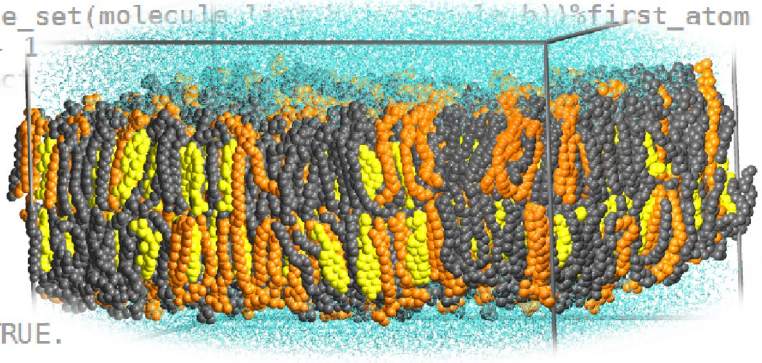
CSC

Sami Ilvonen

Fortran 95/2003

June 17-18, 2013
Hosted by University of Oulu

```
DO imolecule_kind_a=1,nmolecule_kind
   molecule_kind_a => molecule_kind_set(imolecule_kind_a)
   CALL get_molecule_kind(molecule_kind=molecule_kind_a, molecule_list=molecule_list_a,&
        natom=natom_mol_a)
   DO imolecule_a=1,SIZE(molecule_list_a)
      imol_a = imol_a + 1
      DO iatom_mol_a=1,natom_mol_a
         iatom_a = molecule_set(molecule_list_a(imolecule_a))%first_atom + &
             iatom_mol_a - 1
         imol_b = 0
         DO imolecule_kind_b=1,nmolecule_kind
            molecule_kind_b => molecule_kind_set(imolecule_kind_b)
            CALL get_molecule_kind(molecule_kind=molecule_kind_b,&
                 molecule_list=molecule_list_b, natom=natom_mol_b)
            DO imolecule_b=1,SIZE(molecule_list_b)
               imol_b = imol_b + 1
               DO iatom_mol_b=1,natom_mol_b
                  iatom_b = molecule_set(molecule
                      iatom_mol_b - 1
                  ! is this block ac
                  IF (symmetric)
                    IF (iatom_a
                      include_
                    ELSE
                      include_ab
                    END IF
                  ELSE
                    include_ab = .TRUE.
                  END IF
```



Agenda

Monday

9.00-9.45	Getting started with Fortran
10.00-10.45	Fortran arrays
11.00-12.00	Exercises
12.00-13.00	Lunch break
13.00-13.45	Procedures & modules
14.00-15.45	Exercises
15.45-16.00	Wrap-up, Q&A

Tuesday

9.00-9.45	File I/O
10.00-10.45	Derived datatypes
11.00-12.00	Exercises
12.00-13.00	Lunch break
13.00-13.45	Other handy Fortran features
14.00-15.45	Exercises
15.45-16.00	Wrap-up, Q&A

Web resources

- CSC's Fortran95/2003 Guide (in Finnish) for free
  http://www.csc.fi/csc/julkaisut/oppaat
- Fortran wiki: a resource hub for all aspects of Fortran programming
  http://fortranwiki.org
- GNU Fortran online documents
  http://gcc.gnu.org/onlinedocs/gcc-4.5.0/gfortran
- Code examples
  http://www.nag.co.uk/nagware/examples.asp
  http://www.personal.psu.edu/jhm/f90/progref.html
- Mistakes in Fortran 90 Programs That Might Surprise You
  http://www.cs.rpi.edu/~szymansk/OOF90/bugs.html

PART I: GETTING STARTED WITH FORTRAN

Outline

- First encounter with Fortran
- Variables and their assignment
- Control structures

Why learn Fortran?

- Well suited for **numerical computations**
  - Likely over 50% of scientific applications are written in Fortran
- **Fast** code (compilers can optimize well)
- Handy **array data types**
- **Clarity** of code
- **Portability** of code
- Optimized **numerical libraries** available

Fortran through the ages

- John W. Backus et al (1954): The IBM Mathematical **For**mula **Tran**slating System
- Early years development: Fortran II (1958), Fortran IV (1961), Fortran 66 & Basic Fortran (1966)
- Fortran 77 (1978)
- Fortran 90 (1991) major revision, Fortran 95 (1995) a minor revision to it

Fortran through the ages

- Fortran 2003: major revision, adding e.g. object-oriented features
  - ”Fortran 95/2003” is the current *de facto* standard
- The latest standard is Fortran 2008 (approved 2010), a minor upgrade to 2003

Look & Feel

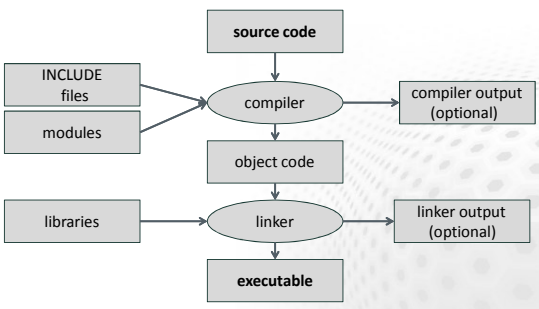
```
PROGRAM square_root_example
! Comments start with an exclamation point.
! You will find data type declarations, couple arithmetic operations
! and an interface that will ask a value for these computations.
 IMPLICIT NONE
 REAL :: x, y
 INTRINSIC SQRT ! Fortran standard provides many commonly used functions

 ! Command line interface. Ask a number and read it in
 WRITE (*,*) 'Give a value (number) for x:'
 READ (*,*) x

 y=x**2+1   ! Power function and addition arithmetic

 WRITE (*,*) 'given value for x:', x
 WRITE (*,*) 'computed value of x**2 + 1:', y
 ! Print the square root of the argument y to screen
 WRITE (*,*) 'computed value of SQRT(x**2 + 1):', SQRT(y)
END PROGRAM square_root_example
```


Compiling and linking



Variables

```
IMPLICIT NONE

INTEGER :: n0

REAL :: a, b
REAL :: r1=0.0

COMPLEX :: c
COMPLEX :: imag_number=(0.1, 1.0)

CHARACTER(LEN=80) :: place
CHARACTER(LEN=80) :: name='James Bond'

LOGICAL :: test0 = .TRUE.
LOGICAL :: test1 = .FALSE.

REAL, PARAMETER :: pi=3.14159
```

Variables must be *declared* at the beginning of the program or procedure

The *intrinsic* data types in Fortran are INTEGER, REAL, COMPLEX, CHARACTER and LOGICAL

They can also be given a value at declaration

Constants defined with the PARAMETER clause – they cannot be altered after their declaration

Assignment statements

```
PROGRAM numbers
  IMPLICIT NONE
  INTEGER :: i
  REAL :: r
  COMPLEX :: c, cc
  i = 7
  r = 1.618034
  c = 2.7182818   !same as c = CMPLX(2.7182818)
  cc = r*(1,1)
  CMPLX(r)
  WRITE (*,*) i, r, c, cc
END PROGRAM
```

Automatic change of representation, works between all numeric intrinsic data types

How can I convert numbers to character strings and vice versa? See "INTERNAL I/O" in the File I/O lecture.

