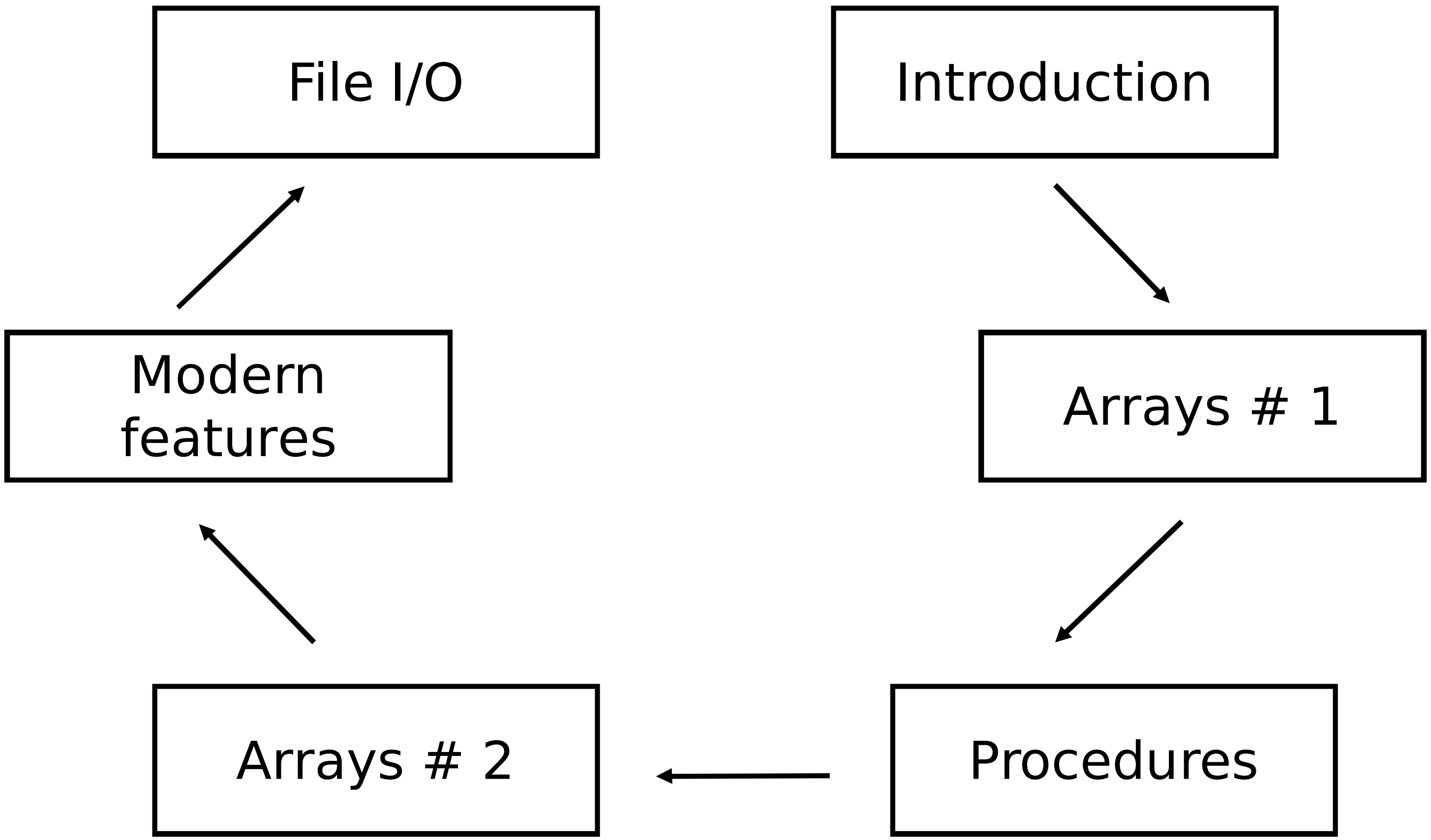
Fortran 95 crash course overview



Part I: Getting started with Fortran 95

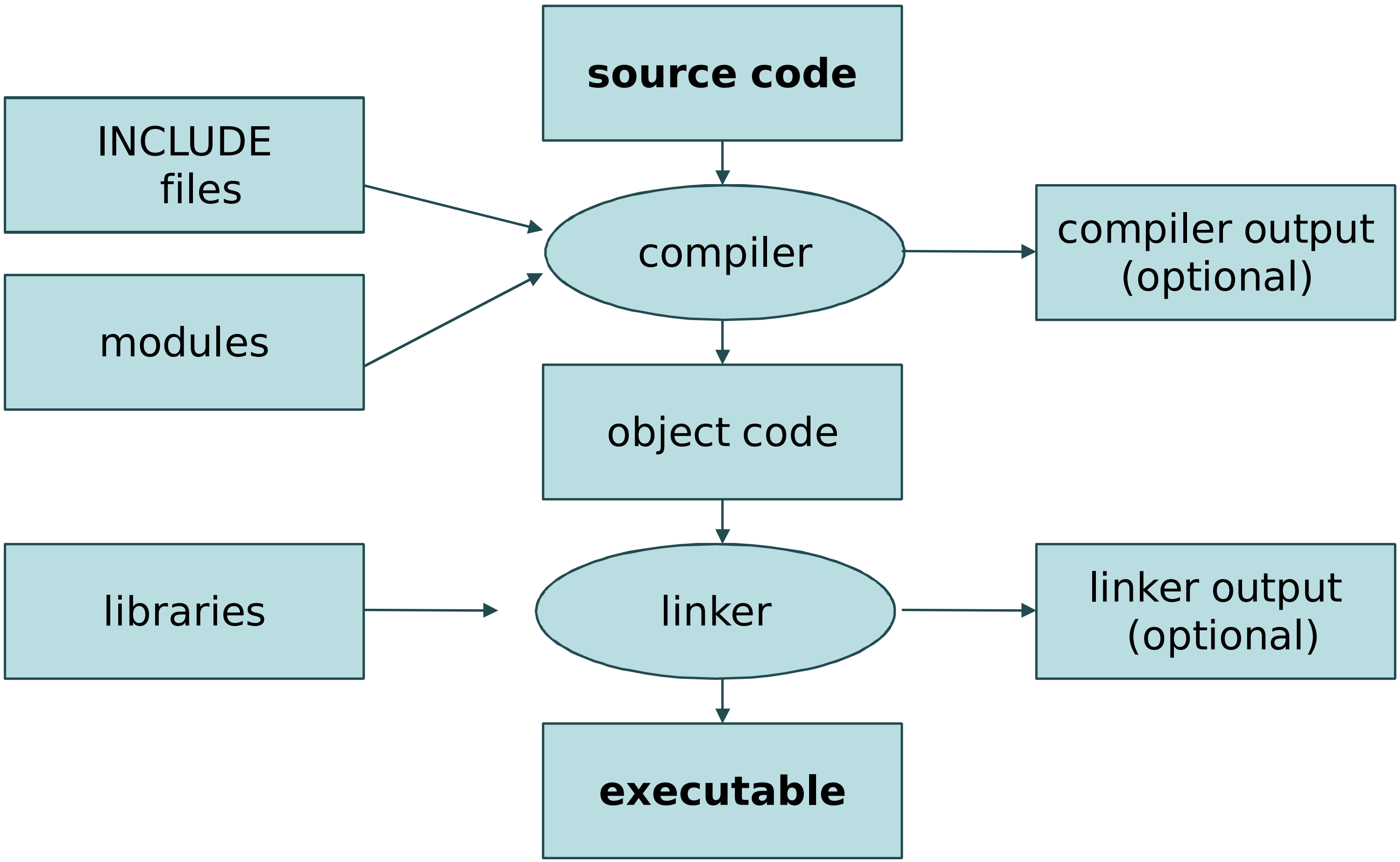
Outline

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

Why learn Fortran 95?

- Well suited for numerical computations
- Fast code (also compilers can optimize well)
- Handy array data types
- Clarity of code
- Portability of code
- Optimized numerical libraries available

Compiling and linking



Look & Feel

```
PROGRAM squarerootexample
! Comments start with an exclamation point.
! Some exponentiation and square root computations.
! 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 !f95 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
! SQRT(y), Return the square root of the argument y
WRITE (*,*) 'computed value of SQRT(x**2 + 1):', SQRT(y)

END PROGRAM squarerootexample
```

Variables

```
IMPLICIT NONE

INTEGER :: n0
INTEGER :: n1=0

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

COMPLEX :: c
COMPLEX :: imag_unit=(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

They can also be given a value at declaration

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

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

Assignment statements

```
PROGRAM numbers
  IMPLICIT NONE
  INTEGER :: i
  REAL :: r
  COMPLEX :: c, cc

  i = 7.3                          !same as i = INT(7.3)
  r = (1.618034, 0.618034)         !same as r = REAL((1.618034, 0.618034))
  c = 2.7182818                    !same as c = CMPLX(2.7182818)
  cc = r*(1,1)                     !at first the variable r is changed
  CMPLX(r)

  WRITE (*,*) i, r, c, cc
END PROGRAM
```

Taking the real part of a complex number

Printout statement

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

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

How can I convert numbers to character strings and vice versa?
Look at the slide “INTERNAL I/O” at the last part.

Source code remarks

- A variable name can be no longer than 31 characters (only letters, digits or underscore, must start with a letter)
- Maximum row length may be 132 characters
- There may be 39 continuation lines, if a line is ended with ampersand, &, it will continued on the next line.
- No distinction between lower and uppercase character
