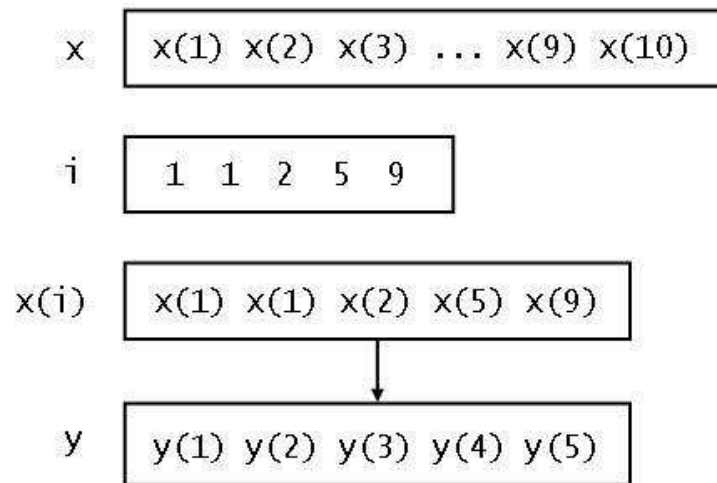
Array-valued expressions in assignments

```
INTEGER, DIMENSION(5) :: ind  
REAL, DIMENSION(10) :: x, y
```

```
ind = (/1,1,2,5,9/)
```

```
y(1:5) = x(ind)
```

```
y(ind) = x(1:5) ! Error : y(1) is not clear
```



Initialization of arrays

- Array construction containing specific values
 - Give the values between the (/ and /) characters.

```
INTEGER, DIMENSION(3), PARAMETER &  
  :: numbers = (/1, 3, 5/)
```

Initialization of arrays (cont'd)

- Implied-DO initialization in a form of
`(array(i), i=lower, upper)`

```
REAL, DIMENSION(8) :: a      ! i.e. a(1:8)
REAL, DIMENSION(n) :: x
INTEGER, DIMENSION(8) :: ii
```

```
a = (/ 0.0, -1.1, (i+0.5, i=3,6), -5.0, 0.0 /)
```

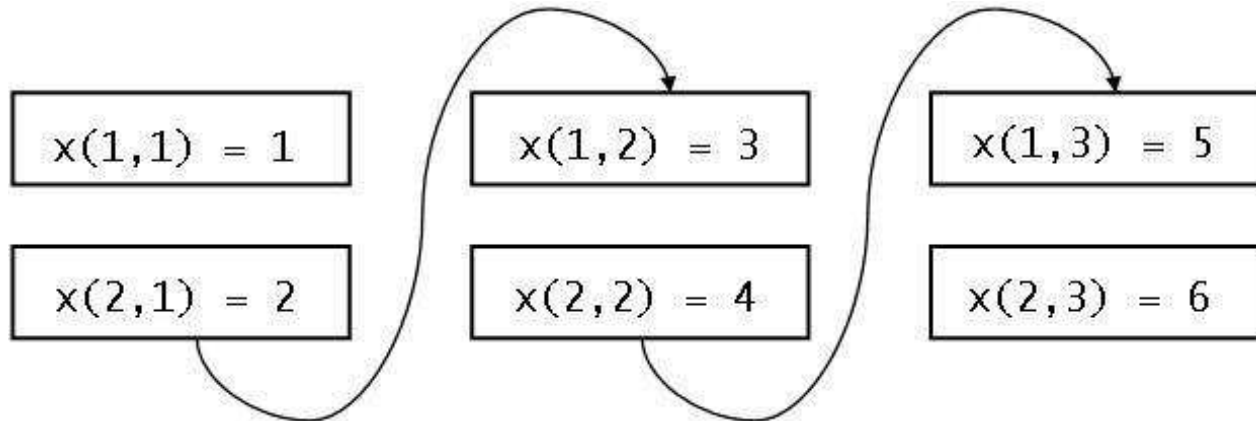
```
x(1:n) = (/ (SIN(pi*i/(n-1.0)), i = 0, n-1) /)
```

```
ii = (/ 0, (1, i=1,4), (2, i=1,3) /)
```

RESHAPE intrinsic function

- An array can be reshaped into any allowable array shape using the RESHAPE intrinsic function

```
x(1:2, 1:3) = RESHAPE( (/ (i, i=1,6) /), (/2, 3/) )
```



```
INTEGER, PARAMETER :: n=100  
REAL, DIMENSION(n,n) :: array  
array = RESHAPE( &  
  & (/ ( (1/(1.0+i+2*j), i=1, n), j=1, n) /), (/ n, n/) )
```

Masked array assignment : WHERE

- Choose a part of an array to be processed

```
WHERE ( x < 0 )  
    x = 0  
END WHERE
```

- Use of ELSEWHERE-token in the same context

```
WHERE ( z < 0 )  
    x = -z  
ELSEWHERE  
    x = z  
END WHERE
```

Masked array assignment : FORALL

- Combination of subscript and mask expressions:

```
INTEGER, PARAMETER :: n = 100
INTEGER :: i, j
REAL, DIMENSION(n, n) :: a
...
FORALL (i=1:n:2, j=1:n:2, a(i,j)>0)
  a(i,j) = i + j
END FORALL

FORALL (i = 1:n) a(i,i) = i**2
```

Array intrinsic functions

- `TRANSPOSE (a)` : Transpose an array of rank of two
- `MATMUL (a , b)` : Matrix multiply
- `DOT_PRODUCT (v1 , v2)` : Dot product of two vectors
- `SUM (array , dim , mask)` : Sum of the elements
- `PRODUCT (array , dim , mask)` : Product of the elements
- `MAXVAL (array , dim , mask)` ,
`MINVAL (array , dim , mask)` :
Maximum, minimum value of the elements
- `MAXLOC (array , mask)` ,
`MINLOC (array , mask)` :
The first position in the array that has the maximum, minimum value

