Fortran 95/2003 Course

Dynamic Data: Arrays and Pointers – Robert Barthel
March 25, 2015

STEINBUCH CENTRE FOR COMPUTING - SCC

Structuring Fortran Code

- Declaration section:
  - Nonexecutable statements

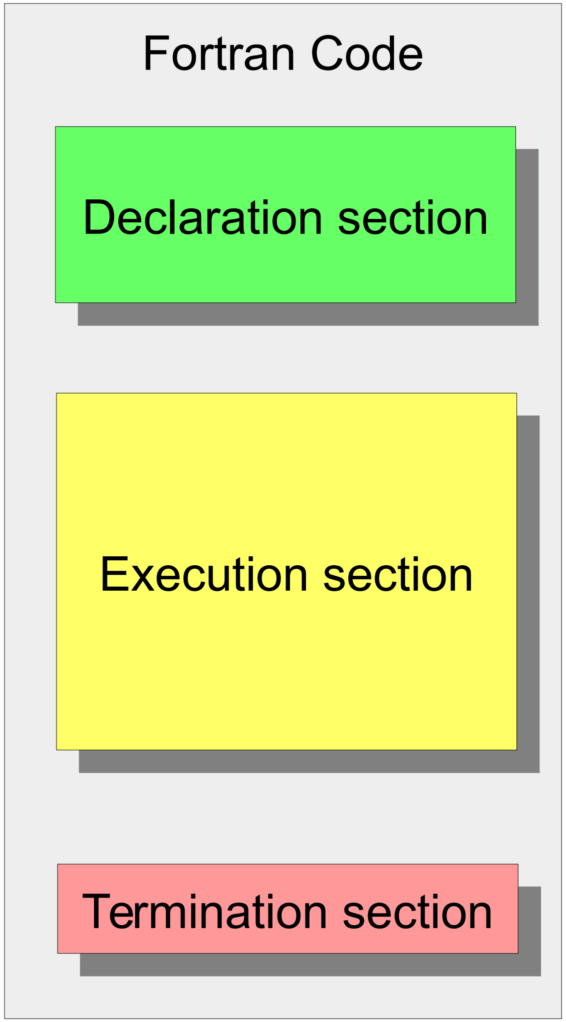
    → `DIMENSION, TYPE(), POINTER`

- Execution section:
  - Statements describing actions

    → **Array Constructors**

- Termination section:
  - Statements stopping execution

    `STOP, END`



Repeat: Good Programming Practice

- Indenting your block structures
- Comment & label your block structures

```
! Comment about loop labelled outer
outer: DO i = 1, dimA
    statements
    ! Comment about loop labelled inner
    inner: DO j = i, dimB
        statements
        ! Comment about if branching
        getlog: IF (a(i,j) >= 0.) THEN
            b(i,j) = log(a(i,j))
            statements
        ENDIF getlog
        statements
    ENDDO inner
    statements
ENDDO outer
```

Karlsruhe Institute of Technology

Array specifications

Array terminology

- Rank: Number of dimensions (Fortran 95 ≤ 7, Fortran 2003 ≤ 15)
- Extent: Number of elements in a dimension
- Shape: Vector of extents
- Size: Product of extents
- Conformance: Same shape

- Example:

```
REAL, DIMENSION(:,:) :: a(-3:4,7)
REAL, DIMENSION(:,:) :: b(8,2:8)
REAL, DIMENSION(:,:) :: d(8,1:8)
```

`a` has rank 2, extents 8 and 7, shape (/ 8, 7 /) and size 56

`a` is conformable with `b`, but not with `d`

Array specifications

```
type [[,DIMENSION (extent-list)] [,attribute]... ::]
  entity-list
```

where

- type — Intrinsic or derived type
- `DIMENSION` — Optional, required to define default dimensions
- (extent-list) — Gives array dimension (integer constant; integer expression using dummy arguments or constants; ":" if array is allocatable or assumed shape
- attribute — as given earlier
- entity-list — list of array names optionally with dimensions and initial values

Array specifications (2)

- If not explicitly changed, array indices start at **1**
- Example specifications
  - Two dimensions

```
REAL, DIMENSION (-3:4,7) :: ra, rb
```

  - Initialization

```
INTEGER, DIMENSION (3) :: ia = (/ 1,2,3 /),      &
                          ib = (/ (i, i=1,3) /)
```

Array specifications (3)

- Automatic arrays (*aka* local explicit-shape array with non-constant bounds):

```
SUBROUTINE sub1 (a,n,m)
Implicit none
integer, intent(in)                 :: n,m ! dummy arg.
real,intent(in),dimension(n,m)      :: a   ! dummy array
REAL, DIMENSION (n,m)               :: tmp ! automatic array
```