- 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
```

Arrays

```
! 1-dimensional character array, not initialized at declaration
INTEGER, PARAMETER :: n_entries = 43
CHARACTER (LEN=30), DIMENSION(n_entries) :: names

! 1-dimensional real array, not initialized
REAL :: marks(n_entries)

! 3-element 1D integer array, lower and upper bound defined,
! initialized
INTEGER, DIMENSION(-1:1) :: x = (/0, 1, 2/)

! Assigning values
names(1)= 'George'
marks(1)= 10.0
names(2)= 'John'
marks(2)= 9.9
...
names(43)= 'Bill'
marks(43)= 4.1
```

This is an alternative to the DIMENSION attribute

By default, the indexing starts from 1

Control structures

- IF THEN ELSE (branching)
- DO (looping)
- SELECT CASE (selecting)

```
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
```

Reading in user input from keyboard

Control structures

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
```

DO WHILE construct (condition controlled loop)

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

Control structures

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 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 :: isprimenumber
...
SELECT CASE (i)
    CASE (2,3,5,7)              ! variables are not allowed on the list
       isprimenumber = .TRUE.
    CASE (1,4,6,8:10)           ! case value range, form low:high
       isprimenumber = .FALSE.
    CASE DEFAULT                ! DEFAULT-branch
       isprimenumber = testprinumber(i)  ! function call
END SELECT
...
```

Control structures example

```
PROGRAM gcd
! Computes the greatest common divisor, Euclidean algorithm
  IMPLICIT NONE
  INTEGER, PARAMETER :: long = SELECTED_INT_KIND(9)
  INTEGER (KIND=long) :: m, n, t
  WRITE(*,*)' Give positive integers m and n :'
  READ(*,*) m, n
  WRITE(*,*)'m:', m,' n:', n
  positivecheck: 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 positivecheck
END PROGRAM gcd
```

These are *tags* that can be given to control structures and used in conjunction with e.g. exit and cycle

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

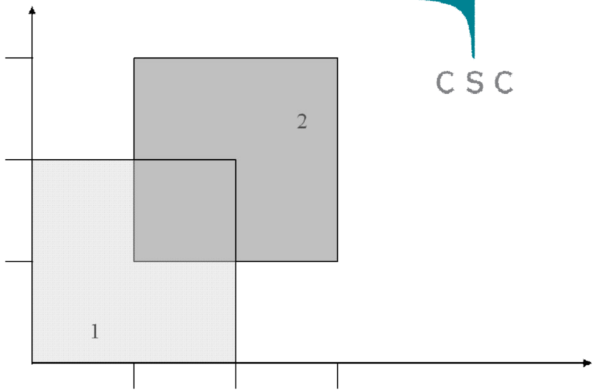
```
.LT. or <     !less than
.LE. or <=    !less than or equal to
.EQ. or ==    !equal to
.NE. or /=    !not equal to
.GT. or >     !greater than
.GE. or >=    !greater than or equal to
```

