



Advanced Features of F90

Tom Clune

NASA GSFC (SIVO – 610.3)



Outline

- Introduction
 - Modules
 - Pointers
 - User defined types
 - Optional arguments (time permitting)
-



Introduction

- Intent is to focus on fairly narrow subset of features introduced in F90
 - Emphasis on examples and best practices
 - Please - ask questions as we go along
-

History of the Fortran Standard

- ISO mandated schedule:
 - Major revision every 10 years
 - Minor revision every 5 years
- FORTRAN 66 – first actual standard
- FORTRAN 77
 - block if/else/endif, overindexing, generic names for intrinsics, implicit none
 - *Capitalization changed* – helps to recognize experts
- Fortran 90 (formerly Fortran 8x)
 - Array syntax, dynamic memory, modules, user defined types, numerous intrinsic functions, free format, long identifiers, explicit procedure interfaces, inline comments, ...
- Fortran 95
 - Very minor/obscure additions
 - Generally still referred to as “F90” by users/vendors
 - Universally implemented by vendors at this time.

Latest Fortran Standard

- F2003
 - Has been the standard for 7 years!
 - Important new features
 - Standardized interoperability with C
 - Object-oriented programming, procedure pointers
 - ASSOCIATE construct
 - Full implementations are only just now becoming available
 - IBM, Cray, NAG
 - Partial implementations are common - intel*, gfortran, g95
- F2008 (F2010?) – minor? Probably not.
 - Co-Array Fortran – parallelism without the pain of MPI

Fortran Resources

- Lots of good material on web
- Usenet - *comp.lang.fortran* lots of spam
- Mailing list – `comp-fortran-90`
 - Low volume, little spam
 - Lots of volunteer experts
 - Interesting discussions of subtle aspects
- `ModelingGuru.nasa.gov` forums
 - Esp. if specific to NASA computing systems

F90 Modules

- program units which act as containers for data and procedures while limiting the visibility to external program units.
- provide *explicit* interfaces for contained procedures and subroutines
 - Provides strong argument checking
 - Argument count
 - TKR (type : kind : rank) of each argument
- Useful for encapsulation – the hiding of implementation details
 - Think firewalls for source code
- Simplest case is just a glorified common block
 - Typical 1st use for developer is to replace commons
 - Guarantees consistent use
 - Does not have memory alignment issues/tricks

F90 Module syntax

- **MODULE / END MODULE**
 - begin/end declaration of a module
- **CONTAINS**
 - Ends declaration of module variables (and derived types)
 - Begins section of subroutines and functions contained in module
 - Use if and only if module has procedures
- **PUBLIC, PRIVATE**
 - Attributes that specify which module entities are visible to external procedures and modules
- **USE <module name> [ONLY: <entity>, ...]**
 - Allow access to all or specified public entities from module

Basic structure of a module

```
module <module name>  
    <use statements>  
    implicit none  
    private  
    <module variable declarations>  
contains  
    <procedure 1>  
    <procedure 2>  
    ...  
end module <module name>
```

Public and private entities

- By default all module entities have the PUBLIC attribute
 - I.e. all module entities are accessible by external procedures and modules.
- Can override with PRIVATE statement
 - No module entities visible externally
 - Also can confirm default with PUBLIC statement
 - But not both
- PUBLIC default is an unfortunate choice
 - Can easily lead to complications with nested modules
 - Strongly recommend always override with PRIVATE
- Individual entities can override with *attribute*

```
real, public :: array(10)
integer :: ifirst
public :: ifirst
```

F90 explicit interfaces

- Vanilla F77 procedure uses *implicit* interfaces
 - Compiler cannot properly check arguments for consistency
 - Permits some interesting tricks by mixing types
- F90 introduces explicit interfaces
 - Checks for consistent argument count and type/kind/rank
 - Functions/Subroutines in modules have explicit interfaces
 - Requires either `USE` statement or `INTERFACE` block in the *caller* routine
 - Not discussing `INTERFACE` blocks today (rarely needed)
 - Common compile time error is “cannot find procedure such-and-such”.
 - Indicates missing “use” statement
 - Compiler “mangles” names under hood so may be difficult to recognize

F90 Pointers

- The `POINTER` *attribute* allows a variable to be associated with the memory of another variable.
 - Think of as sophisticated, dynamic version of F77 `EQUIVALENCE`
 - Some restrictions on which variables can be associated.
 - **Caution**: Related to, but *quite* different from C pointers
- Useful for
 - Dynamic memory allocation
 - Manipulating sub-arrays
 - Avoids copies, improves performance (sometimes)
 - Functions that return arrays instead of scalars
 - Advanced data structures. E.g. linked-lists