- Allocatable arrays (*aka* deferred shape arrays):

```
REAL, DIMENSION (:,:), ALLOCATABLE :: a, b
```

Dimensions defined in subsequent `ALLOCATE` statement

- Assumed shape arrays:

```
REAL, DIMENSION (:,:,:) :: a, b
```

Dimensions taken from actual arguments in calling routine

Array specifications (4)

- Automatic Arrays vs. Allocatable arrays

```
REAL, DIMENSION(dummy_arg):: a
```

```
REAL, DIMENSION(:), ALLOCATABLE :: a
```

- Automatic arrays
  → automatically allocated when procedure entered
- Allocatable arrays
  → more general & flexible (can be generated in differen routines)
  → can be resized during calculation (→ multiple purpose)

Array element order

- Arrays are stored column-wise (opposite to C, Matlab)

- Multidimensional arrays: first index varies fastest
- Example:

```
REAL, DIMENSION(3,3,2) :: b
```

The elements of b are stored in the order

```
b(1,1,1), b(2,1,1), b(3,1,1), b(1,2,1), b(2,2,1), b(3,2,1),
b(1,3,1), b(2,3,1), b(3,3,1), b(1,1,2), b(2,1,2), b(3,1,2),
b(1,2,2), b(2,2,2), b(3,2,2), b(1,3,2), b(2,3,2), b(3,3,2)
```

- Important cases where you need to know the storage order
  - I/O of arrays
  - Array constructors and array constants
  - Optimization (caching and locality)

Array operations

- Can use entire arrays in simple operations

```
c = a+b
```

- Very handy shorthand for nested loops

```
WRITE(*,*) c
```

- Arrays for whole array operation must be conformable
- Evaluate element by element, i.e. expressions evaluated before assignment
- `c = a*b` is not conventional matrix multiplication, but element multiplication → $a_1*b_1 \ldots a_n*b_n$

Array operations (2)

- r.h.s of array syntax expression

  $\longrightarrow$ completely computed before any assignment takes place

  $\rightarrow$ So be careful when the same array appears on both sides of the equal sign

- Scalar broadcast: Scalar is transformed into a conformable array with all elements equaling itself

```
b = a+5
```

Array operations (3)

- Fortran 77

```
      REAL a(20), b(20), c(20)
      DO 10 i = 1,20
        a(i) = 0.0
   10 CONTINUE
      DO 20 i = 1,20
        a(i) = a(i) / 3.1 + b(i) * SQRT(c(i))
   20 CONTINUE
```

- Fortran 90

```
REAL, DIMENSION(20) :: a, b, c
a = 0.0
a = a / 3.1 + b * SQRT(c)
```

Array operations (4)

- Fortran 77

```
      REAL a(5,5), b(5,5), c(5,5)
     DO 20 i = 1,5
       DO 10 i = 1,5
         c(i,j) = a(i,j)+b(i,j)
  10   CONTINUE
  20 CONTINUE
```

- Fortran 90

```
REAL, DIMENSION(5,5) :: a, b, c
c = a+b
```

Array operations with intrinsic procedures

- Elemental procedures specified for array arguments

- May also be applied to conforming array arguments

- Examples
  - To find the square root of all elements of array `a`

```
a = SQRT(a)
```

  - To find the string length excluding trailing blanks for all elements of a character array `words`

```
words = (/'ab   ','abc  ','abcd ','abcde'/))
write(*,*) LEN_TRIM(words)
```

→

```
2  3  4  5
```

`Repeat:` `WHERE` statement (1)

- Form

```
WHERE (logical-array-expr) array-assignment
```

- Operation: Assignment is performed if logical condition is true (element by element)

```
REAL, DIMENSION(5,5) :: ra, rb
WHERE (rb > 0.0) ra = ra/rb
```

→ Note: Mask (`rb>0.0`) must conform with left hand side `ra`

→ Equivalent to:

```
DO j = 1,5
  DO i = 1,5
    IF (rb(i,j) > 0.0) ra(i,j) = ra(i,j)/rb(i,j)
  END DO
END DO
```

`Repeat:` `WHERE` construct (2)

- Used for multiple assignments

```
WHERE (logical-array-expr)
  array-assignments
END WHERE
```

- Used for `IF/ELSE` decision making

```
WHERE (logical-array-expr)
  array-assignments
ELSEWHERE
  other-array-assignments
END WHERE
```

