Workshop On Scientific Problem Solving
In Fortran 90/95 — Derived Types

Spring, 1999
Richard C. Allen, SNL
Paul M. Alsing, UNM/AHPCC
Andrew C. Pineda, UNM/AHPCC
Brian T. Smith, UNM/AHPCC/CS
Topics To Be Covered

- Derived types
  - What are they and why have them?

- Pointers
  - What are they and why have them?

- Data abstraction with modules

- A summary of input/output
  - The new input/output features
    - stream I/O
    - namelist I/O
Derived Types — Why

The mechanism for defining new types

- the new types are called derived types
- uses a strong typing model in the following sense:
  - associations require access to the same type definition

But why?

- Helpful in an application in which objects (variables) are naturally grouped together but are of different “type”
  - a mesh point in a grid where the data at the point represents
    – pressure, temperature, velocity, …
- You want to pass all of the information at a point about the program altogether, not as a list of variables, one for each item -- you want to think of the list as a single object and you want many instances of such objects
And from the practical program side

- you want to avoid long argument lists, long common block lists
- otherwise, the process is very error prone
  - wrong order, mistakes in spelling the names, etc.
- even the compiler may be more effective at compiling your code efficiently because the name space is reduced
- and your code is easier to read and maintain
Topics To Discuss

- Defining a derived type
- Declaring objects of a derived type
- Using derived-type definitions in other scoping units
- Passing objects of derived types to other scoping units
  - as arguments of procedures
  - through common blocks (storage association)
  - from a module (use association)
Topics To Be Covered Continued

- Intrinsically defined operations for derived-type objects
- Constructing objects of derived type
  - the derived-type constructor
- User-defined operations for derived-type objects
- Pointers
- A scientific programming example
  - adaptive grids
A new type is made up of components whose types are either:
- intrinsic,
- other derived types, including the same type

Defining a type is like declaring a template for the parts or components of the type

```plaintext
type polar
  real   magnitude, phase
end type polar
```

A type to represent complex numbers in polar coordinates
- components are named magnitude and phase
- the type name is: polar
The components can be arrays or pointers
There may be any number of components
The general form is:

```plaintext
type  <type_name>
    <any type> [, dimension(…)], [pointer] :: list_of_names
end type [<type_name>]
```
Declaring Objects Of A Derived Type

Data objects of a derived type

- type spelled: `type( <type_name> )`
  ```
  type( polar )  x1, y1
  ```
- variables and constants
  - `x1` and `y1` above are variables
  - a constant can be declared as named constant: for example, `j` in:
    ```
    type( polar ), parameter :: j = polar( 1.0, PI/2.0 )
    ```
- scalars (as are `x1` and `y1` above) or arrays (as is `points` below)
  ```
  type( polar ), dimension(100) :: points
  ```
- can have attributes like any intrinsic object
  ```
  SAVE, DIMENSION, POINTER, TARGET, EXTERNAL, ...
  ```
Accessing And Defining Components

Suppose \( x \) is an object of a derived type

- A component of \( x \) is selected using the selector % as in:
  
  \[
  \text{name} \% \text{component}
  \]

- An example of defining a component is the assignment statement:
  
  \[
  \text{x%magnitude} = 1.0
  \]

  defines the component of \( x \) named \text{magnitude} with the value 1.0. The following defines the component \text{phase} of \( x \):
  
  \[
  \text{x%phase} = \pi/7.0 + \text{y%phase}
  \]

- % is used instead of dot (.) because of ambiguities with the names of the relational operators:
  
  \[
  \text{x .lt. y}
  \]

  Is this \( x < y \) or the component \( y \) of the component \text{lt} of \( x \)?
Values And Operations

Values of the type

- can be defined by a constructor as follows:
  \[ <\text{derived-type-name}>( \text{list_of_values_one_for_each_component} ) \]
- examples:
  \[ \text{polar}( 1.0, \pi/2.0 ) \]
  \[ \text{polar}( R, \text{THETA} ) \]
  ! R and THETA are reals

Operations

- the only intrinsic operations are:
  - assignment (component by component)
  - == and /= (synonyms for .eq. and .ne.)
    - == each corresponding component is equal, and false otherwise
    - /= one or more corresponding components are not equal
ONLY the intrinsic operations for derived types may be redefined:

- for example, polar values can be defined as equal when their values are identical or when the angles are out of phase by a multiple of two $\pi$
- this is done by defining a function with two arguments that returns true when the angles are equal or different by a multiple of $2\pi$
- also, an interface is provided for the relational operation $==$ (or equivalently `.eq.`) that specifies this function