Output (one integer and real and two complex values) :

```
7  1.618034  (2.718282, 0.000000)  (1.618034, 1.618034)
```


Operators

Arithmetic operators

```
REAL :: x,y
INTEGER :: i = 10
x=2.0**(-i)  !power function and negation     precedence: first
x=x*REAL(i)  !multiplication and type change  precedence: second
x=x/2.0      !division                        precedence: second
i=i+1        !addition                        precedence: third
i=i-1        !subtraction                     precedence: third
```

Relational operators

```
.LT. or <     !less than
.LE. or <=    !less than or equal to
.EQ. or ==    !equal to
.NE. or /=    !not equal to
.GT. or >     !greater than
.GE. or >=    !greater than or equal to
```

Logical operators

```
.NOT.      !logical negation              precedence: first
.AND.      !logical conjunction           precedence: second
.OR.       !logical inclusive disjunction precedence: third
```


Arrays

```
INTEGER, PARAMETER :: M = 100, N = 500
INTEGER :: idx(M)
REAL :: vector(0:N-1)
REAL :: matrix(M, N)
CHARACTER (len = 80) :: screen ( 24)

! or

INTEGER, DIMENSION(1:M) :: idx
REAL, DIMENSION(0:N-1) :: vector
REAL, DIMENSION(M, N) :: matrix
CHARACTER(len=80), dimension(24) :: screen
```

By default, indexing starts from 1

Control structures: conditionals

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

 WRITE(*,*)' Give x and y :'
 READ(*,*) x, y
 eps = EPSILON(x)

 IF (ABS(x) > eps) THEN
     t=y/x
 ELSE
     WRITE(*,*)'division by zero'
     t=0.0
 END IF
 WRITE(*,*)' y/x = ',t
END PROGRAM
```


Control structures: loops

```
! DO loop with an integer counter (count controlled)
INTEGER :: i, stepsize, NumberOfPoints
INTEGER, PARAMETER :: max_points=100000
REAL :: x_coodinate(max_points), x, totalsum
...
stepsize=2
DO i = 1, NumberOfPoints, stepsize
   x_coordinate(i) = i*stepsize*0.05
END DO

! Condition controlled loop (DO WHILE)
totalsum = 0.0
READ(*,*) x
DO WHILE (x > 0)
   totalsum = totalsum + x
   READ(*,*) x
END DO
```


Control structures: loops

```
! DO loop without loop control

REAL :: x, totalsum, eps
totalsum = 0.0
DO
  READ(*,*) x
  IF (x < 0) THEN
     EXIT           ! exit the loop
  ELSE IF (x > upperlimit) THEN
     CYCLE          ! do not execute any statements but
                    ! cycle back to the beginning of the loop
  END IF
  totalsum = totalsum + x
END DO
```


Control structures: select case

- SELECT CASE statements matches the entries of a list against the case index
  - Only one found match is allowed
  - Usually arguments are character strings or integers
  - DEFAULT branch if no match found

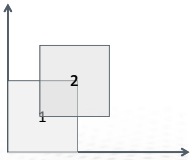
```
...
INTEGER :: i
LOGICAL :: is_prime_number,
test_prime_number
...
SELECT CASE (i)
  CASE (2,3,5,7)
    is_prime_number = .TRUE.
  CASE (1,4,6,8:10)
    is_prime_number = .FALSE.
  CASE DEFAULT
    is_prime_number=test_prime_number(i)
END SELECT
...
```


Control structures example

```
PROGRAM gcd
! Computes the greatest common divisor, Euclidean algorithm
  IMPLICIT NONE
  INTEGER :: m, n, t
  WRITE(*,*)' Give positive integers m and n :'
  READ(*,*) m, n
  WRITE(*,*)'m:', m,' n:', n
  positive_check: IF (m > 0 .AND. n > 0) THEN
     main_algorithm: DO WHILE (n /= 0)
        t = MOD(m,n)
        m = n
        n = t
     END DO main_algorithm
     WRITE(*,*)'Greatest common divisor: ',m
   ELSE
     WRITE(*,*)'Negative value entered'
   END IF positive_check
END PROGRAM gcd
```

Labels can be given to control structures and used in conjunction with e.g. exit and cycle statements

Another example




```
PROGRAM placetest
 IMPLICIT NONE
 LOGICAL :: in_square1, in_square2
 REAL :: x,y

 WRITE(*,*) 'Give point coordinates x and y'
 READ (*,*) x, y
 in_square1 = (x >= 0. .AND. x <= 2. .AND. y >= 0. .AND. y <= 2.)
 in_square2 = (x >= 1. .AND. x <= 3. .AND. y >= 1. .AND. y <= 3.)
 IF (in_square1 .AND. in_square2) THEN      ! inside both
    WRITE(*,*) 'Point within both squares'
 ELSE IF (in_square1) THEN                  ! inside square 1 only
    WRITE(*,*) 'Point inside square 1'
 ELSE IF (in_square2) THEN                  ! inside square 2 only
    WRITE(*,*) 'Point inside square 2'
 ELSE                                       ! both are .FALSE.
    WRITE(*,*) 'Point outside both squares'
 END IF
END PROGRAM placetest
```


Source code remarks

- A variable name can be no longer than 31 characters (containing only letters, digits or underscore, must start with a letter)
- Maximum row length is 132 characters
- There can be max 39 continuation lines
  - if a line is ended with ampersand (&), the line continues onto the next line.
- No distinction between lower and uppercase characters
  - Character strings are case sensitive

Source code remarks

```
! Character strings are case sensitive
CHARACTER(LEN=32) :: ch1, ch2
Logical :: ans
ch1 = 'a'
ch2 = 'A'
ans = ch1 .EQ. ch2
WRITE(*,*) ans     ! OUTPUT from that WRITE statement is: F

! When strings are compared
! the shorter string is extended with blanks
WRITE(*,*) 'A' .EQ. 'A '    ! OUTPUT: T
WRITE(*,*) 'A' .EQ. ' A'    ! OUTPUT: F

! Statement separation: newline and semicolon, ;
! Semicolon as a statement separator
a = a * b; c = d**a
! The above is equivalent to following two lines
a = a * b
c = d**a
```


Summary

- Fortran 95/2003 is – despite its long history - a modern programming language especially for scientific computing
  - Versatile, quite easy to learn, powerful
- In our first encounter, we discussed
  - Variables & data types
  - Control structures: loops & conditionals
  - Operators

PART II: FORTRAN ARRAYS

Outline

- Array syntax & array sections
- Dynamic memory allocation
- Array intrinsic functions
- Pointers to arrays

Significance of Fortran arrays

- *Fortran arrays* enable a natural and versatile way to access multi-dimensional data during computation
  - Matrices, vectors,...
  - Array has particular data type (same for all elements)
  - *Dimension* specified in the variable declaration
  - Fortran 95 supports up to 7 dimensional arrays

Loop order in multi-dimensional arrays

- Always increment the *left-most* index of multi-dimensional arrays in the *innermost* loop (i.e. fastest)
- Some compilers (with sufficient optimization flags) may re-order loops automatically

```
do i=1,N
  do j=1,M
    y(i) = y(i)+ a(i,j)*x(j)
  end do
end do
```

```
do j=1,M
  do i=1,N
    y(i) = y(i)+ a(i,j)*x(j)
  end do
end do
```


Array syntax

- In older Fortran, arrays were traditionally accessed element-by-element basis
- Modern Fortran has a way of accessing several elements in one go: *array syntax*