Array intrinsic functions (cont'd)

- `SIZE(array, dim)`: Number of elements, along the specified dimension
- `COUNT(larray, dim)`: Count of the number of elements, which are `.TRUE.` in `larray`, along the specified dimension
- `ANY(larray, dim)`: `.TRUE.`, if any value in `larray` is `.TRUE.`
- `ALL(larray, dim)`: `.TRUE.`, if all values in `larray` are `.TRUE.`

Array intrinsic functions (cont'd)

- `LBOUND(array, dim)` : Returns the lower bound of an array
- `LUPPER(array, dim)` : Returns the upper bound of an array
- `SHAPE(array)` : Returns the shape of an array i.e. `SIZE` in each of its dimension
- `RESHAPE(source, shape)` : Reconstructs an array with the specified shape using the elements provided by the source

Example: Array intrinsic functions

```
INTEGER, DIMENSION(10, 10) :: A
A = RESHAPE( (/ ( (i+2*j, i=1,10), j=1,10) /), &
            & (/10, 10/) )
WRITE (*,*) SUM(A)
WRITE (*,*) SUM(A, DIM=1)
WRITE (*,*) SUM(A, DIM=2)
WRITE (*,*) SUM(A, MASK = A > 20)
WRITE (*,*) COUNT(A < 20)
WHERE( x /= 0.0 )
    y = y / x
ELSEWHERE
    y = 0.0
END WHERE
```



More array intrinsic functions

- `MERGE (truear , falsear , mask)`: merges two arrays based on a logical mask
- `SPREAD (array , dim , ncopies)`: Replicates an array by adding a new dimension.
- `PACK (array , mask , vector)`: Under the control of a mask, takes elements from an array and packs them into a one-dimensional array
- `UNPACK (vector , mask , array)`: Takes elements from a one-dimensional array and rearranges them into another array.
- `CSHIFT (array , shift , dim)`: Circular shift, shifts the elements along a given dimension of an array.
- `EOSHIFT (array , shift , edge , dim)`: End-off shift, elements shifted off one end are lost.



Dynamic mem alloc, arrays as arguments

Contents

- Dynamic memory allocation
- Pointer assignment
- Passing arrays as arguments



Dynamic memory allocation

- Deferred Shape Arrays:

- Case 1: ALLOCATABLE attribute, array extents are determined when space is allocated.

```
INTEGER :: n, allocstat
REAL, DIMENSION(:), ALLOCATABLE :: A

READ (*,*) n
ALLOCATE (A(n), STAT=allocstat)
IF ( allocstat /= 0 ) STOP 'Memory allocation failed!'
...
DEALLOCATE(A)
```

Dynamic memory allocation (cont'd)

- Deferred Shape Arrays:

- Case 2: POINTER attribute, array extents are determined when space is allocated.

```
INTEGER :: n, allocstat
REAL, DIMENSION(:), POINTER :: A => NULL()
! NULL() define a pointer to be initially disassociated

READ (*,*) n
ALLOCATE (A(n), STAT=allocstat)
IF ( allocstat /= 0 ) STOP 'Memory allocation failed!'
...
DEALLOCATE(A)
```

Dynamic memory allocation (cont'd)

- Automatic arrays : The size is propagated through the argument list

```
SUBROUTINE FOO(n)
  INTEGER, INTENT(IN) :: n
  REAL, DIMENSION(2*n+1,12) :: A
  DO i=1,2*n+1 ; A(i,:) = 0 ; ENDDO
  CALL DO_SOMETHING(A, size(A,dim=1), size(A,dim=2))
END SUBROUTINE FOO
```

- No explicit `ALLOCATE` or `DEALLOCATE` needed
- But also no way to check the success of the allocation before aborting
- Usually allocated from memory `STACK`, which normally has a fairly limited size compared to memory `HEAP`

POINTER assignment

- The TARGET must have the same type and rank as the POINTER

```
PROGRAM ptr
  IMPLICIT NONE
  INTEGER :: i
  REAL, DIMENSION(10), TARGET :: a
  REAL, POINTER :: b => NULL()
  REAL, POINTER, DIMENSION(:) :: c => NULL()
  WRITE(*,*) ASSOCIATED(b),ASSOCIATED(c) !Status of pointers
  a = (/ (i, i = 1, 10) /)
  b => a(3)      ! Rank = 0 i.e. scalar value
  WRITE(*,*) b
  c => a(3:8:2) ! Rank = 1, but 'covers' only every 2nd element
  WRITE(*,*) c
  WRITE(*,*) ASSOCIATED(b),ASSOCIATED(c)
END PROGRAM ptr

F F
  3.000000
  3.000000      5.000000      7.000000
T T
```

Pointer testing example

```
PROGRAM ptr_testing
! Outputs are on comments
  IMPLICIT NONE
  REAL, DIMENSION(:), POINTER :: a => NULL()
  REAL, DIMENSION(:), TARGET, ALLOCATABLE :: b
  INTEGER n1,n2,i,allocstat
  n1=-2; n2=2
  ALLOCATE (a(n1:n2), b(n1:n2), STAT=allocstat)
  IF ( allocstat /= 0 ) STOP 'Memory allocation failed!'
  FORALL (i=n1:n2) a(i)=REAL(i)
  FORALL (i=n1:n2) b(i)=2*REAL(i)+10.
  write(*,*)a; write(*,*)b
!-2.000000      -1.000000      0.000000      1.000000      2.000000
! 6.000000      8.000000      10.00000      12.00000      14.00000
  deallocate(a)
  write(*,*)a; write(*,*)b
!
! 6.000000      8.000000      10.00000      12.00000      14.00000
! Listing continues on next page
```

