Fortran syntax overview

Uwe Küster

Manuela Wossough

Hao Feng

Email: kuester@hlrs.de

Numerics and Libraries

RUS/HLRS

geaendert 16.10.03

Fortran 90 introduction: overview

- Part I
  - Basic Syntax
  - Intrinsic Data Types
  - Kind Type Parameter
  - Derived Types
  - Controlling the flow of a program
  - Program Units and Procedures
    - **Module**
    - **External/Internal Procedures**
    - **Explicit/Implicit Interfaces**
    - **Intrinsic Procedures**
    - **Overloaded/Extended Operators/Assignments**

Fortran 90 introduction: overview

- Part II
  - Pointer
  - Dynamic Data
  - Array Syntax
  - Keywords and Optional Arguments
  - Information Hiding
  - Input / Output
  - Obsolete Features

Part I

ISO Standards for Fortran

- Fortran 77 ISO 1539:1980
- Fortran 90 ISO 1539:1991
- Fortran 95 ISO 1539:1997
- Fortran 2003 ISO 1539:2003

Basic Syntax: character set

- Names of Programs, Procedures and Variables
  up to 31 characters or underscore '_' (Fortran 2000: 63)
  the first character must be a letter.

  subroutine sparse_matrix_solver(...)
- Multiple instructions per line are separated by ';'

  aa=0. ; bb=2.
- operators

  (<=, = =, >=, <, >, ... instead of .LE. , .EQ. , .GE. , .LT. , .GT. , ...)
- Upper and lower case letters are treated as identical
  (but not if calling C-procedures!)
- New declaration syntax
- Fixed and free source form possible

Basic Syntax: traditional Fixed Source Form

- Fortran Statements are located between column 7 and 72
- ‘c‘ or ‘* ‘ in the first column introduces a comment
- multiple statements per line can be separated by ‘;‘(not in F77 !)
- Columns 1 to 5 can contain labels from 1-99999
- Any character in column 6 (even ‘!‘ or ‘;‘) which is not a blank or zero introduces the line as continuation line of the line before. 19 continuation lines are possible.

Basic Syntax: Fixed Source Form example

```
|1234567
|        program solve_eqn_sys
|c     this is an example for fixed source form
|        integer dim
|        parameter(dim=5)
|        real  x (dim), y(dim), a(dim, dim)
|c
|        call   initialize_equation_system(a, x, y, dim)
|c
|c     solving the equation system
|        call solve_equation_system(a, y, x, dim)
|c     print solution
|        print*, ‘ Solution of the equation system : ‘
|        print  100,  x
|  100   format  (f10.1, f10.2, f10.3,
|      1            f10.3, f10.3)
|        end
```

Basic Syntax: Free Source Form

- Up to 132 character per line
- Comments start with '!' in each position of any line
- Continuation line with '&' (up to 39 countinuation lines are possible)

```
subroutine  sparse_matrix_solver(matrix, &
            &   right_hand_sides, x_dimension, &
            &   y_dimension, error_flag)
```

- aa = bb * cc ! this is an array syntax example; aa, bb, cc arrays

Basic Syntax: Free Source Form example

```
program centigrade_to_fahrenheit
implicit none                          ! forces declaration of all variables
!  This program converts a Centigrade temperature ( c )
!  to Fahrenheit ( F ) via the formula    F = ( 9 * C/5) + 32
   real        : :  deg_fah, deg_cen    ! variables declaration
!
   print*, ' What is the Centigrade temperature ? '
   read*,  deg_cen                     ! read the value
   deg_fah = (9.0*deg_cen/5.0) + 32.0   ! conversion
   print*, ' Centigrades:  ' ,deg_cen, '  Fahrenheit: ',deg_fah
end program centigrade_to_fahrenheit
```

Statements Order

- program, function, subroutine, module
- use statements
- implicit none
- derived types definitions, interface blocks, specification statements
- statement functions,
- executable statements

statement ordering Fortran 95

<table>
<tr><td colspan="3">program, function, subroutine, module, blockdata</td></tr>
<tr><td colspan="3">use</td></tr>
<tr><td colspan="3"></td></tr>
<tr><td rowspan="4">format, entry</td><td colspan="2">implicit none</td></tr>
<tr><td>parameter</td><td>implicit</td></tr>
<tr><td>parameter, data</td><td>derived type defs, interface blocks,
type declarations,
procedure decls,
specification statements, statement functions</td></tr>
<tr><td>data</td><td>executable constructs</td></tr>
<tr><td colspan="3">contains statement</td></tr>
<tr><td colspan="3">internal subprograms or module subprograms</td></tr>
<tr><td colspan="3">end statement</td></tr>
</table>

statement ordering Fortran 2000

<table>
<tr><td colspan="3">program, function, subroutine, module, blockdata</td></tr>
<tr><td colspan="3">use</td></tr>
<tr><td colspan="3">import</td></tr>
<tr><td rowspan="4">format, entry</td><td colspan="2">implicit none</td></tr>
<tr><td>parameter</td><td>implicit</td></tr>
<tr><td>parameter, data</td><td>derived type defs, interface blocks, type declarations, type alias defs, enumeration decls, procedure decls, specification statements, statement functions</td></tr>
<tr><td>data</td><td>executable constructs</td></tr>
<tr><td colspan="3">contains statement</td></tr>
<tr><td colspan="3">internal subprograms or module subprograms</td></tr>
<tr><td colspan="3">end statement</td></tr>
</table>

Declaration part:

- the declaration part is a special part at the top of each program unit (different to C, C++ and Java)
- implicit declarations are allowed but not recommended
- intrinsic data types (integer, real, complex, logical, character)
- derived data types (struct in C)
- data types may have attributes
- kind values define properties of data types

Declaration Part: Attributes

Attribute	Keyword
Data Type	INTEGER, REAL (and DOUBLE PRECISION), COMPLEX, LOGICAL, CHARACTER; TYPE(own name)
Array specific	DIMENSION; ALLOCATABLE
Pointer specific	POINTER; TARGET
Data Initialisation	DATA; PARAMETER
Access to Data Objects	PUBLIC, PRIVATE
Usage of Data Objects	INTENT, OPTIONAL, SAVE
Procedure specific	EXTERNAL, INTRINSIC

Declaration Part: Attributes

Declaration:

Type[([kind=]kind_param)] [, attr_list : :] entity_list

Examples:

```
integer imax, i, j
integer, dimension( : , :) , allocatable  : : matrix
real , dimension ( : ) , pointer          : : vector
integer , dimension (100),intent(in)      : :  index_list
```

Intrinsic Data types: integer

- Declaration:

  integer[([kind=]kind_param)] [, attr_list : :] entity_list

- Examples:

```
!* Integer variables
integer imax, jmax                  !* Fortran 77 style
integer                             :: x ,   iterations
!* Static Integer Arrays
integer , dimension(24)             :: int_1d_array !* Static Array
integer, dimension(20, 5, 10)       :: int_3d_array
!*  Integer Constants
integer , parameter                 : : max_iterations = 1000  !*Constant
```