Example:

```
REAL, DIMENSION(5,5) :: ra,rb
WHERE (rb > 0.0)
  ra = ra/rb
ELSEWHERE
  ra = 0.0
END WHERE
```

Array sections

- A subarray, called a section, of an array may be referenced by specifying a range of subscripts, either
  - a simple subscript

```
a(2,3,1)  ! single array element
```

  - a subscript triplet

```
[lower bound]:[upper bound][:stride]
```

```
a(1:6:2)
```

```
→ a(1),a(3),a(5)
```

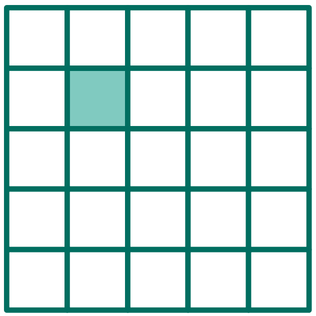
  - a vector subscript

```
a(int_vector)
```

- Array sections can also be used in array operations like whole arrays

Array sections (2)

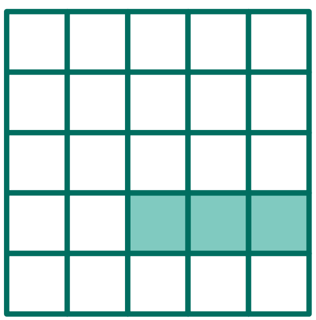
```
REAL, DIMENSION(5,5) :: ra
```



`ra(2,2)` or `ra(2:2:1,2:2:1)`

= an array element

shape (/ 1 /)



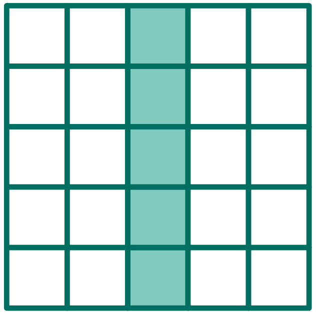
`ra(4,3:5)`

= sub-row

shape (/ 3 /)

Array sections (3)

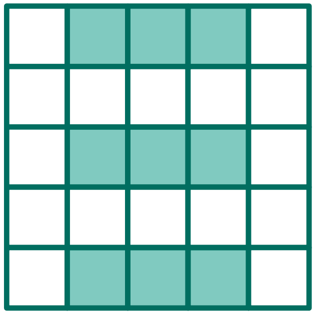
```
REAL, DIMENSION(5,5) :: ra
```



`ra(:,3)`

= whole column

shape (/ 5 /)



`ra(1::2,2:4)`

= repeating subrows

shape (/ 3,3 /)

Vector subscripts (1)

- 1D integer array used as an array of subscripts

  `(/ 3, 2, 12, 2, 1 /)`

- Example:

```
REAL, DIMENSION(:)     :: ra(6), rb(3)
INTEGER, DIMENSION(3) :: iv
iv = (/ 1, 3, 5 /)                ! initialize iv
ra = (/ 1.2, 3.4, 3.0, 11.2, 1.0, 3.7 /)
rb = ra(iv)                       ! iv is the vector subscript
```

Last line equivalent to:

```
rb(1) = ra(1)  ← 1.2
rb(2) = ra(3)  ← 3.0
rb(3) = ra(5)  ← 1.0
```

Vector subscripts (2)

- Vector subscript can be on left hand side of expression

```
iv = (/ 1, 3, 5 /)
ra = 0.0
ra(iv) = (/ 1.2, 3.4, 5.6 /)
! same as ra( (/ 1, 3, 5 /) ) = (/ 1.2, 3.4, 5.6 /)
write(*,*) ra
```

```
→ 1.2   0.0   3.4   0.0   5.6
```

- Must not repeat values of elements on left hand side!

```
iv = (/ 1, 3, 1 /)

ra(iv) = (/ 1.2, 3.4, 5.6 /) ! NOT permitted
! tries to be ra( (/ 1, 3, 1 /) ) = (/ 1.2, 3.4, 5.6 /)
```

Array section assignments

- Operands must be conformable

```
REAL, DIMENSION(5,5) :: ra,rb,rc
INTEGER :: id
…
ra = rb + rc*id
! shape (/ 5,5 /)

ra(3:5,3:4) = rb(1::2,3:5:2) + rc(1:3,1:2)
! shape (/ 3,2 /)

ra(:,1) = rb(:,1) + rb(:,2) + rb(:,3)
! shape (/ 5,1 /)
```

Array constructor (1)

- Already used on previous slides: Method to explicitly create and fill up a 1D array
- Construction of rank 1 array

```
REAL, DIMENSION(6) :: a
a = (/ array-constructor-value-list /)
```

- where `array-constructor-value-list` can be
  - Explicit values

    ```
    (/ 1.2, 3.4, 3.0, 11.2, 1.0, 3.7 /)
    ```

  - Array sections

    ```
    (/ b(i,2),b(i,3),b(i,4),b(1,i+3),b(3,i+3),b(5,i+3) /)
    ```

Array constructor (2)

where `array-constructor-value-list` can be (cont.)

