
Fortran 90 Features

**Science & Technology Support
High Performance Computing**

Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212-1163



Table of Contents

Preliminaries

Source, Types and Control Structures

Procedures and Modules

Array Processing

Pointers

New I/O Features

Intrinsic Procedures

Preliminaries

[Objectives of Fortran 90](#)

[Major new features](#)

[Other new features](#)

[References](#)

[Resource list](#)

[Coding convention](#)

[Acknowledgments](#)

Fortran 90 Objectives

Language evolution

- Obsolescent features
- Keep up with "rival" languages: C, C++

Standardize vendor extensions

- Portability

Modernize the language

- Ease-of-use improvements through new features such as free source form and derived types
- Space conservation of a program with dynamic memory allocation
- Modularisation through defining collections called modules
- Numerical portability through selected precision

Fortran 90 Objectives (cont.)

Provide data parallel capability

- Parallel array operations for better use of vector and parallel processors
- High Performance Fortran built on top of Fortran 90

Compatibility with Fortran 77

- Fortran 77 is a subset of Fortran 90

Improve safety

- Reduce risk of errors in standard code (improved argument checking)

Standard conformance

- Compiler must report non standard code and obsolescent features

Major new features

Free-Form source style

- Forget about those column numbers!

User-defined Data Types

- Heterogeneous data structure: composed of different "smaller" data types

Array processing

- Shorthand for loop nests and more....

Dynamic memory allocation

- Like C malloc & free

Subprogram Improvements

- Function/Subroutine Interface (like C Function Prototype)
- Optional/Keyword Arguments

Major new features (cont.)

Modules

- Replace COMMON block approach to make "global" variables
- Similar to C++ classes

Generic Procedures

Pointers

- Long been needed

Other new features

Specifications/ `IMPLICIT NONE`

Parameterized data types (`KIND`)

- Precision Portability

Operator overloading/defining

- Like in C++

Recursive Functions

New control structures

Other new features (cont.)

New intrinsic functions

- A fairly large library

New I/O features

Obsolescent features

May be removed at the next major revision

- Arithmetic `IF`
- `REAL` and `DOUBLE` precision `DO` variables and control expressions
- Shared `DO` termination, and `DO` termination on a statement other than on a `CONTINUE` or an `END DO` statement
- `ASSIGN` and assigned `GOTO` statements
- Assigned `FORMAT` specifiers
- Branching to `END IF` from outside `IF` block
- Alternate `RETURN`
- `PAUSE` statement
- `H` edit descriptor

References

Information Technology - Programming Languages - Fortran
on-line or paper, the official standard, ISO/IEC 1539:1991 or
ANSI x3.198-1992

Fortran 90 Handbook, J C Adams et. al., McGraw-Hill, 1992
Complete ANSI/ISO Reference

Programmer's Guide to Fortran 90, 2nd edition, WS Brainerd et. al.,
Unicomp, 1994

Fortran 90, M Counihan, Pitman, 1991

References (cont.)

Fortran 90 Programming, TMR Ellis et. al., Wesley, 1994
Fortran 90 for Scientists and Engineers, BD Hahn, Edward Arnold, 1994

Migrating to Fortran 90, J Kerrigan, O'Reilly and Associates, 1993

Fortran 90 Explained, M Metcalf & J Reid, Oxford University Press

Programming in Fortran 90, JS Morgan & JL Schonfelder, Alfred
Waller Ltd., 1993

Programming in Fortran 90, IM Smith, Wiley

WWW Resources

Fortran 90 FAQ

Tutorials and articles available

NAG Fortran 90 Software Repository

Fortran 90 / HPF email discussion group

Fortran newsgroup in Netnews/Usenet (comp.lang.fortran)

Coding convention

Put all Fortran 90 keywords and intrinsic function names in upper case, everything else in lower case. F90 is NOT case sensitive

Indent by 2 columns in the body of program units and `INTERFACE` blocks, `DO` loops, `IF` blocks, `CASE` blocks, etc.

Always include the name of a program, a subroutine and a function on its `END` statement

In `USE` statements, use the `ONLY` clause to document explicitly all entities which are actually accessed from that module

In `CALL` statement and function references, always use argument keywords for optional arguments

Acknowledgments

- Dr. Dave Ennis for developing this course
- The Manchester and North HPC: Training and Education Centre
 - For material on which part of this course is based
 - Walt Brained, Unicomp, Inc. who contributed to this material and it is used with his permission
- Michel Olnagion's Fortran 90 List for links to the Fortran 90 compiler vendors

Sources, Types and Control Structures

Source form

Specifications

IMPLICIT NONE

Kind values

Derived types

Control structures

Free source form

Type in any column you want!

Line lengths up to 132 columns

Lowercase letters permitted

Names up to 31 characters (including underscore)

Semicolon to separate multiple statements on one line

Don't confuse with C's use of ; to mark the end of a statement

Comments may follow exclamation (!)

Free source form (cont.)

Ampersand (&) is a continuation symbol

Character set includes + < > ; ! ? % - " &

New C-like relational operators: '<', '<=', '==', '/=', '>=', '>'

Free Form Example

```
PROGRAM free_source_form
  ! Long names with underscores
  ! No special columns

  IMPLICIT NONE

  ! upper and lower case letters
  REAL :: tx, ty, tz          ! trailing comment

  ! Multiple statements per line
  tx = 1.0; ty = 2.0;
  tz = tx * ty

  ! Continuation symbol on line to be continued
  PRINT *, &
         tx, ty, tz

  END PROGRAM free_source_form
```

Specifications

type [[, attribute]... ::] entity list

type can be INTEGER, REAL, COMPLEX, LOGICAL or CHARACTER with optional kind value:

INTEGER ([KIND=] kind-value)

CHARACTER ([actual parameter list]) ([LEN=] len-value and/or [KIND=] kind-value)

TYPE (type name)

attribute can be PARAMETER, PUBLIC, PRIVATE, ALLOCATABLE, POINTER, TARGET, INTENT(inout), DIMENSION (extent-list), OPTIONAL, SAVE, EXTERNAL, INTRINSIC

Specifications (cont.)

Can initialize variables in specifications

```
INTEGER :: ia, ib
INTEGER, PARAMETER :: n=100, m=1000
REAL :: a = 2.61828, b = 3.14159
CHARACTER (LEN = 8) :: ch
INTEGER, DIMENSION(-3:5, 7) :: ia
```

IMPLICIT NONE

In Fortran 77, implicit typing permitted use of undeclared variables.
This has been the cause of many programming errors.

IMPLICIT NONE forces you to declare all variables.

- As is naturally true in other languages

IMPLICIT NONE may be preceded in a program unit only by USE
and FORMAT statements (see Order of statements).

Kind values

5 intrinsic types:

`REAL`, `INTEGER`, `COMPLEX`, `CHARACTER`, `LOGICAL`

Each type has an associated non negative integer value called the `KIND` type parameter

Useful feature for writing portable code requiring specified precision

A processor must support:

- at least 2 kinds for `REAL` and `COMPLEX`
- 1 kind for `INTEGER`, `LOGICAL` and `CHARACTER`

Many intrinsics for inquiring about and setting kind values

Kind values: REAL

REAL (KIND = wp) :: ra ! or

REAL(wp) :: ra

Declare a real variable, `ra`, whose precision is determined by the value of the kind parameter, `wp`

Kind values are system dependent

An 8 byte (64 bit) real variable usually has kind value 8 or 2

A 4 byte (32 bit) real variable usually has kind value 4 or 1

Literal constants set with kind value: `const = 1.0_wp`

Kind values: REAL (cont.)

Common use is to replace DOUBLE PRECISION:

```
INTEGER, PARAMETER :: idp = KIND(1.0D0)
REAL (KIND = idp) :: ra
```

`ra` is declared as 'double precision', but this is system dependent

To declare `real` in system independent way, specify kind value associated with precision and exponent range required:

```
INTEGER, PARAMETER :: i10=SELECTED_REAL_KIND(10, 200)
REAL (KIND = i10) :: a, b, c
```

`a`, `b` and `c` have at least 10 decimal digits of precision and the exponent range 200 on any machine.