Intrinsic Data Types: real

- Declaration:

  real [([kind=]kind_param)] [, attr_list : :] entity_list

  double precision [, attr_list : :] entity_list

- Examples:

```
!* Real Variables
 real  gradient              !* Fortran  77  style
 real                          : :  epsilon ,  div
!* Static  Real Arrays
real , dimension(8)          : : real_1d_array
real , dimension(100, 2)   : : coordinates , real_2d_array
!* Real Constants
real , parameter                : : eps=0.5E - 5   !*  Constants
!*  Declarations using  'double  precision'
!*          (fortran  77  only  !)
doubleprecision , parameter       : : eps2=0.5E-12
double precision                  : : rot
```

Intrinsic Data Types: Arithmetic with INTEGER and REAL values

- Integer Division

  my_int=3/4 ! my_int is zero

- Sequence of Operations : left to right

```
real        : : a , b
integer     : : ic , id

ic = 1 ;  id = 3 ;  b= 3.0

a = b * ic  /  id      !* a = 1.0
a = ic  /  id  *  b    !* a = 0.0

a = 3.2;  b = 5.6

ic = a + b             !* ic = 8
```

- Type Conversion by intrinsic functions:
  INT, NINT, REAL, DBLE, CMPLX

Intrinsic Data Types: real assignments

```
real(kind=8)                :: dp
real(kind=4)                :: sp
real(kind=my_kind)          :: mk_p
        sp = 3.3 ; dp = sp  ! loss of accuracy
        dp = 3.3            ! dangerous
        dp = 3.3d0          ! better
        dp = 3.3_8
        mk_p = 3.3_my_kind  ! recommended
```

Intrinsic Data Types: complex

- COMPLEX Data
  - Declaration:

    complex[([kind =] kind_param)] [, attr_list ::] entity_list

  - Examples:

```
complex                    :: cmpl1, cmpl2
complex , dimension(4)  :: z1 , z2
complex(kind=8)        :: cpl

z1(1) = (1.5 , 7.3)

cpl = ( -6.2d3 , 9.0d3 )
```

  - Complex arithmetic with operators + , - , *, / and intrinsic functions

Intrinsic Data Types: logical

- LOGICAL Data
  - Declaration:
    logical [([kind =] kind_param)] [, attr_list ::] entity_list
  - Examples:

```
logical     :: var1, allocated_data
var1 = .true. ; allocated_data = .false.
```

  - Logical Operators are .or . , and. , .not. , .eqv. , .neqv.
  - Operators in expressions with logical result
    == , /= , >= , <= , >, <

Intrinsic Data Types: character

- Declaration:

  character [([len =]len , [kind =] kind_param)] &
  & [, attr_list ::] entity_list

- Examples:

```
!* Declaration of  Strings
character*20                name1        !* Fortran 77 style
character                   name2*20     !* Fortran 77 style
character(20)                            :: name3
character(LEN=20)                        :: name4
character(LEN=5*len(name4))              :: name5

!* Static Arrays of Strings
 character(LEN=20) , dimension(100)  ::  name_list
```


Intrinsic Data Types: character

- all character strings have fixed length
- Concatenation Operator //

  character(LEN=7) :: name

  name= “Fred “// “die“

- Substrings

  character(LEN=8) :: string

  string = “Alphabet “

  string(5:5) = “o“ !* => string = “Alphobet “

  string(4:)= “ine “ !* => string= “Alpine “

- Intrinsic functions for string handling etc.

Intrinsic Data Types: character

```
character(len=10)                    :: str
character(len=3),dimension(2)        :: words

str='das ist''s'                     ! das ist's
str(1:)=str(5:7)//' '//str(1:3)      ! ist das
words=(/ 'ls ','cat' /)              ! same size!
str(2:3)=words(2)(2:3)               ! iat das
```


Intrinsic Data Types: IMPLICIT Statement

- Implicit type declaration of Fortran Variables (don't use)
  Variable names starting with letters I-N: Default Integer
  All other variables: Default Real
- Statement for other implicit declarations (don't use)

  implicit complex(a-c)

  implicit logical(1)
- IMPLICIT NONE **(strongly recommended)**

  to switch off the implicit type declaration

  should be the first statement in any program part to avoid errors

Intrinsic Data Types: Kind Values

- Each processor (compiler) has at least
  - one representation for integer , logical , character variables
  - two representations for real , complex variables
- Fortran 90 allows the programmer to specify in a portable way.
  - degree of precision, and/or
  - the range of values required
  - Compiler ensures that the most suitable of the hardware representations is used
- The representation is characterized by the *kind type parameter*
- The *kind type parameter* is positive integer value

Intrinsic Data Types - Kind Values: Integer, Real, Complex

- selected_int_kind (*r*)

  Intrinsic function to specify the range required

  returns a value of the type kind parameter for an integer

  data type that can represent *at least* all integer values *n*

  in the range $-10^r < n < 10^r$

```
integer , parameter      :: int_range = selected_int_kind (20)
integer(kind = int_range)  : int_value
```

  int_value can store values between -10^{20} and 10^{20}

Intrinsic Data Types - Kind Values: Integer, Real, Complex

- selected_real_kind (*p* , *r*).

  Intrinsic function to specify a real precision which is at least

  p decimal digits of accuracy and

  a decimal exponent range of at least *r*

```
integer, parameter   ::  real_prec = selected_real_kind (16 , 40)
real ( kind = real_prec )                  :: real_value
real (selected_real_kind (10, 30) )     ::  r1
```

  r1 : up to 10 decimal digits, Values up to 10^{30}.

Intrinsic Data Types - Kind Values: Integer, Real, Complex

- kind

  Intrinsic inquiry function which returns the kind type parameter value of any integer , real, complex, character or logical entity.

  kind (0) : default integer

  kind (0.0) : default real

  kind (0.0D0) : default double precision

Intrinsic Data Types - Kind Values: Integer, Real, Complex

- Examples:

```
integer , parameter  :: double = kind (0.0D0)
integer , parameter  :: my_real_precision=selected_ real_kind (10,30)
                                  !! min. 10 dec. digits
                                  !! exponent range +/- 30
real (kind = my_real_precision) :: rval
rval = 3.27_my_real_precision
real ( kind = double)        !! double  precision
real (double)
. . .
```


Intrinsic Data Types - Kind Values: Integer, Real, Complex

```
integer (selected_int_kind (9) )
                                  !!  min. 9 decimal digits
real (selected_real_kind (24 , 300) )
                                  !! min. 24 dec. digits
                                  !! exponent range +/- 300
complex(double)        !! double complex
```

- Intrinsic function RANGE: Decimal exponent range
  Intrinsic function PRECISION: Decimal precision of real values
- no support of variable length arithmetic

Intrinsic Data Types - Kind Values: character

- character(kind=kind)
  - Kind type parameter identifies which character set is being used, e.g.

    a natural language character set as Cyrillic or Kanji or
    a character set containing special graphics symbols such as those required for printing music.

Derived Types

- Self defined data structures with components of arbitrary types
- Number of components is fixed at compile time
- Component selection introduced by ‘ % ‘ (**.** for C, C++, Java)
- Nested derived types
- Recursive type definition allowed if components are pointers.
- Structure Constructor uses derived type name as *constructor function* (has nothing to do with C++ Constructor !)
- Attributes for derived type declaration:
  - sequence
    achieves a sequence of the type components
    in the memory (sequence for better optimization: don‘t use sequence)
  - public, private
    attributes for information hiding.