Pointer testing example(cont'd)

```
a=>b(n1:n2:2)
write(*,*)a; write(*,*)b
! 6.000000      10.000000      14.000000
! 6.000000      8.000000       10.000000      12.000000      14.000000
a=>b
write(*,*)a; write(*,*)b
! 6.000000      8.000000      10.000000      12.000000      14.000000
! 6.000000      8.000000      10.000000      12.000000      14.000000
b(n1:0)=0.
write(*,*)a; write(*,*)b
! 0.000000      0.000000      0.000000      12.000000      14.000000
! 0.000000      0.000000      0.000000      12.000000      14.000000
a(n1:0)=-10
write(*,*)a; write(*,*)b
!-10.000000     -10.000000     -10.000000      12.000000      14.000000
!-10.000000     -10.000000     -10.000000      12.000000      14.000000
! What happens if next 3 lines are uncommented?
!ALLOCATE (a(n1:n2), STAT=allocstat)
!IF ( allocstat /= 0 ) STOP 'Memory allocation failed!'
!write(*,*)a; write(*,*)b
END PROGRAM ptr_testing
```



Passing arrays as arguments

Three possible ways:

1. Assumed shape array:

```
REAL, DIMENSION(low1:,low2:) :: matrix
```

or a simpler (but NOT equivalent) form

```
REAL, DIMENSION(:, :) :: matrix
```

- no `ALLOCATABLE` attribute
- Type, Kind and Rank must be the same as in actual argument array
- The extent of a dimension is the extent of the corresponding dimension of the actual argument array
- Procedure must have explicit interface (internal or module procedure, or is defined in an `INTERFACE` block).

Passing arrays as arguments

2. Explicit shape array

```
REAL, DIMENSION(low1:up1,low2:up2) :: matrix
```

or a simpler (but NOT equivalent) form

```
REAL, DIMENSION(up1,up2) :: matrix
```

- The rank do not need to match with the rank of actual argument array, array element ordering is defining the passed elements.
- Array bounds must be passed as arguments or via module.

3. Assumed size array (Fortran 77 feature)

```
REAL, DIMENSION(low1:up1,*) :: matrix
```

- Its declaration specifies the rank and the extents for each dimension except the last.

Example: Arrays and a subroutine

```
PROGRAM array
  INTEGER,PARAMETER :: n = 5
  REAL, DIMENSION(n,n) :: x, y, z
  x = 1
  y = RESHAPE((/(i+j, i=1,n), j=1,n) /), (/n, n/))
  CALL f(x,y,z)
CONTAINS
  SUBROUTINE f(x,y,z)
    REAL, DIMENSION(:,:) :: x, y, z
    IF ( ANY(SHAPE(x) /= SHAPE(y)) .OR. &
        ANY(SHAPE(x) /= SHAPE(z)) ) THEN
      WRITE(*,*) 'array shape error'; STOP
    ELSE
      z = SIN(SQRT(x**2 + y**2))
    END IF
  END SUBROUTINE f
END PROGRAM array
```



Array-valued functions

- Functions can also return arrays:

```
FUNCTION add(v1,v2) RESULT(w)
  IMPLICIT NONE
  REAL, DIMENSION(:), INTENT(IN) :: v1, v2
  REAL, DIMENSION(SIZE(v1)) :: w
  w = v1 + v2
END FUNCTION add
```

- Argument arrays, $v1$, $v2$, can be assumed shape arrays.
- The shape of a return array, w , is computed from the arguments. If the return shape is not known, functions may return array pointers as their result. (If a function is internal or module function, or it is defined in an INTERFACE block its shape can be computed)

Example: Arrays and a function

```
PROGRAM array
  INTEGER,PARAMETER :: n = 5
  REAL, DIMENSION(n,n) :: x, y, z
  x = 1
  y = RESHAPE((/(i+j, i=1,n), j=1,n) /), (/n, n/))
  z = f(x,y)
CONTAINS
  FUNCTION f(x,y) RESULT(z)
    REAL, DIMENSION(:,:) :: x, y
    REAL, DIMENSION(SIZE(x,1), SIZE(x,2)) :: z
    IF ( ANY(SHAPE(x) /= SHAPE(y)) ) THEN
      WRITE(*,*) 'array shape error'; STOP
    ELSE
      z = SIN(SQRT(x**2 + y**2))
    END IF
  END FUNCTION f
END PROGRAM array
```



ALLOCATABLE and POINTER in procedures

- When both the actual and dummy arguments are pointer arrays and the actual argument has been dynamically allocated, the dummy argument pointer receives the pointer association status of the actual argument (see example on the next page)
- If an array that has ALLOCATABLE attribute is passed to a procedure and the array has been dynamically allocated then the dummy argument declaration is defining the subscripting
- A dummy argument can not have ALLOCATABLE attribute
- Procedure can return arrays that have POINTER attribute. If the dummy argument is a pointer, the actual argument shall be a pointer

Example: passing pointer array

```
PROGRAM pass_pointer_array_test
! Outputs are on comments
  IMPLICIT NONE
  REAL, DIMENSION(:), POINTER :: a => NULL()
  INTEGER n1,n2,i,lo_bound,allocstat
  n1=-2;n2=2
  ALLOCATE (a(n1:n2),STAT=allocstat)
  IF ( allocstat /= 0 ) STOP 'Memory allocation failed!'
  FORALL (i=n1:n2) a(i)=REAL(i)
  WRITE(*,*) a;
!-2.000000      -1.000000      0.000000      1.000000      2.000000
  lo_bound=lb_1(a)
  lo_bound=lb_2(a)
  lo_bound=lb_3(a,10)

CONTAINS
! see next page
```