- Implied `DO`-lists

```
(/ ((i+j, i=1,3), j=1,2) /)
! = (/ 2, 3, 4, 3, 4, 5 /)
```

- Arithmetic expressions

```
(/ (1.0/REAL(i), i=1,6) /)
! = (/ 1.0/1.0, 1.0/2.0, 1.0/3.0,
!       1.0/4.0, 1.0/5.0, 1.0/6.0 /)
! = (/ 1.0, 0.5, 0.333, 0.25, 0.20, 0.167 /)
```

Intrinsic function RESHAPE

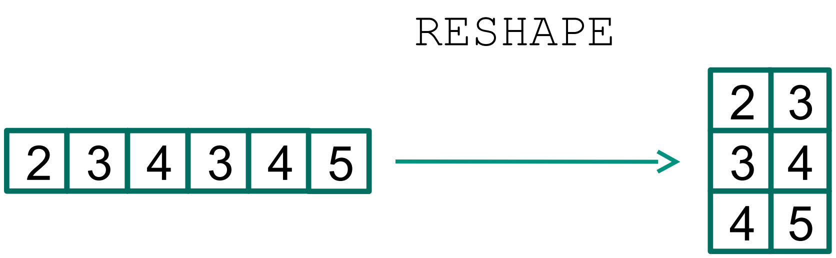
- Change shape of array after having used array constructor to create 1D array
- Syntax:

```
RESHAPE (SOURCE,SHAPE [,PAD] [,ORDER])
```

→ takes an array SOURCE and returns a new array with the elements of SOURCE rearranged to form an array of shape SHAPE

- Example

```
REAL, DIMENSION(3,2) :: ra
ra = RESHAPE((/ ((i+j,i=1,3),j=1,2) /), SHAPE=(/ 3,2 /))
```



Dynamic Arrays

Dynamic arrays

- Fortran 77:
  - Static (fixed) memory allocation at compile time
- Fortran 90:
  - Allocate and deallocate storage as required via allocatable arrays (done during run time)
  - Allow local arrays in a procedure to have different size and shape every time the procedure is invoked via automatic arrays
  - Reduce overall storage requirement
  - Simplify subroutine arguments

Allocatable arrays

- A run-time array which is declared with the `ALLOCATABLE` attribute

```
ALLOCATE (allocate_object_list [, STAT=status])
```

```
DEALLOCATE (allocate_object_list [, STAT=status])
```

- How to use `STAT`:
  → If `STAT=` present, status =0 (success) or status >0 (error)

  → If `STAT=` is not present and an error occurs, the program execution aborts

- Intrinsic function `ALLOCATED` determines status
- Allocatable arrays are created on the **heap** on conventional computers

Allocatable arrays (2)

- Example

```
REAL, DIMENSION(:,:), ALLOCATABLE :: ra
INTEGER :: status, nsize1, nsize2

READ *, nsize1, nsize2

ALLOCATE (ra(nsize1,nsize2), STAT=status)
IF (status > 0) (Error processing code)
...
IF (ALLOCATED(ra)) DEALLOCATE (ra)
```

Automatic arrays (1)

- Typically used as scratch storage within a procedure

```
SUBROUTINE sub1 (a,n,m)
Implicit none
integer, intent(in)              :: n,m ! dummy arg.
real,intent(in),dimension(n,m) :: a   ! dummy array
REAL, DIMENSION (n,m)           :: tmp ! automatic array
```

→ Bounds given when invoking procedure via e.g. dummy arguments

→ Created automatically when entering a subprogram
→ Destroyed when exiting a subprogram
→ Created on the stack on conventional computers
→ May be faster than allocatable arrays

- F95: no checking of sufficient memory for automatic arrays!