Derived Types: Examples

```
type person
   integer          :: age
   character (5)  :: name
end type person
type group
   type (person) :: employee(50)
   real            :: salary(50)
end type group

type(person)                        :: person_a
type(person), dimension(1:3)  :: person_b
type(group)                         :: working_unit
```


Derived Types: Examples

```
   Person_a%age=35; person_a%name='Peter Smart'

   person_b(2)=person(27,'Hugo Schmitt')   ! structure constructor

working_unit%employee(44)%age = 55
```

structure constructor is related to the sequence of definitions in the derived type;

this limits later changes;

usage is not recommended;

same applies for derived type IO

Derived Types: Examples

```
type color
    integer                 :: shade, intensity
   character(LEN=30)   :: name
end type color

type(color), dimension(3)    :: my_ favorite

my_favorite =(/ color(1,2,“red“), color(3,2,“blue“), color(5,1,“green“) /)
                                  ! array constructor
```


Derived Types: f95 allows default values

```
type my_para
    integer,private                          :: dimension
    integer,private                          :: number_of_indices
    integer                                  :: number_of_columns
    integer                                  :: number_of_rows
end type my_para

type sparse_matrix
  type(my_para)                              :: para=my_para(0,0,0,0)
  integer,pointer,dimension(:)               :: length_col => null()
  integer,pointer,dimension(:)               :: begin_col => null()
  integer,pointer,dimension(:)               :: column_number=>null()
  real(kind=my_kind),pointer,dimension(:)    :: values => null()
end type sparse_matrix
```

Derived Types: order of structure components

- the order of the structure components has no significance except for
  - structure constructors
  - IO of the derived type as entity
  - if sequence is defined
- for flexible handling of components avoid these techniques:
  - new components can be introduced at any place
  - old components may be removed

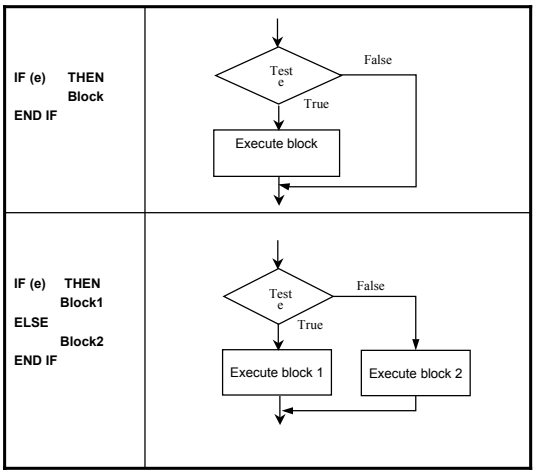
Control constructs

- control constructs for formulation of algorithms
- similar in all programming languages
- loops for iterations
- branches

Control constructs

- if (..) then
  elseif (..) then
  else (..) then
  endif
- select case ()
  case ()
  case default
  end select
- do while ()
  enddo
- do n=1,nmax,diff
  enddo
- do
  cycle | exit
  done

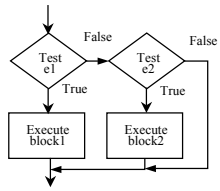
all of these may have names for better identification; controlled by compiler



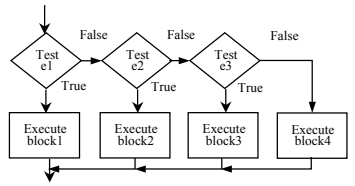
Execute block

FLOW OF CONTROL IN CONSTRUCTS

```
IF (e1)          THEN
        Block1
ELSE IF (e2)     THEN
        Block2
END IF
```




```
IF (e1)          THEN
        Block1
ELSE IF (e2)     THEN
        Block2
 ELSE IF (e3)    THEN
        Block3
ELSE
        Block4
END IF
```



Control constructs: IF Statement, Example

```
IF (x == y) call answer (x,y)

traffic_light : IF  (color ==  'red' ) THEN
    time = 30
    call red(time)
ELSE IF (color == 'green') THEN
     time = 20
     call green(time)
ELSE
    time = 5
    call yellow(time)
END IF traffic_light
```

Control constructs: CASE Statement, Example

```
Surface : SELECT CASE (n)
   CASE  (:0)
      print*, ‘Sorry , negative number‘
   CASE  (1)
       name=‘triangle‘
       a=1. / 2. * g * h
   CASE  (2)
      name=‘circle‘
      a=pi * r * r
    CASE DEFAULT
         CALL error (n)
    END SELECT surface
```

faster
than

if …
elseif …
elseif …
else …
endif

Control constructs: CASE (example from DIGITAL Fortran manual)

```
CHARACTER(len=1) ::  cmdchar

GET_ANSWER: SELECT CASE (cmdchar)
CASE ('0')          ;  WRITE (*, *) "Must retrieve one to nine files"
CASE ('1':'9')      ;  CALL RetrieveNumFiles (cmdchar)
CASE ('A', 'a')     ;  CALL AddEntry
CASE ('D', 'd')     ;  CALL DeleteEntry
CASE ('H', 'h')     ;  CALL Help
CASE DEFAULT
  WRITE (*, *) "Command not recognized; please use H for help"
END SELECT GET_ANSWER
```

Control constructs: DO loops with loop control

- backwards going loop

```
do i=nmax, nmin,-2
    if (i == number) print*,‘ found‘
enddo
```

- no floating points allowed for loop control
  (avoid this in any language!)
- non execution for

```
do i=1,0
...
enddo
```

- loop control variables must not be changed within the loop!
  (difference to C; enables better compiler optimization)

Control constructs: DO WHILE loops

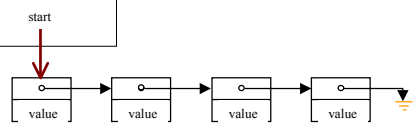
```
do  while (. not . found)
    i=i+1
    if  (i ==number) found= . true .
  enddo
```

do while with exit is slower than explicitly controlled loops!

Control constructs: DO WHILE loops; crossing a linked list

```
type linked_list
    type(linked_list),pointer          :: next
    real(kind=8)                       :: value
end type linked_list

type(linked_list),pointer              :: link
!        the last link%next has to be the null pointer
do while ( associated ( link%next) )
  link => link%next
  if ( link%value <= 0. ) exit
enddo
```



Control constructs: DO loop and IF construct

```
        j=0; k=0; n=0
distribution_loop: do i=1,imax,2

cc26: if ( a(i) < 0._my_kind ) then
            j=j+1      ; b(j)=a(i)
    elseif ( a(i) == 0._my_kind ) then
    cc26
            k=k+1      ; c(k)=a(i)
    else cc26
            n=n+1      ; d(n)=a(i)
    endif cc26

enddo distribution_loop
```

is vectorizable!