Logical operators

```
.NOT.      !logical negation               precedence: first
.AND.      !logical conjunction            precedence: second
.OR.       !logical inclusive disjunction precedence: third
.EQV.      !logical equivalence            precedence: fourth
.NEQV.     !logical nonequivalence         precedence: fourth
```

Operators example

Probably don't need!




```
PROGRAM placetest
! test logical and relational operators
IMPLICIT NONE
LOGICAL :: square1, square2
REAL :: x,y
WRITE(*,*)'Give 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                 !both are .TRUE.
    WRITE(*,*) 'Point within both squares'
ELSE IF (square1) THEN                          !just square1 is .TRUE.
    WRITE(*,*) 'Point in square 1'
ELSE IF (square2) THEN                          !just square2 is .TRUE.
    WRITE(*,*) 'Point in square 2'
ELSE                                            !both are .FALSE.
    WRITE(*,*) 'Point outside'
END IF
END PROGRAM
```

Numerical precision

Can probably just use
real
real(kind=4)
real(kind=8)

- 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 standard functions

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

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

Real number, accurate to *p* decimals

Real number between $-10^r<x<10^r$, accurate to *p* decimals

```
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)
```

Numerical precision

```
PROGRAM test_precision
 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
```

Output:

```
 4  8  4  8
 4 3.4028235E+38 1.1754944E-38 37 6
 8 1.79769313486231 6E+308 2.22507385850720 1E-308 307 15
```

Numerical precision

Other intrinsic functions related to numerical precision

KIND(p)	Returns the kind of the supplied argument
TINY(a)	The smallest positive number
HUGE(a)	The largest positive number
EPSILON(a)	The least positive number that added to 1 returns a number that is greater than 1
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

Part II: Fortran arrays

Outline

- Significance of arrays
- Array declaration and syntax
- Array initialization
- Array sections

Significance of arrays

- Arrays enable a natural way to access vector and/or matrix data during computation
- Fortran language is a very versatile in handling especially multi-dimensional arrays (unlike C or some other languages)

Array declaration & syntax

- Arrays are declared in a pretty much similar fashion to scalar variables
- They all refer to a particular data type but they all have one or more *dimensions* specified in the variable declaration
  - Fortran supports up to 7 dimensional arrays

Array declaration & syntax

```
INTEGER, PARAMETER :: M = 100, N = 500
INTEGER :: idx(M)
REAL(kind = 4) :: vector(0:N-1)
REAL(kind = 8) :: matrix(M, N)
CHARACTER (len = 80) :: screen ( 24)
TYPE(my_own_type) :: object ( 10 )

! or

INTEGER, DIMENSION(1:M) :: idx
REAL(kind = 4), DIMENSION(0:N-1) :: vector
REAL(kind = 8), DIMENSION(1:M, N) :: matrix
CHARACTER(len=80), dimension(24) :: screen
TYPE(my_own_type), Dimension ( 1:10) :: object
```

Array declaration & syntax

- In older Fortran, arrays were traditionally accessed element-by-element basis:

```
INTEGER, PARAMETER :: M = 4,  N = 5
REAL (kind = 8) :: A(M,N) , x(N), y(M)
INTEGER :: I , J

do I=1,M ; y(I) = 0 ; end do

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
```

Note:
Fortran is COLUMN-MAJOR in memory i.e

[a_row,col] =
a11 a12 a13
a21 a22 a23
a31 a32 a33

In memory
addr 0 = a11
addr 1 = a21
addr 2 = a31
addr 4 = a12

Array declaration & syntax

- Already the Fortran 90 standard from 1990's introduced way of accessing several elements in one go, hence the *array syntax*
- The array syntax potentially improves readability of the user code
- It may also give the Fortran compiler a chance for better performance optimization

Array declaration & syntax

- Array syntax allows for less explicit DO loops

```
INTEGER, PARAMETER :: M = 4,  N = 5
REAL (kind = 8) :: A(M,N) , x(N), y(M)
INTEGER :: I , J

y(:) = 0

OUTER_LOOP : do J = 1, N
    INNER_LOOP : do  I = 1, M
        y(:) = y(:)  +  A(: , J) * x(J)
    end do INNER_LOOP
end do OUTER_LOOP
```

Array initialization

- To make a program meaningful, we need to feed its variables with some values
- Arrays can be initialized element-by-element, copied from another array, or by using single line data initialization statements
- More advanced initialization involves use of `FORALL` and `WHERE` statements, or use of `RESHAPE` intrinsic function

Array initialization

- Element-by-element initialization

```
do J = 1, N
    idx ( J ) = J
    vector ( J ) = 0
end do
```

- Initialization by copying from another array

```
REAL(kind=8) :: to(100,100),  from(0:199, 0:199)

to (1:100, 1:100) = from (0:199:2, 0:199:2)
```

Every 2nd

Array initialization

- Using *array construction* and *implied DO* :

```
INTEGER, parameter :: FIXED(2:4) = (/  20, 30, 40 /)

INTEGER :: idx(0:10)
DATA  idx / 0, 1, 2, 3, 7 * 0 /


! or

idx (0 : 10) = (/ 0, (i, i = 1, 3), (0, i = 4,10) /)
```

Array sections

- With Fortran array syntax we can access a part of an array in a pretty intuitive way
- Array sections are perhaps the very reason for Fortran usability in scientific computing

```
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
```

Array sections

- Sections enable us to refer to (say) a sub-block of a matrix, 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

- Be aware of: 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)

! but an error if

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

Array sections

- Also array sections – not necessarily full arrays – can be passed into a procedure:

```
INTEGER :: Array (10, 20)

CALL  SUB ( Array )
CALL  SUB ( Array(5:10, 10:20) )
CALL  SUB ( Array(1:10:2, 1:1) )
CALL  SUB ( Array(1:4, 1:) )
CALL  SUB ( Array(:10, :) )
```

Arrays Sections

- Be aware that an array section is usually *copied* into a hidden temporary array upon calling a procedure and copied back to the array section upon return
- This may have some unwanted *side-effects* (like array overwrite with incorrect values) when using *shared memory based parallel processing*, such as OpenMP

Summary

- Use of arrays makes Fortran language a very versatile vehicle for computationally intensive program development
- Using array syntax, vectors and matrices can be initialized and used in a very intuitive way
- Array sections increase code readability and usually reduce chances of mistakes