Kind values: INTEGER

Integers usually have 16, 32 or 64 bit

16 bit integer normally permits $-32768 < i < 32767$

Kind values for each supported type

To declare integer in system independent way, specify kind value associated with range of integers required:

```
INTEGER, PARAMETER :: i8=SELECTED_INT_KIND(8)  
INTEGER (KIND = i8) :: ia, ib, ic
```

i_a , i_b and i_c can have values between 10^8 and -10^8 at least (if permitted by processor)

Kind values: Intrinsic Functions

```
INTEGER, PARAMETER :: i8 = SELECTED_INT_KIND(8)  
INTEGER (KIND = i8) :: ia
```

```
PRINT *, HUGE(ia), KIND(ia)
```

This will print the largest integer available for this integer type (2147483674), and its kind value.

```
INTEGER, PARAMETER::i10 = SELECTED_REAL_KIND(10, 200)  
REAL (KIND = i10) :: a
```

```
PRINT *, RANGE(a), PRECISION(a), KIND(a)
```

This will print the exponent range, the decimal digits of precision and the kind value of a.

Derived types

Defined by user (also called structures)

Can include different intrinsic types and other derived types

– Like C structures and Pascal records

Components accessed using percent operator (%)

Only assignment operator (=) is defined for derived types

Can (re)define operators - see later operator overloading

Derived types: Examples

Define the form of derived type:

```
TYPE card
  INTEGER :: pips
  CHARACTER (LEN = 8) :: suit
END TYPE card
```

Create the structures of that type:

```
TYPE (card) :: card1, card2
```

Assign values to the structure components:

```
card1 = card(8, 'Hearts')
```

Derived types: Examples (cont.)

Use % to select a component of the structure

```
print *, "The suit of the card is ", card1%suit
```

Assigning structures to each other done component by component

```
card2=card1 ! card2%pips would get 8
```

Arrays of derived types are possible:

```
TYPE (card), DIMENSION (52) :: deck  
deck(34)=card(13, "Diamonds")
```

Control structures

Three block constructs:

IF

DO

CASE (new to Fortran 90)

All can be nested

All may have construct names to help readability or to increase flexibility

IF..THEN..ELSE Statement

General form:

```
[name:] IF (logical expression) THEN
```

```
    block
```

```
[ELSE IF (logical expression) THEN [name]
```

```
    block]...
```

```
[ELSE [name]
```

```
    block]
```

```
END IF [name]
```

IF..THEN..ELSE Statement (cont.)

Example:

```
selection: IF (i < 0) THEN
```

```
        CALL negative
```

```
ELSE IF (i == 0) THEN
```

```
        CALL zero
```

```
ELSE
```

```
        CALL positive
```

```
END IF selection
```

DO Loops

General form:

```
[name:] DO [control clause]
```

```
    block
```

```
END DO [name]
```

Control clause may be:

- an iteration control clause `count = initial, final [,inc]`
- a `WHILE` control clause `WHILE (logical expression)`
- or nothing (no control clause at all)

DO Loops (cont.)

Iteration control clause:

```
rows: DO i = 1, n
cols:   DO j = 1, m
        a(i, j) = i + j
        END DO cols
      END DO rows
```

WHILE control clause:

```
true: DO WHILE (i <= 100)
      ... body of loop ...
    END DO true
```

DO loops: EXIT and CYCLE Features

Use of EXIT and CYCLE:

- smooth exit from loop with EXIT
- transfer to END DO with CYCLE (i.e., skip the rest of the loop)
- EXIT and CYCLE apply to inner loop by default but can refer to specific, named loop

Example:

```
DO
  READ *, number
  IF (number == 0.0) EXIT
  IF (number <= 0.0) CYCLE
  sum = sum + SQRT(number)
END DO
```

CASE construct

Structured way of selecting different options, dependent on value of single expression

- Similar to C `switch` statement

Replacement for:

- `computed GOTO`
- "else if" ladders

General form:

```
[name:] SELECT CASE (expression)
[CASE (selector) [name]
    block]
    ...
END SELECT [name]
```

CASE construct (cont.)

General form:

```
[name:] SELECT CASE (expression)
[CASE (selector) [name]
        block]
        ...
END SELECT [name]
```

`expression` = character, logical or integer

`selector` = one or more values of same type as `expression`:

- single value
- range of values separated by: (character or integer only), upper or lower value may be absent
- list of values separated by commas
- keyword `DEFAULT`

Example

```
SELECT CASE (ch)

    CASE ('C', 'D', 'G':'M')
        color = 'red'
    CASE ('X':)
        color = 'green'
    CASE DEFAULT
        color = 'blue'

END SELECT
```

Procedures and Modules

Program units

Procedures

Internal procedures

INTERFACE blocks

Procedure arguments

Array-valued functions

Recursive procedures

Generic procedures

Modules

Overloading operators

Defining operators

Assignment overloading

Program Structure

Program Units

Main Program

- Form:

```
PROGRAM [name]
```

```
[specification statements]
```

```
[executable statements]
```

```
...
```

```
END [PROGRAM [name]]
```

Main Program

Form:

```
PROGRAM [name]
  [specification statements]
  [executable statements]
  ...
END [PROGRAM [name]]
```

Example:

```
PROGRAM test
  ...
  ...
! END
! END PROGRAM
END PROGRAM test
```

Procedures: Functions and Subroutines

Structurally, procedures may be:

- **External** - self contained (not necessarily Fortran, but good luck with type conversion for arguments)
- **Internal** - inside a program unit (new to Fortran 90)
- **Module** - member of a module (new to Fortran 90)

Fortran 77 has only external procedures

External procedures

```
SUBROUTINE name (dummy-argument-list)
[specification-statements]
[executable-statements]
...
END [SUBROUTINE [name]]
```

or

```
FUNCTION name (dummy-argument-list)
[specification-statements]
[executable-statements]
...
END [FUNCTION [name]]
```

Note: RETURN statement no longer needed.

Internal procedures

Each program unit can contain internal procedures

Internal procedures are collected together at the end of a program unit and preceded by a `CONTAINS` statement

Same form as external procedures except that the word `SUBROUTINE`/`FUNCTION` must be present on the `END` statement

Variables defined in the program unit remain defined in the internal procedures, unless redefined there

- New way of making "global" variables

Nesting of internal procedures is not permitted

Internal Procedure: Example

```
PROGRAM main
  IMPLICIT NONE
  REAL :: a=6.0, b=30.34, c=98.98
  REAL :: mainsum
  mainsum = add()
CONTAINS
  FUNCTION add()
    REAL :: add      ! a,b,c defined in 'main'
    add = a + b + c
  END FUNCTION add
END PROGRAM main
```

INTERFACE Blocks

If an 'explicit' interface for a procedure is provided, compiler can check argument inconsistency

Module and internal procedures have an 'explicit' interface by default

External procedures have an 'implicit' interface by default

An `INTERFACE` block can be used to specify an 'explicit' interface for external procedures

Always use an `INTERFACE` block in the calling program unit for external procedures

Similar in form and justification for to C function prototypes

INTERFACE blocks: Syntax

General form:

```
INTERFACE
    interface_body
END INTERFACE
```

where `interface_body` is an exact copy of the subprogram specification, its dummy argument specifications and its `END` statement

Example:

```
INTERFACE
    REAL FUNCTION func(x)
    REAL, INTENT(IN) :: x
    END FUNCTION func
END INTERFACE
```

INTENT Attribute

Argument intent - can specify whether an argument is for:

input (IN)

output (OUT)

or both (INOUT)

Examples:

```
INTEGER, INTENT(IN) :: in_only
```

```
REAL, INTENT(OUT) :: out_only
```

```
INTEGER, INTENT(INOUT) :: both_in_out
```