Example: passing pointer array(cont'd)

```
INTEGER FUNCTION lb_1(x) RESULT(lb)
  IMPLICIT NONE
  REAL, DIMENSION(:) :: x
  WRITE(*,*) x(1:2)
! -2.000000      -1.000000
  WRITE(*,*) LBOUND(x,1),UBOUND(x,1)
! 1           5
  lb_1 = LBOUND(x,1)
END FUNCTION lb_1

INTEGER FUNCTION lb_2(x) RESULT(lb)
  IMPLICIT NONE
  REAL, DIMENSION(:), POINTER :: x
  WRITE(*,*) x(1:2)
! 1.000000      2.000000
  WRITE(*,*) LBOUND(x,1),UBOUND(x,1)
! -2           2
  lb = LBOUND(x,1)
END FUNCTION lb_2
```



Example: passing pointer array(cont'd)

```
INTEGER FUNCTION lb_3(x,new_lower_bound) RESULT(lb)
  IMPLICIT NONE
  REAL, DIMENSION(new_lower_bound:) :: x
  INTEGER new_lower_bound
  WRITE(*,*) x(new_lower_bound:new_lower_bound+1)
!-2.000000      -1.000000
  WRITE(*,*) LBOUND(x,1),UBOUND(x,1)
! 10           14
  lb = LBOUND(x,1)
END FUNCTION lb_3
END PROGRAM pass_pointer_array_test
```

Example: passing pointer array(cont'd)

!change the main program following way what happens/why?

```
PROGRAM pass_pointer_array_test
  IMPLICIT NONE
  REAL, DIMENSION(:), POINTER :: a => NULL()
  REAL, DIMENSION(:), TARGET, ALLOCATABLE :: b !This line is new
  INTEGER n1,n2,i,lo_bound,allocstat
  n1=-2;n2=2
  ALLOCATE (b(n1:n2),STAT=allocstat) !changed line
  IF ( allocstat /= 0 ) STOP 'Memory allocation failed!'
  FORALL (i=n1:n2) b(i)=REAL(i) !changed nine
  a=>b(n1+1:) !new line
  lo_bound=lb_1(a)
  lo_bound=lb_2(a)
  lo_bound=lb_3(a,10)
```



Modules and derived data types

Contents

- Using modules
- Basic structure of modules
- Compiling a module
- Procedures
- Derived data types
- Abstract data types
- PRIVATE and PUBLIC keywords
- Object oriented programming
- Generic procedures
- Overloading operators and custom operators (extra)

Using modules

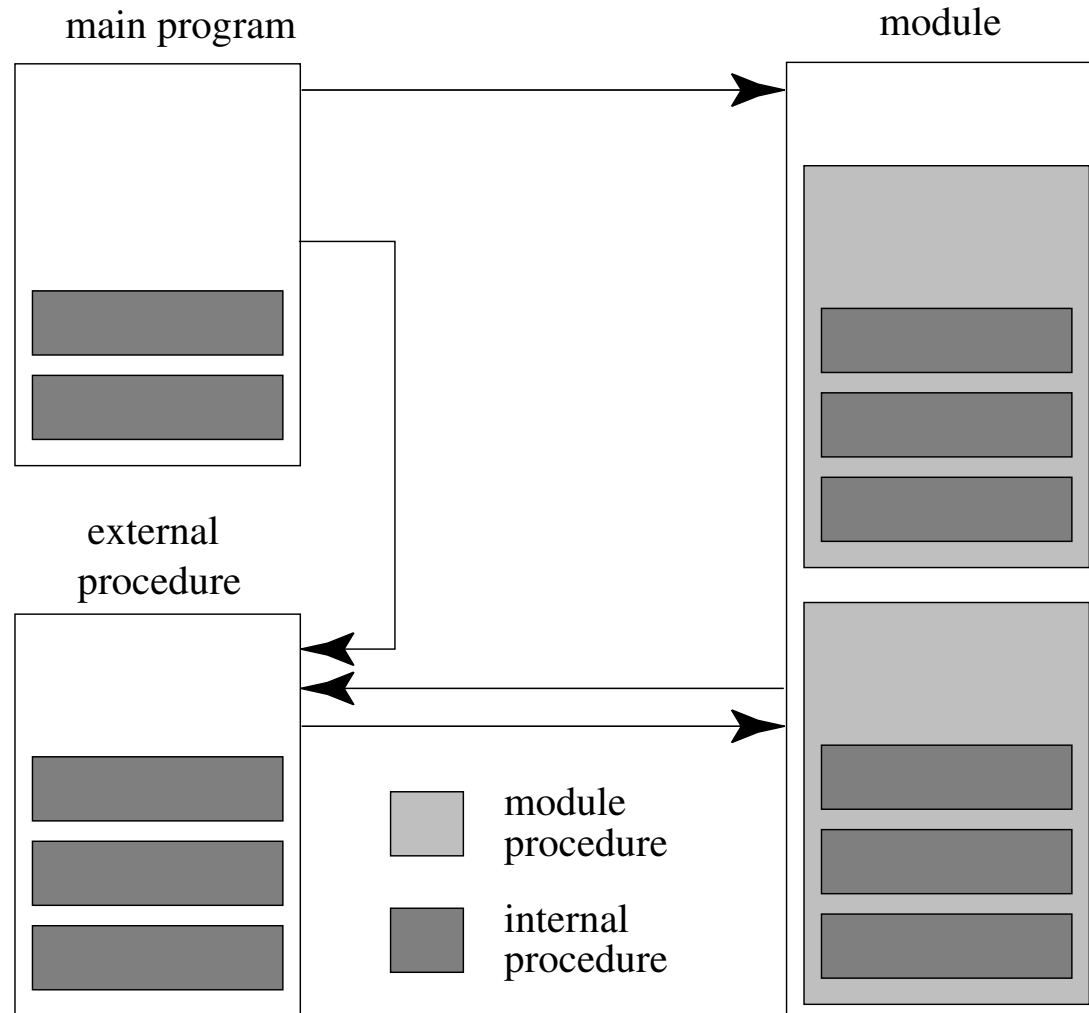
- *Modularity* means dividing a program into small, minimally dependent units called modules.
- Advantages:
 - Constants, variables, data types and procedures can be defined in modules.
 - Data structures and the related algorithms can be collected into a same program unit.
 - Makes possible to divide program into smaller self contained units.