Control constructs: **DO loops, cycle and exit**

- do construct (infinite loop) with exit or cycle statement; slower

```
i=0; j=0
 even_i: do
   j=j+1
   if (mod(j , 2)==0) then    !!  even number
        i=i+1; number(i)=j ; print*, ‘ even number  ‘, j
   else
      cycle  even_i
   endif
   if ( i == maximum_numbers ) exit even_i
enddo even_i
```


Program Units and Procedures

- Program Units:
  - Main Program
  - External Procedure
  - Module
- Procedures
  - External Procedure
  - Internal Procedure
  - Module Procedure
  - Intrinsic Procedure

Program Units and Procedures: main program

- Form:
  - [PROGRAM program-name]
  - [specification part]
  - [execution part]
  - [internal subprogram part]
  - END [PROGRAM[program-name]]
- No arguments can be passed to main programs like in C, C++ or Java;
  this will be possible in Fortran 2000 ;
  most todays compilers provide for command line arguments

Program Units and Procedures: Functions and Subroutines

- Subroutine
  [RECURSIVE] &
  & SUBROUTINE subroutine-name [([dummy-argument list])]
  [specification part]
  [execution part]
  END [SUBROUTINE [subroutine-name]]
- Function
  [RECURSIVE] &
  & FUNCTION function-name [([dummy-argument- list])] &
  & [RESULT (result-name)]
  [specification part]
  [execution part]
  END [FUNCTION [function-name]]

function example

```
function circle_of(center,radius,number) result(result_value)
complex(kind=dk)                          :: center
complex(kind=dk)                          :: radius
complex(kind=dk),pointer,dimension(:)     :: result_value
integer                                   :: number
integer                                   :: ii
allocate(result_value(number))
result_value=center +radius &
        & *exp(img *2._dk*pi &
        & *(/(real(ii-1,kind=dk)/real(number-1,kind=dk),ii=1,number)/) &
        & )
end function circle_of
```

deallocation to be done in the calling program unit

main program example

```
program kurve
use complex_array_module
integer,parameter                         :: nmax=801
real(kind=dk),dimension(nmax)             :: unit_intervall
complex(kind=dk),pointer,dimension(:) :: gg_circle=>null()
complex(kind=dk)                          :: mid

mid=(0.125_dk,.30_dk)
gg_circle=>circle_of(center=mid,radius=-mid,number=size(unit_intervall))
call write(gg_circle,give_unit_of("gg_circle.dat"))
end program kurve
```

Program Units und Procedures: modules

module

```
module this_module
use other_module
type my_type
end type my_type
real  :: pi=3.142
interface ....
contains
    subroutine
    function
end module this_module
```

access to other modules

definition of types

global variables

interfaces

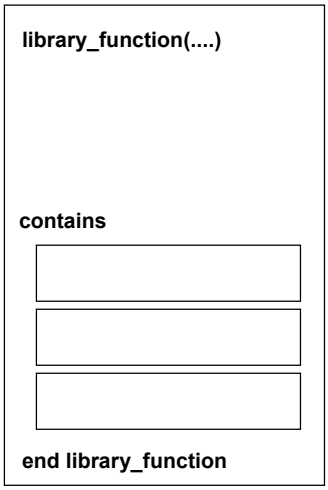
module procedures

Program Units und Procedures: Nesting of subprograms

Internal Procedures

```
use my_module
contains
    contains
        [ ]
        [ ]
    contains
        [ ]
        [ ]
        [ ]
```

Program Units und Procedures: external procedures



External Procedure

Internal Procedures

Modules

module is a program Unit, which may contain

- Type Declarations
- Variable Declarations
- Interface Declarations
- Specification Statements (implicit, sequence, use, ..)
- Module Procedures

Modules : structure

Form of a Module :

MODULE name

[(specification part)]

[CONTAINS

(Module procedures)]

END [MODULE [name]]

generates .mod file

Modules: encapsulation

- A module encapsulates
  - shared (global) data
    (replaces COMMON blocks)
  - abstract data types
    (data type plus operations/methods)
  - procedures and interfaces
    (library)
- private and public control export to other modules (visibility)
- *use* controls import from other modules
  (use *module_name*)

Modules: example 1

```
module planes
real , parameter  ::  pi=3.1415926535  !  Global Variable
                                       !  Known in 'planes'
                                       !  and 'circle'
contains
  function circle_surface(radius) result(ci_sur)
    real, intent(in)   ::  radius
    real               ::  ci_sur
    ci_sur = pi *  radius ** 2
  end function circle_surface
    :
end module planes
```


Modules: example 2

```
program circle
use planes
real  ::  radius, surface
   :
read*, radius
surface = circle_surface(radius)
   :
end program circle
```

Interfaces: implicit versus explicit

- implicit Interfaces Call to external procedure
  (Procedure name may be specified in an EXTERNAL-Statement)
  - traditional form (as in FORTRAN 77)
  - typical for libraries
- explicit Interfaces (Fortran 90)
  - Call to internal procedure in the same program unit
  - Call to module procedure,
    either from another statement in the same module or from a statement after a use statement in another program unit
  - Call to intrinsic procedures
  - Have much more information than implicit interfaces
  - Most interfaces can be made explicit

Interfaces: functionality of explicit

- Explicit Interfaces are necessary for
  - Optional arguments
  - Keyword actual argumants
  - Generic procedures
  - Functions with array as result value
  - Functions with character variables of dynamical length as result value
  - Assumed shape dummy arguments
  - Dummy arguments with pointer oder target attribute
  - Overloaded and user defined operators
  - Overloaded assignment

Interfaces: interface blocks

- **Interface block** to make an implicit interface explicit.
  Describes the characteristica of an external procedure and its arguments.
  (An interface ist an exact copy of the procedure without the local specification part and the execution part)

```
interface
   real function fun(x)
      real, intent(in)  ::  x
   end function fun
end  interface
```


Interfaces: new operators

- Explicit interface to define a new (user defined) operator

```
interface operator (.new.)
  module procedure set_new
end interface
```

a=b.new.c

Interfaces: overloaded operators

- Explicit interface to expand an existing operator (operator overloading)

```
interface  operator(+)
   module procedure add_vectors
end interface
```

v1=v2+v3

add_vectors is called

Interfaces: overloaded assignment

- Explicit interface to overload the assignment('='),

```
interface assigment(=)
   module procedure assign_matrix_to_matrix
end interface
```

matrix1=matrix2

assign_matrix_to_matrix is called

Interfaces: generic procedures

- Explicit interface to give a procedure a generic name

```
interface axpy
   subroutine saxpy(n,alpha,x,incx,y,incy)
     integer n,incx,incy
     real(kind=single) alpha,x(incx*n),y(incy*n)
   end subroutine saxpy
end interface
interface axpy
   subroutine daxpy(n,alpha,x,incx,y,incy)
     integer n,incx,incy
     real(kind=double) alpha,x(incx*n),y(incy*n)
   end subroutine daxpy
end interface
```


Interfaces: generic procedures

```
  :
real(kind=single) xs(n),ys(n)
real(kind=double) xd(n),yd(n)
  :
call axpy(n,7.6D0,xd,1,yd,1)
call axpy(n,7.4,xs,1,ys,1)
```

Modules with Interfaces: example

```
module obj_info_module
!* Global constant defined in a separate module
 use my_precision_module

 integer,parameter :: prec=selected_real_kind(10,30)
 integer,parameter ::cmax=20   !* Constants Declaration

  type obj_info                    !* Data Type Declaration
     character(LEN=cmax) :: name
     real(kind=prec) :: aflag
   end type obj_info

   interface assignment(=)         !* overloaded assignment
     module procedure objinfo_to_objinfo
     module procedure name_to_objinfo
  end interface
```


Modules with Interfaces : example

```
!* Module subroutines
contains

subroutine name_to_objinfo(objinfo,name)
   ...
end subroutine name_to_objinfo
   ...
end module obj_info_module
```