Any other operator may be defined (say, $+$)

- for example, $x_1+y_1$ can be defined for type polar objects
Interfaces For ==

- To have a user function be used for ==
  - an interface for the operator == must be defined
  - it is accomplished with an interface specification as follows:
    ```
    interface  operator( == )
      module procedure  eq_polar
    end interface
    ```
  - In essence, this interface says that any use of == with each operand of type polar uses the procedure eq_polar; that is,
    ```
    result = (x == y)
    ```
  - is the same as:
    ```
    result = eq_polar( x, y )
    ```
Redefining The Operator ==

Module polar_module

  ...  

  interface operator(==)  
      module procedure eq_polar( x, y )  
  end interface

CONTAINS

  logical function eq_polar( x, y )
      type( polar ), intent(in) :: x, y
      ...  
      eq_polar = x%magnitude == y%magnitude .and. &
      (mod(x%phase,2*PI) - mod(y%phase,2*PI)) < &
      2*EPS
  end function eq_polar

end module polar_module
Redefining The Assignment Operator

Module polar_module
  . . .
  interface assignment(=)
    module procedure assign_polar( x, y )
  end interface
CONTAINS
  subroutine assign_polar( x, y )
    type( polar ), intent(out) :: x
    type( polar ), intent(in) :: y
    x%magnitude = y%magnitude
    x%phase = mod(y%phase, TWO_PI)
  end subroutine assign_polar
end module polar_module
Because of local scoping rules, the definition of a type is known only in the unit where the definition appears.

That definition may be inherited into other units by:

- a USE statement -- USE association
- host association -- inherit from an enclosed scope
  - in an internal procedure from its host
  - in a module procedure from its module

What about external units with no USE?
Passing Derived Types To External Units

A repetition of the type definition is a different and new type

- when passing objects, this results in a type mismatch
  - derived types are strongly typed
- an “escape” is provided to handle this case and storage layouts for derived types
  - the escape is the SEQUENCE attribute
  - the model is that the compiler must layout the components in a predictable (and natural) storage sequence
    - the components one after the other in order with no padding (essentially)
When Do Sequenced Types Match?

The rules are:

- the type name must be the same
- the names of the components must be the same and in the same order
- the attributes of the corresponding components must be the same
- they both must have the SEQUENCE attribute
An Example Of “Sequenced Types”

Program sequence_types
    ... ! Sequence type definition for type person
    type(person) patients(10)
    interface
        real function average_weight( p )
            ... ! Sequence type definition for type person
            type( person ), dimension(:), intent(in) :: p
        end function average_weight
    end interface
    ... new_avg_wt = average_weight( patients ) + 10.3
Example Continued

Sequence type definition for type person

type person
  sequence
  character(12) name
  integer ssn
  real height, weight
end type person
External Procedure With Derived Type Argument

Function average_weight( p )
    type person
        sequence
            character(12) name
            integer ssn
            real height, weight
    end type person
    type( person), dimension(:), intent(in) :: p
    real average_weight
    average_weight = sum( p(:)%weight )
end function average_weight
Importance Of Sequenced Type

Sequence types MUST be used when derived types are place in common
  ♦ common utilizes a concept of storage association which requires a strict layout of components in storage

Sequence types are virtually required when passing derived-type objects to other languages such as C
  ♦ both languages have a concept of storage layout that is in most cases compatible -- however, it is often vendor specific
  ♦ Fortran 2002 will likely specify interoperability requirements with C to make passing of sequenced derived types more dependable and portable
**Pointers**

- Why have them?
  - Supports arbitrary dynamic structures
    - more and more scientific computation needs this capability as simulations become bigger and bigger
    - also simulations are multi-problem or phased, with different algorithms and modeling techniques used in each phase
  - Recognition of the need to simplify the programming methodologies with high level capabilities
    - as opposed to simulating high level programming techniques with static programming facilities
    - the con is that this feature introduces potential execution inefficiencies that are subtle and difficult to avoid, particularly with poor compiler implementations
General Characteristics Of Fortran Pointers

- Strongly typed
- Tightly specified to permit traditional optimizations of non-pointer code
  - requires the use of the TARGET attribute for any variable that can be pointed to
- No pointer arithmetic allowed
- Automatic de-referencing in non-pointer contexts
  - the target is the object of interest, not the pointer
  - it is design mostly as an restricted name aliasing facility
Automatic De-Referencing

Use of the pointer is a reference to its target in most cases rather than its pointer.