```
y(:) = y(:)  +  A(:,j) * x(j)
```

- Array syntax improves code readability and performance

Array syntax

- Element-by-element initialization

```
do j = 0, 10
   vector (j) = 0
   idx (j) = j
end do
```

- Using array syntax in initialization

```
vector = 0
! or
vector(:) = 0

idx(0:10) = (/ (j, j = 0, 10) /)
```


Array syntax

- Array syntax allows for less explicit DO loops

```
integer :: m = 100, n = 200
real    :: a(m,n), x(n), y(m)
integer :: i, j

y = 0.0
outer_loop: do j = 1, n
  inner_loop: do i = 1, m
    y(i) = y(i) + a(i,j) * x(j)
  end do inner_loop
end do outer_loop
```

```
integer :: m = 100, n = 200
real    :: a(m,n), x(n), y(m)
integer :: j

y = 0.0
outer_loop: do j = 1, n
    y(:) = y(:) + a(:,j) * x(j)
end do outer_loop
```


Array sections

- With Fortran array syntax we can access a part of an array in a pretty intuitive way: *array sections*

```
Sub_Vector(3:N+8)  = 0
Every_Third(1:3*N+1 : 3) = 1
Diag_Block(i-1:i+1, j-2:j+2) = k
```

- Sections enable us to refer to (say) a sub-block of a matrix, or a sub-cube of a 3D array:

```
REAL(kind = 8) :: A ( 1000, 1000)
INTEGER(kind = 2) :: pixel_3D(256, 256, 256)
A(2:500, 3:300:3) = 4.0
pixel_3D(128:150, 56:80, 1:256:8) = 32000
```

Array sections

- When copying array sections, then both left and right hand sides of the assignment statement has to have conforming dimensions

```
LHS(1:3, 0:9) = RHS(-2:0, 20:29)  ! This is OK

! but here is an error :

LHS(1:2, 0:9) = RHS(-2:0, 20:29)
```


Dynamic memory allocation

- Memory allocation is *static* if the array dimensions have been declared at compile time
- If the sizes of an array depends on the input of the program, its memory should be *allocated* at runtime
  - Memory allocation becomes *dynamic*

Dynamic memory allocation

- Fortran provides two different mechanisms to allocate memory dynamically through arrays:
  - Array variable declaration has an ALLOCATABLE attribute
    - memory is allocated through the ALLOCATE statement
    - and freed through DEALLOCATE
  - A variable, which is declared in the procedure with size information coming from the argument list or from a module, is an *automatic array*
    - no ALLOCATE or DEALLOCATE is needed

Dynamic memory allocation

```
INTEGER :: M=100, N=200, alloc_stat
INTEGER, ALLOCATABLE :: idx(:)
REAL, ALLOCATABLE :: mat(:,:)

ALLOCATE(idx(0:M-1), STAT=alloc_stat)
IF (alloc_stat /= 0) CALL abort()

ALLOCATE(mat(M,N), STAT=alloc_stat)
IF (alloc_stat /= 0) CALL abort()
...
DEALLOCATE(idx, mat)
```


Memory allocation with automatic arrays

```
SUBROUTINE CALCULATE(M, N)
  INTEGER, INTENT(IN) :: M, N ! Intended dimensions
  INTEGER :: idx(0:M-1)       ! An automatic array
  REAL :: mat(M,N)            ! An automatic array

! No explicit ALLOCATE - but no checks upon failure either
  ...
  CALL DO_SOMETHING(M, N, idx, mat)
  ...
! No explicit DEALLOCATE - memory gets reclaimed automatically

END SUBROUTINE CALCULATE
```


Array intrinsic functions

- Built-in functions can apply various operations on whole array, not just array elements
- As a result either another array or just a scalar value is returned
- A subset selection through *masking* is possible
  - Masking and use of array (intrinsic) functions is often accompanied with use of FORALL and WHERE array statements

Array intrinsic functions

- **`SIZE(array[, dim])`** returns # of elements in the array, optionally along the specified dimension
- **`SHAPE(array)`** returns an INTEGER vector containing SIZE of array with respect to each of its dimension
- **`COUNT(L_array[, dim])`** returns the count of elements which are .TRUE. in the LOGICAL L_array
- **`SUM(array[, dim][, mask])`** : sum of the elements, optionally along a dimension, and optionally under mask

Array intrinsic functions

- **`ANY(L_array[, dim])`** returns a scalar value of .TRUE. if any value in LOGICAL L_array is found to be .TRUE.
- **`ALL(L_array[, dim])`** returns a scalar value of .TRUE. if all values in LOGICAL L_array are .TRUE.
- **`MINVAL/MAXVAL(array[, dim][, mask])`** return the minimum/maximum value in a given array [along specified dimension] [, under mask]
- **`MINLOC/MAXLOC(array[, mask])`** return a vector of location(s) [, under mask], where the minimum/maximum value(s) is/are found

Array intrinsic functions

```
INTEGER :: M, N
REAL :: X(M,N), V(N)
PRINT *, SIZE(X), SIZE(V) ! M, N, N
PRINT *, SHAPE(X)         ! M, N
PRINT *, SIZE(SHAPE(X))   ! 2
PRINT *, COUNT(X >= 0)
PRINT *, SUM(X, DIM=2, MASK=X < 1)
```

```
PRINT *, ANY(V > -1 .and. V < 1)
PRINT *, ALL(X >= 0, DIM=1)
PRINT *, MINVAL(V), MAXVAL(V)
PRINT *, MINLOC(V), MAXLOC(V)
```


Array intrinsic functions

- **`RESHAPE(array, shape)`** returns a reconstructed array with different shape than in the input array, for example:
  - Can be used as a single line statement to initialize an array (often in expense of readability)
  - Create from an M-by-N matrix a vector of length MxN

```
INTEGER :: M, N
REAL :: A(M,N), V(M*N)
! "Carbon"-copy A to V without loops
V = RESHAPE(A, SHAPE(V))
```


Array intrinsic functions

- Some array functions manipulate vectors/matrices effectively :
  - **`DOT_PRODUCT(a, b)`** returns a dot product of two vectors
  - **`MATMUL(a, b)`** returns matrix multiply of two matrices
  - **`TRANSPOSE(a)`** returns transposed of the input matrix

```
INTEGER :: L, M, N
REAL :: A(L,M), B(M,N), C(L,N)
REAL :: A_tr(M,L)
REAL :: V1(N), V2(N), DOTP
```

```
A_tr = TRANSPOSE(A)
C = MATMUL(A, B)
DOTP = DOT_PRODUCT(V1, V2)
```


Array intrinsic functions

- Array control statements **FORALL** and **WHERE** are commonly used in the context of manipulating arrays
  - These are frankly speaking not array intrinsic functions, but very closely related to
- They can provide a masked assignment of values using effective vector operations

Array intrinsic functions

- Examples of array control statements

```
INTEGER :: j, ix(5)

ix(:) = (/ (j, j=1, size(ix)) /)

WHERE (ix == 0) ix = -9999

WHERE (ix < 0)
    ix = -ix
ELSEWHERE
    ix = 0
END WHERE
```

```
INTEGER :: j
REAL :: a(100,100), b(100), c(100)

! Fill in diagonal matrix
FORALL (j=1:100) a(j,j) = b(j)