Using modules (cont'd)

With modules one can

- make global definitions
- use same derived data types and procedures in different program units
- improve compile-time error checks
- “hide” implementation details (object-oriented programming)
- group routines and data structures
- define generic procedures and custom operators

Relation between program units



Basic structure of modules

- Definition (in a module file):

```
MODULE name  
  [definitions]  
[CONTAINS  
  SUBROUTINEs and/or FUNCTIONs ]  
END [MODULE [name]]
```

- Module is accessed in another program unit by
USE name
- Using keyword ONLY only a part of the module is used:
USE name, ONLY: label
- Labels can be renamed:
USE name, ONLY: new_label => label

Example: Kind parameters in modules

- Define KIND-parameter for REAL-variables (accmod.F90):

```
MODULE accuracy
  IMPLICIT NONE
  INTEGER, PARAMETER :: prec= SELECTED_REAL_KIND(12,100)
END MODULE accuracy
```

- Accuracy can thus be defined for a whole program in a single module
- An example of the main program:

```
PROGRAM example
  USE accuracy, ONLY : prec
  ...
  REAL (prec) :: x, y
```

- Accuracy can now be easily modified later if necessary

Compiling a module

- Module file must be compiled before the program units which use it.
- Produces (usually) two files (depends on computer/compiler):
 - Usually a “.mod” file named after the module name
 - An object file “.o” file named after the original source file
- For example:

```
% f95 -c accmod.f90
```

```
% f95 -o prog prog.f90 accmod.o
```

```
% ls
```

```
accmod.f90  accmod.o  accuracy.mod
```

```
prog  prog.f90
```

Global variables in modules

- COMMON definitions FORTRAN 77

```
COMMON/EQ/N,NTOT  
COMMON/TOL/ABSTOL,RELTOL
```

- The same using module

```
MODULE globals  
  INTEGER, SAVE :: n, ntot  
  REAL, SAVE :: abstol, reltol  
END MODULE globals
```

Option `SAVE` will save the values of the variables between consecutive calls

- Example

```
USE globals, ONLY : n, ntot  
USE globals, ONLY : tol => abstol
```

Module procedures

- Procedures (SUBROUTINEs and/or FUNCTIONs) can be defined in modules
- They are placed in CONTAINS section:

```
MODULE name  
  [definitions]  
  [CONTAINS  
   procedures]  
END [MODULE [name]]
```

- Procedure definition is otherwise similar to the standard procedure definition
- Usually placing procedures in modules gives compiler chance to discover errors better and even obtain better optimization



Example

- KIND parameter `r4` and subroutine `testSub` are defined in module `test`:

```
MODULE test
  IMPLICIT NONE
  INTEGER, PARAMETER :: &
    r4 = SELECTED_REAL_KIND(6,30)
CONTAINS
  SUBROUTINE testSub(x,y)
    IMPLICIT NONE
    REAL(r4), INTENT(IN) :: x
    REAL(r4), INTENT(OUT) :: y
    y = x + 1.0
  END SUBROUTINE testSub
END MODULE test
```

Example (cont'd)

● Main program

```
PROGRAM prog
  USE test
  IMPLICIT NONE
  REAL(KIND=r4) :: x, y
  x = 1.0
  CALL testSub(x,y)
  PRINT *, x, y
END PROGRAM prog
```

● Compilation

```
% f95 -c test.f90
% f95 -o prog prog.f90 test.o
```

Derived data types

- A derived data type is a structure of data types defined by the programmer

- Structure:

```
TYPE name_of_type
    [list of variable definitions]
END TYPE name_of_type
```

- Its components can comprise of any data types, including other derived types
- By combining modules and derived data types one can define *abstract data types*, which merge the data and the data handling operations
- It is most convenient to define derived data types within modules

Definition of a derived type

- Type is defined in a definition section of a program unit or a procedure

```
TYPE cowtype
  CHARACTER (LEN=50) :: name
  INTEGER :: born
  REAL :: milklitres
END TYPE cowtype
```

- A new variable of a derived type can now be defined as

```
TYPE(cowtype) :: cow
```

- Similarly for an array

```
TYPE(cowtype), DIMENSION (100) :: cattle
```

Initializing and accessing the variable

- Elements of the derived type are accessed with '%' operator:

- Syntax : type % element

- For example

```
cow % name = 'Heluna'  
age = 2008 - cow % born  
cattle(1) % milk_litres = 8772.3
```

- Initialization can be done by filling all the details separately

- Another way to initialize variable is

```
cow = cowtype('Heluna', 2005, 8772.3)
```

- Initialization can also be done within the variable definition

```
TYPE(cowtype) :: cow = cowtype('Heluna', 2005, 8772.3)
```

Derived data types

- Elements of a derived data type can also be derived types.
- Let's first create another derived type:

```
TYPE bulltype
  CHARACTER (LEN=50) :: name
  INTEGER :: born
END TYPE bulltype
```

- Let's now use both of the types for a new one:

```
TYPE farm
  REAL :: area
  TYPE(bulltype) :: bull
  TYPE(cowtype), DIMENSION(50) :: cattle
END TYPE farm
```

