# Pointers and OOP in Fortran

**Dr. Axel Kohlmeyer**

Assistant Dean for High-Performance Computing
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

**a.kohlmeyer@temple.edu**

# Pointers in Fortran vs. C/C++

- In Fortran “POINTER” is an attribute to be used when declaring a variable (like “PARAMETER” or “ALLOCATABLE”)
- In C/C++ a pointer is kind of variable (stores a memory address) with an associated type
- Unlike in C/C++ when using a pointer variable in Fortran, you access the data it is pointing to, i.e. it is more like a reference
- To make a pointer variable point to a “target”, you need to use the association operator: =>
  Associate with NULL() to “disconnect” a pointer
- ASSOCIATED() returns .true. for pointer associated to a target
- Only variables that have the “TARGET” attribute added or other “POINTER” variables or NULL() can be used for associations

# Simple Example using Pointers

```
PROGRAM pointers
  REAL, POINTER :: q => NULL()
  REAL, TARGET  :: c = 0.0, d = -1.0

  PRINT*,ASSOCIATED(q)  ! prints F
  q => c
  PRINT*,ASSOCIATED(q)  ! prints T
  PRINT*,c,q            ! prints 0.0  0.0
  c = 1.0
  PRINT*,c,q            ! prints 1.0  1.0
  q = 2.0
  PRINT*,c,q            ! prints 2.0  2.0
  q => d
  PRINT*,c,q            ! prints 2.0 -1.0
END PROGRAM pointers
```

# POINTER versus ALLOCATABLE

- Variables with the POINTER attribute can also be used with ALLOCATE() and DEALLOCATE()
- This is considered equivalent to defining an “anonymous” “ALLOCATABLE,TARGET” variable and then associating the pointer with it
- Because of the “anonymous” nature there is no automatic deallocation when the pointer goes out of scope (unlike for ALLOCATABLE), so the POINTER behaves like ALLOCATABLE, SAVE
- => use DEALLOCATE() to avoid memory leaks

# Pointers in Derived Types

- The POINTER attribute may also be used for members of a derived type and include references to its own type. Derived types can then be allocated and associated. Example:

```
TYPE llitem
    TYPE(llitem), POINTER :: next => NULL()
    INTEGER :: val
END TYPE llitem
TYPE(llitem), POINTER :: head, item => NULL()

ALLOCATE(item)
item%val = 1
head => item
```

# Building a Data Structure

- The code from the previous example can be extended to build a longer linked list:

```
TYPE(llitem), POINTER :: item, head

ALLOCATE(item)
item%val = 1   ! item%next defaults to NULL()
head => item
ALLOCATE(item)
item%val = 2
item%next => head
head => item
ALLOCATE(item)
item%val = 3
item%next => head
head => item
```

# Traversing a Data Structure

- After the code from the previous example we have a linked list with 3 items. Now we want to output its values:

```
item => head
DO WHILE (ASSOCIATED(item))
    PRINT*,item%val
    item => item%next
END DO
```

- We can delete the linked list in a similar fashion:

```
DO WHILE (ASSOCIATED(head))
  item => head%next
  DEALLOCATE(head)
  head => item
END DO
```

# Type-Bound Procedures (1)

- First step toward classes in Fortran is to include functions and subroutines into derived types. these are called type-bound procedures:

```
MODULE zoo
  PRIVATE
  TYPE animal
     CHARACTER(len=8) :: word=' '
  CONTAINS
     PROCEDURE :: say
  END TYPE ANIMAL
  PUBLIC :: animal
CONTAINS
  SUBROUTINE say(this)
  CLASS(animal) :: this
    PRINT*,'This animal says',this%word
  END SUBROUTINE say
END MODULE zoo
```

# Type-Bound Procedures (2)

- Derived type needs to be defined in a module
- Type-bound procedure must have the derived type as the first argument, usually named “self” or “this” (same as in Python)
- Instead of TYPE(name) use CLASS(name) for first argument referring to the instance itself
- Contained procedure can be any subroutine or function inside the module (may be private)
- Access with <name>%<procedure>

# Constructors

- The constructor is a function contained in the module (but not type-bound) that has the type of the class as return value
- It is made a constructor by defining an interface that has the same name as the derived type
  → that also allows to overload the constructor

```
MODULE zoo
! [...]
  INTERFACE animal
    MODULE PROCEDURE :: init_animal
  END INTERFACE animal
CONTAINS
  TYPE(animal) init_animal FUNCTION(w)
    init_animal%word = w
  END SUBROUTINE init_animal
END MODULE zoo
```

```
PROGRAM sounds
  USE zoo
  IMPLICIT NONE
  TYPE(animal) :: one, two

  one = animal('woof')
  two = animal('meow')
  CALL one%say
  CALL two%say
END PROGRAM sounds
```

# Destructor

- A destructor is a type-bound procedure that is called automatically if an allocated derived type is deallocated
- It must have a TYPE (not CLASS) as first argument

```
MODULE zoo
  TYPE animal
      CHARACTER(len=10) :: word
    CONTAINS
      PROCEDURE :: say
      FINAL :: theend
  END TYPE animal
CONTAINS
  SUBROUTINE theend(self)
    TYPE(animal) :: self
    PRINT*,'the end of me ',self%word
  END SUBROUTINE theend
  SUBROUTINE say(self)
    CLASS(animal) :: self
    PRINT*, 'Say: ‘,self%word)
  END SUBROUTINE say
```

```
PROGRAM sounds
  USE zoo
  IMPLICIT NONE
  TYPE(animal),ALLOCATABLE :: one
  TYPE(animal) :: two

  ALLOCATE(one)
  one = animal(‘woof’)
  two = animal(‘meow’)
  CALL one%say
  CALL two%say
! this will trigger the destructor
  DEALLOCATE(one)
END PROGRAM sounds
```

# Pointers and OOP in Fortran

**Dr. Axel Kohlmeyer**

Assistant Dean for High-Performance Computing
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

**a.kohlmeyer@temple.edu**