! Fill in lower bi-diagonal matrix
FORALL (j=2:100) a(j,j-1) = c(j)
```


Pointers to arrays

- The POINTER attribute enables to create array (or scalar) *aliasing variables*
- Pointer variables are usually employed to *refer* to another array or array section
- A pointer variable can also be a sole variable itself, but requires ALLOCATE
  - This is not a recommended practice – use the ALLOCATABLE attribute and employ POINTERs for aliasing only
- C programmers: a "pointer" has a slightly different meaning in C and Fortran

Pointers to arrays

- A POINTER can refer to an already allocated memory region

```
INTEGER, POINTER :: p_x(:) => NULL()
INTEGER, TARGET :: x(1000)
...
p_x => x
p_x => x(2 : 300)
p_x => x(1 : 1000 : 5)
...
p_x(1) = 0
NULLIFY(p_x)
```

Initialized to point to nothing

Pointers provide a neat way for array sections

This would change also x(1) to 0

Disconnects p_x's connection to x

Summary

- *Arrays* make Fortran language a very versatile vehicle for computationally intensive program development
- Using its *array syntax*, vectors and matrices can be initialized and used in a very intuitive way
- *Dynamic memory allocation* enables sizing of arrays according to particular needs
- *Array intrinsic functions* further simplify coding effort and improve code readability

PART III: PROCEDURES & MODULES

Outline

- Structured programming
- Procedures: functions and subroutines
- Procedure arguments
- Modules

Structured programming

- Structured programming based on program sub-units (*functions*, *subroutines* and *modules*) enables
  - testing and debugging separately
  - re-use of code
  - improved readability
  - re-occurring tasks
- The key to success is in well defined data structures and scoping, which lead to clean procedure interfaces

What are procedures?

- With procedures we mean *subroutines* and *functions*
- Subroutines exchange data through its argument lists

```
CALL mySubroutine(arg1, arg2, arg3)
```

- Functions return a value

```
value = myFunction(arg1, arg2)
```

- Both can also interact with the rest of the program through module (global) variables

Declaration

Function

```
[TYPE] FUNCTION func(arg1,
    arg2,...) [RESULT(arg3)]
  [declarations]
  [statements]
END FUNCTION func
```

- Call convention

```
res = func(arg1, arg2,...)
```

Subroutine

```
SUBROUTINE sub(arg1, arg2,...)
  [declarations]
  [statements]
END SUBROUTINE sub
```

- Call convention

```
CALL sub(arg1, arg2,...)
```


Declaration

```
REAL FUNCTION dist(x, y)
 IMPLICIT NONE
 REAL :: x, y
 dist = SQRT(x**2 + y**2)
END FUNCTION dist


PROGRAM do_something
...
 r = dist(x, y)
...
```

```
SUBROUTINE dist(x, y, d)
 IMPLICIT NONE
 REAL :: x, y, d
 d = SQRT(x**2 + y**2)
END SUBROUTINE dist


PROGRAM do_something
 ...
 call dist(x, y, r)
 ...
```


Procedure types

- There are four procedure types in Fortran 90: *intrinsic, external*, *internal* and *module* procedures
- Procedure types differ in
  - Scoping, i.e. what data and other procedures a procedure can access
  - Interface type, explicit or implicit
- In Fortran the procedure arguments are always passed by reference, i.e. just as a pointer to a location in memory
- Compiler can check the argument types of the at compile time only if the interface is explicit

Procedure types, cont.

- The interfaces of the intrinsic, internal and module procedures are explicit
- The interfaces of the external procedures, such as many library subroutines, are implicit. You can write an explicit interface to those, though.
- Intrinsic procedures are the procedures defined by the programming language itself, such as

```
INTRINSIC SIN
```

Internal procedures

- Each program unit (program/subroutine/function) may contain *internal procedures*

```
SUBROUTINE mySubroutine
    …
    CALL myInternalSubroutine
    ...
CONTAINS
    SUBROUTINE myInternalSubroutine
        …
    END SUBROUTINE myInternalSubroutine
END SUBROUTINE mySubroutine
```


Internal procedures, cont.

- Declared at the end of a program unit after the CONTAINS statement
  - Nested CONTAINS statements are not allowed
- Scoping: internal procedure can access the parent program unit's variables and objects
- Often used for "small and local, convenience" subroutines within a program unit

External procedures

- Declared in a separate program unit
  - Referred to with the EXTERNAL keyword
  - Compiled separately and linked to the final executable
- Do not use them within a program, module procedures provide much better compile time error checking
- External procedures are often needed when using
  - procedures written with different programming language
  - library routines (e.g. BLAS & MPI libraries)
  - old F77 subroutines

Procedure arguments

- Call by reference:
  - Means that only the memory addresses of the arguments are passed to the called procedure
  - Any change to argument changes the actual argument
  - Compiler can check the argument types only if the interface is explict, i.e. compiler has information about the called procedure at compile time.
  - INTENT keyword adds readability and possibility for more compile-time error catching

INTENT keyword

- Declares how formal argument is intended for transferring a value
  - in: the value of the argument cannot be changed
  - out: the value of the argument must be provided
  - inout (default)
- Compiler uses INTENT for error checking and optimization

```
SUBROUTINE foo(x, y, z)
  IMPLICIT NONE
  REAL, INTENT(in)    :: x
  REAL, INTENT(inout) :: y
  REAL, INTENT(out)   :: z

  x = 10 ! Compilation error
  y = 10    ! Correct
  z = y * x ! Correct
END SUBROUTINE foo
```


Passing array arguments

- Two (three) ways to pass arrays to procedures
  - Explicit shape array (dimensions passed explicitly, F77'tish)

```
subroutine foo(size1, size2, ..., matrix, ...)
    implicit none
    integer :: size1, size2
    real, dimension(size1, size2) :: matrix
    …
```

  - Assumed shape array (requires explicit interface)

```
subroutine foo(matrix)
   real, dimension(:,:) :: matrix
```

    - One can use the intrinsic function SIZE for checking the actual dimensions

Procedure arguments

- We may pass into procedures also *other procedures* (i.e., not only data)
- Internal procedures cannot be used as arguments

```
PROGRAM degtest
   IMPLICIT NONE
   INTRINSIC ASIN, ACOS, ATAN
   WRITE (*,*) 'arcsin(0.5): ', deg(ASIN,0.5)
   WRITE (*,*) 'arccos(0.5): ', deg(ACOS,0.5)
   WRITE (*,*) 'arctan(1.0): ', deg(ATAN,1.0)
CONTAINS
 REAL FUNCTION deg(f, x)
   IMPLICIT NONE
   INTRINSIC ATAN
   REAL, EXTERNAL :: f
   REAL, INTENT(IN) :: x
   deg = 45 * f(x) / ATAN(1.0)
 END FUNCTION deg
END PROGRAM degtest
```


Modular programming

- Modularity means dividing a program into minimally dependent *modules*
  - Enables division of the program into smaller self-contained units
- Where to employ Fortran modules
  - Global definitions of procedures, variables and constants
  - Compilation-time *error checking*
  - Hiding *implementation details*
  - *Grouping* routines and data structures
  - Defining *generic procedures* and custom operators

Module procedures & variables

Declaration

