



Derived Types



Reusing this material



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

http://creativecommons.org/licenses/by-nc-sa/4.0/deed.en_US

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Note that this presentation may contain images owned by others. Please seek their permission before reusing these images.



Derived data types

- Fortran allows the use of derived data types
 - Groups of data structures
 - Enables building of more sophisticated types than the intrinsic ones, i.e. linked data structures, lists, trees etc...
- Imagine we wish to specify objects representing persons
 - Each person is uniquely distinguished by a name and room number
 - We can define a corresponding “person” data type as follows:

```
type person
  character (len=10) :: name
  integer           :: officeNumber
end type person
```



Derived data types

- To create a derived type variable you use the syntax:

```
type(person) :: fred, me
```

- Initialisation (construction) possible as well:

```
fred = person("Fred Jones", 21)
```

- **fred** is a variable containing 2 elements: **name**, **officeNumber**

- Elements (individual components) of derived type can be accessed by component selector: %

```
fred%name           ! contains the name of you
```

```
fred%officeNumber  ! contains the age of you
```



Derived data types

- Can perform computations using derived type variables as follows:

- Difference in officeNumber between variables `fred` and `me`

```
integer :: officeNumberDiff
```

```
officeNumberDiff = fred%officeNumber - me%officeNumber
```

- I/O access components in defined order, i.e.:

```
fred%name
```

```
fred%officeNumber
```



Supertypes

- Derived type can be used in other derived types:

```
type corridor
  type (person), dimension (:), allocatable :: rooms (:)
  integer :: numberOfRooms
end type corridor
```

```
type (corridor) :: a1
...
a1%rooms(1)%name
a1%numberOfRooms = 10
```



Further example

```
TYPE COORDS_3D
  REAL :: x, y, z
END TYPE COORDS_3D
TYPE SPHERE
  TYPE(COORDS_3D) :: centre
  REAL :: radius
END TYPE SPHERE
TYPE(SPHERE) :: ball
type(coords_3d) :: pt1
pt1 = COORDS_3D(3.0, 4.0, 5.0)
ball = SPHERE(centre=pt1, radius=5.0)
```



Summary

- Derived types can provide class like features for data
 - Package up similar/related data together
 - Use composition to build on other types
 - Don't bring functions together with the data
- Derived types can be included in modules
 - Together with module procedures can provide class like functionality
 - Module private can restrict data to module procedures only
 - Module private can restrict procedures to module procedures only
- Derived types and modules together can provide basic OO-like functionality
 - Does not necessarily provide proper data/procedure control
 - Does not provide inheritance
 - Can provide composition and basic polymorphism



Exercise

- Derived type basic exercises
- A separate CFD exercise
- Create appropriate derived types for the percolate exercise
 - What data structures could be grouped together? Which module should it be created in?