Automatic arrays (2)

- Example 1: Bounds of automatic arrays depend on dummy arguments (`work1` and `work2` are the automatic arrays)

```
SUBROUTINE sub(n,a)
  IMPLICIT NONE
  INTEGER :: n
  REAL, DIMENSION(n,n) :: a
  REAL, DIMENSION(n,n) :: work1
  REAL, DIMENSION(SIZE(a,1)) :: work2

  ...
END SUBROUTINE sub
```

Automatic arrays (3)

- Example 2: Bounds of an automatic array are defined by the global variable in a module

```
MODULE auto_mod
  INTEGER :: n
CONTAINS
  SUBROUTINE sub
    REAL, DIMENSION(n) :: w
    PRINT *, 'Bounds and size of w: ', &
      LBOUND(w), UBOUND(w), SIZE(w)
  END SUBROUTINE sub
END MODULE auto_mod

PROGRAM auto_arrays
  USE auto_mod
  n = 10
  CALL sub
END PROGRAM auto_arrays
```

Assumed shape arrays

- Shape of actual and dummy array arguments must agree (in all Fortran versions)

```
REAL, DIMENSION (:,:,:) :: a, b
```

- F90: not necessary to pass array dimensions
  → Assumed array shape uses dimension of actual arguments
  → Can specify lower bound of the assumed shape array
  → External procedures: one must provide an INTERFACE block

Assumed shape arrays (2)

```
… ! calling program unit
INTERFACE
  SUBROUTINE sub (ra, rb, rc)
    REAL, DIMENSION (:, :) :: ra, rb
    REAL, DIMENSION (0:, 2:) :: rc
  END SUBROUTINE sub
END INTERFACE
REAL, DIMENSION (0:9,10) :: ra ! Shape (/ 10, 10 /)
CALL sub(ra, ra(0:4, 2:6), ra(3:7, 5:9))
…
SUBROUTINE sub(ra, rb, rc) ! External
  REAL, DIMENSION (:, :) :: ra ! Shape (/10, 10/)
  REAL, DIMENSION (:, :) :: rb ! Shape (/ 5, 5 /)
  ! = REAL, DIMENSION (1:5, 1:5) :: rb
  REAL, DIMENSION (0:, 2:) :: rc ! Shape (/ 5, 5 /)
  ! = REAL, DIMENSION (0:4, 2:6) :: rc
  …
END SUBROUTINE sub
```

Karlsruhe Institute of Technology

Pointers

Pointer

= address of variable (**target**)
→ contains no data at all

} i.e. stores in its memory location the address of ordinary variable

- Declaration via:
  - type attribute: pointer ↔ target (but not both!)
- Assignment syntax:

```
pointer => target
```

```
allocate(pointer)
```

- Use of Pointers
  - more flexible alternative to allocated arrays
  - linked lists and other dynamic data structures (binary trees)

Pointer specifications

- `type [[, attribute] … ::] list of variables`

  where attribute must include
  - `POINTER` for a pointer variable, or
  - `TARGET` for a target variable

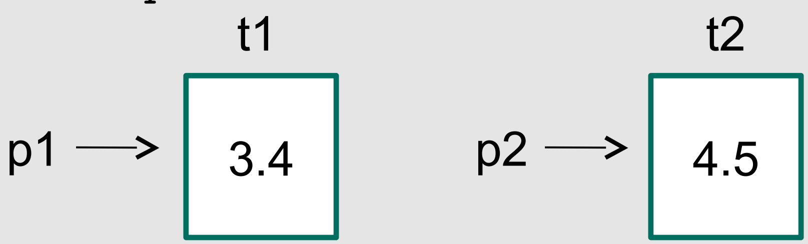
- The type, type parameters and rank of a pointer must be the same as the type and rank of any target it points to

- If a pointer is an array pointer, only the rank, not the shape can be defined

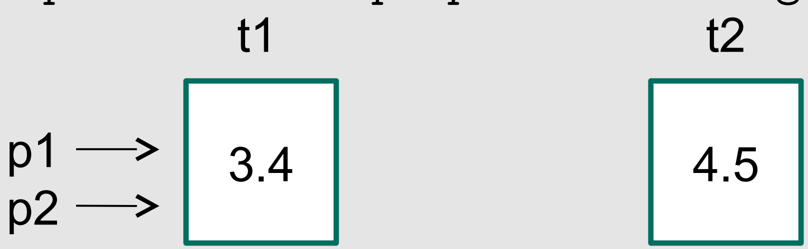
```
REAL, DIMENSION(:), POINTER :: p  ! legal
REAL, DIMENSION(20), POINTER :: p ! illegal
```

Pointer assignment

```
REAL, POINTER :: p1, p2
REAL, TARGET :: t1 = 3.4, t2 = 4.5
p1 => t1; p2 => t2
```




```
PRINT *, t1, p1  ! 3.4 printed out twice
PRINT *, t2, p2  ! 4.5 printed out twice
p2 => p1  ! valid: p2 points to target of p1
```




```
PRINT *, t1, p1, p2  ! 3.4 printed out 3 times
```