```
MODULE check
 IMPLICIT NONE
 INTEGER, PARAMETER :: &
   longint = SELECTED_INT_KIND(8)
CONTAINS
  FUNCTION check_this(x) RESULT(z)
   INTEGER(longint):: x, z
   ...
  END FUNCTION
END MODULE check
```

Module procedures are declared after the CONTAINS statement

Usage

Procedures defined in modules can be referred to in any other program unit with the USE clause

```
PROGRAM testprog
  USE check
  IMPLICIT NONE
  INTEGER(KIND=longint) :: x,test
  test=check_this(x)
END PROGRAM testprog
```

- A good habit

```
USE check, ONLY: longint
```


Global data/variables

- Global variables can be accessed from any program unit
- F90 module variables provide controllable way to define and use global variables

```
MODULE commons
 INTEGER, PARAMETER :: r = 0.42
 INTEGER, SAVE :: n, ntot
 REAL, SAVE :: abstol, reltol
END MODULE commons
```

  - Explicit interface: type checking, limited scope
- Implemented as *common blocks* in old F77 codes

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

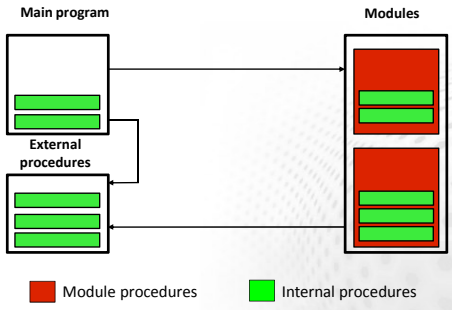
  - Extremely error prone

Visibility of module objects

- Variables and procedures in modules can be PRIVATE or PUBLIC
  - PUBLIC = visible for all program units using the module (default)
  - PRIVATE will hide the objects from other program units

```
REAL :: x, y
PRIVATE :: x
PUBLIC :: y
```


Program units



Summary

- Procedural programming makes the code more readable and easier to develop
  - Procedures encapsulate some piece of work that makes sense and may be worth re-using elsewhere
- Fortran uses *functions* and *subroutines*
  - Values of procedure arguments may be changed upon calling the procedure
- Fortran *modules* are used for modular programming and data encapsulation

PART IV: INPUT/OUTPUT

Outline

- Input/output (I/O) formatting
- Internal I/O
- File I/O
  - File opening and closing
  - Writing and reading to/from a file
  - Formatted and unformatted (binary) files
  - Stream I/O

Input/Output formatting

- To prettify output and to make it human readable, use FORMAT descriptors in connection with the WRITE statement
- Although less often used nowadays, it can also be used with READ to input data at fixed line positions and using predefined field lengths
- Use either through FORMAT statements, CHARACTER variable or embedded in READ / WRITE fmt keyword

Output formatting

Data type	FORMAT descriptors	Examples

Integer	`Iw, Iw.m`	`WRITE(*,'(I5)') J`
		`WRITE(*,'(I5.3)') J`
		`WRITE(*,'(I0)') J`

Real (decimal and exponential forms, auto-scaling)	`Fw.d`
	`Ew.d`
	`Gw.d`
		`WRITE(*,'(F7.4)') R`
		`WRITE(*,'(E12.3)') R`
		`WRITE(*,'(G20.13)') R`

Character	`A, Aw`	`WRITE(*,'(A)') C`
Logical	`Lw`	`WRITE(*,'(L2)') L`

w=width of the output field, **d**=number of digits to the right of decimal point, **m**=minimum number of characters to be used.
Variables: Integer :: J, Real :: R, Character :: C, Logical :: L

Output formatting: miscellaneous

- With complex numbers provide format for both real and imaginary parts:

```
COMPLEX :: Z
WRITE (*,'(F6.3,2X,F6.3)') Z
```

- Line break and tabbing:

```
WRITE (*,'(F6.3,/,F6.3)') X, Y
WRITE (*,'(I5,T20,I5)') I, J
```

- It is possible that an edit descriptor will be repeated a specified number of times

```
WRITE (*,'(5I8)')
WRITE (*,'(3(I5,F8.3))')
```


Internal I/O

- Often it is necessary to filter out data from a given character string
- Or to pack values into a character string
- Fortran internal I/O with READ & WRITE becomes handy
- No actual (physical) files are involved at all

Internal I/O : Examples

- Extract a number from a given character string

```
CHARACTER(LEN=13) :: CL ='Time step# 10'
INTEGER :: ISTEP
READ(CL,fmt='(10X,I3)') ISTEP
```

- Write data to a character string

```
INTEGER :: njobs
CHARACTER(LEN=60) :: CL
WRITE(CL,'(A,I0)') 'The number of jobs completed = ', njobs
```


Opening & closing files : basic concepts

- Writing to or reading from a file is similar to writing onto a terminal screen or reading from a keyboard
- Differences
  - File must be opened with an OPEN statement, in which the unit number and (optionally) the file name are given
  - Subsequent writes (or reads) must to refer to the given unit number
  - File should be closed at the end

Opening & closing a file

- The syntax is (the brackets [ ] indicate optional keywords or arguments)

```
OPEN([unit=]iu, file='name' [, options])
CLOSE([unit=]iu [, options])
```

- For example

```
OPEN(10, file= 'output.dat', status='new')
CLOSE(unit=10, status='keep')
```


Opening & closing a file

- The first parameter is the unit number
- The keyword `unit=` can be omitted
- The unit numbers 0, 5 and 6 are predefined
  - 0 is output for standard (system) error messages
  - 5 is for standard (user) input
  - 6 is for standard (user) output
  - These units are opened by default and should not be re-opened nor closed by the user

Opening & closing a file

- The *default input/output unit* can be referred with a star:

```
WRITE(*, ...)
READ(*, ...)
```

  - Note that these are *not* necessarily the same as the stdout and stdin unit numbers 6 and 5
- If the file name is omitted in the OPEN, the file name will based on unit number being opened, e.g. for unit=12 this usually means the filename 'fort.12'

File opening options

- *status* : existence of the file
  - 'old', 'new', 'replace', 'scratch', 'unknown'
- *position* : offset, where to start writing
  - 'append'
- *action* : file operation mode
  - 'write', 'read', 'readwrite'
- *form* : text or binary file
  - 'formatted', 'unformatted'

File opening options

- *access* : direct or sequential file access
  - 'direct', 'sequential', 'stream',
- *iostat* : error indicator, (output) integer
  - Non-zero only upon an error
- *err* : the label number to jump upon error
- *recl* : record length, (input) integer
  - For direct access files only
  - Warning (check): may be in bytes *or* words

File opening: file properties

- Use INQUIRE statement to find out information about
  - file existence
  - file unit open status
  - various file attributes
- The syntax has two forms, one based on file name, the other for unit number

```
INQUIRE(file='name', options ...)
INQUIRE(unit=iu, options ...)
```


File opening: file properties

- exist : does file exist ? (LOGICAL)
- opened : is file / unit opened ? (LOGICAL)
- form : 'formatted' or 'unformatted' (CHAR)
- access : 'sequential' or 'direct' or 'stream' (CHAR)
- action : 'read', 'write', 'readwrite' (CHAR)
- recl : record length (INTEGER)
- size : file size in bytes (INTEGER)

File opening: file properties

- Find out about existence of a file

```
LOGICAL :: file_exist

INQUIRE (FILE='foo.dat', EXIST=file_exist)
IF (.NOT. file_exist) THEN
  WRITE(*,*) 'The file does not exist'
ELSE
  ! Do something with the file 'foo.dat'
ENDIF
```

File writing and reading

- Writing to and reading from a file is done by giving the corresponding unit number (iu) as a parameter :