Modules with Interfaces : example 2

```
module tabular_module
implicit none                                  ! recommended
type tabular
  integer                                      :: nmax
  real(my_kind),pointer,dimension(:,:)         :: values
end type tabular

interface print                                ! generic name
  module procedure print_tabular               ! specific name
end interface print
contains
  ...    ! here are the procedures; see next slide !
end module tabular_module
```


Modules with Interfaces: subroutine example

```
subroutine print_tabular(string,tabular,index)
character(len=*)                :: string
type(tabular_type)              :: tabular
integer,dimension(:)            :: index

print*,trim(string)             ! trailing blanks are cutted
write(*,'(10i10)') (index(mm),mm=1,size(index))
do nn=1,tabular%nmax
  do mm=1,size(index)
    write(*,'(f10.2)',advance='no') tabular%values(nn, index(mm))
  enddo
    write(*,advance='yes')
enddo
end subroutine print_tabular
```

Modules with Interfaces: how to be called

```
type(tabular_type)                    :: tabular
integer,dimension(:),pointer          :: index

allocate(index(tabular%nmax-2))

index=(/nn,nn=size(index),1,-1/)      ! reverse order

call print("test_tab ",tabular,index)  ! nonspecific generic name
call print("this is a matrix""",matrix)  ! nonspecific generic name
```

simply to use!

Modules with Interfaces: generic interfaces for object oriented approach

```
interface allocate
  module procedure allocate_tabular
end interface
interface deallocate
  module procedure deallocate_tabular
end interface deallocate
interface print
  module procedure print_tabular
end interface
interface assignment (=)
  module procedure tabular_to_tabular
end interface
```

Modules with Interfaces: how to use the module procedures

```
use tabular_module
type(tabular_type)                    :: test_tabular, tabular_1,tabular_2
...
call allocate(test_tabular &
            &,number_of_lines=nmax & ! comment
            &,no_of_properties=mmax)
call read(test_tabular,from_file='file_1.tbl')
tabular_1=test_tabular                ! overloaded assignment
test_tabular=change(tabular_1)        ! using a function change
call print( 'test in main',tabular_1,(/3,2,5/) ) ! print columns 3,2,5
```

combination of generic procedures, assignment, functions with derived type result

Modules with Interfaces: overloaded operator +, definition

```
module test_module
type var_type
    real(kind=real_kind)  :: var
end type var_type
interface operator (+)
    module procedure var_add_var
end interface
contains
function var_add_var(aa,bb) result(result_value)
type(var_type),intent(in)  :: aa,bb
type(var_type)             :: result_value
    result_value%var=aa%var+bb%var
end function var_add_var

end module test_module
end module test_module
```

Modules with Interfaces: overloaded operator +, usage

```
use test_module

type(var_type),dimension(mmx) :: aa,bb,cc

 do mm=1,mmax
   aa(mm)=bb(mm)+cc(mm)
  enddo
```

inlining is essential

use only for large objects

(e.g. sparse matrices)

Summary - Part I

- Fortran 90 programs should be written in free source form
- Intrinsic Data Types: integer, real, complex, logical, character
- Fortran 90 knows a kind type parameter to specify precision and exponent range of variables and constants.
- Derived Data Types

```
TYPE type_name
  component1_type :: component1
  ...
END TYPE type_name
```

Summary - Part I

- Fortran 90 knows IF-statement, CASE-Contructs and several different DO-Loops.
- Fortran 90 has 3 kinds of program units: Main Programs, External Procedures,Modules
- Fortran 90 has 4 kinds of procedures: External Procedures,Internal Procedures,Module Procedures, Intrinsic Procedures
- There are 2 kinds of interfaces: Implicit Interfaces,Explicit Interfaces
- Fortran 90 offers the possibility to define new operators and to overload existing operators and assignment

Part II

Pointer 1

- Pointer in FORTRAN 90 is not a data object of special type, pointer is an attribute of a data object.

```
Integer, pointer      :: ip,iq
real, pointer         :: rp,p_feld(:)
```

- pointers are complete data object descriptors, not simply addresses.
- No arrays with pointer components
  (=> array of pointer via derived type with pointer component)
- pointer assignment('=>')associates pointer with existing memory.
- A Fortran-90 Pointer is the pointer and the data object
  Context determines whether pointer or referenced object is used.

```
allocate(q)
iq = 17
ip => iq
rp => p_feld(ip)
```


Pointer 2

- Targets can be:
  - an array element,
  - a part of an array
  - a simple variable,
  - a derived type component,
  - a whole derived type
- Pointer can only point to objects with target or pointer attribute.
- Targets allocated by a pointer implicitly have the target attribute.
- Other objects are never target of pointers
  (difference to C; important for optimization)

Pointer 3

```
real, dimension(5,4), target :: D2_array
real, dimension(3), target    :: D1_array
real, dimension(:), pointer   :: array_p ! single dimension
integer, pointer              :: p,q
integer, target               :: r
      :
allocate(q)
q = 4; p => q          !! p,q have the value 4
      :
r = 8; q => r          !! q has the value 8
                       !! P has value 4(!)
array_p =>  D1_array
      :
array_p =>  D2_array(3,:)  !! Array_p has the values
                           !! D2_array(3,1),D2_array(3,2)
                           !! D2_array(3,3),D2_array(3,4)
```


pointer 3.1

```
program pointer
real,pointer :: pp,qq
real,target :: rr
allocate(qq)
qq=1.4; rr=3.
pp => qq
print*,'after pp => qq :'
print'(a,3f5.1)','pp,qq,rr=',pp,qq,rr
pp=9.
print*,'after pp=9.  :'
print'(a,3f5.1)','pp,qq,rr=',pp,qq,rr
qq => rr
print*,'after qq=>rr :'
print'(a,3f5.1)','pp,qq,rr=',pp,qq,rr
end program pointer
```

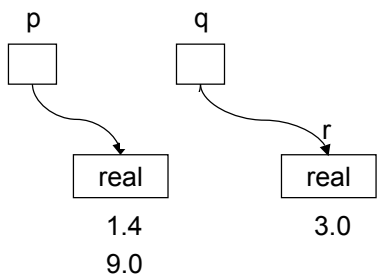
```
after pp => qq :
pp,qq,rr=  1.4  1.4  3.0

after pp=9.  :
pp,qq,rr=  9.0  9.0  3.0

after qq=>rr :
pp,qq,rr=  9.0  3.0  3.0
```

pointer 3.2

```
real,pointer :: p,q
real,target   :: r
allocate(q)
q = 1.4
r  = 3.
p  =>  q
p  =  9.
q  => r
```



Pointer 4

- Definition and Association Status
  - At beginning of program execution:
    association status and definition status undefined.
  - With **nullify** or **allocate** the association status becomes defined.
    (=> intrinsic function associated)
  - Definition status is defined, if target is completely defined
  - check for successfull memory allocation with status variable

```
integer                          :: error
integer, pointer, dimension(:)  :: p
        :
allocate(p(3),STAT=error)
        :
if (.not.associated(p)) allocate(p(5))
```

  - Intrinsic function **associated** to test whether a pointer is associated with a target.