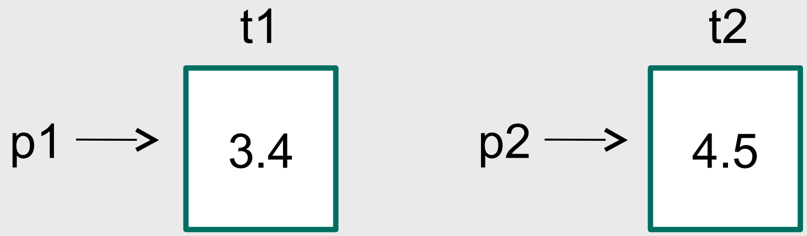
- Associated pointer:

  → returns value of the target variable

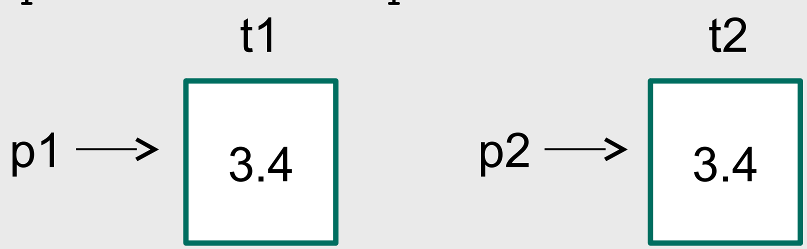
  → pointer just acts as an **alias** for the target variable

Pointer assignment (2)

```
REAL, POINTER :: p1, p2
REAL, TARGET :: t1 = 3.4, t2 = 4.5
p1 => t1; p2 => t2
```




```
PRINT *, t1, p1  ! 3.4 printed out twice
PRINT *, t2, p2  ! 4.5 printed out twice
p2 = p1  ! valid: equivalent to t2=t1
```




```
PRINT *, t1, t2, p1, p2  ! 3.4 printed out four times
```

Array pointers

- Target of a pointer can be an array

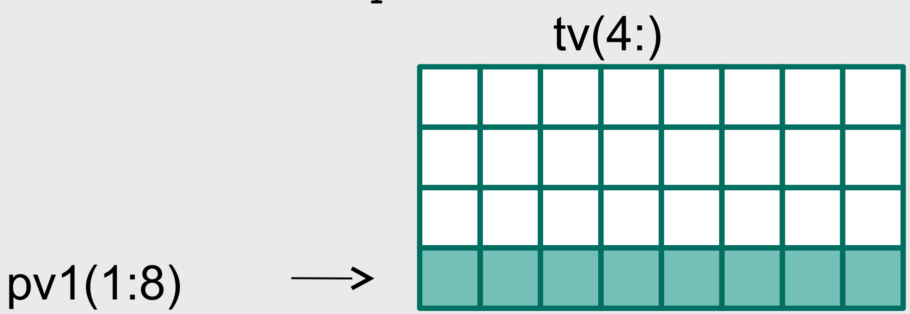
```
REAL, DIMENSION(:), POINTER   :: pv1
REAL, DIMENSION(-3:5), TARGET :: tv1
pv1 => tv1           ! pv1 aliased to tv1
```


```
pv1 => tv1(:)        ! aliased with section subscript
```

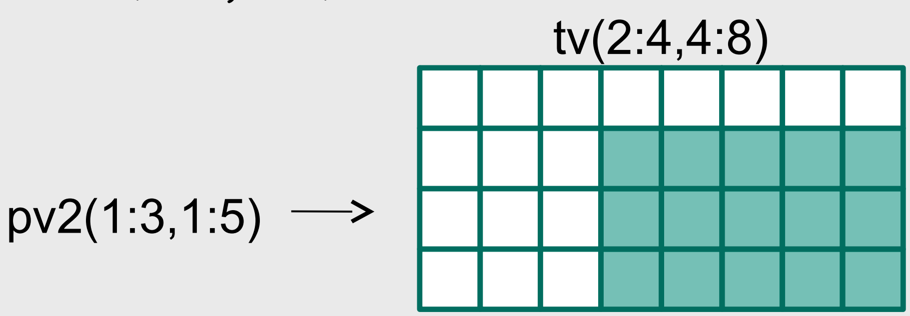

```
pv1 => tv1(1:5:2) ! aliased with section triplet
```