Part III: Procedures

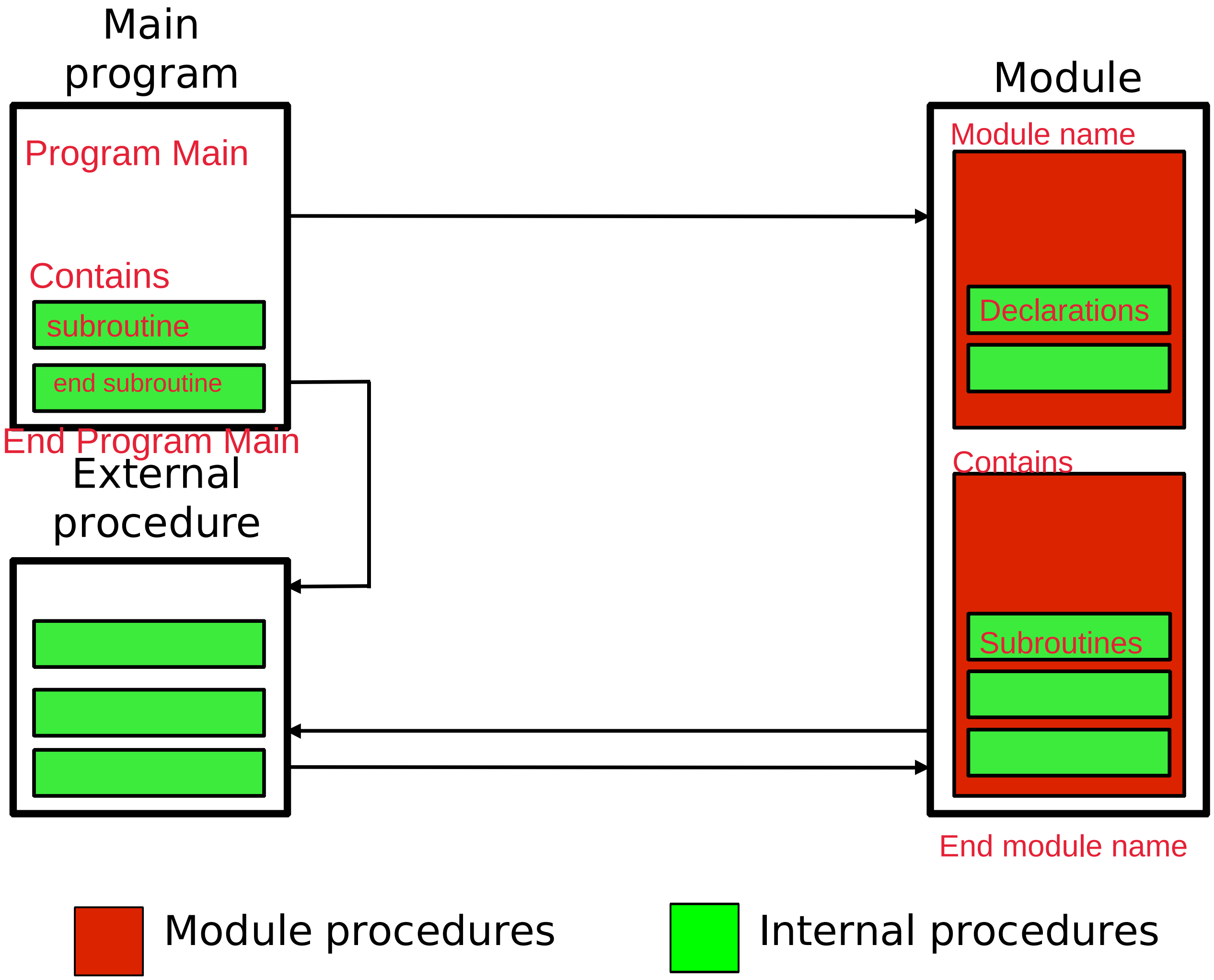
Outline

- Procedures
- Procedure types
- Arguments
- Miscellaneous remarks

Structured programming

- Structured programming based on functions, subroutines and modules
  - testing and debugging separately
  - recycling of code
  - improved readability
  - re-occurring tasks

Program units



What are procedures?

- By procedures we mean subroutines and functions
  - Subroutines exchange data with arguments only
  - Functions return value according to its declared data type

Procedure types

- Internal, external and module procedures
  - Internal: within program structure
  - External: independently declared, may be other language
  - Module procedure: defined in module (see later lecture on modules)
- Internal and module procedures provide a defined interface, compiler uses this to check arguments

Internal procedures

- Each program unit may contain internal procedures
- Declared at the end of program unit after the `CONTAINS` statement
- Inherits variables and objects from the program unit

External procedures

- Declared in separate program units, included with the EXTERNAL keyword
- Modules are much easier
- Library routines are external procedures

Declarations

Function

```
[TYPE] FUNCTION func(ARGS)
[RESULT(arg)]

[declarations]

[statements]

END SUBROUTINE func
```

Subroutine

```
SUBROUTINE sub(ARGS)

[declarations]

[statements]

END SUBROUTINE sub
```

Function call

```
res = func(ARGS)
```

Subroutine call

```
CALL sub(ARGS)
```

Declarations

These two do exactly the same thing.

```
INTEGER FUNCTION test(s)
IMPLICIT NONE
INTEGER::s
test=10*s
END FUNCTION test

PROGRAM dosomething
...
result=test(s)
...
```

```
SUBROUTINE test(s,test)
IMPLICIT NONE
INTEGER::s,test
test=10*s
END FUNCTION test

PROGRAM dosomething
...
call(s,result)
...
```

Procedure arguments

- Call by reference: any change to arguments value changes the actual argument
- Compiler checks arguments if the interface of procedure is known at compilation time
  - internal and module procedures
- Behavior of the arguments can be controlled with the INTENT keyword

INTENT keyword

- Declares how formal argument is intended for transferring a value
  - in
  - out
  - inout (default)
- Compiler uses this for error checking and optimization

```
SUBROUTINE func(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 func
```

Passing array arguments

- Three ways to pass arrays to procedures
  - Assumed shape array
    REAL, DIMENSION(:,:) :: matrix
  - Explicit shape array
    REAL, DIMENSION(size1,size2) :: matrix
  - Assumed size array
    REAL, DIMENSION(low:up,*) :: matrix OLD! Don't use!