Pointer 5

```
Integer, dimension(:), pointer :: p     !! Ass: undefined
                                        !! Def: undefined
allocate(p(3))                          !! Ass: defined (associated)
                                        !! Def: undefined
p=(/25,50,100/)                         !! Ass: defined (associated)
                                        !! Def: defined
nullify(p)                              !! Ass: defined (disassociated)
                                        !! Def: undefined
```


Pointer 6

- Example: Double Linked List, Binary Tree

```
        type tree
            integer                 :: value
            type(tree), pointer     :: left, right
        end   type tree
        type(tree), pointer         :: root, temp
            . . .
        allocate(root, new)
         root%value = 3
    nullify(root%left, root%right)
    temp = root                 ! ordinary assignment (object copied)
    root%left  =>  new          ! pointer    assignment (pointer copied)
```

Pointer 7: behaviour of a procedure interface

Actual and dummy arguments having the attributes pointer, target or none of these attributes

		Dummy Argument		
		POINTER	TARGET	OTHERS
Actual Argument	POINTER	TKR	TKR AESA	TKR AESA
	TARGET	not allowed	TKR AESA	TKR AESA
	OTHERS	not allowed	TKR AESA	TKR AESA

- TKR:‘type, kind, rank‘.
- AESA:‘array element sequence association ‘.

Dynamic Data

- dynamic data consist on arrays of any type allocated at run time
- pointer arrays
- allocatable arrays
- automatic arrays
- (malloc, free in C; constructor/destructor in C++; new in Java)
- never forget: allocation and deallocation is expensive

Dynamic Data 1: allocatable arrays

- Dynamic (Allocatable) Arrays
  - Before execution of the program, only rank and name of the array is specified.
  - controlled memory management with allocate and deallocate
  - intrinsic function **allocated**
  - dynamic arrays are created on the heap on conventional computers
  - dynamic array may not be a dummy argument of a procedure
  - Fortran 90: dynamic array must not be a derived type component. Allowed in Fortran 2000

Dynamic Data 2: allocatable arrays, example

```
real, dimension( : , :), allocatable  ::  a
integer, allocatable                  ::  b(: , : , :)
read*, nmax
allocate(a(nmax , 3))
allocate(b(5 , nmax**2 , 3*nmax+1))
        :
deallocate(a , b)
```

Dynamic Data 3: automatic arrays

- Automatic Arrays
  - are defined as variable arrays in procedures
    (=> local arrays !)
  - are created automatically entering a subprogram.
    (on conventional computers on stack (run time problem!))
  - are destroyed automatically with exiting the subprogram

subroutine automatic(imax, dummy_array)

```
integer                          :: imax
integer, dimension(:)            :: dummy_array

real, dimension(imax)                          :: local_array1
real,dimension(size(dummy_array))              :: local_array2
        :
end subroutine automatic
```


Dynamic Data 4: differences

- Example to show the differences of all dynamic arrays

```
subroutine swap(A,B, nmax)
        real, dimension(nmax)                  :: A
                                               !! Conventional
                                               !! Fortran 77
                                               !! array argument
        real, dimension(:)                     :: B
                                               !! assumed-shape array
        real, dimension(size(A))               :: auto_array
                                               !! automatic array
        real, dimension(:),allocatable         :: dyn_array
                                               !! dynamic array
        real, dimension(:,:),pointer           :: array_pointer
                                               !! array pointer
            :
        allocate (dyn_array(size(B)))
        allocate (array_pointer(size(A),size(B)))
            :
        deallocate(...)
  end subroutine swap
```

Dynamic Data: declaration of array dummy parameters

```
subroutine up(n,p,r,s,t)
real, dimension(1:15,5,10)          :: p        ! explicit
real                                :: r(1:15,n,*)        ! assumed size
real,dimension(:,5:)                :: s        ! assumed shape
real,allocatable,dimension(:,:)     :: u        ! dynamic
real,pointer,dimension(:)           :: v,t      ! array pointer
real,dimension(size(s,2),n)         :: bb       ! automatic array
...
end subroutine up
! default lower bound is 1!
```


Dynamic Data: different handling of implicit lower bounds 1

```
subroutine sub(aa,bb)
real, dimension(:)          :: aa      ! lower bound 1
real,pointer,dimension(:) :: bb      ! actual arg must be pointer
        print*,aa(1),bb(1)
end subroutine sub
```

Dynamic Data: different handling of implicit lower bounds 2

```
real,pointer,dimension(:)    :: xx
real,pointer,dimension(:)    :: yy
allocate(xx(-1:+1),yy(-1:+1))
xx=(/ (ii, ii=lbound(xx,1),ubound(xx,1)) /)        ! -1,0,+1
yy=(/ (ii, ii=lbound(yy,1),ubound(yy,1)) /)        ! -1,0,+1
call sub(xx,yy)
```

output: -1 1

Array Syntax 1

- A whole array can be handled as a single data object

```
   real , dimension(5,5)  ::  array_a, array_b, array_c
              :
 array_a = 3.2
 array_b =  array_a * 1.5
              :
 array_c =  array_a * array_b !! no matrix multiplication
```

- Array processing possible with whole arrays and array sections.

```
   real, dimension(2,3)  :: array_a
   real, dimension(3,2)  :: array_b
   real, dimension(3)    :: vector
              :
 array_a = 1.5
 array_b = 2.5
              :
 array_a(1,:) = array_b(:,2) - 1.2 * vector
```

Array Syntax 2

- masked array assignments with WHERE

```
real,dimension(10,10)   :: A
     :
where  (A/=0.0)
     recip_A = 1.0
elsewhere       ! Fortran 95
    recip_A = 2.0
endwhere
```

- Array Constructors(rank one arrays)

```
Form:Array=(List-of-values)
AA=(/B(I,1),B(I+1,1),c(2:6,1:4)/)
BB=(/1,3,5,7,9/)                    !! BB and CC are
CC=(/(2*i-1,i=1,5)/)                !! identical
```

- many intrinsic procedures for arrays.

Keywords and Optional Arguments

- Example

```
subroutine sub (x, y, z, upper, lower)
integer                          :: x
real, optional, pointer          :: y
real, optional, dimension(:)     :: z
real, optional                   :: upper, lower
    :
if (present(upper)) then
   u_bound = upper
else
   u_bound = 0
endif
    :
end subroutine sub
    :
    :
    :
call sub (x, y, z)
call sub (x)
call sub (x, upper = x0, lower = x1)
call sub (x, lower = x1, y = start)
```

very simple to use; self documenting interface

Optional Arguments: example

```
subroutine beep(number)
! beeps number of times. If number not present only one time
! call beep or call beep(5) or call beep(number=5)
integer,optional          :: number
integer                   :: no
integer                   :: nn

if(present(number) ) then
  no=number
else
  no=1
endif

do nn=1,no ;  write(*,*) char(7) ; enddo

end subroutine beep
```


Information Hiding 1

- Access specific attributes: PUBLIC; PRIVATE
- Setting the attributes handles access to objects of a module
- Default is PUBLIC

Information Hiding 2: Derived Types

- Attributes for derived type declarations:
  - public
    Data structure is globally visible.
  - private
    Data structure is only visible in the surrounding host.
- Type declaration public, components (all) private

```
module declaration
  type point
    private
    real  ::  x, y
end module declaration
```

- 'local' type declaration

```
type, private    :: straight_line
   integer       :: number
   type(punkt) :: point1, point2
end type straight_line
```


Information Hiding 3: Private Module Information

```
module exmpl_access_1
   private
   public   ::  a1, a2, a3, assignment(=), operator(*)
        ...
end module exmpl_access_1
```

```
module exmpl_access_2
     public        !  not necessary
     real, private  ::  b1, b2, b3
        ...
end module exmpl_access_2
```