```
WRITE(iu,*) str
WRITE(unit=iu, fmt=*) str
READ(iu,*) str
READ(unit=iu, fmt=*) str
```

The star format (*) indicates list-directed output (i.e. programmer does not choose the input/output styles)

- Formats and other options can be used as needed
- If keyword 'unit' used, also 'fmt' keyword must be used ('fmt' is applicable to formatted, text files only)

Formatted vs. unformatted files

- Text or *formatted* files are
  - Human readable
  - Portable i.e. machine independent
- Binary or *unformatted* files are
  - Machine readable only, *not* portable
  - Much *faster to access* than formatted files
  - Suitable for large amount of data due to *reduced file sizes*
  - Internal data representation used for numbers, thus no number conversion, no rounding of errors compared to formatted data

Unformatted I/O

- Write to a sequential binary file

```
REAL rval
CHARACTER(len = 60) string
OPEN(10, file='foo.dat', form='unformatted')
WRITE(10) rval
WRITE(10) string
CLOSE(10)
```

- No FORMAT descriptors allowed
- Reading similarly

```
READ(10) rval
READ(10) string
```


Stream I/O

- A binary file write adds extra record delimiters (hidden from programmer) to the beginning and end of records
- In Fortran 2003 using access method 'stream' avoids this and implements a C-like approach
  - One should move to use stream I/O
- Create a stream (binary) file

```
REAL dbheader(20), dbdata(300)
OPEN(10,file='my_database.dat', access='stream')
WRITE(10) dbheader
WRITE(10) dbdata
CLOSE(10)
```

- Reading similarly

Summary

- Input/Output formatting
- Files: communication between a program and the outside world
  - Opening and closing a file
  - Data reading & writing
- Use unformatted (binary) I/O for all except text files
- Stream I/O
- Internal I/O

PART V: DERIVED DATA TYPES

Outline

- Recalling Fortran built-in data types
- Rationale behind derived data types
- Data type declaration and visibility with examples

Fortran built-in types

- Standard Fortran already supports a wide variety of fundamental data types to represent integers, floating point numbers (real), truth values (logical) and variable length character strings
- In addition each of these built-in types may have declared as multi-dimensional array
- Furthermore, reals and integers can be declared to consume less memory in expense of reduced numerical precision through kind parameter (e.g. 8 or 4)

A few words about numerical precision

- The variable representation method (precision) may be declared using the KIND statement

```
! SELECTED_INT_KIND(r)
! SELECTED_REAL_KIND(p)
! SELECTED_REAL_KIND(p,r)

INTEGER, PARAMETER :: short=SELECTED_INT_KIND(4)
INTEGER, PARAMETER :: double=SELECTED_REAL_KIND(12,100)
INTEGER (KIND=short) :: index
REAL (KIND=double) :: x, y, z
COMPLEX (KIND=double) :: c

x=1.0_double; y=2.0_double * ACOS(x)
```

Integer between $-10^r < n < 10^r$

Real number accurate to *p* decimals

A real number between $-10^{100} < x < 10^{100}$, accurate to 12 decimals

Numerical precision

```
PROGRAM Precision_Test
IMPLICIT NONE

INTEGER, PARAMETER :: sp = SELECTED_REAL_KIND(6,30), &
                      dp = SELECTED_REAL_KIND(10,200)
REAL(KIND=sp) :: a
REAL(KIND=dp) :: b
WRITE(*,*) sp, dp, KIND(1.0), KIND(1.0_dp)
WRITE(*,*) KIND(a), HUGE(a), TINY(a), RANGE(a), PRECISION(a)
WRITE(*,*) KIND(b), HUGE(b), TINY(b), RANGE(b), PRECISION(b)

END PROGRAM Precision_Test

Output:
 4  8  4  8
 4 3.4028235E+38 1.1754944E-38 37 6
 8 1.797693134862316E+308 2.225073858507201E-308 307 15
```


F2008 standard module ISO_FORTRAN_ENV

```
MODULE prec
    USE ISO_FORTRAN_ENV, ONLY: INT32, INT64, REAL32, REAL64
    IMPLICIT NONE
    PRIVATE
    INTERGER, PARAMETER :: i4 = INT32 &
                           i8 = INT64 &
                           r4 = REAL32 &
                           r8 = REAL64
    PUBLIC :: i4, i8, r4, r8
END MODULE prec
```


Numerical precision

Other intrinsic functions related to numerical precision

`KIND(A)`	Returns the kind of the supplied argument
`TINY(A)`	The smallest positive number
`HUGE(A)`	The largest positive number
`EPSILON(A)`	The smallest positive number added to 1.0 returns a number just greater than 1.0
`PRECISION(A)`	Decimal precision
`DIGITS(A)`	Number of significant digits
`RANGE(A)`	Decimal exponent
`MAXEXPONENT(A)`	Largest exponent (of the `kind(A)`)
`MINEXPONENT(A)`	Smallest exponent (of the `kind(A)`)

What is derived data type ?

- Derived data type is a data structure composed of built-in data types and possibly other derived data types
  - Equivalent to structs in C programming language
- Derived type is defined in the variable declaration section of programming unit
  - Not visible to other programming units
    - Unless defined in a module and used via USE clause, which is most often the preferred way ☺

Derived data types – rationale

- Properly constructed data types make the program more readable, lead to clean interfaces and less errors
- Variables used in the same context should be grouped together, using modules and derived data types
- Please do not forget computationally efficient data layout when diving into object oriented programmin in Fortran (or any other language)

Data type declaration

- Type declaration

```
TYPE playertype
  CHARACTER (LEN=30) :: name
  INTEGER :: number, goals, assists
END TYPE playertype
```

- Declaring variables using a derived data type

```
TYPE(playertype) :: ville, pekka
TYPE(playertype), DIMENSION(30) :: players
```


Accessing data types

- Initialization

```
ville%name = ‘Ville Nieminen'
ville%number = 17
ville%goals = 10
ville%assists = 8
```

- Alternatively

```
ville = playertype(‘Ville Nieminen', 17, 10, 8)
```

- Vector of derived data type: element-wise addressing

```
players(1)%name = 'Pekka Saravo'
players(1)%number = 6
players(1)%goals = 2
players(1)%assists = 4
```


Nested derived types

- Declaration of a derived type using another derived type

```
TYPE hockeyteam
  CHARACTER (LEN=80) :: name
  TYPE(playertype) :: players(30)
  TYPE(goalietype) :: goalies(3)
END TYPE hockeyteam
```

- Declaring variables:

```
TYPE(hockeyteam) :: tappara, ilves, karpat
```

- Initialization / access example:

```
tappara%name = 'Tappara'
tappara%players(2)%name = 'Ville Nieminen'
tappara%players(2)%number = 17
```


Visibility of derived data types

- When declared in the same programming unit derived data types are visible to that unit only
  - and subunits under CONTAINS statement
- When declared in a module unit, a derived data type can be accessed outside the module through USE-statement

Summary

- Derived data types enables grouping of data to form logical objects
- A Fortran program becomes more readable and modular with sensible use of derived data types
- Handling of linked lists or binary trees becomes more manageable with use of data structures
- Enables the use of object oriented programming concepts

PART VI: OTHER HANDY FORTRAN 95/2003 FEATURES

Outline

- Interface definition & Generic procedures
- Special procedure attributes & optional procedure attributes
- Command-line arguments

Interface definition

- It is a good practice to define the interfaces for external procedures
  - Enables compilation time error checking
- The INTERFACE block can be also used for defining sc. generic procerudes