Saving local variables

- By default objects in procedures are dynamically allocated at 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`

Recursive procedures

Probably don't need this!

- Recursion means calling a procedure within itself
- RECURSIVE keyword for the compiler

```
RECURSIVE FUNCTION recurse(n) RESULT(test)
  IMPLICIT NONE
  INTEGER, INTENT(IN) ::n
  INTEGER :: test
  IF (n<1) then
    test=n
    WRITE(*,*) test
  ELSE
    test=recurse(n-1)
  END IF
END SUBROUTINE recurse
```

Summary

- Procedural programming makes the code more readable and easier to modify
  - Procedures encapsulate some piece of work that makes sense
- Fortran uses *functions* and *subroutines*
- Procedure arguments will be changed upon calling the procedure
  - Can be controlled with the INTENT keyword
  - Arrays can easily be procedure arguments

Part IV: More about Fortran arrays

Outline

- Dynamic memory allocation
- Array intrinsic functions
- Pointers to arrays

Dynamic memory allocation

You probably only need STATIC for now!

- Sizing of arrays may be *static* or *dynamic*
- For small array sizes a static dimensioning is usually not a problem
- For large arrays dynamic memory allocation is maybe the only option – or otherwise *the program may not fit into the memory* – and will not be able to run
- Effective runtime sizing of data arrays

Dynamic memory allocation

- Fortran provides two different mechanisms to dynamically allocate memory for arrays:
  1. Variable declaration has an `ALLOCATABLE` (or a `POINTER`) attribute, and memory is allocated through `ALLOCATE` statements
  2. A variable, which is declared in the procedure with size information coming from the argument list or a module, is an *automatic array*

Dynamic memory allocation

- An example of using ALLOCATE :

```
INTEGER :: M, N, alloc_stat
INTEGER, ALLOCATABLE :: idx ( : )
REAL(kind = 8), ALLOCATABLE :: mat (: , :)

M = 100 ; N = 200
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)
```

Dynamic memory allocation

- Identical example with automatic arrays
- No explicit ALLOCATE/DEALLOCATE

```
SUBROUTINE SUB (M)
 USE some_module, ONLY : N
 INTEGER, INTENT(IN) :: M

 INTEGER :: idx ( 0 : M – 1 )
 REAL(kind = 8) :: mat ( M , N )

END SUBROUTINE SUB
```

M and N come through the arguments

Dynamic memory allocation

- When using the `ALLOCATE` statement, it is always recommended to use `ALLOCATABLE` rather than `POINTER` attribute in dynamic variable declaration
- To avoid unexpected memory growth (”memory leak”), remember to use `DEALLOCATE` for every `ALLOCATE` statement ever used

Array intrinsic functions

Some of these are very useful!

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

Array Intrinsic Functions

VERY HANDY!!!

- Perhaps the most commonly used array functions are the following
  - SIZE, SHAPE, COUNT, SUM
  - ANY, ALL
  - MINVAL / MAXVAL , MINLOC / MAXLOC
  - RESHAPE
  - DOT_PRODUCT, MATMUL, TRANSPOSE

Array Intrinsic Functions

- SIZE (array [, dim]) returns # of elements in the array [, along the specified dimension]
- SHAPE (array) returns an INTEGER vector containing SIZE of array in each dimension
- COUNT (L_array [,dim]) returns count of elements which are .TRUE. in L_array
- SUM (array[, dim][, mask]) : sum of the elements [, along dimension] [, under mask]

Array Intrinsic Functions

Logical variable functions

- ANY (L_array [, dim]) returns a scalar value of .TRUE. if any value in L_array is .TRUE.
- ALL (L_array [, dim]) returns a scalar value of .TRUE. if all values in L_array are .TRUE.

Array Intrinsic Functions

- Some examples

```
INTEGER :: j, IA(4, 2)
IA(:, 1)=(/ (j, j = 1,SIZE(IA,dim=1)) /)
IA(:, 2)=(/ (SIZE(IA,dim=1) + j, j = 1, SIZE(IA,dim=1)) /)

PRINT *, SHAPE(IA)
PRINT *, COUNT(IA > 0), COUNT(IA <= 0, dim = 2)
PRINT *, SUM(IA), SUM(IA, dim=2, mask = IA > 3)

IF (ANY(IA <  0)) PRINT *,'Some IAs less than zero'
IF (ALL(IA >= 0)) PRINT *,'All IAs non-negative'
```

Array Intrinsic Functions

- MINVAL (array [,dim] [, mask]) returns the minimum value in a given array [along specified dimension] [, under mask]
- MAXVAL is the same as MINVAL, but returns the maximum value in a given array
- MINLOC (array [, mask]) returns a vector of location(s) [, under mask], where the minimum value(s) is/are found
- MAXLOC similar to MINLOC, for maximums

Array Intrinsic Functions

- RESHAPE (array, shape) returns a reconstructed array with different shape than in the input array
  - Can be used as a single line statement to initialize an array (in expense of readability)
  - Create for example from an existing N-by-N matrix a 1D-array (vector) of length N x N

Array Intrinsic Functions

- RESHAPE example :

1 2 3 4

5 6 7 8

```
INTEGER :: j, IA(4, 2)
IA(:,1) = ( / (j, j = 1,SIZE(IA,dim=1)) /)
IA(:,2) = (/ (SIZE(IA,dim=1)+j, j = 1, SIZE(IA,dim=1)) /)

! The same with RESHAPE:

IA = RESHAPE ( (/ (j, j = 1, SIZE(IA) /), (/ 4, 2 /) )
```

Array Intrinsic Functions

- Some other array functions manipulate vectors and matrices effectively :
  - `DOT_PRODUCT (a_vec, b_vec)` returns a scalar value – dot product – of two vectors
  - `MATMUL (a_mat, b_mat)` returns a matrix containing matrix multiply of two matrices
  - `TRANSPOSE (a_mat)` returns a transposed matrix of the input matrix

!!!