- in an expression
- as the variable on the left side of an assignment
- in an I/O item list
- as an actual argument in an argument list, unless the corresponding dummy argument has the pointer attribute
- see the examples on the next slide
  - convention -- ..._ptr is the pointer, ... is the target
real, pointer :: A_PTR, B_PTR
real, target :: A = 3.1, B = 4.2
integer, pointer :: C_PTR
integer, target :: C = -32

A_PTR => A;   B_PTR => B;  C_PTR => C  ! Pointers on the left
A_PTR = B_PTR + C            ! Targets referenced here on both sides
                            ! A_PTR becomes -27.8.

C_PTR => A      ! Invalid -- A and C_PTR are not the same type
C_PTR = sqrt(A_PTR + B_PTR) + abs(C_PTR)    ! All are targets
A variable with the pointer attribute contains either:

- an address of a target (associated with a target)
  - more generally, the address of a descriptor for the target which indicates where the target is
- a null value (disassociated, points to null)
- an undefined (unpredictable, not testable) value

Cases above correspond to one of three states for a pointer

- associated, disassociated, undefined

Initially, a pointer is in the **undefined** state

- the programmer can specify an initial value or state other than undefined in F95
The Disassociated State For Pointers

A pointer becomes disassociated in one of three ways:

- nullified as in the statement
  \[
  \text{nullify( PTR )}
  \]
- deallocated as in the statement
  \[
  \text{deallocate( PTR )}
  \]
- pointer-assigned to a disassociated pointer
  \[
  \text{nullify( A_PTR )} \quad ! \text{A_PTR is disassociated}
  \]
  \[
  \text{PTR => A_PTR} \quad ! \text{Now PTR is disassociated}
  \]
Associated State For Pointers

A pointer becomes associated by being:

- pointer assigned to a target or to a pointer that is associated with a target using pointer assignment:
  
  A_PTR => A  ! Assume A is a target

- allocated using the ALLOCATE statement
  
  allocate( A_PTR )
Initial States

When the program starts initially:

- Pointer A_PTR: Undefined
- Target A: Undefined
Associated States

After the following statements are executed:

\[ A = 3.1; \quad A\_PTR \rightarrow A \]

- Pointer A\_PTR
- Address of A
- =>
- 3.1
- Target A

Pointer assignment
After the following statements are executed:

\[ A = 3.1; \ A\_PTR \Rightarrow A; \ B\_PTR \Rightarrow A\_PTR \]

- Pointer B_PTR
- Address of A
- Pointer assignment
- 3.1
- Target A
Associated States Continued

After the execution of the following statement:
allocate( A_PTR ); A_PTR = 13.3

Pointer A_PTR

Allocate statement

Anonymous target

Address of anonymous target

allocate

13.3
Disassociated States

After the execution of the following statement:
nullify( A_PTR )

Pointer A_PTR

Address of null target

nullify

null

NULLIFY statement
Examples Of Pointers

real, pointer :: PTR
real, target :: A, B, C

PTR => A        ! Pointer assignment.
! PTR now has A as its target.
PTR => B        ! Change the target of PTR to B.
allocate( PTR ) ! Create storage for a target and
! point PTR to this storage
! (target is anonymous, has no name).
! (PTR no longer points to B.)
Rules For Pointer Assignment

**Form**

\[ \text{<pointer>} \Rightarrow \text{<target>} \]

- \text{<pointer>} and \text{<target>} must have the same type, kind, and rank
- \text{<target>} must be a variable or function with the TARGET or POINTER attribute
- \text{<pointer>} must be a variable with the POINTER attribute
Rules For Reference To A Pointer

- In a pointer assignment:
  - `<pointer>` is a reference to the address (or descriptor) of the target

- As an actual argument corresponding to a dummy argument with the POINTER attribute
  - it is the address of the target if it is associated

- In all other executable constructs:
  - it is a reference to the target
Major Uses In Scientific Computing

- Avoiding the copy operation for large data structures -- for example, large arrays
- Dynamically sized data structures
  - arrays
  - lists of objects of arbitrary lengths and with arbitrary sized objects
    - link lists of grids
    - sparse matrices
Avoiding Large Copy Operations

- Iterating with a large grid of points with a large amount of data associated with the grid points (pressures, temps, velocities, etc)
  - may be megabytes of storage

  OLD_GRID = NEW_GRID
  NEW_GRID = ... OLD_GRID ... ! A calculation

  ! with the old grid

  - first assignment moves lots of data
    - from the storage for NEW_GRID to the storage for OLD_GRID
No Data Movement With Pointers

T => OLD_GRID
OLD_GRID => NEW_GRID
NEW_GRID => T
NEW_GRID = ... OLD_GRID ...