Sample Program: NO INTERFACE

```
PROGRAM test
  INTEGER :: i=3,j=25
  PRINT *, 'The ratio is ',ratio(i,j)
END PROGRAM test

REAL FUNCTION ratio(x,y)
  REAL,INTENT(IN):: x,y
  ratio=x/y
END FUNCTION ratio
```

Output

```
Floating point exception
Beginning of Traceback:
  Started from address 453c in routine 'RATIO'.
  Called from line 3 (address 421b) in routine 'TEST'.
  Called from line 316 (address 21531b) in routine '$START$'.
End of Traceback.
Floating exception (core dumped)
```

Sample Program: WITH INTERFACE

```
PROGRAM test
  INTERFACE
    REAL FUNCTION ratio(x,y)
      REAL, INTENT(IN)::x,y
    END FUNCTION ratio
  END INTERFACE
  INTEGER :: i=3,j=25
  PRINT *, 'The ratio is ',ratio(i,j)
END PROGRAM test
```

```
REAL FUNCTION ratio(x,y)
REAL,INTENT(IN):: x,y
ratio=x/y
END FUNCTION ratio
```

Output

```
cf90-1108 f90: ERROR TEST, File = ratio_int.f90, Line = 3, Column = 33
  The type of the actual argument, "INTEGER", does not match "REAL",
  the type of the dummy argument.
```

Sample Program: INTERNAL PROCEDURE

```
PROGRAM test
  INTEGER :: i=3,j=25
  PRINT *, 'The ratio is ',ratio(i,j)

CONTAINS

  REAL FUNCTION ratio(x,y)
    REAL,INTENT(IN):: x,y
    ratio=x/y
  END FUNCTION ratio

END PROGRAM test
```

Output

```
cf90-1108 f90: ERROR TEST, File = ratio_int.f90, Line = 3, Column = 33
  The type of the actual argument, "INTEGER", does not match "REAL",
  the type of the dummy argument
```

Using Keywords with Arguments

With intrinsic Fortran functions, have always been able to use a keyword with arguments

```
READ (UNIT=10 , FMT=67 , END=789) x , y , z
```

If interface provided, programmer can use keywords with their own f90 procedure arguments. **ADVANTAGES:** Readability and override order of arguments

Using Keywords with Arguments (cont.)

Example Interface:

```
REAL FUNCTION area (start, finish, tol)
    REAL, INTENT(IN) :: start, finish, tol
    ...
END FUNCTION area
```

Call with:

```
a = area(0.0, 100.0, 0.01)
b = area(start = 0.0, tol = 0.01, finish = 100.0)
c = area(0.0, finish = 100.0, tol = 0.01)
```

Once a keyword is used, all the rest must use keywords

Optional arguments

If interface provided, programmer can specify some arguments to be optional

Coder's responsibility to ensure a default value if necessary (use `PRESENT` function)

The `PRESENT` intrinsic function will test to see if an actual argument has been provided.

Optional arguments

Example Interface:

```
REAL FUNCTION area (start, finish, tol)
    REAL, INTENT(IN), OPTIONAL :: start, finish, tol
    ...
END FUNCTION area
```

Call with:

```
a = area(0.0, 100.0, 0.01)
b = area(start=0.0, finish=100.0, tol=0.01)
c = area(0.0)
d = area(0.0, tol=0.01)
```

Example

```
REAL FUNCTION area (start, finish, tol)
  IMPLICIT NONEReAL, INTENT(IN), OPTIONAL :: start, finish, tol
  REAL :: ttol
  ...
  IF (PRESENT(tol)) THEN
    ttol = tol
  ELSE
    ttol = 0.01
  END IF
  ...
END FUNCTION area
```

Need to use local variable `ttol` because dummy argument `tol` cannot be changed because of its `INTENT` attribute

Using Structures as Arguments

Procedure arguments can be of derived type if:

- the procedure is internal to the program unit in which the derived type is defined
- or the derived type is defined in a module which is accessible from the procedure

Array-valued functions

In Fortran 90, functions may have an array-valued result:

```
FUNCTION add_vec (a, b, n)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n
  REAL, DIMENSION (n), INTENT(IN) :: a, b
  REAL, DIMENSION (n) :: add_vec
  INTEGER :: i
  DO i = 1, n
    add_vec(i) = a(i) + b(i)
  END DO
END FUNCTION add_vec
```

Recursive procedures

In Fortran 90, procedures may be called recursively:

- either A calls B calls A, or
- A calls A directly (`RESULT` clause required).

The `RESULT` clause specifies an alternate name (instead of the function name) to contain the value the function returns.

`RESULT` clause **required** for recursion

Recursive procedures (cont.)

Must be defined as recursive procedure:

```
RECURSIVE FUNCTION fact(n) RESULT(res)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n
  INTEGER :: res

  IF (n == 1) THEN
    res = 1
  ELSE
    res = n * fact(n - 1)
  END IF

END FUNCTION fact
```

Generic procedures

In Fortran 90, can define your own generic procedures

Need distinct procedures for specific type of arguments and a 'generic interface':

```
INTERFACE generic_name
    specific_interface_body
    specific_interface_body
    ...
END INTERFACE
```

Each distinct procedure can be invoked **using the generic name only**

The actual procedure used will depend on the type of arguments

Example Generic Subroutine swap

Consider the following two external subroutines for swapping the values REAL and INTEGER variables:

```
SUBROUTINE swapreal (a, b)

  REAL, INTENT(INOUT) :: a, b

  REAL :: temp

  temp = a;

  a = b;

  b = temp

END SUBROUTINE swapreal
```

```
SUBROUTINE swapint (a, b)

  INTEGER, INTENT(INOUT) :: a, b

  INTEGER :: temp

  temp = a;

  a = b;

  b = temp

END SUBROUTINE swapint
```

Example Generic Subroutine swap (cont.)

The extremely similar routines can be used to make a generic swap routine with the following interface:

```
INTERFACE swap    ! generic name
```

```
  SUBROUTINE swapreal (a,b)
```

```
    REAL, INTENT(INOUT)::a,b
```

```
  END SUBROUTINE swapreal
```

```
  SUBROUTINE swapint (a, b)
```

```
    INTEGER, INTENT(INOUT)::a,b
```

```
  END SUBROUTINE swapint
```

```
END INTERFACE
```

In the main program, only the generic name is used

```
INTEGER :: m,n
```

```
REAL :: x,y
```

```
...
```

```
CALL swap(m,n)
```

```
CALL swap(x,y)
```

Modules

Very powerful facility with many applications in terms of program structure

Method of sharing data and/or procedures to different units within a single program

Allows data/procedures to be reused in many programs

- Library of useful routines
- "Global" data
- Combination of both (like C++ class)

Very useful role for definitions of types and associated operators

Modules (cont.)

Form:

```
MODULE module-name

    [specification-stmts]

    [executable-stmts]

    [CONTAINS

        module procedures]

END [MODULE [module-name]]
```

Accessed via the USE statement

Modules: Global data

Can put global data in a module and each program unit that needs access to the data can simply "USE" the module

- **Replaces the old common block trick**

Example:

```
MODULE globals
  REAL, SAVE :: a, b, c
  INTEGER, SAVE :: i, j, k
END MODULE globals
```

Note the new variable attribute `SAVE`

Modules: Global data (cont.)

Examples of the `USE` statement:

```
USE globals                ! allows all variables in the
                           ! module to be accessed

USE globals, ONLY: a, c    ! Allows only variables
                           ! a and c to be accessed

USE globals, r => a, s => b ! allows a, b and c to be accessed
                           ! with local variables r, s and c
```

`USE` statement must appear at very beginning of the program unit
(right after `PROGRAM` statement)

Module procedures

Modules may contain procedures that can be accessed by other program units

Same form as external procedure except:

- Procedures must follow a `CONTAINS` statement
- The `END` statement must have `SUBROUTINE` or `FUNCTION` specified.