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 so closely related
- 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)) /)
REAL, DIMENSION(100,100) :: a
FORALL (j=1:100) a(j,j) = b(j)      !processing lower
FORALL (j=2:100) a(j,j-1) = c(j)    !bidiagonal matrix

WHERE (ix == 0) ix = -9999

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

Pointers to arrays

- `POINTER` attribute enables to create array (or even scalar) *alias variables*
  - `POINTER` – if misused – leads to a hard-to-detect programming error

!!!

- 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`

Pointers to arrays

- A POINTER example with 1D-array

```
INTEGER, POINTER :: p_x ( : ) => NULL ( )
INTEGER, TARGET :: x ( 1000 )

p_x => x
p_x => x ( 2 : 300 )
p_x => x ( 1 : 1000 : 5 )

NULLIFY(p_x)
```

Pointers to arrays

- A POINTER example with 2D-array

```
REAL(kind = 8), POINTER :: &
    p_mat ( : , : ) => NULL ( )
REAL(kind = 8), TARGET :: mat ( 100, 200 )

p_mat => mat
p_mat => mat(1:50,1:50)
p_mat => mat(55:70,101:150)
p_mat => mat(10:100:10,10:SIZE(mat,dim=2):5)

NULLIFY(p_mat)
```

Pointers to arrays

- Whether a POINTER points to anything, use ASSOCIATED – function to check :

```
REAL(kind = 8), POINTER :: &
        p_mat ( : , : ) => NULL ( )
REAL(kind = 8), TARGET :: &
        mat ( 100, 200 )

p_mat => mat
IF ( ASSOCIATED (p_mat) ) &
    PRINT *,'Points to something'

NULLIFY(p_mat)
IF ( .not. ASSOCIATED (p_mat) ) &
    PRINT *,'Points to nothing'
```

Summary

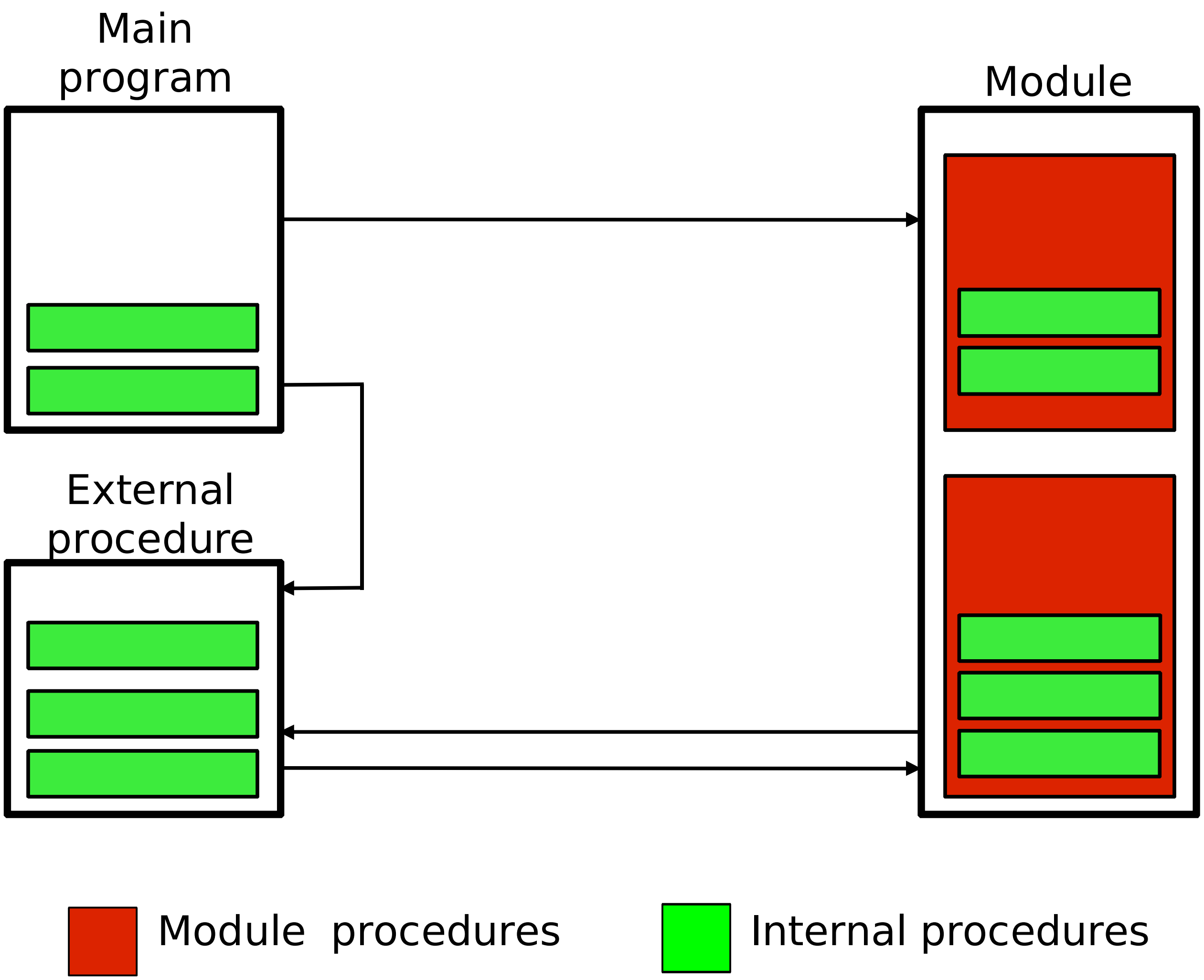
- Dynamic memory allocation enables sizing of arrays according to particular needs
- Array intrinsic functions further simplify coding efforts and improve program code readability when using Fortran arrays
- Pointers offer a versatile alias mechanism to refer into the existing arrays or array sections

Part V: Modern features of Fortran 95

Outline

- Modules in Fortran 95
- Generic procedures
- Derived datatypes

Program units



Modular programming

- Modularity means dividing a program into small minimally dependent modules
- Advantages
  - Constants, variables, data types and procedures can be defined in modules
  - Makes possible to divide program into smaller self contained units

Usefulness of Fortran modules

- Global definitions
- The same procedures and data types available in different program units
- Compile-time error checks
- Hide implementation details (OOP or object oriented programming)
- Group routines and data structures
- Define generic procedures and custom operators

