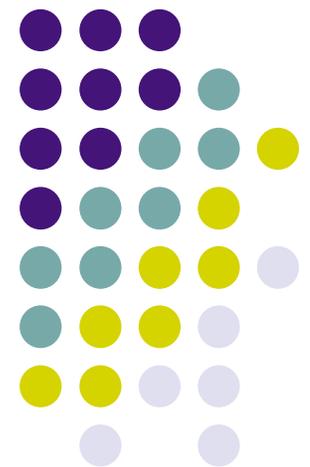


OOP: Polymorphism in F2003

Tom Clune
SIVO Fortran 2003 Series
May 20, 2008





Logistics

- Materials for this series can be found at <http://modelingguru.nasa.gov/clearspace/docs/DOC-1375>
 - Contains slides and source code examples.
 - Latest materials may only be ready at-the-last-minute.
- Please be courteous:
 - Remote attendees should use “*6” to toggle the mute. This will minimize background noise for other attendees.



Outline

- Procedure variables
- Class polymorphism
 - CLASS
- Template polymorphism
 - Parameterized Types



Procedure Variables

- F2003 permits the declaration of variables associated with procedures.
 - Can have explicit or implicit interface
 - Can have the POINTER attribute.
 - Cannot be generic nor elemental.
- Common uses:
 - Dummy arguments
 - Dynamic pointers
 - Components in derived types



Procedure Pointer Syntax

- A typical declaration of a procedure pointer:
PROCEDURE (*[proc-interface]*), POINTER :: p [\Rightarrow null()]
 - *Proc-interface* can be any of:
 - Empty - implicit interface (external subroutine)
 - Interface block or reference procedure - explicit interface
 - Type - implicit interface (external function)
 - " \Rightarrow null() " is optional
- Other combinations are possible. E.g.
`real, external, pointer :: realFunc`
- Note that the syntax is cleaner than that for C, but a bit more verbose in most cases.



Assigning Procedure Pointers

- Pointer assignment follows the same conventions as other portions of the standard:

```
ptr => target
```

- Example 1: subroutine reference

```
proc => mySub  
call proc(a,b)
```

- Example 2: function evaluation:

```
fp => exp  
sum = sum + fp(1.45)
```



Explicit Interface

- Example 1: Prototype procedure

```
procedure (square), pointer :: power => null()
```

```
...
```

```
real function square(x)
```

```
    square = x*x
```

```
end function square
```

- Pointer “power” can point to any real valued function which has 1 real argument.
- Example 2: *Abstract* interface - useful when no prototype interface is available in current scope:

```
abstract interface
```

```
    subroutine processInterface(...)
```

```
...
```

```
    end subroutine processInterface
```

```
end interface
```

```
procedure (processInterface), pointer :: process
```

- Pointer “process” can point to any procedure with the same interface as defined in the interface block.



Function return

- Procedure pointers can be the return value of a function:

```
function getMethod(name) result(fPtr)
  character(len=*) :: name
  procedure(interface), pointer :: fPtr

  select case (trim(name))
  case ('a')
    fPtr => method1
  case('b')
    fPtr => method2
  case default
    fPtr => null()
  end select case
end function getMethod

...
procedure(interface)  pointer :: p
p => getMethod('a')
```



Procedure Components

- Procedure pointers can be components within a derived type.
 - Essentially the same syntax as for type-bound procedures
 - Default behavior is to pass the “object” as the first argument when using the component.
 - PASS/NOPASS attribute can be used to override default
 - Key differences
 - Procedure pointers are *data* components (go above CONTAINS clause)
 - Procedure pointers are more general - can be used to point to arbitrary “behavior”.
 - Best to use type-bound procedures if same behavior is expected of all instances for a given derived type.



Proc Component Example

- The following example is an example that might be useful within the context of a nonlinear solver.
 - Procedure components can use other data elements without explicit reference within the solver.
 - Especially powerful with inheritance

```
type :: Nonlinear
  integer :: coefficient
  real :: tuningParam
  procedure(NonlinInterface), pointer :: eval
end type Nonlinear
type (Nonlinear) :: myObj
...
myObj % eval => complicatedFunctionOf_x
...
subroutine solver(obj, x0, x1)
  type (Nonlinear) :: obj
  real :: x0, x1
  xGuess = ...
  y = obj % eval(x)! Uses obj%coefficient and obj%tuningParam
end subroutine solver
```