Particularly useful for a collection of derived types and associated functions that employ them

Module Example: Adding Structures

```
MODULE point_module
  TYPE point
    REAL :: x, y
  END TYPE point
  CONTAINS
    FUNCTION addpoints (p, q)
      TYPE (point), INTENT(IN) :: p, q
      TYPE (point) :: addpoints
      addpoints%x = p%x + q%x
      addpoints%y = p%y + q%y
    END FUNCTION addpoints
END MODULE point_module
```

A program unit would contain:

```
USE point_module
TYPE (point) :: px, py, pz
...
pz = addpoints(px, py)
```

Modules: Generic procedures

A common use of modules is to define generic procedures, especially those involving derived types.

```
MODULE genswap
  TYPE point
    REAL :: x, y
  END TYPE point
  INTERFACE swap ! generic interface
    MODULE PROCEDURE swapreal, swapint, swaplog, swappoint
  END INTERFACE
CONTAINS
  SUBROUTINE swappoint (a, b)
    TYPE (point), INTENT(INOUT) :: a, b
    TYPE (point) :: temp
    temp = a; a=b; b=temp
  END SUBROUTINE swappoint
  ... ! swapint, swapreal, swaplog procedures are defined here
END MODULE genswap
```

Modules: Public and private object

By default all objects in a module are available to a program unit which includes the `USE` statement

Can restrict the use of certain objects to the guest program

May wish to update module subroutines at any time, keeping the purpose and interface the same, but changing the meaning of specific variables

Achieved via `PRIVATE` attribute or `PRIVATE` statement:

```
INTEGER, PRIVATE :: keep, out
```

Overloading operators

Can extend the meaning of an intrinsic operator to apply to additional data types - operator overloading

Need an `INTERFACE` block with the form:

```
INTERFACE OPERATOR (intrinsic_operator)

    interface_body

END INTERFACE
```

Overloading operators (cont.)

- Example: Define '+' for character variables

```
MODULE over
```

```
    INTERFACE OPERATOR (+)
        MODULE PROCEDURE concat
    END INTERFACE
```

```
CONTAINS
```

```
    FUNCTION concat(cha, chb)
        CHARACTER (LEN=*) , INTENT(IN) :: cha, chb
        CHARACTER (LEN=LEN_TRIM(cha) + & LEN_TRIM(chb)) :: concat
        concat = TRIM(cha) // TRIM(chb)
    END FUNCTION concat
```

```
END MODULE over
```

Overloading operators (cont.)

Here is how this module could be used in a program:

```
PROGRAM testadd
  USE over
  CHARACTER (LEN=23) :: name
  CHARACTER (LEN=13) ::
wordname='Balder'
word='convoluted'
  PRINT *,name // word
  PRINT *,name + word
END PROGRAM testadd
```

Output

```
Balder      convoluted
Balderconvoluted
```

Defining operators

Can define new operators - especially useful for user defined types

Operator name must have a '.' at the beginning and end

Need to define the operation via a function which has one or two non-optional arguments with `INTENT (IN)`

Defining operators (cont.)

Example - find the straight line distance between two derived type 'points'

```
PROGRAM main

    USE distance_module

    TYPE (point) :: p1, p2
    REAL :: distance
    ...
    distance = p1 .dist. p2
    ...

END PROGRAM main
```

Defining operators (cont.)

Where the "distance_module" looks like this:

```
MODULE distance_module
  TYPE point
    REAL :: x, y
  END TYPE point

  INTERFACE OPERATOR (.dist.)
    MODULE PROCEDURE calcdist
  END INTERFACE

CONTAINS

  REAL FUNCTION calcdist (px, py)
    TYPE (point), INTENT(IN) :: px, py
    calcdist = SQRT ((px%x-py%x)**2 & + px%y-py%y)**2 )
  END FUNCTION calcdist
END MODULE distance_module
```

Assignment overloading

When using derived data types, may need to extend the meaning of assignment (=) to new data types:

```
REAL :: ax
TYPE (point) :: px
...
ax = px ! type point assigned to type real
        ! not valid until defined
```

Need to define this assignment via a subroutine with two non-optional arguments, the first having INTENT(OUT) or INTENT(INOUT), the second having INTENT(IN) and create an interface assignment block:

```
INTERFACE ASSIGNMENT (=)
    subroutine interface body
END INTERFACE
```

Assignment overloading (cont.)

In the following example we will define the above assignment to give `ax` the maximum of the two components of `px`

```
MODULE assignoverload_module
  TYPE point
    REAL :: x,y
  END TYPE point
  INTERFACE ASSIGNMENT (=)
    MODULE PROCEDURE assign_point
  END INTERFACE
CONTAINS
  SUBROUTINE assign_point(ax,px)
    REAL, INTENT(OUT)::ax
    TYPE (point), INTENT(IN)::px
    ax = MAX(px%x,px%y)
  END SUBROUTINE assign_point
END MODULE assignoverload_module
```

Program structure: Using Interface Blocks

When a module/external procedure is called:

- Which defines or overloads an operator, or the assignment
- Using a generic name

Additionally, when an external procedure:

- Is called with keyword/optional argument
- Is an array-valued/pointer function or a character function which is neither a constant nor assumed length
- Has a dummy argument which is an assumed-shape array, a pointer/target
- Is a dummy or actual argument (not mandatory but recommended)

Program structure: Summary

Fortran 77 style:

- Main program with external procedures, possibly in a library
- No explicit interfaces, so argument inconsistencies are not checked by compiler.

Simple Fortran 90:

- Main program with internal procedures
- Interfaces are 'explicit', so argument inconsistencies are trapped by compiler.

Fortran 90 with modules:

- Main program and module(s) containing interfaces and possibly specifications, and external procedures (possibly precompiled libraries).

Program structure: Summary (cont.)

A Fortran 90 version of the Fortran 77 style - with interfaces to permit compiler checking of argument inconsistencies.

Main program and module(s) containing specifications, interfaces and procedures. No external procedures.

Expected for sophisticated Fortran 90 programs.

ARRAY PROCESSING TOPICS

Terminology

Specifications

Whole array operations

WHERE statement and construct

Array sections

Array constructors

Allocatable arrays

Automatic arrays

Assumed shape arrays

Array intrinsic procedures

Terminology

Rank = Number of dimensions

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)
```

ahas:

- rank 2
- shape (/ 8, 7 /)
- extents 8 and 7
- 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 (extent-list)	Optional, but required to define default dimensions 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 (cont.)

Example specifications:

- Two dimensions

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

- Initialization (replace DATA statement):

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

Array Specifications (cont.)

Automatic arrays:

```
LOGICAL, DIMENSION (SIZE(loga)) :: logb
```

– where `loga` is a dummy array argument

Allocatable arrays (deferred shape):

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

– Dimensions are defined in subsequent `ALLOCATE` statement.

Assumed shape arrays:

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

Dimensions taken from actual arguments in calling routine.

Whole array operations (Array Syntax)

Can use entire arrays in simple operations $c=a+b$

Very handy shorthand for nested loops `PRINT *, 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

Whole array operations (Array Syntax)

RHS of array syntax expression completely computed before any assignment takes place

So be careful when the same array appears on both sides of the “=” sign

Scalars broadcast-scalar is transformed into a conformable array with all elements equaling itself $b=a+5$

Array Syntax Examples

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 Syntax Examples (cont.)

Fortran 77:

```
REAL a(5, 5), b(5, 5), c(5, 5)
DO 20 j = 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
```

Using Array Syntax with Intrinsic Procedures

Elemental procedures specified for scalar arguments

May also be applied to conforming array arguments

Work as if applied to each element separately.

Example:

```
! To find 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
  lengths = LEN_TRIM(words)
```

WHERE Statement

Form:

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

- Operation: assignment is performed if logical condition is true [again, element by element]

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

```
WHERE (rb > 0.0) ra = ra / rb
```

- Note mask (rb > 0.0) must conform with LHS 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
```

WHERE construct

- Used for multiple assignments:

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

- Or, 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

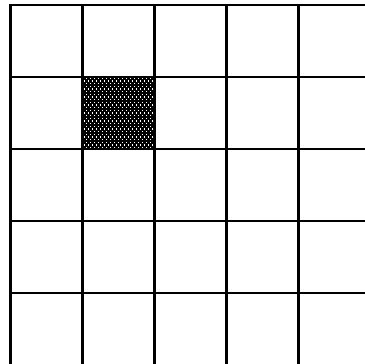
A subarray, called a section, of an array may be referenced by specifying a range of sub-scripts, either:

- A simple subscript `a(2, 3, 1)` ! single array element
- A subscript triplet defaults to declared bounds and stride 1
`[lower bound]:[upper bound] [:stride]`
- A vector subscript

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

Array Sections: Example

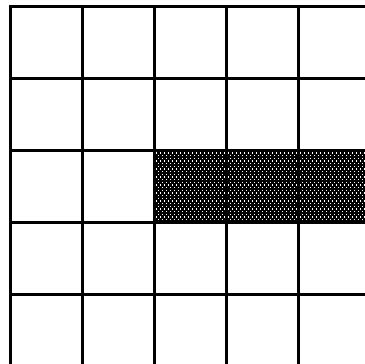
```
REAL, DIMENSION (5,5) :: ra
```



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

An Array Element

Shape (/1/)



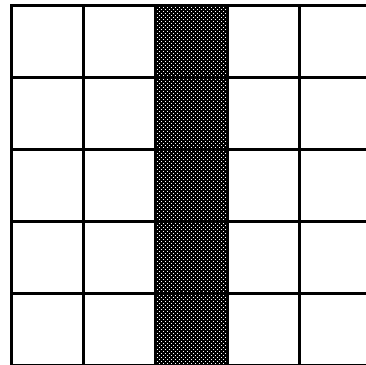
```
ra (3,3:5)
```

Sub-Row

Shape (/3/)

Array Sections: Examples (cont.)

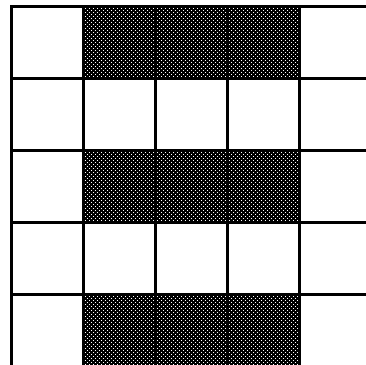
```
REAL, DIMENSION (5,5) :: ra
```



```
ra(:,3)
```

Whole Column

Shape (/5/)



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

Stride 2 for rows

Shape (/3,3/)

Vector subscripts

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 (cont.)

Vector subscript can be on LHS of expression

```
iv = (/ 1, 3, 5 /)
```

```
ra(iv) = (/ 1.2, 3.4, 5.6 /)
```

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

Must not repeat values of elements on LHS (many to one)

```
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 /)
```

```
iv = (/ 1, 3, 5 /)
```

```
ra(iv) = (/ 1.2, 3.4, 5.6 /) ! permitted
```

Array Section Assignments

Operands must be conformable

Example:

```
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 /)
```

Array Constructor

Have already used in previous pages: method to explicitly create and fill up a 1D array

Construction of rank 1 array:

```
REAL, DIMENSION (6) : : a, b (6,6)
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/)
 (/b (1, 2:4), b (1:5:2, i + 3) /)

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 (cont.)

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/)
! = (/1.0,0.5,0.33,0.25,.0.20,0.167 /)
```

RESHAPE Array Intrinsic Function

Once you have used a constructor to make a 1D array can change its shape with the `RESHAPE` functions

Syntax:

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

Operation: `RESHAPE` returns takes an array `SOURCE` and returns a new array with the elements of `SOURCE` rearranged to form an array of shape `SHAPE`

- 2D array constructed from 1D array elements a column at a time (by default)

RESHAPE Array Intrinsic Function (cont.)

Example:

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

Resulting array `ra` looks like:

2	3
3	4
4	5

Shape (/ 3,2 /)

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_obj_list [, STAT= status])
```

When `STAT=` is present, `status = 0` (success) or `status > 0` (error).

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

Allocatable arrays

Example:

```
REAL, DIMENSION (:, :), ALLOCATABLE :: ra
INTEGER :: status
READ (*, *) nsize1, nsize2
ALLOCATE (ra(nsize1, nsize2), STAT = status)
IF (status > 0) ! Error processing code goes here
    ! Now just use ra as if it were a "normal" array
IF (ALLOCATED(ra)) DEALLOCATE (ra)
...
```

Intrinsic function `ALLOCATED` returns present status of its array argument

Automatic arrays

- Automatic arrays typically used as scratch storage within a procedure
 - Advantage: no storage allocated for the automatic array unless the procedure is actually executing
- An automatic array is an explicit shape array in a procedure (not a dummy argument), whose bounds are provided when the procedure is invoked via:
 - dummy arguments
 - variables defined by use or host (internal procedure) associations
- Automatic arrays must not appear in `SAVE` or `NAMelist` statement, nor be initialized in type declaration
- Be aware that Fortran 90 provides no mechanism for checking whether there is sufficient memory for automatic arrays. If there is not, the outcome is unpredictable - the program will probably abort

Automatic arrays: Examples

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: Examples (cont.)

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
    WRITE (*, *) 'Bounds and size of a: ', &
               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 Fortrans)

Fortran 77: pass array dimensions as arguments

Fortran 90: not necessary to pass array dimensions

- Assumed shape array uses dimension of actual arguments
- Can specify a lower bound of the assumed shape array
- Interface required, so must provide an INTERFACE block if using an external procedure

Assumed shape arrays: Example

```
... ! 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
```

Array intrinsic functions

Reduction:

`ALL (MASK [, DIM])`

returns `.TRUE.` if all values of `MASK` are `.TRUE.` along dimension `DIM`;
otherwise `.FALSE.`

`ANY (MASK [, DIM])`

returns `.TRUE.` if any values of `MASK` are `.TRUE.` along dimension `DIM`;
otherwise `.FALSE.`

`COUNT (MASK [, DIM])`

returns the number of `.TRUE.` values in `MASK` along dimension `DIM`

`MAXVAL (ARRAY [, DIM] [, MASK])`

returns the maximum value of `ARRAY` along dimension `DIM`
corresponding to the `.TRUE.` values of `MASK`

Array intrinsic functions (cont.)

`MINVAL (ARRAY [, DIM] [, MASK])`

returns the minimum value of `ARRAY` along dimension `DIM` corresponding to the `.TRUE.` values of `MASK`

`PRODUCT (ARRAY [, DIM] [, MASK])`

returns the product of elements in `ARRAY` along dimension `DIM` corresponding to the `.TRUE.` values of `MASK`

`SUM (ARRAY [, DIM] [, MASK])`

returns the sum of elements in `ARRAY` along dimension `DIM` corresponding to the `.TRUE.` values of `MASK`

Array Intrinsic Functions (cont.)

Here and in the following slides, MASK is an LOGICAL of array of LOGICALs

Inquiry:

ALLOCATED (ARRAY)

returns **.TRUE.** if ARRAY is currently allocated; otherwise returns **.FALSE.**

LBOUND (ARRAY [, DIM])

returns the lower index bound(s) for ARRAY

SHAPE (SOURCE)

returns the shape of SOURCE as a rank-1 array of integers

SIZE (ARRAY [, DIM])

returns either the total number of elements in ARRAY (if DIM is not present) or the number of elements in ARRAY along dimension DIM

UBOUND (ARRAY [, DIM])

returns the upper index bound (s) for ARRAY

Array intrinsic functions (cont.)

Construction:

`MERGE (TSOURCE , FSOURCE , MASK)`

merges `TSOURCE` and `FSOURCE` into a single array based on values in `MASK`

`PACK (ARRAY , MASK [, VECTOR])`

packs `ARRAY` into rank-1 array based on values in `MASK`

`UNPACK (VECTOR , MASK , FIELD)`

unpacks the elements of `VECTOR` into an array based on values in `MASK`

`SPREAD (SOURCE , DIM , NCOPIES)`

returns an array by making `NCOPIES` copies of `SOURCE` along dimension `DIM`

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

returns an array of shape `SHAPE` using the elements of `SOURCE`

Array intrinsic functions (cont.)

Array manipulation:

`CSHIFT(ARRAY, SHIFT[, DIM])`

returns `ARRAY` circular-shifted `SHIFT` times (along dimension `DIM` only, if present)

`EOSHIFT(ARRAY, SHIFT[, BOUNDARY][, DIM])`

returns `ARRAY` end-off-shifted `SHIFT` times (along dimension `DIM` only, if present)

`TRANSPOSE(MATRIX)`

returns the matrix transpose of `MATRIX`

Array intrinsic functions (cont.)

Array Location:

`MAXLOC (ARRAY [, MASK])`

returns the location of the maximum element of `ARRAY`

`MINLOC (ARRAY [, MASK])`

returns the location of the minimum element of `ARRAY`

Vector and matrix arithmetic:

`DOT_PRODUCT (VECTOR_A , VECTOR_B)`

returns the dot products of `VECTOR_A` and `VECTOR_B`

`MATMUL (MATRIX_A , MATRIX_B)`

returns the matrix multiplication of `MATRIX_A` and `MATRIX_B`

Array Intrinsic Functions: Example

Three students take four exams. The results are stored in an INTEGER array:

85	76	90	60
71	45	50	60
66	45	21	55

score(1:3,1:4)

Largest score:

```
MAXVAL (score) ! = 90
```

Largest score for each student:

```
MAXVAL (score, DIM = 2) ! = (/ 90, 80, 66 /)
```

Student with largest score:

```
MAXLOC (MAXVAL (score, DIM = 2))  
! = MAXLOC((/ 90, 80, 66 /)) = (/ 1 /)
```

Average score:

```
average = SUM (score) / SIZE (score) ! = 62
```

Array Intrinsic Functions: Example (cont.)

Number of scores above average:

```
above = score > average
```

```
!above(3, 4) is a LOGICAL array
```

```
!above =
```

T	T	T	F
F	F	F	T
T	F	F	F

```
n_gt_average = COUNT (above) ! = 6
```

Pack all scores above the average:

```
INTEGER, ALLOCATABLE, DIMENSION (:) :: &  
    score_gt_average  
ALLOCATE (score_gt_average(n_gt_average))  
scores_gt_average = PACK (score, above)  
    ! = (/ 85, 71, 66, 76, 90, 80 /)
```

Array Intrinsic Functions: Example (cont.)

Did any student always score above the average?

```
ANY (ALL (above, DIM = 2)) ! = .FALSE.
```

Did all students score above the average on any of the tests?

```
ANY (ALL (above, DIM = 1)) ! = .TRUE.
```

Array example: Conjugate gradient algorithm

```
INTEGER :: iters, its, n
LOGICAL :: converged
REAL :: tol, up, alpha, beta
REAL, ALLOCATABLE :: a(:, :), b(:), x(:), r(:), &
    u(:), p(:), xnew(:)

READ (*, *) n, tol, its
ALLOCATE (a(n, n), b(n), x(n), r(n), u(n), p(n), xnew(n))

OPEN (10, FILE='data')
READ (10, *) a; READ(10, *) b

x = 1.0
r = b - MATMUL(a, x)
p = r
iters = 0
```

Conjugate gradient algorithm (cont.)

```
DO
  iters = iters + 1
  u = MATMUL(a, p)
  up = DOT_PRODUCT(r, r)
  alpha = up / DOT_PRODUCT(p, u)
  xnew = x + p * alpha
  r = r - u * alpha
  beta = DOT_PRODUCT(r, r) / up
  p = r + p * beta
  converged = ( MAXVAL(ABS(xnew-x))/MAXVAL(ABS(x)) < tol )
  x = xnew
  IF (converged .OR. iters == its) EXIT
END DO
```

Pointers Topics

[What is a pointer?](#)

[Specifications](#)

[Pointer Operation](#)

[Pointer assignment](#)

[Array Pointers](#)

[Pointer status](#)

[Dynamic storage](#)

[Pointer arguments](#)

[Pointer functions](#)

[Arrays of pointer](#)

[Linked list](#)

What is a Fortran 90 pointer?

A pointer variable has the `POINTER` attribute and may point to:

- Another data object of the same type, which has the `TARGET` attribute, or
- An area of dynamically allocated memory

The use of pointers provides:

- A more flexible alternative to allocatable arrays
- The tool to create and manipulate linked lists and other dynamic data structures (binary trees)

Fortran 90 pointer does not contain any data itself and **should not** be thought of containing an address (as in C)

A Fortran 90 pointer is best viewed as an alias for another "normal" variable that actually contains data

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 to which it is pointing
- If a pointer is an array pointer, only the rank, not the shape, should be defined

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

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

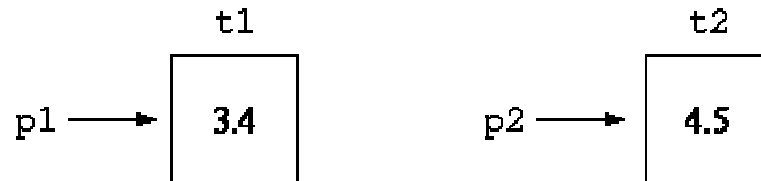
Pointer Operation

Once a pointer is associated with a target (see next page), and is then used in a Fortran statement where a value is expected, the value returned is that of the target variable.

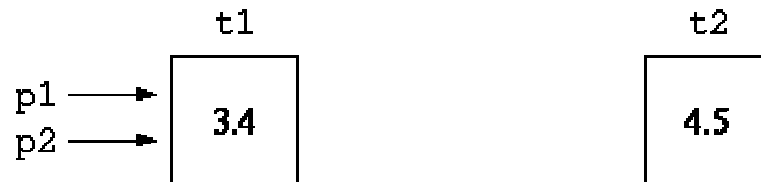
i.e., the pointer just acts as an alias for the target variable.

Pointer Assignment Operator =>

```
REAL, POINTER :: p1, p2
REAL, TARGET :: t1 = 3.4, t2 = 4.5
p1 => t1
PRINT *, t1, p1 ! 3.4 printed out twice
```



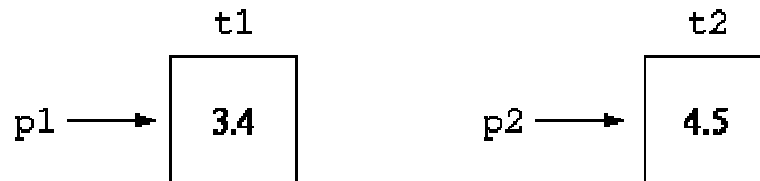
```
PRINT *, t2, p2 ! 4.5 printed out twice
p2 => p1 ! Valid: p2 points to the target of p1
```



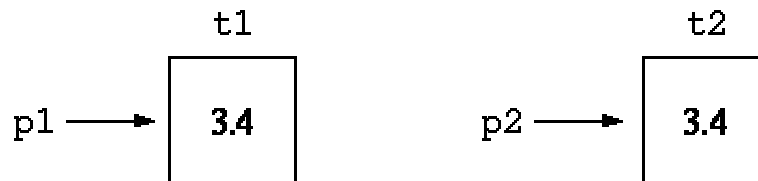
```
PRINT *, t1, p1, p2 ! 3.4 printed out three times
```

Pointer Assignment vs. Ordinary Assignment

