Derived types

Arjen Markus

Deltares

June 6, 2018

Derived types

Often data belong together:

- Components of an address or a book reference
- Variables solved on a grid (pressure, flow components, temperature, ...)
- Hydraulic system:
  - Pipes and their connections
  - Pumps and reservoirs

Derived types (2)

You can combine them in a single type:

```
type grid
  real, dimension(:,:), allocatable :: x, y, depth
end type grid

type(grid) :: g
```

Derived types (3)

Pointer or allocatable: dimension can be adjusted to the problem at hand

```
allocate( g%x(10,10), g%y(10,10), g%depth(10,10) )
```

The % is used to identify the parts

Derived types (4)

You can "chain" derived types:

```
type solution
   type(grid) :: g
   type(statevar) :: temp, turb_energy, turb_diss
end type
type(solution) :: s

call print_grid( s%g%x, s%g%y )
```

Derived types (5)

You can refer to the same type:

```
type list
   type(list), pointer :: next
   type(list_data) :: data
end type
```

Useful to build recursive data structures (lists, trees, ...)

Derived types (6)

You can define arrays of derived types:

```
type(grid), dimension(10) :: multigrid
multigrid(1)%g = g
```

Note: this uses *default assignment*:

- For fixed-size and allocatable components: the values are copied
- For *pointer* components: the reference to the memory pointed to is copied

Another use ...

Computer data do not have a unit:

```
length      = 10 ! meters
temperature = 12 ! degrees Centigrade

write(*,*) length+temperature
```

Nothing prevents this! But it is nonsense

Another use (2)

But what if:

```
type length_t
    real :: len
end type length_t

type temp_t
    real :: tmp
end type

type(length_t) :: length      = 10
type(temp_t)   :: temperature = 12

write(*,*) length+temperature
```

The compiler will complain

Another use (3)

Define operations for `length_t`:

```
module lengths
  type length_t
      real :: len
  end type
  interface operator(+)
     module procedure add_lengths
  end interface
contains
function add_lengths( x, y ) result(r)
  type(length_t), intent(in) :: x, y
  type(length_t) :: r
  r%len = x%len + y%len
end function add_lengths
end module
```

Another use (4)

Then the following is legal:

```
type(length_t) :: len1, len2

len1%len = 10
len2%len = 20

write(*,*) len1+len2
```

Assigning and printing

You can do:

```
type(length_t) :: len1, len2

len1 = len2

write(*,*) len1, len2
```

The compiler provides default assignment and printing.
However: printing like this will not work with pointer components.

Assigning and printing (2)

Limitations:

- With pointer components no automatic allocation! (Instead a reference to the same memory!)
- Printing of derived types with pointer components is not allowed
- Rationale: *endless loops are possible!*

Solutions:

- Define your own assignment
- Print individual components or define "user-defined I/O routines"

Assigning and printing – caveat

Copying grids (x is a pointer component):

```
type grid_t
    real, dimension(:,:), pointer :: x, y
end type grid_t

type(grid_t) :: grid_out, grid_in

grid_out = grid_in

! Shift the new grid ...
grid_out%x = grid_out%x + 10.0
```

Problem: as `grid_out%x` refers to the same memory as `grid_in%x`, the original grid is also shifted!
If you use *allocatables*, this will not happen.

Assigning and printing – user-defined assignment

Supplying your own assignment routines:

```
module grids
   type grid
       real, dimension(:,:), pointer :: x, y
   end type
   interface assignment(=)
      module procedure copy_grid
   end interface
contains
subroutine copy_grid( out, in )
   type(grid), intent(out) :: out
   type(grid), intent(in)  :: in
   allocate( out%x(...), ... )
   out%x = in%x
   ...
end subroutine
```

Initialisation

You can set the values of each component separately or via a "constructor":

```
type(address_t) :: address
address%name = "Jansen"
address%city = "Delft"

address = address_t( "Jansen", "Delft") ! Alternative
```

Also: Make the initial value part of the definition

```
type grid
   integer :: type = 1
   real, dimension(:,:), pointer :: x => null()
   real, dimension(:,:), pointer :: y => null()
end type
```

Parameterised types

Derived types can have *parameters*:

```
type matrix_t(real_kind,n,m)
   integer, kind :: real_kind
   integer, len  :: n, m

   real(kind=real_kind) :: value(n,m)
end type matrix_t

type(matrix_t(kind(0.0),17,23) :: matrix
type(matrix_t(real_kind=kind(0.0),n=17,m=23)) :: matrix
```

Parameterised types (2)

In a routine the declaration would look like this (comparable to strings):

```
subroutine print_matrix( matrix )
   type(matrix_t(kind(0.0),n=*,m=*), intent(in) :: matrix
   ...
end subroutine
```

Note: the kind is fixed!

Pointers/allocatables:

```
type(matrix_t(kind(0.0),n=:,m=:), pointer    :: pm
type(matrix_t(kind(0.0),n=200,m=300), target :: matrix

pm => matrix
```

Parameterised types (3)

Default parameters:

```
type matrix_t(real_kind,n,m)
   integer, kind :: real_kind = kind(0.0)
   integer, len  :: n, m

   real(kind=real_kind) :: value(n,m)
end type matrix_t

type(matrix_t(n=10,m=10)) :: matrix
```

Inquiring about the type parameters:

```
write(*,*) matrix%real_kind
write(*,*) matrix%n, matrix%m
```