Information Hiding 4: Private Module Information

```
  charater(5), public, save  ::  access_name = ‘alpha‘
  charater(7), private       :: password=‘rosebud‘
  type, private  :: vehicle
    integer  no_wheels
    real       weight
  end type vehicle
!* same as
  character(5) :: access_name
  character(7)  :: password
  data access_name/‘alpha‘/,password/‘rosebud‘/
  type vehicle
    integer  no_wheels
    real        weight
  end   type vehicle

  private password, vehicle
  public access_name
  save     access_name
```


Information Hiding 5: Private Module Information

```
module obj_info_module
   implicit none
   use my_precision_module
   private
!* Constants Declaration
  integer, parameter  ::  cmax=20
!* Data Type Declaration
  type obj_info
      character(LEN=cmax)  ::  name
      real(kind=prec)  ::  aflag
   end type obj_info
!* Interfaces Declaration
  interface assignment(=)
     module procedure objinfo_to_objinfo
     module procedure name_to_objinfo
  end interface
...
```

Information Hiding 5: Private Module Information

```
!* Public Module Interface
  Public ::  obj_info, assignment(=)
!* Module subroutines
contains
  subroutine name_to_objinfo(objinfo, name)
    . . .
  end subroutine name_ to_objinfo
        . . .
end module obj_info_module
```


Information Hiding 6: More Access control with Use Statement

- Rename same module entities in different modules
  USE module_name, rename-list
  Every element in rename-list looks like
  local_name => name_in_module
  Example:
  USE geometry, define_line => gen_line
- Rename generic defined module entities with own names
  USE geometry, &
  & circle_def => circle, line_def => line, &
  & point_def => point, line => gen_line

Information Hiding 7: More Access Control with Use Statement

- Example:

```
PROGRAM rename_example
  USE geometry, &
        & circle_def  => circle, line_def => line, &
        & point_def => point, line => gen_line
  IMPLICIT NONE
  TYPE(point_def)       :: pt1, pt2
  TYPE(line_def)        :: ln1,ln2,ln3
  TYPE(circle_def)      :: cir1
        . . .
  call line (ln1,pt1,pt2)
  call line (ln2,pt1,ln1)
  call line (ln3,pt2,cir1, 'xlarge')
              . . .
END PROGRAM rename_example
```


Information Hiding 8: More Access Control with Use Statement

- A program unit using a module can restrict the module access
  USE module_name, ONLY: only-list
  The only-list consists of name or renames of module entities
- Example :

```
USE geometry, ONLY: circle, line, point, &
                    &   define_line => gen_line
```

Input/Output

- read, write, print
- format
- open, close
- inquire

Input/Output: List-directed I/O; INPUT Editing

- Input Statement
  READ*, *input_list*
  READ *'(edit_descriptor)', input_list*
  READ *label, input_list*

Input/Output: List-directed I/O; INPUT Edit descriptors

I*w*	Read next *w* characters as an integer
F*w.d*	Read next *w* characters as a real number with *d* digits after the decimal place if no decimal point is present
E*w.d*	Read next *w* characters as a real number with *d* digits after the decimal place if no decimal point is present
A*w*	Read next *w* characters as characters
A	Read sufficient character to fill the input list item, stored as characters
L*w*	Read next *w* characters as the representation of a logical value
*n*x	Ignore the next n characters
*T*c	Next character to be read is at position *c*

Input/Output: List-directed I/O; INPUT Editing

Examples

```
read*, i1
read ‘(I9)‘, n
read ‘(4X, I5, F8.4, F12.6)‘, num, vall, val2
read ‘(A15,A20)‘, first_name, last_name
read  100 , first_name, last_name, age
100 format  (A15, 5X, A20, 2x, I3)
```

Input/Output: List-directed I/O; OUTPUT Editing

- Output Statement print
  - PRINT*, *output_list* ! list directed
  - PRINT *'(edit_descriptor)', output_list*
  - PRINT *label, output_list*
- Output statement write
  - WRITE(iunit,*) output list
  - WRITE(*iunit,'(edit_descriptor)') output_list*
  - WRITE(*iunit,label*) *output_list*
    label format*'(edit_descriptor)*
- *edit descriptor* in formats is evaluated at run time

Input/Output: OUTPUT Editing, format list

I*w*	Output an integer in the next *w* character positions
F*w*.d	Output a real number in the next *w* character positions with *d* decimal places
E*w*.d	Output a real number in the next *w* character positions using an exponent format with *d* decimal places in the mantissa and four character for the exponent

Input/Output: OUTPUT Editing, format list

A*w*	Output a character string in the next *w* character positions
A	Output a character string , starting at the next character position, with no leading or trailing blanks
L*w*	Output *w-1* blanks followed by T or F represent a logical value
nX	Ignore the next n character positions
Tc	Output the next item at character position c
'*rst*'	Output the string of characters rst starting at the next character position

Input/Output: List-directed I/O; OUTPUT Editing

Examples

```
print* , 'The integer value is ', i1

x = 3.14159; y= -275.3024; z= 0.0000361764
print '(F10.3,F10.3,2X,E10.4) ',x, y, z
!* => output string '  3.142    -275.302   0.3618E-04

print 100, first_name, last_name, age
100 format (A15, 5X, A20, 2X, I3)
```

Input/Output: READ, WRITE Statements

- More general form of read, print statements
  READ*(control information list) input_list*
  WRITE*(control information list), output_list*
- The control information list consists of one or more items, known as specifiers, separated by commas.

Form of I/O Control Specifiers

Specifier
UNIT = io-unit
FMT = format
NML = namelist-group-name
ADVANCE = scalar-default-character-expression
END = label
EOR = label
ERR = label
IOSTAT = scalar-default-integer-variable
REC = scalar-integer-expression
SIZE = scalar-default-integer-variable

Input/Output: READ, WRITE Statements

- There must always be a unit specifier of the from UNIT = unit
  - unit is the input (output) device, from which input is to be taken (to which output is to be written)
  - unit is an integer value or 0 or an asterisk(*) to indicate default input/output unit
  - a unit number identifies exactly one external file in all program units of a Fortran program
  - connection between unit number and external file by the:
    OPEN Statement
- The format specifier FMT = format identifies the needed format, format can be
  - default character expression, providing the format specification
  - an asterisk(*) to indicate list-directed formatting
  - scalar default integer variable as the label of a FORMAT statement

Input/Output: READ, WRITE Statements

- Examples

```
READ(*,*) a, b, c  !* same as READ*, a, b, c
READ(UNIT = 5, FMT = '(3F6.2)') x, y, z
WRITE(5,100)  x, y, z
WRITE(FMT=100, UNIT=5) x, y, z
READ(5 ,'(3F6.2)') x, y, z
100 FORMAT (3F6.2)
```


Input/Output: READ, WRITE Statements

- NML = namelist group name specifies the name of a namelist group declared in a NAMELIST statement.
- ADVANCE = scalar default character expression specifies advancing / nonadvancing I/O where
  - NO indicates nonadvancing formatted sequential data transfer
  - YES indicates advancing formatted sequential data transfer(default)
- END = label specifies what to do if the end of the file is reached label is the label of a branch target statement taken when an end-of-file condition occurs
- EOR = label specifies what to do if the end of the record is reached label is the label of a branch target statement taken when an end-of-record condition occurs