Array pointers (2)

```
REAL, DIMENSION(:), POINTER   :: pv1
REAL, DIMENSION(:,:), POINTER :: pv2
REAL, DIMENSION(4,8), TARGET  :: tv
pv1 => tv(4,:)   ! pv1 aliased to 4th row of tv
```




```
pv2 => tv(2:4,4:8)
```



Pointer status

- Undefined at start of program (after declaration)
- Must not reference undefined pointer

- Defining status with:
  → `NULLIFY` or `ptr=>null()`
  → `ALLOCATE`

- Test status with: `ASSOCIATED` intrinsic function

```
REAL, POINTER :: p        ! p undefined
REAL, TARGET  :: t
PRINT *, ASSOCIATED(p)    ! not valid
NULLIFY(p)                ! point at "nothing"
PRINT *, ASSOCIATED(p)    ! .FALSE.
p => t
PRINT *, ASSOCIATED(p)    ! .TRUE.
```

Dynamic storage for pointers

- Can **allocate** storage for a pointer:

  → creates an unnamed variable/array of specified size & implied target attribute

```
REAL, POINTER :: p
REAL, DIMENSION(:,:), POINTER :: pv
INTEGER :: m,n
ALLOCATE (p, pv(m,n))
```

- Can **release** storage when no longer required:

```
DEALLOCATE (pv)   ! pv is in null status
```

- Value assignment only possible after:

  `ALLOCATE` or pointer assignment statement

```
REAL, POINTER :: p
p = 3.4

→ Segmentation fault,
invalid memory reference
```

Potential problems with pointers

Karlsruhe Institute of Technology

- Dangling pointer

```
REAL, POINTER :: p1,p2
ALLOCATE (p1)
p1 = 3.4
p2 => p1
DEALLOCATE (p1)
```

Dynamic variable p1 and p2 both pointed to is gone.

Reference to p2 now gives unpredictable results!

- Unreferenced storage

```
REAL, DIMENSION(:), POINTER :: p
ALLOCATE (p(1000))
NULLIFY(p)
```

Nullify p without first deallocating it.

Big block of memory not released and unusable!

Using Pointers in Procedures (1)

→ as dummy or actual arguments

→ as function result

```
SUBROUTINE sub1(ptr)
REAL, DIMENSION(:), POINTER :: ptr
...
END SUBROUTINE sub1
```

- Restrictions:
  - Dummy arguments has attribute `POINTER, TARGET`
    → procedure must have an explicit `INTERFACE` (construct)
  - Dummy argument is `POINTER`
    → passed argument must be `POINTER` of same `TYPE, KIND` & `RANK`
  - Pointer dummy argument
    → can not have attribute `INTENT`

- Combinations:
  - Pointer actual argument + ordinary dummy argument
    → dummy argument becomes associated with target of pointer

Using Pointers in Procedures (2)

```
INTERFACE ! do not forget interface in calling unit
  SUBROUTINE sub2(b)
    REAL, DIMENSION(:,:), POINTER :: b
  END SUBROUTINE sub2
END INTERFACE
REAL, DIMENSION(:,:), POINTER :: p
ALLOCATE (p(50,50))
CALL sub1(p) ! both sub1 and sub2
CALL sub2(p) ! are external procedures
...
SUBROUTINE sub1(a) ! a is not a pointer
  REAL, DIMENSION(50,50) :: a
  ...
END SUBROUTINE sub1
...
SUBROUTINE sub2(b) ! b is a pointer
  REAL, DIMENSION(:,:), POINTER :: b
  DEALLOCATE(b)
END SUBROUTINE sub2
```

Pointer-valued Functions

- A function result may also have the `POINTER` attribute
- Useful if the result size depends on calculations performed in the function
- The result can be used in an expression, but must be associated with a defined target

```
INTEGER, DIMENSION(100) :: x
INTEGER, DIMENSION(:), POINTER :: p
p => gtzero(x)
...
CONTAINS
  ! function to get all values .gt. 0 from a
FUNCTION gtzero(a)
  INTEGER, DIMENSION(:), POINTER :: gtzero
  INTEGER, DIMENSION(:) :: a
  INTEGER :: n
  … ! n = find number of values .gt. 0
  ALLOCATE (gtzero(n))
  … ! put found values into gtzero
END FUNCTION gtzero
...
```

Adv: Array of pointers

- An array of pointers cannot be declared directly

```
REAL, DIMENSION(20), POINTER :: p ! illegal
```

- An array of pointers can be simulated by means of a derived type having a pointer component:

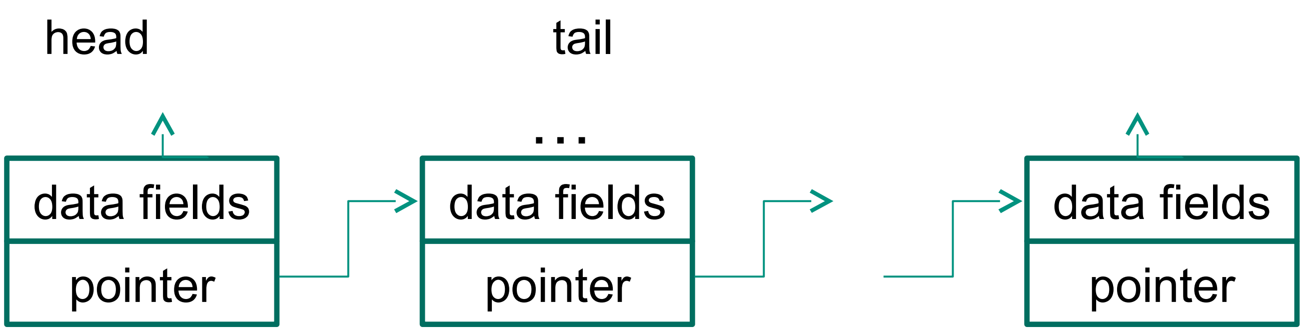
```
TYPE cell
  REAL, DIMENSION(:), POINTER :: column
END TYPE cell
TYPE (cell), DIMENSION(:), POINTER :: matrix
INTEGER, DIMENSION(100) :: rows
INTEGER :: i,j,n
READ *, n, (rows(j), j=1,n)
ALLOCATE (matrix(1:n))
DO j = 1, n
  ALLOCATE (matrix(j)%column(1:rows(j)))
END DO
```

Linked list

- A pointer component of a derived type can point at an object of the same type; this enables a linked list to be created

```
TYPE node
  INTEGER :: value ! data field
  TYPE (node), POINTER :: next ! pointer field
END TYPE node
```

- A linked list typically consists of objects of a derived type containing fields for the data plus a field that is a pointer to the next object of the same type in the list



Linked list (2)

- Dynamic alternative to arrays

- In a linked list, the connected objects
  - are not necessarily stored contiguously
  - can be created dynamically at execution time
  - may be inserted at any position in the list
  - may be removed dynamically

- The size of a list may grow to an arbitrary size as a program is executing

- Trees or other dynamic data structures can be constructed in a similar way

Linked list (3)

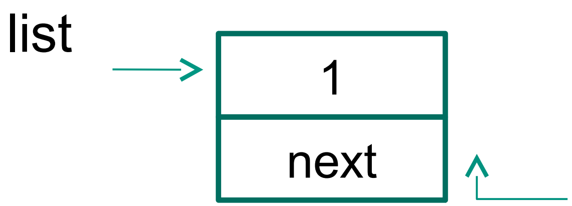
```
TYPE node
  INTEGER :: value                    ! data field
  TYPE (node), POINTER :: next        ! pointer field
END TYPE node
TYPE (node), POINTER :: list, current
INTEGER :: num

NULLIFY(list)        ! initially nullify list (mark its end)
DO
  READ *, num               ! read num from keyboard
  IF (num == 0) EXIT        ! until 0 is entered
  ALLOCATE (current)        ! create new node
  current%value = num
  current%next => list      ! point to previous one
  list => current           ! update head of list
END DO
...
```

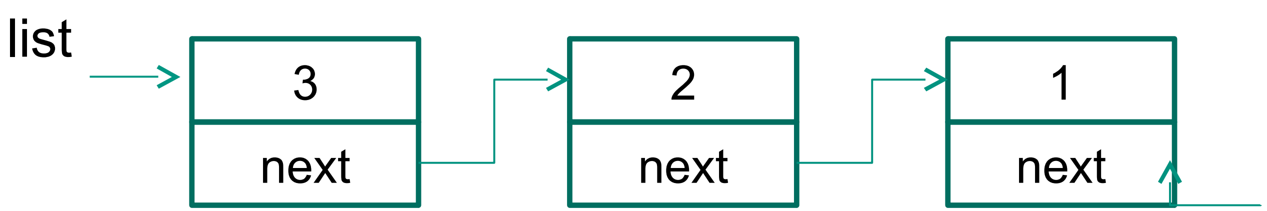
Linked list (4)

- If, for example, the values 1, 2, 3 are entered in that order, the list looks like (progressively):
  - After `NULLIFY(list)`

  - After the first `num` is read



  - After all 3 numbers are read



Karlsruhe Institute of Technology

Thank you for your attention!