- As seen, a component (here: `cattle`) can also be an array of a derived type



Derived data types (cont'd)

- One can now define a new variable

```
TYPE(farm) :: piippola
```

- Using data types

- Consecutive components are referred using "%"-operator.
- Arrays are indexed in the usual manner:

```
piippola % cattle(1) % name = 'Heluna'  
piippola % cattle(1) % born = 2005  
piippola % cattle(1) % milklitres = 8772.3  
piippola % cattle(2) = cowtype('Mansikki', 2004, 9459.0)  
piippola % area = 128.4
```

Derived types and modules

- Suppose we want to define a type `point` in a main program:
 - ```
type point
 real :: x, y
end type point
```
  - Next, we want to make a subroutine which sums two `point`-types
- Problem: *Only internal subroutines can see the type which is defined in the main program*
  - Defining the similar type in a subroutine won't help, because it would be the subroutine's internal type
- The answer: **define a derived type in a module**

# Abstract data type

## ● Module

```
MODULE pointmod
 IMPLICIT NONE
 TYPE point
 ...
 END TYPE point
CONTAINS
 SUBROUTINE sum_points(p1,p2,p3)
 IMPLICIT NONE
 TYPE (point) :: p1, p2, p3

 END SUBROUTINE sum
END MODULE pointmod
```

## ● Main program

```
PROGRAM points
 USE pointmod
 IMPLICIT NONE
 TYPE (point) :: p1, p2, p3
 ...
 CALL sum_points(p1,p2,p3)
 ...
END PROGRAM points
```

- Type `point` (and subroutine `sum_points`) can also be accessed in other program units using the `USE`-statement

# Example: Vector calculus

- A module for vector calculus: Need
  - a vector type
  - a function for vector addition

```
MODULE vecmod
 IMPLICIT NONE
 TYPE vectype
 REAL :: x, y, z
 END TYPE vectype
CONTAINS
 FUNCTION addvec(v1,v2)
 IMPLICIT NONE
 TYPE(vectype), INTENT(IN) :: &
 v1, v2
 TYPE(vectype) :: addvec
 addvec%x = v1%x + v2%x
 addvec%y = v1%y + v2%y
 addvec%z = v1%z + v2%z
 END FUNCTION addvec
END MODULE vecmod
```



# Example: Vector calculus (cont'd)

## ● Main program

```
PROGRAM vectest
 USE vecmod, ONLY : vectype, &
 addvec
 IMPLICIT NONE
 TYPE(vectype) :: vec1, vec2, res
 vec1 = vectype(0.0, 1.0, 2.0)
 vec2 = vectype(-1.0, -1.0, -1.0)
 res = addvec(vec1,vec2)
 WRITE (*,'(3F7.3)') vec1
 WRITE (*,'(3F7.3)') vec2
 WRITE (*,'(3F7.3)') res
END PROGRAM vectest
```

## ● Output:

```
0.000 1.000 2.000
-1.000 -1.000 -1.000
-1.000 0.000 1.000
```

# PRIVATE and PUBLIC objects

- Objects in modules can be *PRIVATE* or *PUBLIC*
- By default they are all *PUBLIC*, i.e. visible to the program using the module
- Objects can be hidden from other program units by defining them as *PRIVATE*:

```
MODULE stuff
 IMPLICIT NONE
 REAL, PARAMETER :: pi = 3.14159265359
 INTEGER, PRIVATE :: io_unit
 INTEGER, PRIVATE :: i, j, k
END MODULE stuff
```

- Options are *PRIVATE* and *PUBLIC*.

# PRIVATE and PUBLIC objects

- Instead of defining each object PUBLIC or PRIVATE separately, a command PRIVATE or PUBLIC can be used:

```
MODULE stuff
 IMPLICIT NONE
 PRIVATE ! Nothing will be visible outside this module ...

 REAL, PARAMETER :: pi = 3.14159265359
 INTEGER :: io_unit
 INTEGER :: i, j, k
 PUBLIC :: pi ! .. except this variable

 CONTAINS
 ...
END MODULE stuff
```

# Nested modules: an example

- A module can use other modules as well:

```
MODULE accuracy
 IMPLICIT NONE
 ! Real number accuracy
 INTEGER, PARAMETER :: &
 r4 = &
 SELECTED_REAL_KIND(6, 30)
END MODULE accuracy
```

```
MODULE constants
 USE accuracy, ONLY : r4
 IMPLICIT NONE
 REAL(r4), PRIVATE :: x
 REAL(r4), PARAMETER, PUBLIC :: &
 MachEps = EPSILON(x), &
 MachHuge = HUGE(x), &
 MachTiny = TINY(x), &
 MachPrec = PRECISION(x)
END MODULE constants
```

# F95 and object oriented programming

- Objects of a derived type are PUBLIC by default, unless defined PRIVATE. For example:

```
MODULE vector
 IMPLICIT NONE
 TYPE vectortype
 PRIVATE
 REAL :: x, y, z
 END TYPE vectortype
END MODULE vector
```

- Now the components of `vectortype` can't be accessed from other program units.
- One needs a function to initialize the vector, as well as a function to access its components, etc.



# Object oriented programming (cont'd)

```
MODULE vector
...
CONTAINS
 FUNCTION initialize(x, y, z) RESULT(vect)
 IMPLICIT NONE
 REAL :: x, y, z
 TYPE(vectortype) :: vect
 vect % x = x ; vect % y = y ; vect % z = z
 END FUNCTION initialize

 FUNCTION comp(vect) RESULT(result)
 IMPLICIT NONE
 TYPE(vectortype) :: vect
 REAL, DIMENSION(3) :: result
 result(1)=vect%x; result(2)=vect%y; result(3)=vect%z
 END FUNCTION comp
END MODULE vector
```



# Object oriented programming (cont'd)

- Data structure can be accessed only via the interface defined in the module
- Module procedures behave like JAVA class methods
- This will help to detect and track errors in large programming projects
- If the data structure is altered, only the module procedure needs to be modified ...
- ... but that means there will be implicit change also in the procedures that are consuming services of such a module ...
- ... thus a small change in a critical data structure may require nearly complete recompilation of the software!

