Derived Types

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 only describe the former.

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 90/95 Restriction

Fortran 90/95 was much more restrictive
You couldn't have ALLOCATABLE arrays
Had to use pointers instead

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

Be sure to check your own compiler

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 values 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 that I am omitting

Assignment (2)

Each derived type is unique

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.)
```

Fortran 2003 allows keywords for components

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

Default Initialization

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 in Fortran 2003

```
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 embedded derived types)

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

```
 1.230000  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 :: width, 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 :: width, length
   END TYPE Wombat
   CONTAINS
    . . .
END MODULE Marsupial
```

Wombat IS exported from Marsupial

But its components (width, 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

Example: 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

Code with Derived Types

```
MODULE Delocal
    INTEGER, PARAMETER :: MaxCoords = 1000
    INTEGER, PARAMETER :: MaxAtoms = 100
    TYPE MolecularMechanicsCoords
        INTEGER :: nmm
        INTEGER :: imm(5,MaxCoords)
        REAL :: vmm(MaxCoords)
    END TYPE MolecularMechanicsCoords
    TYPE MMFactors
        REAL :: fudge_a, fudge_b
        REAL :: fvdws(6)
    END TYPE MMFactors
```

```
    TYPE Geometry
        INTEGER :: nat
        INTEGER :: ielem(MaxAtoms)
        REAL :: xyz(3,MaxAtoms)
    END TYPE Geometry
END MODULE Delocal

SUBROUTINE make_vmm(mmCoords, geom, factors)
    USE Delocal
    TYPE(MolecularMechanicsCoords) :: mmCoords
    TYPE(Geometry) :: geom
    TYPE(MMFactors) :: factors
END SUBROUTINE make_vmm
```

Original Code

```
SUBROUTINE make_vmm(xyz, nat, imm, nmm, &
                    vmm, ielem, fudge_a, fudge_b, fvdws)
    INTEGER, INTENT(IN) :: nat, nmm
    INTEGER, INTENT(IN) :: imm(5,nmm), ielem(nat)
    REAL, INTENT(IN) :: xyz(3,nat), fudge_a, fudge_b, &
                        fvdws(6)
    REAL, INTENT(OUT) :: vmm(nmm)
END SUBROUTINE make_vmm
```

Modify the Original Code

```
SUBROUTINE make_vmm(xyz, nat, imm, nmm, &
    vmm, ielem, fudge_a, fudge_b, fudge_c, fvdws)
    INTEGER, INTENT(IN) :: nat, nmm
    INTEGER, INTENT(IN) :: imm(5,nmm), ielem(nat)
    REAL, INTENT(IN) :: xyz(3,nat), fudge_a, fudge_b, &
                                    fudge_c, fvdws(7)
    REAL, INTENT(OUT) :: vmm(nmm)
END SUBROUTINE make_vmm

REAL :: fudge_c, fvdws(7)
CALL make_vmm(xyz, nat, imm, nmm, &
    vmm, ielem, fudge_a, fudge_b, fudge_c, fvdws)
```

Modify the Derived Type Code

```
MODULE Delocal
    . . .
    TYPE MMFactors
        REAL :: fudge_a, fudge_b, fudge_c
        REAL :: fvdws(7)
    END TYPE MMFactors
    . . .

CALL make_vmm(mmCoords, geom, factors)
```