Example: Table of procedures

```
type ProcPtr ! To support array
  character(len=MAXLEN) :: name
  procedure(interfc), pointer :: p => null()
end type ProcPtr
type ProcedureTable
  type (ProcPtr), allocatable :: list(:)
contains
  procedure :: addProcedure
  procedure :: getProcedure
end type
```



Polymorphic variables

- A variable declared with the keyword **CLASS** (instead of TYPE) is *polymorphic*:
CLASS (declaredType) ...
 - Variable can take on the specified type *or any* of its extensions during execution.
 - Type at any given point in execution is 'dynamic type'.
 - Type in declaration is the 'declared type'.
- Restrictions: must be either pointer, allocatable, or a dummy argument.
 - Variable gets dynamic type from allocation, pointer assignment, or argument association.
- Unlimited polymorphic entities: **CLASS** (*)
 - Type compatible with *any* data type *including* intrinsics.



Polymorphic assignment

- Derived-type *intrinsic* assignment
 - Extended to allow RHS (**NOT** LHS) to be polymorphic
 - Types must conform - RHS may have dynamic type that is extension of LHS
 - Components of LHS are copied to corresponding components of RHS
- *Pointer* assignment
 - Pointer is required to be type compatible with target
 - Kind-type parameters must be the same (see parameterized types)
 - If polymorphic - assumes the dynamic type of the target
 - Exception: pointer may be of a sequence derived type when the target is unlimited polymorphic and has that derived type as its dynamic type.



Example usage

```
Type Point
```

```
    real :: x,y
```

```
End Type Point
```

```
Type, extends(Point) :: Point3D
```

```
    real :: z
```

```
End Type Point3D
```

```
Type, extends(Point) :: ColorPoint
```

```
    real :: r, g, b
```

```
End type ColorPoint
```

```
...
```



Example cont'd

```
Type (Point) :: p0
```

```
Type (Point3D), target :: p1
```

```
Type (ColorPoint), target :: p2, p4
```

```
Class (Point), pointer :: p3 => null()
```

Call doSomething(p3) ! No dynamic type yet

p3 => p1

Call doSomething(p3) ! doSomething works with generic type

p4 = p3 ! Not allowed - wrong types

p3 => p2

Call doSomething(p3)

p4 = p3 ! Copy x,y,z,r,g,b from p3

P0 = p3 ! Copy x,y back



Extension to Allocate

- Allocate statement now accepts an optional argument that can specify the dynamic type of a polymorphic object:

```
allocate(var, SOURCE=other)
```

- Allocation also copies *source* into *var*
- Declaration type and dynamic type of *source* must be type compatible

- Examples:

```
type (extended) :: foo
type (other) :: bar
class(base), allocatable :: var
allocate(var, SOURCE=foo)
! Var dynamic type is "extended"
deallocate(var)
allocate(var, SOURCE=bar)
! Var dynamic type is now "other"
```



Inquiry intrinsic functions

- `SAME_TYPE_AS (A , B)`
 - Returns scalar default logical
 - True if A and B have the same *dynamic* type
- `EXTENDS_TYPE_OF (A , MOLD)`
 - Returns scalar default logical
 - True if dynamic type of A is an extension of the type of MOLD
 - True if MOLD is unlimited polymorphic, disassociated pointer, or unallocated allocatable.



Select Type Construct

- Compiler does not ‘know’ about dynamic type of polymorphic entity.
 - Use can only access properties (methods/components) of *declared* type
- Access to other components of *dynamic* type is through the SELECT TYPE construct
 - Similar to SELECT CASE, but for dynamic types
 - Analog of ‘dynamic_cast’ in C++
 - TYPE IS (*type*) - dynamic type exactly matches *type*
 - CLASS IS (*type*) - dynamic type is *type* or any extension
 - If more than one matches, the one that is an extension of all others is chosen.
 - CLASS DEFAULT - matches any.
 - Within each block, variable acts as if declared type is given on the TYPE IS or CLASS IS clause.