Input/Output: READ, WRITE Statements

- ERR = label specifies what to do if an error occurs
  label is the label of a branch target statement taken when an error occurs
- IOSTAT = scalar default integer variable checks the IO status
  - positive integer indicates an error condition occured
  - negative integer indicates an end-of-file or end-of-record condition occured
  - 0 indicates that no error , end-of-file or end-of-record condition occured

Input/Output: READ, WRITE Statements

- REC = scalar integer expression indicates the record number to be read or written
- SIZE = scalar default integer variable indicates the number of characters read before an end-of-record condition occured

Input/Output: READ, WRITE Statements

- Examples

```
WRITE(9, IOSTAT=is, ERR=99)
do i = 1, imax
   WRITE (* ,'(I1)', ADVANCE="NO") SSN(i)
 enddo
 READ(7, FMT=108, REC=32, ERR=99) A
 READ(UNIT=10, FMT=185, &
          &  ERR=99, IOSTAT = io_err) a, b, (c(i), i=1, 40)
```


Input/Output: OPEN Statement

- the OPEN statement
  - establishes a connection between a unit and an external file
  - determines the connection properties
- Connection Specifiers

Input/Output: OPEN Statement

- Connection Specifiers

Specifier	Possible values	Default value
ACCESS=	DIRECT, SEQUENTIAL	SEQUENTIAL
ACTION=	READ; WRITE; READWRITE	processor dependent
BLANK=	NULL;ZERO	NULL
DELIM=	APOSTROPHE, QUOTE, NONE	NONE
ERR=	label	no default
FILE	character expression	processor determined
FORM=	FORMATTED,	(ASCII)
	UNFORMATTED	binary IO

Input/Output: OPEN Statement

Specifier	Possible values	Default value
IOSTAT=integer	<0 EOF or EOR | 0 regular|	> 0 error
PAD=	YES, NO	YES
POSITION=	ASIS, REWIND, APPEND	ASIS
RECL=	positive scalar integer	processor dependent
STATUS=	OLD, NEW, UNKNOWN	UNKNOWN
	REPLACE, SEARCH	
UNIT=	Scalar integer expression	no default

Input/Output: OPEN Statement

- Examples

```
OPEN(UNIT=9; FILE= input_file, STATUS='OLD', IOSTAT=open_stat)
READ(UNIT=9, FMT='(2I6)', IOSTAT= read_stat) int1, int2

OPEN(UNIT=11, FILE=output_file, IOSTAT=open_stat)
WRITE(11,*) ' This is my output file ! '
```


Input/Output: Close Statement

- The CLOSE statement terminates the connection of file to a unit
- CLOSE specifiers

Specifier	Meaning
UNIT= scalar integer expression	unit number of the file
IOSTAT= integer variable	I/O Status
ERR= label	where to continue after error
STATUS= scalar default character expression (KEEP, DELETE)	specifies whether the file will exist after closing or not.

Input/Output: Close Statement

- Examples:

  CLOSE(11)

  CLOSE(9, STATUS='DELETE')

  CLOSE(15, IOSTAT=close_stat)

Input/Output: inquire 1

- inquire analyses the file status of a given file
- inquire(file=filename,specifier1,specifier2,....) for a given named file
  or
  inquire(unit=iounit,specifier1,specifier2,....) for a given unit number
- inquire(..,err=4711,...) on error jump to 4711
- specifier:
  access=acc
  'SEQUENTIAL', 'DIRECT','UNDEFINED'
  action=act
  'READ','WRITE','READWRITE','UNDEFINED'
  blank=blnk
  'NULL','ZERO','UNDEFINED'
  direct=dir
  'YES','NO','UNKNOWN'
  exist=ex
  .TRUE.,.FALSE.

Input/Output: unformatted IO

- formatted (ASCII) data are directly exchangeable between all systems. But the formatting process is compute intensive. Rounding of data necessary on reading and writing.
- unformatted IO is much faster for large amounts of data. Exchange of binary data. Because nearly all systems are using the IEEE-754 data standard, exchange is possible without accuracy loss.
- no standard in writing unformatted data; but exchange possible by no or minor modifications; conversion between Little and Big Endian may be a problem. No problem if pure data (only real*4, only integer*8, ..) are used

Input/Output: unformatted IO, example

- access to each record

```
open(unit=17,form='unformatted')
write(17) nmax
do n=1,nmax
        write(17) array(n)  !  new record control words
enddo
close(17)
```

- **much faster:**

```
open(unit=17,form='unformatted')
write(17) nmax
write(17) (array(n),n=1,nmax)
close(17)
```

Input/Output: inquire 2

- specifier:
  form=fm
  ‘FORMATTED‘, ‘UNFORMATTED‘,‘UNDEFINED‘
  formatted=fmt
  ‘YES‘,NO‘,‘UNKNOWN‘
  unformatted=unf
  ‘YES‘,‘NO‘,‘UNKNOWN‘
  name=nme
  ‘NULL‘,‘ZERO‘,‘UNDEFINED‘
  named=nmd
  .TRUE.,.FALSE.
  nextrec=nr
  next record number for direct access
  number=num
  unit of connected file or -1 if not connected
  opened=od
  .TRUE.,.FALSE.

Input/Output: inquire 3

- specifier:
  opened=od
  .TRUE.,.FALSE.
  pad=pd
  ‘YES‘,‘NO‘
  position=pos
  ‘REWIND‘,‘APPEND‘,‘ASIS‘,‘UNDEFINED‘
  read=rd
  ‘YES‘,‘NO‘,‘UNKNOWN‘
  readwrite=rdwr
  ‘YES‘,‘NO‘,‘UNKNOWN‘
  write=wr
  ‘YES‘,‘NO‘,‘UNKNOWN‘
  recl=rcl
  maximum allowed record length in bytes for formatted data
  in 4-bytes for unformatted

Input/Output: inquire 4

- specifier:
  sequential=seq
  ‘YES‘,‘NO‘,‘UNKNOWN‘

Input/Output 10: Overview over all I/O Statements

- I/O Statements

READ	Input statement dor data transfer
WRITE	Output statement for data transfer
PRINT	Output statement for data transfer
OPEN	Statement to connect a file
CLOSE	Statement to disconnect a file
INQUIRE	Statement to inquire a file
BACKSPACE	Statement for file positioning
ENDFILE	Statement for file positioning
REWIND	Statement for file positioning

Obsolete Features

- Storage association
  - EQUIVALENCE
  - COMMON BLOCKS
  - BLOCK DATA
  - ENTRY-Statements
- New obsolete Constructs
  - Do WHILE - Construct
- Old obsolete Constructs
  - Fixed source form
  - Computed GoTo
  - DOUBLE PRECISION
  - Arithmetic IF
  - Alternate return
  - RETURN and STOP statements
  - INCLUDE statement
  - ...

Summary - Part II

- Pointer in Fortran 90 is not a data object of a special type, pointer is an attribute.
- Three types of dynamic arrays:
  Array Pointer, Allocatable Arrays, Automatic Arrays
- With Array Syntax an array can be handled as a single data object
- It is possible to have **Optional Arguments** and use dummy argument names as **Keywords**
- **Hide Information** in modules via access attributes PUBLIC, PRIVATE
- Module access can be limited by the USE statement
- Fortran I/O is a large chapter