```
REAL, POINTER :: p1, p2
REAL, TARGET :: t1 = 3.4, t2 = 4.5
p1 => t1
PRINT *, t1, p1 ! 3.4 printed out twice
```



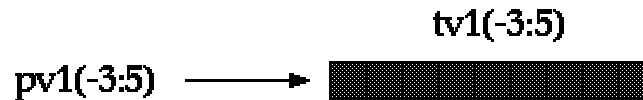
```
PRINT *, t2, p2 ! 4.5 printed out twice
p2 = p1 ! Valid: equivalent to t2=t1
```



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

Array Pointers: Target of a pointer can be an array

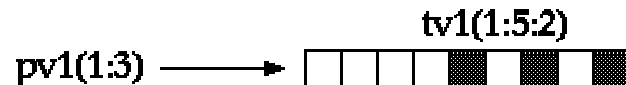
```
REAL, DIMENSION (:), POINTER :: pv1
REAL, DIMENSION (-3:5), TARGET :: tv1
pv1 => tv1          ! pv1 aliased to tv1
```



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

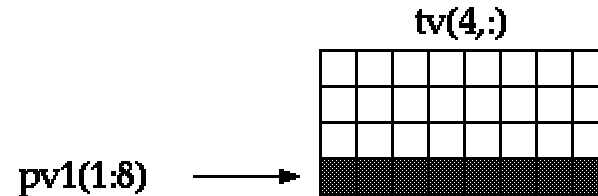


```
pv1=tv1(1:5:2)      ! aliased with section triplet
```

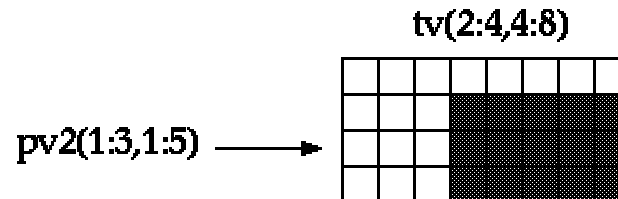


Array Pointers: 2D

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



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



Pointer status

Undefined as at start of program (after declaration)

Null not the alias of any data object

Must not reference undefined pointer, so set to null with NULLIFY statement

Associated

- The alias of a data object
- The status can be tested by ASSOCIATED intrinsic function, for example:

```
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 to create an un-named variable or array of specified size with 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
```

Before assignment like `p = 3.4` is made, `p` must be associated with its target via `ALLOCATE`, or aliased with target via pointer assignment statement (as before)

Potential problems

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
```

Pointer arguments

Pointers, whether allocated or not, are allowed to be procedure arguments (efficient way to pass an entire array)

- In contrast, allocatable arrays can not be used as dummy arguments: must therefore be allocated and deallocated in the same program unit.

Rules:

- If a procedure has a pointer or target dummy argument, the interface to the procedure must be explicit
- If a dummy argument is a pointer, then the actual argument must be a pointer with the same type, type parameter and rank
- A pointer dummy argument can not have the intent attribute
- If the actual argument is a pointer but the dummy argument is not, the dummy argument becomes associated with the target of the pointer

Pointers as arguments: Sample Program

```
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(:, :) ::
  ...
END SUBROUTINE sub1
SUBROUTINE sub2(b) ! b is a pointer
  REAL, DIMENSION(:, :), POINTER :: b
  DEALLOCATE(b)
END SUBROUTINE sub2
```

Pointer 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

The interface to a pointer function must be explicit

Pointer functions: Sample Program

```
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
    ... ! find the number of values .gt. 0 (put in n)
    ALLOCATE (gtzero(n))
        ... ! put the found values into gtzero
END FUNCTION gtzero
...
END
```

Arrays of pointers

An array of pointers can not 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 real_pointer
    REAL, DIMENSION(:), POINTER :: p
END TYPE real_pointer
TYPE(real_pointer), DIMENSION(100) :: a
INTEGER :: i
    ... ! possible to refer to the ith pointer
        by a(i)%p
DO i = 1, 100
    ALLOCATE (a(i)%p(i))
END DO
```

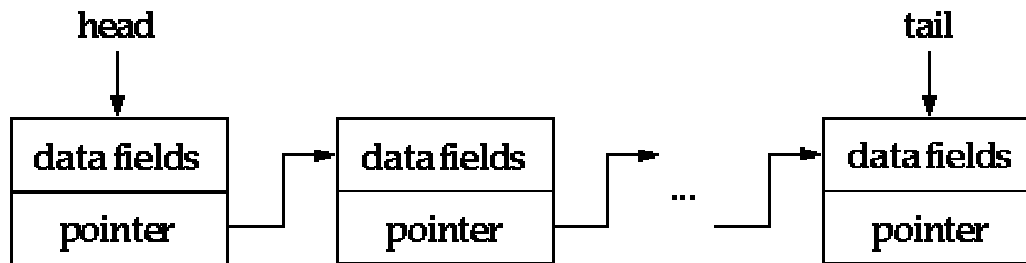
Q: Just what is a(10)%p ???????

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 Properties

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: Creation Program

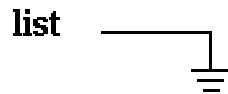
```
TYPE node
    INTEGER :: value ! data field
    TYPE (node), POINTER :: next ! pointer field
END TYPE node

INTEGER :: num
TYPE (node), POINTER :: list, current
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
...
```

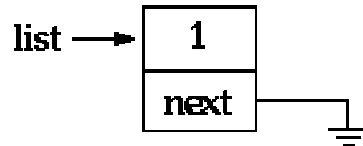
Linked list: Creation Program (cont.)

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

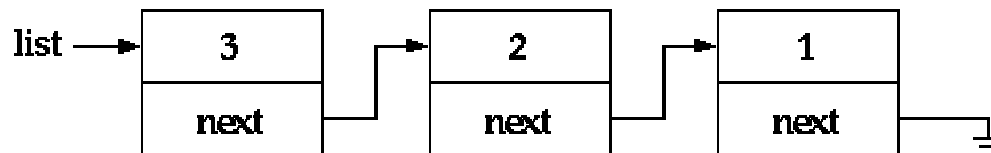
After `NULLIFY(list)`



After the first num is read



After all 3 numbers are read



New IO Features

Non-advancing I/O

INQUIRE by I/O list

New edit descriptors

New statement specifiers

Non-advancing I/O

Each READ or WRITE normally involves full records

Non-advancing I/O permits READ or WRITE without advancing position to new record

ADVANCE= 'NO' specifier:

```
WRITE(*, '("Input size:")', ADVANCE='NO')  
READ(*, '(I5)') n
```

On Screen, would see: Input size:34

Non-advancing I/O not applicable with list directed I/O

INQUIRE by I/O list

Syntax:

```
INQUIRE (IOLENGTH=length) output-list
```

To determine the length of an unformatted output item list

May be used as value of RECL specifier in subsequent OPEN statement

Example:

```
INTEGER :: rec_len
INQUIRE (IOLENGTH = rec_len) name, title, age, &
        address, tel
OPEN (UNIT = 1, FILE = 'test', RECL = rec_len, &
      FORM = 'UNFORMATTED')
WRITE(1) name, title, age, address, tel
```

New edit descriptors

- EN (Engineering) Same as E but exponent divisible by 3, value before decimal point between 1 and 1000
- ES (Scientific) Same as E but value before decimal point is from 1 to 10
- B Binary
- O Octal
- Z Hexadecimal
- G Generalized edit descriptor now applicable for all intrinsic types

New edit descriptors (cont.)

To compare the differences among E, EN, ES and G edit descriptors consider the following code:

```
PROGRAM e_en_es_g_compare
REAL, DIMENSION(4) :: &
x=(/1.234, -0.5, 0.00678, 98765.4/)
PRINT '(4E14.3/4EN14.3/4ES14.3/4G14.3)', x, x, x, x
END PROGRAM e_en_es_g_compare
```

Which produces this output:

0.123E+01	-0.500E+00	0.678E-02	0.988E+05
1.234E+00	-500.000E-03	6.78E-03	98.765E+03
1.234E+00	-5.000E-01	6.78E-03	9.877E+04
1.23	-0.500	0.678E-02	0.988E+05

New statement specifiers

INQUIRE

```
POSITION = 'ASIS' 'REWIND' 'APPEND'  
ACTION = 'READ' 'WRITE' 'READWRITE'  
DELIM = 'APOSTROPHE' 'QUOTE' 'NONE'  
PAD = 'YES' 'NO'  
READWRITE = )  
READ = ) 'YES' 'NO' 'UNKNOWN'  
WRITE = )
```

OPEN