Select Type Example

```
TYPE (animal) :: a
TYPE (vertebrate) :: b ! Extends animal
TYPE (mammal) :: c ! Extends vertebrate
TYPE (cat) :: d ! Extends mammal
TYPE (primate) :: e ! Extends mammal
CLASS (animal), pointer :: pet
```

```
SELECT TYPE (pet) ! Can also use assoc. (p => pet)
TYPE IS (primate)
    call pet % swingOnBranch()
TYPE IS (cat)
    call pet % sharpenClaws()
CLASS IS (mammal)
    call pet % shedHair()
CLASS DEFAULT
    ...
END SELECT TYPE
```



Advanced Example - Decorate

- Combine polymorphism and “aggregation” to get new effects
 - Override class methods for *any* extension of base class
 - Still acts as proper subclass
 - E.g. add a diagnostic to certain methods

```
type base
  ...
end type base
type, extends(base) :: decorator
  class (base) :: reference
  ...
end type decorator

type (someExtension) :: a
type (decorator) :: b
b = newDecorator(a) ! Store "a" as reference
```



Parameterized Types

- Sometimes referred to as *parametric polymorphism*
- Allows user-defined derived types that are *parameterized* by ‘kind’ and ‘length’ parameters.
 - Similar capabilities as those provided for intrinsic types
 - KIND parameters are constant (fixed at compile time) and can be used as a KIND parameter for other intrinsic or derived types.
 - Can have a default value
 - LEN parameter is akin to that of the length parameter for character and can be used for
 - Character lengths of character components
 - *Bounds* of array components
- Limited compared to templates in other languages
 - E.g. cannot overload integer and floating point



Parameterized Syntax

```
Integer, parameter :: DP = kind(0.0d0)
```

```
Integer, parameter :: SP = kind(0.0)
```

```
type subregion(kind, im, jm)
```

```
  integer, KIND :: kind = DP ! Default value
```

```
  integer, LEN  :: im, jm
```

```
  real (kind)  :: patch(im, jm)
```

```
End type subregion
```

```
...
```

```
type (subregion(DP, 10, 20)) :: dp
```

```
type (subregion(SP, 20, jm=20)) :: sp ! Named arg
```

```
type (subregion(10, 20)) ! Default DP
```



Parameter Enquiry

- Value of TYPE and KIND parameters can be obtained as:

```
print*, obj % kind  
print*, obj % im, obj % jm
```

- Extension for intrinsic types to match this style

```
character(len=MAXLEN) :: string  
real (Kind=KIND(0.0D0)) :: x
```

```
print*, string % len  
print*, x % kind
```



Allocatable Parameterized

- Deferred types for parameterized allocatable variables are specified at allocation.
 - Non-deferred types must agree

```
Type (subregion(SP, im=10, jm=5) :: a
```

```
Type (subregion(SP, :, :), ALLOCATABLE ::  
    myRegion, b, c
```

! This one copies from the source

```
Allocate(myRegion, SOURCE=a) ! Or
```

! This one does not copy

```
Allocate(subregion(SP, im=10, jm=5) :: b, c )
```



Dummy Variables

- For a dummy argument, an asterisk (*) may be used to indicate an *assumed* value for LEN parameters
 - KIND parameters must be an initialization expression

```
type (subregion(SP,im=7,jm=8)) :: a
call proc(a)
```

...

```
subroutine proc(arg)
  ! im,jm from actual argument
  type (subregion(SP,*,*)) :: arg
end subroutine proc
```



Supported features

- IBM XLF has procedure pointers
- Polymorphic variables generally supported by both IBM XLF and NAG F95
 - Support is somewhat fragile at this time
 - No support yet for Unlimited Polymorphic Entities.
- Parameterized types generally not available at this time.



Pitfalls and Best Practices

- Polymorphic variables must have either the ALLOCATABLE or POINTER attribute or be a dummy variable. (Always get dynamic type from other entity.)
- Polymorphism combined with subclassing is generally safer and clearer than procedure pointers.



Resources

- **SIVO Fortran 2003 series:**
<https://modelingguru.nasa.gov/clearspace/docs/DOC-1390>
- **Questions to Modeling Guru:** <https://modelingguru.nasa.gov>
- **SIVO code examples on Modeling Guru**
- **Fortran 2003 standard:**
<http://www.open-std.org/jtc1/sc22/open/n3661.pdf>
- ***John Reid summary:***
 - <ftp://ftp.nag.co.uk/sc22wg5/N1551-N1600/N1579.pdf>
 - <ftp://ftp.nag.co.uk/sc22wg5/N1551-N1600/N1579.ps.gz>
- ***Newsgroups***
 - <http://groups.google.com/group/comp.lang.fortran>
- ***Mailing list***
 - <http://www.jiscmail.ac.uk/lists/comp-fortran-90.html>