# Generic procedures

- Procedures which perform similar tasks can be defined as “generic procedures” (overloading).
  - Procedures can be called using the generic name, and the compiler chooses the correct procedure based on the argument type, number and dimensions.
- Generic name is defined in an `INTERFACE` section
- Both external and module procedures can be overloaded
- Overloading should be used with caution, as it may yield code which may be difficult to comprehend
- It is recommended to define generic procedures using modules

# Generic procedures: example

A generic procedure `swap` to exchange 2 REALs or characters:

```
MODULE swapmod
 IMPLICIT NONE
 INTERFACE swap
 MODULE PROCEDURE swap_real, swap_char
 END INTERFACE
CONTAINS
 SUBROUTINE swap_real(a, b)
 REAL, INTENT(INOUT) :: a, b
 REAL :: temp
 temp = a; a = b; b = temp
 END SUBROUTINE swap_real
 SUBROUTINE swap_char(a, b)
 CHARACTER, INTENT(INOUT) :: a, b
 CHARACTER :: temp
 temp = a; a = b; b = temp
 END SUBROUTINE swap_char
END MODULE swapmod
```



# Generic procedures: example

```
PROGRAM swap_main
 USE swapmod, ONLY : swap
 IMPLICIT NONE
 REAL :: a, b
 CHARACTER :: c, d
 a = 7.1; b = 10.9
 c = 'a'; d = 'b'
 WRITE(*,*) a, b, c, d
 CALL swap(a, b)
 CALL swap(c, d)
 WRITE(*,*) a, b, c, d
END PROGRAM swap_main
```

## Output:

```
7.0999999 10.8999996 ab
10.8999996 7.0999999 ba
```

Calling `swap` is independent of parameters.



# Operator overloading and custom operators

- It is possible to overload standard operators (e.g. +, \*, / etc.) and e.g. define them for derived data types.

- One can, for example, overload \* to perform matrix multiplications:  
 $C=A * B$

- Overloading is done using INTERFACE-command:

```
INTERFACE OPERATOR(oper)
 [definitions]
END INTERFACE
```

- It is strongly recommended to use modules for operator overloading.
- Modifying the action and definition range of standard internal operators must be done with caution!

# Operator overloading: +

- Let us define + operation for strings:

```
MODULE plus
 IMPLICIT NONE
 INTERFACE operator(+)
 MODULE PROCEDURE addst
 END INTERFACE
CONTAINS
 FUNCTION addst(string1,string2) RESULT(result)
 IMPLICIT NONE
 CHARACTER (LEN=*), INTENT(in):: string1, string2
 CHARACTER (LEN=LEN(string1) + LEN(string2)):: result
 result=string1//string2
 END FUNCTION addst
END MODULE plus
```

# Operator overloading: + (cont'd)

- Main program:

```
PROGRAM test
 USE plus
 IMPLICIT NONE
 PRINT *, 'Hip ' + 'hei!'
END PROGRAM test
```

- Output:

```
Hip hei!
```

# Overloading substitution

- Substitution operator (=) can also be overloaded:

```
INTERFACE ASSIGNMENT(=)
 [definition]
END INTERFACE
```

- Assignment is done using subroutine, whose arguments correspond to the left and right hand sides of the substitution.
- Subroutines first argument is type `INTENT ( OUT )` and second one `INTENT ( IN )`

# Overloading substitution: example

## ● Substituting an integer into a string

```
MODULE assgn
 IMPLICIT NONE
 INTERFACE ASSIGNMENT(=)
 MODULE PROCEDURE i2s
 END INTERFACE
CONTAINS
 SUBROUTINE i2s(string,n)
 IMPLICIT NONE
 CHARACTER (LEN=*), &
 INTENT(out):: string
 INTEGER, INTENT(in):: n
 WRITE(string,*) n
 END SUBROUTINE i2s
END MODULE assgn
```

## ● Usage:

```
PROGRAM test
 USE assgn
 IMPLICIT NONE
 CHARACTER (LEN=10):: string
 INTEGER:: n
 n=10
 string=n
 PRINT *, string
END PROGRAM test
```

## ● Output:

```
10
```

# Custom operators

- User can also define custom operators
- Operators name begin and end with a dot: `.OP.`
- Operator refers to a function, with arguments of type `INTENT ( IN )`.
- For example:

```
s = x .ip. y
h = .norm. x
```

- Unary operators precede binary operators in the execution order (i.e. unary operators are processed first)

# Custom operators: inner product

- Let us define an inner product operation `.ip.` for vectors

```
MODULE vecmod
 IMPLICIT NONE
 TYPE vectype
 REAL :: x, y, z
 END TYPE vectype
 INTERFACE OPERATOR(.ip.)
 MODULE PROCEDURE inner
 END INTERFACE
CONTAINS
 REAL FUNCTION inner(v1,v2)
 TYPE(vectype), INTENT(IN) :: v1, v2
 inner = v1%x*v2%x + v1%y*v2%y + v1%z*v2%z
 END FUNCTION inner
END MODULE vecmod
```

# Custom operators: inner product

## ● Usage:

```
program vecprod
 use vecmod
 implicit none
 type(vectype) v1, v2
 real s
 v1=vectype(2.0,0.0,-1.0)
 v2=vectype(3.5,1.5,2.5)

 s = v1 .ip. v2

 write(*,*) 'v1=', v1
 write(*,*) 'v2=', v2
 write(*,*) 's=', s

end program
```

# Transition to F95

## Contents

- Outlines
- Bad features of FORTRAN 77
- Transition from Fortran 77 to Fortran 95

# Outlines

- Most important advice when moving to new standard:  
*If it works, don't fix it!*
- Fortran 90/95 -compiler can be used with FORTRAN 77 -code.
- Old code can possibly be used with the new code, similarly as subroutine libraries.
- New programs should be made using Fortran 90/95.