- No data is moved when => is used
- The computation of the new grid values are stored in the location for NEW_GRID which was the location of the OLD_GRID on the previous iteration
Consider the Gaussian elimination algorithm

```fortran
subroutine column_eliminate_elementary( a, i, pivot )
    real, dimension(:), pointer :: pivot_row
    . . .
    pivot_row => a(i+1:,i)
    do j = i+1, n
        a(i+1:,j) = a(i+1:,j) - pivot_row*a(i,j)
    enddo
    . . .
end subroutine column_eliminate_elementary
```
Consider a simulation on a region which changes its mesh point density and configuration as the algorithm proceeds with the simulation:

- changes to adapt to periods of high activity or change:
  - mesh becomes finer to obtain an accurate enough solution
- changes to adapt to periods of low activity or change:
  - mesh becomes coarser to improve the efficiency of the computation
Suppose the changing grids are represented by a linked list of grids

- each node contains the grid (and associated data)
- a link to a grid that is coarser for efficient computation
  - or maybe to allow for computation of estimated errors
- a link to a grid that is finer for a more accurate solution
  - or maybe to allow for computation of estimated errors

```fortran
type mesh
  type (mesh), pointer :: finer
  real, pointer, dimension(:) :: mesh_points
  type (mesh), pointer :: coarser
end type mesh
```
Creating A Node In The Linked List

- **Declare the head of the list**
  ```
  type (mesh) my_grid_head
  ```

- **Allocate the mesh points in the top node**
  ```
  allocate (my_grid_head%mesh_points(3))
  ```

- **Make this the only node in the list**
  ```
  nullify (my_grid_head%finer)
  nullify (my_grid_head%coarser)
  ```

- **Provide a set of mesh points for this node**
  ```
  my_grid_head%mesh_points(:) = (/ 1.0, 2.0, 3.0 /)
  ```
Printing A Contents Of A Node

Printing the node:

```fortran
  type (mesh), pointer :: current_node
  current_node => my_grid_head
  call print_node( current_node )
```

The print procedures:

```fortran
  subroutine print_node( node )
    type(mesh), pointer :: node
    call print_grid( node%mesh_points )
  end subroutine print_node

  subroutine print_grid(points)
    real, dimension(:) :: points
    print *, points
  end subroutine print_grid
```
More On The Link List Of Mesh Points

There are three codes giving examples of more and more sophisticated uses of pointers, linked lists, and recursive procedures

- see mesh1.f90 in directory dervtype
- see mesh2.f90 in directory dervtype
- see mesh3.f90 in directory dervtype
The following procedure prints the contents of a node:

```fortran
subroutine print_node( node )
   type(mesh), pointer :: node
   call print_grid( node%mesh_points )
end subroutine print_node
```

- The argument is a pointer of type `mesh`
  - the `mesh_points` component is passed to `print_grid`
  - the target is always used as the parent with the `%` selector
The Reference On the Calling Side

- To pass this node to print_node (pointer argument), a pointer must be supplied.

- The code looks like:

  current_node => my_grid_head
  call print_node( current_node )

  The pointer assignment creates a pointer to the target my_grid_head which is a target, not a pointer
Data Abstraction with Modules

Data abstraction and data hiding are programming concepts to limit access to parts of program with the following goals:

- make available only what is needed
- leave the details of the implementation hidden so that
  - the details can be optimized separately to develop a more efficient or environment dependent implementation without changing the application
Control of Names From Modules

This is done in Fortran with modules using:

- attributes PRIVATE and PUBLIC for all items in a module
  - an item is made public if the user of the module is expected to use and need it in his applications
  - an item that should not or need not ever be used
- the USE statement with ONLY: and the rename options to control and manipulate the accessibility of names
Introduction and Summary

- The Fortran I/O model
  - Records, files, access methods, and units
  - Existence of files, inquiry, open, and closing files
- The access method
  - Sequential or direct access
- The kinds of files
  - External and internal files
- The form of the file
  - Unformatted, formatted (user-controlled formats, list-directed and name directed)
New I/O Features With Application To Scientific Computing

- **Internal files**
  - read and write to and from a character string as opposed to an external file (disk, printer, tape, …)
  - see the example of adjustable formats

- **Stream input/output**
  - no advancing of the record on a read or a write
  - continue from the position in the record left by a previous read or write

- **Name directed input/output**
  - the input is specified by names in the input record
  - the output prints the name of the variable in the output record.