Introduction to Modern Fortran

Derived Types

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Summary

There is one important new feature to cover

It is not complicated, as we shall do it

- But we won't cover it in great depth

Doing it fully would be a course in itself
The same applies in other languages, too

What Are Derived Types?

As usual, a hybrid of two, unrelated concepts
C++, Python etc. are very similar

- One is structures – i.e. composite objects

Arbitrary types, statically indexed by name

- The other is user–defined types

Often called semantic extension
This is where object orientation comes in

- This course will describe only the former

Why Am I Wimping Out?

Fortran 2003 has really changed this

- full object orientation
- semantic extension
- polymorphism (abstract types)
- and lots more

The course was already getting too big
And, yes, I was getting sick of writing it!

This area justifies a separate course
About one day or two afternoons, not three days
Please ask if you would like it written

Simple Derived Types

```
TYPE :: Wheel
    INTEGER :: spokes
    REAL  :: diameter, width
    CHARACTER(LEN=15) :: material
END TYPE Wheel
```

That defines a derived type Wheel
Using derived types needs a special syntax

```
TYPE(Wheel) :: w1
```

More Complicated Ones

You can include almost anything in there

```
TYPE :: Bicycle
    CHARACTER(LEN=80) :: description(100)
    TYPE(Wheel) :: front, back
    REAL, ALLOCATABLE, DIMENSION(:) :: times
    INTEGER, DIMENSION(100) :: codes
END TYPE Bicycle
```

And so on ...

Fortran 95 Restriction

Fortran 95 was much more restrictive
You couldn't have ALLOCATABLE arrays
You had to use POINTER instead

Fortran 2003 removed that restriction
You may come across POINTER in old code
It can usually be replace by ALLOCATABLE

Ask if you hit problems and want to check

Component Selection

The selector '%' is used for this
Followed by a component of the derived type

It delivers whatever type that field is
You can then subscript or select it

```
TYPE(Bicycle) :: mine

mine%times(52:53) = (/ 123.4, 98.7 /)
PRINT *, mine%front%spokes
```

Selecting from Arrays

You can select from arrays and array sections
It produces an array of that component alone

```
TYPE :: Rabbit
    CHARACTER(LEN=16) :: variety
    REAL :: weight, length
    INTEGER :: age
END TYPE Rabbit
TYPE(Rabbit), DIMENSION(100) :: exhibits
REAL, DIMENSION(50) :: fattest

fattest = exhibits(51:)%weight
```

Assignment (1)

You can assign complete derived types
That copies the value element–by–element

```
TYPE(Bicycle) :: mine, yours

yours = mine
mine%front = yours%back
```

Assignment is the only intrinsic operation

You can redefine that or define other operations
But they are some of the topics I am omitting

Assignment (2)

Each derived type is a separate type
You cannot assign between different ones

```
TYPE :: Fred
    REAL :: x
END TYPE Fred
TYPE :: Joe
    REAL :: x
END TYPE Joe
TYPE(Fred) :: a
TYPE(Joe) :: b
a = b    ! This is erroneous
```

Constructors

A constructor creates a derived type value

```
TYPE Circle
    REAL :: X, Y, radius
    LOGICAL :: filled
END TYPE Circle

TYPE(Circle) :: a
a = Circle(1.23, 4.56, 2.0, .False.)
```

Or use keywords for components (Fortran 2003)

```
a = Circle(X = 1.23, Y = 4.56, radius = 2.0, filled = .False.)
```

Default Initialisation

You can specify default initial values

```
TYPE :: Circle
    REAL :: X = 0.0, Y = 0.0, radius = 1.0
    LOGICAL :: filled = .False.
END TYPE Circle

TYPE(Circle) :: a, b, c
a = Circle(1.23, 4.56, 2.0, .True.)
```

This becomes much more useful with keywords

```
a = Circle(X = 1.23, Y = 4.56)
```

I/O on Derived Types

Can do normal I/O with the ultimate components
A derived type is flattened much like an array
[recursively, if it includes derived types]

```
TYPE(Circle) :: a, b, c
a = Circle(1.23, 4.56, 2.0, .True.)
PRINT *, a ;   PRINT *, b ;   PRINT *, c

  1.2300000   4.5599999   2.0000000 T
  0.0000000E+00   0.0000000E+00   1.0000000 F
  0.0000000E+00   0.0000000E+00   1.0000000 F
```

Private Derived Types

When you define them in modules

A derived type can be wholly private
I.e. accessible only to module procedures

Or its components can be hidden
I.e. it's visible as an opaque type

Both useful, even without semantic extension

Wholly Private Types

```
MODULE Marsupial
    TYPE, PRIVATE :: Wombat
        REAL :: weight, length
    END TYPE Wombat
    REAL, PRIVATE :: Koala
CONTAINS
    . . .
END MODULE Marsupial
```

Wombat is not exported from Marsupial
No more than the variable Koala is

Hidden Components (1)

```
MODULE Marsupial
    TYPE :: Wombat
        PRIVATE
        REAL :: weight, length
    END TYPE Wombat
CONTAINS
    . . .
END MODULE Marsupial
```

Wombat IS exported from Marsupial
But its components (weight, length) are not

Hidden Components (2)

Hidden components allow opaque types
The module procedures use them normally

- Users of the module can't look inside them

They can assign them like variables
They can pass them as arguments
Or call the module procedures to work on them

An important software engineering technique
Usually called data encapsulation

Trees

E.g. type A contains an array of type B
Objects of type B contain arrays of type C

```
TYPE :: Leaf
    CHARACTER(LEN=20) :: name
    REAL(KIND=dp), DIMENSION(3) :: data
END TYPE Leaf
TYPE :: Branch
    TYPE(Leaf), ALLOCATABLE :: leaves(:)
END TYPE Branch
TYPE :: Trunk
    TYPE(Branch), ALLOCATABLE :: branches(:)
END TYPE Trunk
```

Recursive Types

Pointers allow that to be done a little more flexibly
You don't need a separate type for each level

People often use more complicated structures
You build those using derived types
E.g. linked lists (also called chains)

Both very commonly used for sparse matrices
And algorithms like Dirichlet tesselation

We shall return to this when we cover pointers

Opaque Types etc.

This is another using aspect that has been omitted
It's there in the notes, if you are interested

- Skip the practical that needs that facility