```
SUBROUTINE nag_rand(table)
 INTERFACE
   SUBROUTINE g05faf(a, b, n, x)
     REAL, INTENT(IN) :: a
     REAL, INTENT(IN) :: b
     INTEGER, INTENT(IN) :: n
     REAL, INTENT(OUT), DIMENSION(n) :: x
   END SUBROUTINE g05faf
 END INTERFACE
 REAL, DIMENSION(:), INTENT(OUT) :: table
 CALL g05faf(-1.0, 1.0, SIZE(table), table)
END SUBROUTINE nag_rand
```

Defining an interface for the g05faf subroutine of the NAG library (generates a set of random numbers)

Generic procedures

- Procedures which perform similar actions but for different data types can be defined as **generic procedures**
- Procedures are called using the **generic name** and compiler uses the correct procedure based on the argument number, type and dimensions
  - Compare with ”overloading” in C++
- Generic name is defined in INTERFACE section

Generic procedures example

```
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
  SUBROUTINE swap_char(a, b)
    CHARACTER, INTENT(INOUT) :: a, b
    CHARACTER :: temp
    temp = a; a = b; b = temp
  END SUBROUTINE
END MODULE swapmod
```

```
PROGRAM switch
 USE swapmod
 IMPLICIT NONE
 CHARACTER :: n, s
 REAL :: x, y
 n = 'J'
 s = 'S'
 x=10
 y=20
 PRINT *, x, y
 PRINT *, n, s
 CALL swap(n,s)
 CALL swap(x,y)
 PRINT *, x, y
 PRINT *, n, s
END PROGRAM
```


Special attributes for procedures: RECURSIVE

- Recursion means calling a procedure within itself
- Triggered via RECURSIVE keyword

```
RECURSIVE FUNCTION factorial(n) RESULT(fac)
  INTEGER, INTENT(IN) :: n
  INTEGER :: fac
  IF (n == 0) THEN
    fac = 1
  ELSE
    fac = n * factorial(n - 1)
  END IF
END FUNCTION factorial
```


Special attributes for procedures: PURE

- PURE keyword indicates that the function is free of side effects
  - Such as a change in value of an input argument or global variable
- Intrinsic functions are always pure
- No (external) I/O is allowed in PURE procedures
- Pure procedure must specify intents of its all arguments
- The motivation is **efficiency**: Enables more aggressive compiler optimization and parallelization with e.g. OpenMP

Special attributes for procedures: ELEMENTAL

- The ELEMENTAL attribute allows for declaring procedures that operate element-by-element
- The argument can be a scalar or an array of any shape
- This is another way for defining more general procedures

```
ELEMENTAL REAL FUNCTION f(x, y)
  REAL, INTENT(IN) :: x, y
  f = SQRT(x**2 + y**2)
END FUNCTION f

...
REAL, DIMENSION(n,n) :: a, b, c
REAL, DIMENSION(n) :: t, u, v
...
c = f(a, b)
v = f(t, u)
```

SAVEd variables

- By default objects in procedures are dynamically allocated upon invocation
- Only saved variables keep their value from one call to the next
  - SAVE attribute
    ```
    REAL, SAVE :: a
    ```
  - Variables assigned with a value upon declaration are equal to SAVE attribute (C programmers should note this!)
    ```
    REAL :: a = 1.0
    ```


Optional procedure arguments

- Procedure arguments can be defined as *optional,* i.e., using some predefined value for arguments not provided

Counting an average of a set of real numbers – optionally numbers outside [low,up] can be omitted from the average. The function can be called with either 1, 2 or 3 argumets, but the set of numbers has to be provided.

```
REAL FUNCTION average(x, low, up)
 IMPLICIT NONE
 REAL, DIMENSION(:), INTENT(IN) :: x
 REAL, INTENT(IN), OPTIONAL :: low, up
 REAL :: a, b
 INTEGER :: i, icount
 a = -HUGE(a)
 b = HUGE(b)
 IF (PRESENT(low)) a = low
 IF (PRESENT(up)) b = up
 average = 0.0
 icount = 0
 DO i = 1, SIZE(x)
   IF (x(i) >= a .AND. x(i) <= b) THEN
     average = average + x(i)
     icount = icount + 1
   END IF
 END DO
 average = average/icount
END FUNCTION average
```


Command line input

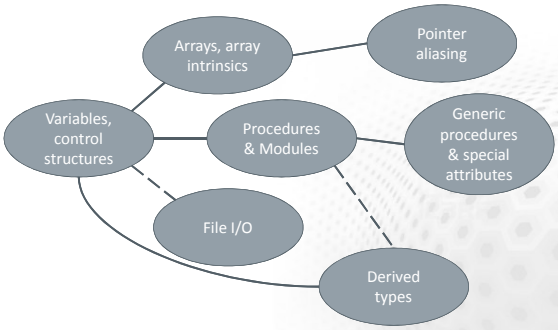
- In many cases, it is convenient to give parameters for the program directly during program launch
  - Instead of using a parser, reading from an input file etc.
- Fortran 2003 provides a way for this
  - `COMMAND_ARGUMENT_COUNT()` : compute the number of user-provided arguments
  - `GET_COMMAND_ARGUMENT(integer i, character arg(i))` extract the argument from position i
  - You will need internal I/O to convert e.g. integer-valued arguments into values of integer variables

Command line input

- Example: reading in two integer values from the command line
- The (full) program should be launched as (e.g.)
  ```
  % ./a.out 100 100
  ```