Pointer syntax

- Declaration

```
real, pointer :: x ! Scalar  
real, pointer :: vec(:) ! 1D array  
real, pointer :: arr(:, :) ! 2D, etc
```

or

```
real :: x  
pointer :: x
```

- Usage

```
real, target :: y
```

...

```
x => y ! Pointer assignment
```

```
x = y ! Copy value of y into target of x
```

- Pointer and target must have same TYPE, KIND, and RANK
- Can give provide initial value (F95)

```
real, pointer :: z => null()
```

- SAVE attribute is implicit, as with all other initialized variables
- Strongly recommended for global pointers

Simple pointer example

```
real, pointer :: ptr  
real, target :: A, B
```

```
A = 1.
```

```
ptr => A
```

```
A = 2.
```

```
print*, ptr ! Should print "2"
```

```
ptr = 3.
```

```
print*, A ! Should print "3"
```

```
ptr => B
```

```
ptr = 7.
```

```
print*, A ! Should still print "3."
```

Useful intrinsic functions

- **Nullify** - restores pointer to pristine state
 - Caution: does *not* deallocate memory (other pointers may still point to the same target)
- **Associated** – returns `.true.` iff pointer is associated with a target or a specific target
 - Useful to test result of function which sets a pointer
 - Note that results are undefined for uninitialized pointers!
 - Usage:
 - Any target
`if (associated(ptr)) then`
 - Specific target (or pointer with same target)
`if (associated(ptr, targ)) then`
 - Do not confuse with allocated intrinsic for allocatables.

Dynamic memory allocation

- In addition to associating with a target variable, pointer variables can be allocated just like **ALLOCATABLE** variables.
 - Target is *implicit* and not associated with any actual variable
- Important diffs from allocatable variables:
 - Deallocation nullifies pointer
 - Pointer local vars not automatically deallocated
 - Memory leak potential
 - Pointers can be reallocated without deallocation
 - Pointers can reassigned to other var without deallocation
- Best practice: use **ALLOCATABLE** attribute unless **POINTER** is required.

Dynamic memory example

```
integer, pointer :: indices(:)
real, pointer :: ptr(10) ! Not allowed
...
if (.not. Associated(indices)) then
    allocate(indices(100))
end if
indices = ...
...
deallocate(indices)
allocate(indices(5:10)) ! Specify different lbound
...
```

Pointers and array slices

- Pointers can be associated with slices (subsections) of multidimensional arrays
 - Mostly straightforward, but there are some subtleties lurking about

- Examples:

real, target :: A(0:100,2:100)

real, pointer :: p1(:,:), p2(:,:), p3(:,:), p4(:,:)

p1 => A ! Whole array

p2 => A(:,:) ! *Not* whole array

p3 => A(2:4,51:)

p4 => A(::10,50:80:10) ! strided

- What are the lbound, ubound, and shape of p1, p2, p3?

Pointer	L Bound	U Bound	Shape
p1	[0,2]	[100,100]	[101,99]
p2	[1,1]	[101,99]	[101,99]
p3	[1,1]	[3,50]	[3,50]
P4	[1,1]	[11,4]	[11,4]

Pointers actual arguments

- An associated pointer variable can be passed as an actual argument as though it was a regular variable
 - Must be associated
 - Usual Fortran rules against aliasing apply

```
real :: tracer1(100,10)
real :: tracer2(100,10)
real, pointer :: a(:, :)
a => tracer1
call sub(a) ! Same as call sub(tracer1)
a => tracer2
call sub(a) ! Same as call sub(tracer2)
```

Pointer dummy arguments

- Dummy arguments can also have the `pointer` and `target` attributes
- *Pointer dummy*:
 - *requires* a *pointer actual*
 - association status of *dummy* is that of *actual* at entry (can be null)
 - association status of *actual* is that of *dummy* at exit (can be null)
 - *requires explicit* interface
 - cannot specify “intent”
 - Ambiguous meaning
 - Permitted in F2003

Pointer Function Return

- Can be useful to have a function or subroutine return a dynamically sized array.
 - POINTER attribute is necessary if array is to be allocated by the procedure
 - Potential source of memory leaks – *caller* responsible for deallocation
 - E.g. suppose we wanted to have a function that returns a dynamically sized array:

Example: Pointer function value

```
function newGridArray() result(array)
  real, pointer :: array(:, :, :)
  allocate(array(IM, JM, LM))
end function newGridArray
```

...

```
real, pointer, dimension(:, :, :) :: u, v, w
```

```
u => newGridArray()
```

```
v => newGridArray()
```

```
w => newGridArray()
```

- Later we will see how to use derived types to pass in the grid dimensions

User-defined Data Types

- Provides ability to declare new “types”
 - Collections of entities of intrinsic types and/or other user-defined types
 - Esp. useful for multiple instances of such collections
 - Introduces higher-level structure: treat many as one
 - Concept is to group items that are closely associated
- Nearly impossible to underestimate the usefulness
 - Encapsulation and reuse
 - Replacing common blocks, short meaningful argument lists
 - Generic programming
 - First step on path to object-oriented programming
- Perhaps difficult to appreciate at first
 - Effective design requires experience

F90 User-defined types

- F90 terminology is different than most languages. E.g.
 - Define types are called “derived types”
 - Member entities are referred to as “components”
- Components can have default initial values
- Declare *type* with **TYPE** / **END TYPE** block
 - In same part of procedure/module as other variable declarations

```
TYPE <type name>  
  <component1> [= <initial value>]  
  <component2> [= <initial value>]  
  ...  
END TYPE
```
 - Declare *variable of* a derived type via

```
TYPE (<type>) :: myVar ! Instance of <type>
```
 - Access components with selector “%”:
 - (Note: in C the selector is “.”)

```
x = myVar%foo
```

Example 1: Complex numbers

- Suppose Fortran did not provide complex numbers. We might do something like:

```
type Complex  
    real :: real = 0. ! not required  
    real :: imag = 0.  
end type Complex
```

```
function add(z1, z2) result(z3)  
    type (Complex) :: z1, z2, z3  
    z3%real = z1%real + z2%real  
    z3%imag = z1%imag + z2%imag  
end function add
```

Example 2: LatLonGrid

- We can have dynamically sized components, but must use POINTER instead of ALLOCATABLE
 - Fixed in F2003
- Computational grids involve a number of highly related values that should be grouped together.
 - Reduces duplication when model has multiple grids

```
type LatLonGrid
  integer :: numLat, numLon, numLev
  real    :: dlat, dlon
  real, pointer :: latitudes(:)
  real, pointer :: longitudes(:)
  real, pointer :: pressureLevels(:)
end type LatLonGrid

type (LatLonGrid) :: atmosGrid
type (LatLonGrid) :: oceanGrid
```

Example 2: cont'd

- We can now return to the array allocation example from the pointer section:

```
function newGridArray(grid) result(array)
  type (LatLonGrid) :: grid
  real, pointer :: array(:, :, :)
  allocate(array(grid%numLat, grid%numLon, &
    & grid%numLev))
end function newGridArray

...
real, pointer, dimension(:, :, :) :: u, v, w
type (LatLonGrid) :: atmosGrid
u => newGridArray(atmosGrid)
v => newGridArray(atmosGrid)
w => newGridArray(atmosGrid)
```

Example 3: Array of pointers

- Might want to try:

```
real, pointer :: ptrs(:)
```

- Unfortunately this is the syntax for an array pointer
 - Ambiguity inherent in F90 notation
- Instead, one can do this:

```
type MyPointer
    real, pointer :: ptr
end type MyPointer
type (MyPointer) :: ptrs(:)
```

- Arises more often than one might think
-

Example 4: Nesting

```
type Field
  real, pointer :: values(:, :, :) ! 3d
  type (Grid), pointer :: gridReference
  character(len=80) :: longName
  character(len=80) :: shortName
  character(len=80) :: units
end type Field

type Bundle
  type (Field), pointer :: fields(:)
end type Bundle
```

Example 5: Circular

- F90 permits circular type definitions *if* a pointer is used to interrupt infinite regress:

```
type LinkedList
  type (LinkedList), pointer :: next
  type (LinkedList), pointer :: parent
  real :: value
end type LinkedList
```

```
function getNext(list) result(next)
  type (LinkedList) :: list
  type (LinkedList), pointer :: next
  next => list%next
end function getNext
```