Modules

- Declaration

```
MODULE accuracy
 IMPLICIT NONE
 INTEGER, PARAMETER :: &
  realp = SELECTED_REAL_KIND(12,6)
 INTEGER, PARAMETER :: &
   intp =SELECTED_INT_KIND(4)
END MODULE accuracy

MODULE check
 USE accuracy
 IMPLICIT NONE
 INTEGER(KIND=intp) ::y
 CONTAINS
  FUNCTION check_this(x) RESULT(z)
   INTEGER:: x, z
   z = HUGE(x)
  END FUNCTION
END MODULE check
```

- Usage

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

  - Good habit

```
USE accuracy, ONLY: realp
```

Module procedures

- Procedures defined in modules can be used in any other program unit
- Placing procedures in modules helps compiler to detect programming errors and to optimize the code
- Module procedures are declared after `CONTAINS` statement

Common variables with modules

- In many Fortran 77 codes, sc. common variables are used

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

- The Fortran 95 way is to do it with modules

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

Visibility of objects

- Objects in modules can be PRIVATE or PUBLIC
- Default is PUBLIC, i.e. visible for all program units using the module
- PRIVATE will hide the objects from other program units

```
INTEGER, PRIVATE :: x
INTEGER, PUBLIC  :: y
PRIVATE :: z
```

Generic procedures

Not likely to use this!

- Procedures which perform similar tasks 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 the ”templates” in C++ and ”generics” in Java
- 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
```

Output
```
 JS
   10.00000       20.00000
 SJ
   20.00000       10.00000
```

Derived data types

- Derived data type is a structure of data types which is defined by the programmer
  - Equivalent to structs in C or classes in C++
- Comprises of any data types including other derived types
- Abstract data type includes data type definitions and procedures
- Derived type defined in variable definition section of programming unit
  - Not visible to other programming units
    - Unless defined in modules and used via USE clause

Derived data types

- Type declaration

```
TYPE playertype
  CHARACTER (LEN=30) :: name
  INTEGER :: number
  REAL :: rating
END TYPE playertype
```

- Declaring derived type variables

```
TYPE(playertype) :: john, luiz
TYPE(playertype), DIMENSION(10) :: players
```

- Element addressing

```
players(1)%name = 'Phil'
players(1)%number = 4
players(1)%rating = 5.5
```

Derived data types example

```
MODULE playertype
 IMPLICIT NONE
 TYPE playertype
  CHARACTER (LEN=30) :: name
  INTEGER :: number
  REAL :: rating
 END TYPE playertype
END MODULE playertype

PROGRAM team
 USE playertype
 IMPLICIT NONE
 TYPE(playertype), dimension(10) :: players

 players(1)%name='John'
 players(1)%number=10
 players(1)%rating=5.5
 players(2)=playertype(‘Luiz’,4,9.0)
 print *,players(1)
 print *,players(2)
END PROGRAM team
```

spot the errors!

```
John    1    5.600000
Luiz    4    9.000000
```

Summary

- In Fortran 95, there is a set of features that makes Fortran 95 meet all standards of a modern programming language
  - Modules for modular programming and data encapsulation
  - Generic procedures to operate with templates
  - Derived data types for class-like objects

Part VI: File I/O

Outline

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

File I/O motivation

- File interface with other applications
- Data reading
- Data writing

Basic concepts

- Writing to or reading from a file is basically similar to writing onto a terminal screen or reading from a keyboard
- Differences
  - File must be opened first with OPEN-statement, in which the unit number and (optionally) a 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 closed

read(5,*) same as read(*,*)

write(6,*) same as write(*,*)

Opening & closing a file

- You can also refer to the default output or input unit with an asterisk
  ```
  WRITE(*, ...) ! or READ(*, ...)
  ```
- Note that they are NOT necessarily the same as the unit numbers 5 and 6
- If the file name is omitted in the OPEN, the a file based on unit number will be opened, e.g. for unit=12 →'fort.12'

File opening options

- Investigating the options-flags in
  `OPEN([unit=]iu, file='name' [,options])`
- The options-flags can be one (or a suitable combination) of the following :
  - `status, position, action, form`
  - `access, iostat, err, recl`

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'
- iostat : error indicator, (output) integer
  - Non-zero only upon 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 attributes etc.
- 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

- The options contains one or more (keyword , variable) pairs
- The corresponding variable contains the information that was inquired
  - Depending on context, the variable is either LOGICAL, CHARACTER-string or INTEGER

File opening: file properties

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

File opening: file properties

- Find out about file existence

```
LOGICAL :: exfile
INQUIRE (FILE='foo.dat', EXIST=exfile)
IF (.NOT. exfile) THEN
WRITE(*,*) ‘The file does not exist‘
ELSE
   ...
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
```

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

File writing

- If the file unit (iu) has not been explicitly OPENed, the very first WRITE on that unit will trigger an implicit OPEN
- In most UNIX systems this means opening a file named as ’fort.<iu>’, where <iu> is the unit number in concern
- Star (‘*’) format indicates list directed output (a programmer do not choose the output style)

Output formatting

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

Output formatting

Lots of details to prettifying!

w=number of characters to use, d=number of digits to the right of decimal point, m=minimum number of characters to be used, e=number of digits in the exponent. Variables: Integer :: J, Real :: R, Character :: C, Logical :: T

Data type	**Basic data edit descriptors**	**Examples**
Integer	Iw, Iw.m	`WRITE(*,'(I5)') J` `WRITE(*,'(I5.3)') J`
Real (decimal and exponential forms)	Fw.d Ew.d, Ew.dEe	`WRITE(*,'(F7.4)') R` `WRITE(*,'(E12.3E4)') R`
Character	A, Aw	`WRITE(*,'(A)') C`
Logical	Lw	`WRITE(*,'(L7)') T`

Output formatting

Probably do not need!

Control edit descriptors

Variables: Integer :: I, J

(n = number of characters)

Task	**Descriptor**	**Example**
New line	`/`	`write(*,'(I5,/,I5)') I, J`
Tabbing	`Tn`	`write(*,'(I5,T20,I5)') I, J`
Tabbing	`TRn`	`write(*,'(I5,TR5,I5)') I, J`
Tabbing	`TLn`	`write(*,'(I5,TL3,I5)') I, J`
Number of blanks	`nX`	`write(*,'(I5,5X,I5)') I, J`
Do not read blanks	`BN`	`read(*,'(BN,I5)') I`
If blanks -> zeros	`BZ`	`read(*,'(BZ,I5)') I`
Switch on plus sign	`SP`	`write(*,'(SP,I5)') I`
Switch off plus sign	`SS`	`write(*,'(SP,I5,SS,I5)') I, J`

Output formatting: miscellaneous

- Complex number case, give data format for both parts:

```
Complex :: Z
WRITE(*,'(F6.3,F6.3)') Z
```

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

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

Output formatting: miscellaneous

- Matrix style (2D), row by row, output example (4x3 matrix):

```
INTEGER :: j
INTEGER, PARAMETER :: ind1=4, ind2=3
REAL, DIMENSION(ind1,ind2) :: R
DO j=1,ind1
  WRITE(*,'(3(F7.3,1X))') R(j,:)
END DO
```

Formatted vs. unformatted files

- Text or *formatted* files are
  - Human readable
  - Portable i.e. machine independent
- Binary or *unformatted* files are
  - Machine readable only
  - 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
  - Not necessarily portable

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
```

Internal I/O

IGNORE!

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

Internal I/O

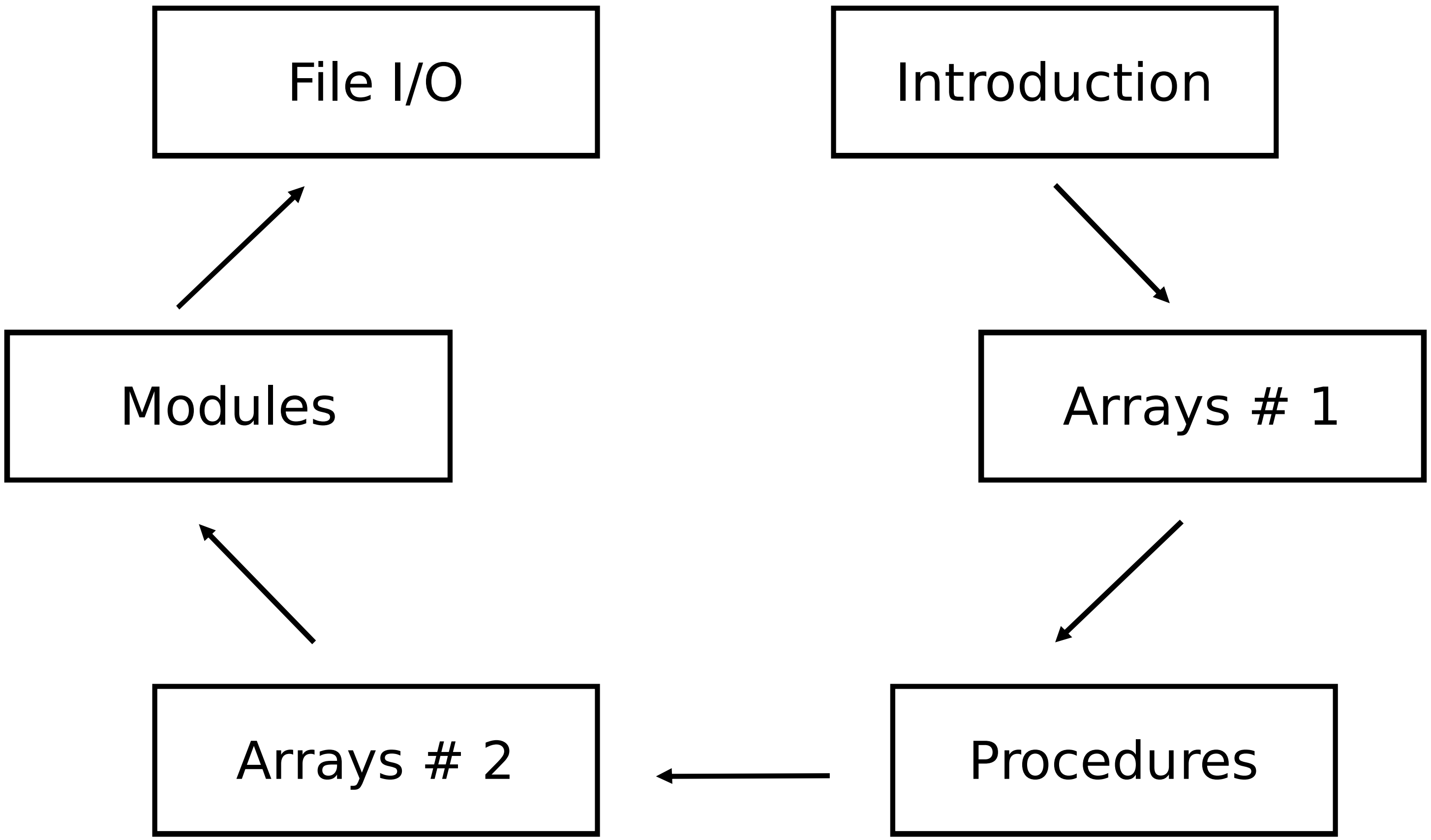
```
character(len=8), :: tstamp = ‘20100613'
integer :: yr, mon, day
character(len=3), parameter :: mons(12) = (/ &
’Jan’, ’Feb’, ’Mar’, ’Apr’, ’May’, ’Jun’, &
’Jul’, ’Aug’, ’Sep’, ’Oct’, ’Nov’, ’Dec’ /)
character(len=11) :: pretty

READ(tstamp, fmt='(i4,i2,i2)') yr, mon, day
WRITE(*,*) ‘yr, mon, day = ', yr, mon, day
WRITE(pretty, fmt='(i2.2,”-”,a3,”-”,i4)') &
      day, mons(mon), yr
WRITE(*,*) pretty
```

Summary

- Disk files: communication between a program and the outside world
  - Opening and closing a file
- Data reading
- Data writing

Fortran 95 modules overview



Web resources

- Get CSC’s Fortran95/2003 Guide (in Finnish) for free
  http://www.csc.fi/csc/julkaisut/oppaat
- GNU Fortran online documents
  http://gcc.gnu.org/onlinedocs/gcc-4.5.0/gfortran
- Examples repository
  http://www.nag.co.uk/nagware/examples.asp
- More examples
  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