Inheritance

Carlos Cruz

NASA GSFC Code 606 (ASTG)
Greenbelt, Maryland 20771

carlos.a.cruz@nasa.gov

October 25, 2018

Agenda

Inheritance

- Variables as Objects
  - Type bound procedures
- Type Extension and Inheritance
  - Extends attribute
  - Abstract Types
  - Abstract Interfaces

Type-bound Procedures

Or "Procedures Bound to a Type by Name"

- Allows a Fortran subroutine or procedure to be treated as a method on an object of the given type.

  ```
  call object%method(...)
  x = object%func(...)
  ```

  which is equivalent to

  ```
  call method(object, ...)
  x = func(object, ...)
  ```

- This syntax encourages an object-oriented style of programming that can improve clarity in some contexts.
- Procedures can also be bound by type to operators: [=, +, etc.]

Type-bound Syntax

The following example defines a derived type with 2 type bound procedures compute() and retrieve()

```
1  type my_type
2     real :: value
3  contains
4     procedure :: compute
5     procedure :: retrieve
6  end type my_type
```

- Type-bound procedures must be module procedures or external procedures with explicit interfaces.
- By default type-bound names are public, but each entity in a type (including components) can be have a PUBLIC/PRIVATE attribute.

Example

```
1   module myType_mod
2      private  ! information hiding
3      public :: my_type ! except
4      type my_type
5           real :: my_value(4) = 0.0
6      contains
7           procedure :: write
8           procedure :: reset
9      end type my_type
10  contains
11     subroutine write (this, unit)
12     class(my_type) :: this
13        integer, optional :: unit
14        if(present(unit)) then
15           write (unit, *) this %
           my_value
16        else
17           print *, this%my_value
18         endif
19     end subroutine write
20  ...
```

```
1   ...
2   subroutine reset(variable)
3        class(my_type) :: variable
4   variable%my_value = 0.0 end subroutine
           reset
5   end module myType_mod
```

Usage:
use mytype_mod
type(myType) :: var
...
call var%write(unit=6)
call var%reset()

Passed-object dummy arg

By default, type-bound procedures pass the object as the first argument.

- Can override behavior with NOPASS attribute.
  procedure, NOPASS :: method
  ...
  call thing % method(...) ← No object is passed
- Can also specify which argument is to be associated with the passed-object with the PASS attribute:
  procedure, PASS(obj) :: method
  ...
  subroutine method(x,obj,y)
  ...
  call thing % method(x,y) ← Thing is 2nd obj
- The default can be explicitly confirmed by
  procedure, PASS :: method

Strongly recommend that you always use the default.

Renaming and Generic

- Type-bound procedures can specify an alternative public name using a mechanism analogous to that for the module ONLY clause:

  procedure :: write => writeInternal

  ...

  call thing % method(...) ← No object is passed

- Similarly, an external name can be **overloaded** for multiple interfaces with the GENERIC statement:

  type myType
  contains
    procedure :: addInteger
    procedure :: addReal
    generic :: add => addReal, addInteger
  end type

Inheritance

Fortran 2003 introduces OOP inheritance via the EXTENDS attribute for user defined types.

- Implementation is restricted to single inheritance.
  - Inheritance always forms hierarchical trees.
- Implementation is designed to be efficient such that offsets for components and type-bound procedures can be computed at compile time. ("single lookup")

With *type extension*, a developer may add new components and type-bound procedures to an existing derived type even *without* access to the source code for that type.

Inheritance Terminology

- An extensible type without the EXTENDS attribute is considered to be a 'base type'.
  - Base types need not have any components.
  - Extension need not add any components.
- A type with the EXTENDS attribute is said to be an extended type.
  - 'Parent type' is used for the type from which the extension is made.
  - All the components, and bound procedures of the parent type are inherited by the extended type and they are known by the same names.

Syntax for Extends

```
1  type Location2D
2     real :: latitude, longitude
3  end type Location2D
4
5  type, EXTENDS (Location2D) :: Location3D
6     real :: pressureHeight
7  end type Location3D
8     ...
9  type (Location3D) :: location
10 lat = location % latitude
11 lon = location % longitude
12 height = location % pressureHeight
```

The Parent Component

- Every extended type has an *implicit component* associated with the parent type
  - The component name is the type name of the parent.
  - Provides *multiple* mechanisms to access components in parent type

From the previous example we could do:

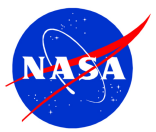
```
1 type (Location2D) :: latLon
2 latLon = location % Location2D
3 lat = location % Location2D % latitude
```

Extends and Type-bound

- Type-bound procedures in the parent may be invoked within extended types.
- Extended types may add additional type-bound procedures in the natural fashion.
- An extended type can *override* a type-bound procedure in the parent - specifying new behavior in the extended type.
  - The keyword NON_OVERRIDABLE can be used to prohibit extended classes from overriding behavior:
    procedure, NON_OVERRIDABLE :: foo

Overriding Example

```
1   type square
2      real :: length
3   contains
4      procedure :: area => square_area
5   end type square
6
7   type, extends(square) :: rectangle  ! inherits area
8      real :: width
9   contains
10     procedure :: area => rectangle_area  ! overriding area
11  end type rectangle
12
13  real function square_area(this)
14     square_area = (this % length) ** 2
15
16  real function rectangle_area(this)
17     rectangle_area = (this % length) * (this % width)
```



Abstract Types

- It is often useful to have a base type that declares methods (type-bound procedures) that are not implemented except in extended classes.
- Fortran 2003 uses the ABSTRACT attribute to denote such a type.
  - The DEFERRED attribute is used for those methods which are not to be implemented.
  - No variables can be declared to be of an abstract type.

Abstract Example

```
1  type, ABSTRACT :: abstract_shape
2  contains
3     procedure (area_interface), DEFERRED :: area
4  end type abstract_shape
5  ...
6  ABSTRACT interface
7     subroutine area_interface(obj)
8        import abstract_shape
9        class (abstract_shape) :: obj
10    end subroutine area_interface
11 end interface
12 ...
13 type, EXTENDS(abstract_shape) :: square
14    real :: length
15 contains
16    procedure :: area => square_area !  Provide concrete
17 end type square
```

Shape Class Exercise

Let's look at the shape class files. Then

- Create a new file triangle_mod.F90 that contains
  - a constructor that creates a triangle object.
  - a function that calculates the area of a triangle.
- edit test_shapes.F90 and add code to print the result

Then build the executable and run the code using Makefile_exercise.