Object-oriented programming – advanced features

Arjen Markus

Deltares

July 4, 2020

More syntax: pass and nopass

Control how the object is passed:

```
module mytypes
    type, extends(mytype) :: mynewtype
        real :: extra
    contains
        procedure, pass(value) :: write => write_mynewtype
    end type mytype

contains
subroutine write_mynewtype( lun, value )
    class(mynewtype) :: value
    integer :: lun
    call value%mytype%write( lun )
    write( lun, * ) 'Extra: ', value%extra
end subroutine
end module
```

More syntax: pass and nopass (2)

Invoking it:

```
use mytypes
type(mynewtype) :: v
call v%write(lun)     ! <= Not different than before!

call write_mynewtype( lun, v ) ! Alternative
```

More syntax: pass and nopass (3)

Do not pass it implicitly:

```
procedure, nopass :: write => write_mynewtype

type(mynewtype) :: v
call v%write( lun, v )    ! <= Now we need to pass it
                          !    ourselves
```

Useful if it works on the class, for instance – the module variables.

Abstract classes and deferred procedures

Types can be made "abstract" – they serve as a template
One or more procedures can be "deferred" – define them in an extended type

```
type, abstract :: mytemplate
    ...
contains
    procedure(calculate_template), deferred :: calc
end type mytemplate
```

Abstract classes and deferred procedures (2)

Extending this abstract type:

```
type, extends(mytemplate) :: mydata
    ...
contains
    procedure(calculate_template) :: calc => calc_mydata  ! <= Concrete
end type mydata
```

Abstract classes and deferred procedures (3)

Abstract interfaces:

```
abstract interface
    subroutine calculate_template( t, x, y )
        import :: mytemplate
        class(mytemplate) :: t
        real :: x, y
    end subroutine calculate_template
end interface
```

Abstract classes and deferred procedures (4)

- Abstract interfaces are applicable to ordinary types too.
- And for procedure pointers, see below

Procedure pointers

New class of pointers:

```
procedure(calculate_template), pointer :: p

p => my_calculation

call p( t, x, y )
call my_calculation( t, x, y )
```

Note: *no syntactical difference!*
The signature must match

Procedure pointers (2)

Each object can have its own routine:

```
type :: mydata
    real :: x
contains
    procedure(calculate), pointer :: calc
end type

type(mydata), dimension(10) :: data

data%calc => calculation_1
data(2)%calc => calculation_2   ! <= slightly different,
                                !    same interface

do i = 1,10
    call data(i)%calc( y ) ! Object data(2) does it differently
enddo
```

Finalizers

- You may need to do something special if the object ceases to exist.
- For instance: close a file, release memory that was accessed via a pointer.
- This happens if the object is local to a routine or is explicitly deallocated.
- That is what finalizers are for.

```
module mytypes
    type :: mydata
        real, dimension(:), pointer :: px
    contains
        final :: mydata_release_px
    end type
contains
subroutine mydata_release_px( data )
    type(mydata) :: data
    deallocate( data%px )
end subroutine my_data_release_px
end module mytypes
```

Generic procedures

Generic type-bound procedures work in a similar way as ordinary generic procedures. But they require a slightly different syntax.

```
module mytypes
    type :: mycollection
        integer :: ivalue
        real    :: rvalue
    contains
        procedure :: get_r :: get_real
        procedure :: get_i :: get_int

        generic   :: get => get_real, get_int
    end type
contains
subroutine get_real( c, x )  ! Similar for integer
    type(mycollection) :: c
    real               :: x
end subroutine get_real
end module mytypes
```

More details

Type-bound procedures (methods) can be "protected" against overriding:

```
type :: mycollection
    ...
contains
    procedure, non_overridable :: get => get_int
end type
```

Differences with C++

Some differences with c++:

- F2003 does not define C++-style constructor routines.
- It does define finalizer routines
- F2003 does not allow multiple inheritance
  - You can extend the interface
  - Or use "aggregation" to handle a lot of the things you would want to do
- In F2003 no difference between calling routines via pointers or via fixed names! This is therefore transparent in the user's code.