# Bad features of FORTRAN 77

- Fixed form input
- Spaces were not necessary:

```
DO 15 I = 1,10
```

```
DO 15 I = 1.10
```

```
DO15I=1.10
```

- Implicit types are error prone:

```
INTEGER VALUE, VAL1
```

```
VALUE = 12
```

```
VAL1 = FUN(VALUE)
```

- Numeric types are difficult to port
- No dynamic or structured data types

# Bad features of FORTRAN 77

- Dependencies how data is stored (COMMON, EQUIVALENCE):

```
PROGRAM WHOOPS
REAL NUMBER, A, B
COMMON /OOPS/ NUMBER, A, B
CALL NOGOOD
WRITE (*,*) 'NUMBER,A,B: ', NUMBER, A, B
END

SUBROUTINE NOGOOD
COMMON /OOPS/ NUMBER, A, B
NUMBER = 3
A = 4
B = 5
END
```

## Output:

```
NUMBER,A,B: 4.2038954E-45 4.0000 5.0000
```

# Fortran 77 → Fortran 95

- Generate test material to check the correctness of code
- Do not use fixed format input
- Do not use implicit variable definition  
Instead, always use `IMPLICIT NONE`
- Do not use `DIMENSION-` and `PARAMETER-` definitions, but the corresponding type definitions:

```
INTEGER N
PARAMETER (N = 100)
REAL X
DIMENSION X(N)
```

→

```
INTEGER, PARAMETER :: n = 100
REAL, DIMENSION(n) :: x
```

- Do not use non-standard features such as `DOUBLE COMPLEX` and `REAL*16`

# Fortran 77 → Fortran 95

- Replace COMMON-blocks and BLOCK DATA -units with modules

```
COMMON/EQ/N,NTOT
COMMON/FCALLS/NC
```

→

```
MODULE global_var
 IMPLICIT NONE
 INTEGER, SAVE :: n,ntot,nc
END MODULE global_var
```

- Avoid EQUIVALENCE-definitions, because they will worsen the portability of code
- Use dynamic arrays instead of fixed work space arrays.

# Fortran 77 → Fortran 95

- Use modules, internal subroutines and `INTERFACE`-blocks to ensure error checks on procedure calls
- Replace *statement function* by internal procedures
- *External procedures* should be replaced by internal procedures or modules
- Use control structures
  - `IF ... THEN ... ELSE ... END IF`
  - `SELECT CASE ... CASE ... END SELECT`
  - `DO ... END DO`
  - `DO WHILE ... END DO`
  - `WHERE` and `FORALL`

# The good, the bad and the ugly features of F95

## Contents

- Good features
- Occasionally useful features
- Potentially dangerous features

# Good features

- `IMPLICIT NONE` removes the implicit types
- New parameter definitions  
`REAL, PARAMETER :: ...`  
make the code clearer and easier to read.
- Kind parameters (`KIND`) improve portability
- Fortran 95 control structures ease programming
- Free format source code is easier to read

# Good features...

- Modules improve reliability and effectiveness of programming (eg. compile time checks of argument lists)
- Dynamic memory allocation eases programming
- `INTENT`-options for procedure arguments improve compile time error checks
- New notations for logical symbols (`<`, `>`, `<=`, `>=`, `==` and `/=`) improve readability of the code
- Array syntax eases and improves effective coding and readability

# Occasionally useful features

- Derived data types enables creating complex structures (eg. lists, etc)
- Pointer types enables dynamical references to variables and arrays (or parts of them)
- Internal procedures are practical for making small auxiliary routines.
- Optional arguments make it possible to pass only part of the arguments – default values will be used for the others
- Keyword arguments can clarify the purpose and meaning of the procedure arguments

# Occasionally useful features...

- Generic procedures enable similar procedure calls for different argument types – but can make code more difficult to understand
- NAMELIST-structure is practical for small input (eg. program input data) but it should not be used for large amount of data
- Options PUBLIC and PRIVATE can be used in modules to enlarge/shrink visibility of module objects
- Control structure DO WHILE is practical in some situations. Compare for example:

```
DO WHILE (condition)
 ...
END DO
```

```
DO
 IF (.NOT. condition) EXIT
 ...
END DO
```

# Potentially dangerous features

- Using custom operators can make code difficult to understand, and ineffective
- Using Fortran 77:n features such as `COMMON`, `EQUIVALENCE`, `GOTO` etc. make code difficult to port and read
- Antiquated fixed form source code makes code difficult to read

# Fortran 2003 (and 2008)

- Lots of additions to standard
- Compiler support not yet sufficient
  - parametrized derived types
  - dynamic objects of types, parametrized types
  - object oriented programming: expandable types, inheritance
  - procedure pointers
  - interrupt (IEEE 754) handling
  - compatibility with C programs
  - operating system support: environment variables, command line arguments
  - asynchronic and derived type transfer
  - international character sets
  - etc.

# General instructions

- When code is easy to read, understand and support, it is also easy for the compiler.
- Write well constructed, clear and readable code. The next person trying to figure out the code will most probably be you – and after couple of months you have already forgotten what the code is supposed to do.
- Express what you want simply, directly and clearly; avoid any tricks.
- Let the program and the computer do the “dirty” work. Use existing subroutine libraries and do not reinvent the wheel.