Put Type Definitions in Modules

- When defined inside procedures can only be used in that procedure
 - Cannot use “same text” to declare type elsewhere – treated as *different* types
- Passing types to procedures requires *explicit* interface – trivial for modules
- Type can be declared public/private just like module variables
 - Oddity: can have PUBLIC variables of a PRIVATE type
- Type *components* are default PUBLIC, but can be declared PRIVATE
 - Cannot specify per component (fixed F2003)
 - PRIVATE enables encapsulation

Module/Type Design Philosophy

- Choose *good meaningful* names – make code readable
- Module defines one derived type
 - One module per file (make it easy to name & find)
 - Declare derived type PUBLIC
 - Declare derived type *components* PRIVATE
 - Few if any module variables
 - possibly some parameters
- Module contains any procedures that need to directly manipulate type components
 - Create trivial “accessor” procedures that get/set components
 - Other modules/procedures work with type at an abstract level
 - Make 1st argument the derived type argument
 - Think of subroutines as methods which act on the type
- Many/most changes to derived type will not require changing external code - *encapsulation*

Eliminating Common Blocks

- Why would we want to?
 - Promote reuse
 - Global entities complicate reusing procedure in other context
 - Using procedure arguments does not
 - But ... passing long lists of arguments is also bad
- Derived types give us means to pass large collection of data to subroutine with a modest argument list.

Recipe: Eliminating a Common Block

1. Create a container module
2. Declare derived type
 - a) For each member of common block create a similar type *component*
 - b) Declare *components* PUBLIC (for now)
3. Declare a global variable of new type
4. Pass variable as argument to routines that use common block
 - a) Replace references to common with references to *components*
 - b) For large codes, create 2 routines to copy type to/from common
 - Place in container module
 - Enables gradual transition as opposed to wholesale slaughter
 - c) Try to identify suite of “helper” procedures that could contain most component references – push them out of external procedures
 - If successful – declare components PRIVATE
5. Delete common block (rinse and repeat)

Generic programming

- Use F90 “overloading” to express functional similarities
 - Enhances understanding of code
 - Reduces magnitude of change if a type changes (procedure name does not)
- E.g. checkpoint()

```
interface checkpoint
  module procedure checkpoint_grid
  module procedure checkpoint_field
end interface

...
subroutine checkpoint_grid(this, unit)
  type (Grid) :: this
  integer :: unit
  ...
end subroutine checkpoint_grid
subroutine checkpoint_field(this, unit)
  type (Field) :: this
  integer :: unit
  ...
end subroutine checkpoint_field
```

Keyword arguments

- F90 allows arguments to be passed by keyword instead of by order
 - Requires *explicit* interface
 - All arguments after first keyword must also be passed by keyword
- Example

```
subroutine print(name, unit)
...
end subroutine

...
call print(unit=5, name='Bob')
call print('Bob', unit) ! equivalent
```

Optional arguments

- F90 allows arguments to be *optional*
 - No actual argument is required for corresponding optional dummy.
 - Declare with **OPTIONAL** attribute on dummy
 - Optional args must be *after* all non-optional args
 - Best practice: use keyword for actual argument
- Illegal/undefined to reference optional dummy if no actual was passed
 - Intrinsic **PRESENT**(<dummy>) returns .true. if-and-only-if an actual has been passed
 - **Caution:** Fortran does not “short-circuit”
Common mistake:

```
if (present(x) .and. x > 0) ... ! illegal
if (present(x)) then
    if (x > 0) ...
end if
```

Default values and optional args

- Quite often an optional argument is associated with a *default* value that should be used when actual is not PRESENT.
 - Consistent style can improve legibility
 - Introduce similarly named *local* variable
 - Provide default value
 - Override if actual is present

```
Subroutine foo(x, y, flag)
...
logical, optional :: flag
logical :: flag_
flag_ = .true. ! Default value
if (present(flag)) flag_ = flag
...
```

Best practices summary

- Pointers
 - Use ALLOCATABLE for dynamic allocation except for
 - Data structure components
 - Procedure args and function return values
 - Initialize global pointers with NULL()
 - Do not use associated() with uninitialized pointers
 - Modules
 - Prefer default PRIVATE
 - Data structures
 - Choose *good* names
 - Group entities that are tightly *related*
 - Private components, public type
 - Co-locate structure definition with routines that make heavy use of components
-

Best practices (cont'd)

- Optional arguments
 - Always check with **PRESENT** ()
 - Use keyword with optional
 - Use sparingly
 - Generally at most one optional argument
 - Prefer overloading in most situations



Questions