```
subroutine read_command_line(height, width)
  integer, intent(out) :: height, width
  character(len=10) :: args(2)
  integer :: n_args, i
  n_args = command_argument_count()
  if (n_args /= 2) then
    write(*,*) ' Usage : ./exe height width '
    call abort()
  end if
  do i = 1, 2
    call get_command_argument(i, args(i))
    args(i) = trim(adjustl(args(i)))
  end do
  read(args(1), *) height
  read(args(2), *) width
end subroutine read_command_line
```


Fortran 95/2003 crash course summary



Towards Fortran 2008

- The Fortran 2008 standard features e.g. the following capabilities on top of Fortran 2003
  - Submodules
  - **Coarray Fortran** – a parallel execution model
  - The DO CONCURRENT construct
  - The CONTIGUOUS attribute
  - The BLOCK construct

Towards Fortran 2008

"Coarrays were designed to answer the question: '*What is the smallest change required to convert Fortran into a robust and efficient parallel language?*'

The answer: a simple syntactic extension.

It looks and feels like Fortran and requires Fortran programmers to learn only a few new rules."

John Reid,
ISO Fortran Convener

Towards Fortran 2008

See **http://fortranwiki.org/fortran/show/Fortran+2008+status**

Fortran Wiki
Fortran 2008 status

Home Page | All Pages | Recently Revised | Authors | Feeds | Export | Search

Compiler Support for the Fortran 2008 Standard

Fortran 2008 features	Absoft	Cray	g95	GNU	HP	IBM	Intel	NAG	Oracle	PathScale	PGI
Compiler Version Number				4.6			12.0	5.2		4.0	
Submodules	N	Y	?	N	N	N	N	N	N	N	N
Coarrays	N	Y	P	P (200)	N	N	Y	N	N	N	N
Performance enhancements	**Absoft**	**Cray**	**g95**	**GNU**	**HP**	**IBM**	**Intel**	**NAG**	**Oracle**	**PathScale**	**PGI**
do concurrent	N	N	?	P	N	N	Y	N	N	N	N
Contiguous attribute	N	Y	?	Y	N	N	Y	N	N	N	N
Simply contiguous arrays	N	Y	?	Y	N	N	Y	N	N	N	N
Data Declaration	**Absoft**	**Cray**	**g95**	**GNU**	**HP**	**IBM**	**Intel**	**NAG**	**Oracle**	**PathScale**	**PGI**
Maximum rank	N	N	?	N	N	Y	Y	N	N	N	N
Long integers	Y (100)	Y	?	Y	N	Y	Y (100)	Y	Y	Y	Y
Allocatable components of											

Don’t stop here!

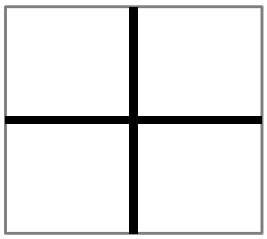
- See for more PRACE training opportunities at **www.prace-ri.eu/training**
- CSC’s course calendar: **www.csc.fi/courses**

EXERCISE ASSIGNMENTS

Fortran 95/2003 exercises

1. Playing around with control structures and arrays

a. Declare an integer array of dimensions 100 by 100 and initialize it to zero otherwise but where the first index is equal to 50 or the second index is equal to 50 the array elements get a value of 1.



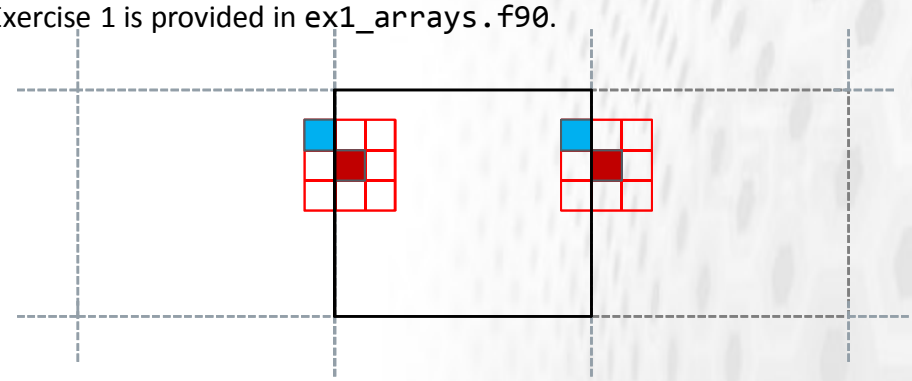
Value 1 on the black rows, value 0 elsewhere

b. Then initialize another array with the same dimensions, but its values are computed from the first array such that

- The cells with exactly two elements equal to 1 in the first array get the same value in the new array as in the first array.
- Any cell with exactly three neighbors with value 1 gets the value 1 in the new array.
- Otherwise, i.e. new array cells with less than two or more than three neighbors with value 1 in the old array get a value 0.

At this stage, just run the indices from 2 to 99, i.e. not referencing to the boundaries at all.

c. Modify the program from exercise 1b such that the array becomes periodic – that is, the boundaries depend on the cells on the other side of the array. The solutions for all three items of Exercise 1 is provided in `ex1_arrays.f90`.



2. Getting acquainted with procedures

a. Modify the program such that you can produce new arrays iteratively, i.e. taking the array from the previous iteration and obtaining a new array by applying the same rules for it.

b. Modify the exercise 2a so that it uses a function or a subroutine to produce the new array.

c. Change the initialization of the board such that the board starts from a random configuration. The intrinsic procedure is called `RANDOM_NUMBER`. Wrap also this board initialization into its own procedure. The solutions for all three items of Exercise 2 is provided in `ex2_procedures.f90`.

Fortran 95/2003 exercises

3. Game of Life

The *Game of Life* (GoL) is a cellular automaton devised by John Horton Conway in 1970, see http://en.wikipedia.org/wiki/Conway's_Game_of_Life.

You can compile a reference executable (since the file `ex3_gol.f90` will contain the solution) with

```
% f90 -o gol gol_io.f90 ex3_gol.f90
```

Run the program of e.g. 200x200 board for 100 iterations. With the command `xview` or `eog` you can view the images (.pbm) and see how the automaton looks like after those. You can also animate the board development by first using convert as

```
% convert -delay 40 -geometry 512x512 life_*.pbm life.gif
```

(on a single line) and then displaying the animation with

```
% animate life.gif
```

a. See the file `ex3_gol0.f90`, get acquainted with the program and complete the missing parts of the code (search: "TODO"). Refer back to assignments 1 and 2.

b. Experiment, how the game evolves if you replace the starting pattern ('plus', c.f. exercise 1a) to a random one (c.f. Exercise 2c).

c. Modify the GoL program such that the board is manipulated through a derived datatype `GoL_board`, which contains the actual board, its dimensions as well as how many iterations it has gone through. No solution has been prepared.

d. Now we will examine the I/O module of the Game of Life program. It visualizes the board in the netpbm image format, see http://en.wikipedia.org/wiki/Netpbm_format. Implement the writing of the board as pbm images - or in some other image format if you want to go your own way. Consult the `gol_io.f90` when in trouble. *Shortcut*: study the **draw** subroutine in `gol_io.f90` and make sure you understand the piece of code.

e. Modify the program such that the user input is read directly from the command line instead of parsering, i.e. the program is launched as ./gol (# iterations) (board height) (board width), for example

```
% ./gol 200 100 100
```

The answer is provided in gol_io.f90.

More bonus exercises

Fortran Quiz

a. Are the following Fortran statements written correctly?

```
character_string = ’Awake in the morning,
    & asleep in the evening.’
x = 0.4-6
answer = ’True & false’
low-limit = 0.0005E10
y = E6
```

b. Are the following declarations legimate in Fortran?

```
DOUBLE :: x
CHARACTER(LEN=*), PARAMETER :: "Name"
REAL :: pi = 22/7
REAL :: x = 2., y = -3
REAL :: pii = 22.0/7.0
REAL x = 1.0
```

c. What are the iteration counts of the following DO loops, the values of loop variable i inside the loop, and the value of the loop variable after the DO construct?

```
DO i = 1, 5
DO i = 5, 0, -1
DO i = 10, 1, -2
DO i = 0, 30, 7
DO i = 3, 2, 1
```

Derived types

a. Declare the derived type which can save the birth date in the form:
21 01 1990
This derived type thus contains three integers, which have different KIND values: SELECTED_INT_KIND(2) and SELECTED_INT_KIND(4).

b. Add the field the for a name to the derived type. Write a function, which returns the name and date in a character string in the following form
Charlie Brown (01.01.1999)

Recursion

Write a recursive function, which calculates ”Tribonacci numbers”:

$$\begin{cases} x_1 = 1, \ x_2 = 1, \ x_3 = 1, \\ x_n = x_{n-1} + x_{n-2} + x_{n-3}. \end{cases}$$

Calculate x_{12}. Carry out the computation also using a loop structure.