```
POSITION, ACTION, DELIM, PAD are same as above  
STATUS = 'REPLACE'
```

New statement specifiers (cont.)

READ/WRITE

```
NML = namelist_name  
ADVANCE = 'YES' 'NO'
```

READ

```
EOR = label  
SIZE = character_count
```

Intrinsic Procedures Topics

[Intrinsic procedure categories](#)

[List of new intrinsic procedures](#)

Categories

1. Elemental procedures

A set of functions and one subroutine, specified for scalar arguments, but applicable for conforming array arguments

2. Inquiry functions

Return properties of principal arguments that do not depend on their values

3. Transformational functions

Usually have array arguments and an array result whose elements depend on many of the elements of the arguments

4. Nonelemental subroutines

New intrinsic procedures - Elemental functions

Numeric

`CEILING (A)`

returns smallest integer $\geq A$

`FLOOR (A)`

returns largest integer $\leq A$

`MODULO (A , P)`

returns remainder of A/P

Character

`ACHAR (I)`

returns character corresponding to I th character in ASCII

New intrinsic procedures - Elemental functions

Character (cont.)

ADJUSTL (STRING)

returns STRING with leading blanks removed

ADJUSTR (STRING)

returns STRING with trailing blanks removed

IACHAR (C)

returns ASCII code corresponding to character C

INDEX (STRING , SUBSTRING [, BACK])

returns starting position of SUBSTRING within STRING

LEN_TRIM (STRING)

returns length of STRING, neglecting trailing spaces

New intrinsic procedures - Elemental functions

Character (cont.)

`SCAN(STRING, SET [, BACK])`

returns the position of the first occurrence of any of the characters in SET within STRING

`VERIFY(STRING, SET [, BACK])`

returns 0 if every character in STRING is also in SET;
otherwise, returns the position of the first character in STRING that is not in SET

If BACK (a logical) is used and `.TRUE.`, then searches are made right to left rather than left to right (the default).

New intrinsic procedures - Elemental functions

Bit manipulation

`BTEST(I, POS)`

returns `.TRUE.` if bit `POS` of `I` is 1; otherwise `.FALSE.`

`IAND(I, J)`

returns bitwise `AND` of `I` and `J`

`IBCLR(I, POS)`

returns `I` with bit `POS` set to 0

`IBITS(I, POS, LEN)`

returns a right-adjusted sequenced of bits from `I` of length `LEN` starting with bit `POS`; all other bits 0

New intrinsic procedures - Elemental functions

Bit manipulation (cont.)

`IBSET(I, POS)`

returns `I` with bit `POS` set to 1

`IEOR(I, J)`

returns the bitwise exclusive OR (XOR) of `I` and `J`

`IOR(I, J)`

returns the bitwise OR

`ISHFT(I, SHIFT)`

returns `I` shifted bitwise `SHIFT` bits to the left; vacated position filled with 0

New intrinsic procedures - Elemental functions

Bit manipulation (cont.)

`ISHFTC(I, SHIFT [, SIZE])`

returns `I` circular-shifted bitwise `SHIFT` bits to the left

`NOT(I)`

returns the bitwise complement (negation) of `I`

- If `SHIFT` is negative, then bit shifts are done to the right rather than to the left.

Kind

`SELECTED_INT_KIND(R)`

returns a `KIND` value representing all `INTEGERS` $< 10^{**R}$

`SELECTED_REAL_KIND(P, R)`

returns a `KIND` value representing all `REALs` with at least `P` significant digits and a decimal exponent range of at least `R`

New intrinsic procedures - Elemental functions

Floating point manipulation

EXPONENT (X)

returns exponent of x in base 2 ($=1+\log_2(\text{int}(X))$)

FRACTION (X)

returns fractional part of x

NEAREST (X , S)

returns nearest machine-representable number of x in direction S ($S > 0$ forward; $S < 0$ backward)

RRSPACING (X)

returns reciprocal of relative spacing of numbers near x

New intrinsic procedures - Elemental functions

Floating point manipulation (cont.)

`SCALE (X , I)`

returns $X^{(2^I)}$

`SET_EXPONENT (X , I)`

returns a number whose fractional part is the same as `X` and whose exponent is `I`

`SPACING (X)`

returns the absolute spacing of numbers near `X`

Logical

`LOGICAL (L [, KIND])`

returns a logical value converted from the `KIND` of `L` to `KIND`

New intrinsic procedures - Elemental functions

Elemental subroutine

MVBITS (FROM , FROMPOS , LEN , TO , TOPOS)

copies a sequence of bit in FROM, starting at FROMPOS and LEN bits long, into TO beginning at TOPOS

New Intrinsic Procedures - Inquiry functions

Inquiry functions

`PRESENT (A)`

returns `.TRUE.` if A is present; otherwise returns `.FALSE.`

`ASSOCIATED (POINTER [, TARGET])`

returns `.TRUE.` if POINTER is associated and TARGET is not present, or if POINTER is associated to TARGET; otherwise returns `.FALSE.`

`KIND (X)`

returns the KIND value of X

`BIT_SIZE (I)`

returns the number of bits in I

New Intrinsic Procedures - Inquiry functions

Numeric

DIGITS (X)

returns the number of significant digits in X

EPSILON (X)

returns the smallest difference representable by the type and
KIND of X

HUGE (X)

returns the largest positive number representable by the type and
KIND of X

MAXEXPONENT (X)

returns of the maximum exponent of the type and KIND of X

New Intrinsic Procedures - Inquiry functions

Numeric (cont.)

`MINEXPONENT (X)`

returns of the minimum exponent of the type and `KIND` of `X`

`PRECISION (X)`

returns of the precision of the type and `KIND` of `X`

`RADIX (X)`

returns of the radix base of the type and `KIND` of `X` (typically 2)

`TINY (X)`

returns the smallest positive number representable by the type and `KIND` of `X`

New Intrinsic Procedures - Transformational functions

Transformational functions

REPEAT (STRING , NCOPIES)

returns a CHARACTER value contain NCOPIES copies of STRING

TRIM (STRING)

returns STRING without any trailing spaces

TRANSFER (SOURCE , MOLD [, SIZE])

returns a scalar or rank-1 array with the same physical (i.e. internal) representation as SOURCE but in the same type and KIND as MOLD

New Intrinsic Procedures - Non elemental subroutines

Non elemental intrinsic subroutines:

`DATE_AND_TIME ([DATE] [, TIME] [, ZONE] [, VALUES])`

returns the current date and time

`SYSTEM_CLOCK ([COUNT] [, COUNT_RATE] [, COUNT_MAX])`

returns data from the system real-time clock

`RANDOM_NUMBER (HARVEST)`

returns a (psuedo-)random number in HARVEST

`RANDOM_SEED ([SIZE] [, PUT] [, GET])`

restarts the psuedo-random number generator used by

`RANDOM_NUMBER`, or returns parameters about the generator

(Array intrinsic procedures: See section on array processing)

Random Number Program: Clock Seed

```
PROGRAM random
  INTEGER, DIMENSION(8):: time_info
  INTEGER :: msec,i,n
  REAL, DIMENSION(10) :: num
  CALL DATE_AND_TIME(VALUE=time_info)
  msec=time_info(7)*1000+time_info(8)
  CALL RANDOM_SEED(SIZE=n)
  CALL RANDOM_SEED(PUT=(/ (msec,i=1,n) /))
  CALL RANDOM_NUMBER(num)
  WRITE (*, '(10F5.2)') num
END PROGRAM random
```

Output from several runs of the program:

```
0.80 0.44 0.46 0.27 0.17 0.45 0.29 0.05 0.64 0.21
0.92 0.47 0.05 0.99 0.87 0.36 0.37 0.03 0.68 0.81
0.12 0.19 0.41 0.81 0.47 0.49 0.08 0.00 0.92 0.46
0.55 0.52 0.88 0.38 0.51 0.70 0.45 0.49 0.45 0.85
```