Large-scale simulations with Fortran 95: An object-based approach

Lesson 2

Modules and Derived Types

Paul F. Dubois

X-Division

dubois1@llnl.gov
Fortran 90/95 are significantly different than Fortran 77.

(From the Digital Fortran Reference Manual)
In the first lesson, we concentrated on modules and arrays.

Modules are the replacement for common blocks, and they can hold both data and functions.

We learned that F95 has an array language similar to that in Matlab or Basis, and that arrays can now be passed as self-contained arguments and returned from functions.

However, to receive this and many other benefits we must make the interface to a subprogram explicit by
- putting the subprogram in a module, or
- putting an interface declaration for it in a module, and
- use’ing that module where appropriate
Answers to the homework

1. Write and demo this function:

function stats(x)
    x is a one-dimensional real array. (Use a kind for the array).
    return value is a real array y containing the following items:
    y(1) = mean of x
    y(2) = percentage of the array actually greater than the mean
    y(3) = variance (average the squares of the differences between each
          element and the mean).

2. Write and demo a function which returns the outer product of two
   vectors given as arguments. (If x and y are vectors, the outer product w is
   defined as having elements w(i,j) = x(i) * y(j)).

3. Go rip out a common block in one of your codes. Feel good?
module hw1
    use precision
    implicit none

    contains

    function stats(x) result(y)
      real(adequate), intent(in), dimension(:):: x
      real(adequate) y(3), mean

      mean = sum(x) / size(x)
      y(1) = mean
      y(2) = real(count(x > mean), adequate) / size(x)
      y(3) = sum((x - mean)**2) / size(x)
    end function stats

    function outer (x, y) result(z)
      real(adequate), intent(in):: x(:), y(:)
      real(adequate) z(size(x), size(y))
      integer i

      do i = 1, size(x)
        z(i, :) = x(i) * y
      enddo
    end function outer

end module hw1
program test1
  use hw1
  implicit none

  real(adequate) x(5), y(5)

  x = (/ 1., 2., 3., 4., 5./)
  y = 10 * x

  print *, stats(x)
  print *, outer(x, y)
end program test1
The suggestion was made to use the kind function to make a subroutine even more useful.

That trick unfortunately doesn’t work for arguments, however:

```fortran
function version3 (x) result(y)
! same ideas work for arguments and function results
   use precision
   implicit none
   real(kind(x)), intent(in):: x(:)
   real(kind(x)) y(size(x))

   y = sin(x * pi)
end function version3
```

You can see why: the compiler would have to wait until the routine was called in order to compile the correct version of the routine.
In this lesson we will learn more about
modules and then learn how to define our own
data types.

Each of these ideas is important on their own.

Together, they enable an object-based approach to Fortran programming.
Modules fill several needs.

1. Provide a replacement for common/include.
2. Allow defining of user-defined types and operators.
3. Permit the construction of self-contained software libraries.
4. Provide a means for data abstraction.
You have precise control over the names in a module.

module mod1
  use precision
  private ! makes names private by default
  real(adequate),public:: x, z
  real(adequate) y
  public adequate, precise, display
contains
  subroutine display(label)
    character(len=*),intent(in),optional:: label
    if(present(label)) then
      write(11, *) label
    endif
    write(11, *) x, y, z
  end subroutine display
end module mod1
subroutine example1
  use mod1
  implicit none
  integer y ! does not conflict with the private y in mod1
  x = 3
  y = 2
  call display ("example1")
end subroutine example1
Always begin a module with any use statements followed by implicit none.

module mod2
    use mod1
    ! left out the implicit none, trouble follows...
    private

    public y ! creates a new variable y of default type real
    ! it doesn't succeed in making mod1::y public
end module mod2

subroutine example2
    use mod2
    implicit none
    ! real x does not compile, x still public
    x = 9.0
    y = 7.0
    call display("example2")
end subroutine example2
It is just a matter of style as to which default you use and which method of declaring non-defaults.

I personally favor default private, and a separate “public” statement. This requires an affirmative step to make things public and also makes a nice visible list of the public names.

module foo
    implicit none
    private
    public x, y, z, w
    real, dimension(:), allocatable:: x, y, z, w
    real, dimension(:), allocatable:: a_temporary
contains
    function hidden
    ...
    allocate(a_temporary(size(x)))
    ...
    deallocate(a_temporary)
end module foo
Defining your own types and the operations they support is the key to specializing Fortran for your application area.

A data type comprises:
• a name for the type
• the set of values the instances of the type can assume
• the set of operations the type supports
• a form for writing constants of that type
The native types in a language have names, values, operations, and constants that are already determined.

- Name: integer
- Values: positive and negative integers within some processor-dependent bounds.
- Operations: plus, minus, times, divide, ...
- Constant form: 123, 456_mykind
The derived-type declaration is simply the type name with a list of the desired components.

```fortran
module mod3
  use precision
  implicit none
  public

  type atom
    real(adequate) mass
    integer charge
  end type atom

end module mod3
```
type(atom) now is a type we can use in the same way we use “integer”.

subroutine example3
    use mod3
    implicit none
    type(atom) h, he, hep
    type(atom), parameter:: omm = atom(16.0, -2)

    h = atom ( 1.0, 0)
    he = atom (omm%mass/4, 0)
    hep = he
    hep%charge = 1
    write(11, *) "h ", h
    write(11, *) "he ", he
    write(11, *) "hep ", hep
end subroutine example3
This example illustrates all the main facts about derived types.

• To access a component value, we use the % operator. (Too bad it isn’t a period, but Fortran had to live with .lt., .true., etc.).

• Assignment is equivalent to a series of assignment statements of each corresponding component value.

• The type name also becomes the “constructor” for both constant and transient values.
The type declaration defines three of the four properties of the type.

1. Name: the name given in the declaration
2. Values: the set product of the values possible for each component.
3. Operations: Other operations may be defined, but the type declaration doesn’t do it.
4. The name of the type becomes the name used to form constants of that type. The argument list must list one appropriate value for each component.
To discuss what happens when a component is a pointer, we must digress and learn a little more about pointers.

- Pointers can be “associated” with some target, or “disassociated”. You can test whether a pointer is associated or not, or associated to a certain target, with the intrinsic function `associated`.
- There are two “assignment” operators for pointers:
  1. Regular assignment statement: `the_pointer = some_value`
     This assigns `some_value` to the object to which `the_pointer` is currently “associated”, called its “target”.
  2. Pointer assignment statement: `the_pointer => some_target`
     This changes the association of `the_pointer` to `some_target`. Or, if `some_target` is a pointer also, `the_pointer` is associated to its target.
If a derived type has a pointer component, the normal “assignment” operator does pointer assignment on components which are pointers.

subroutine example3a
  use precision

  type curve
    real(adequate), dimension(:,), pointer:: x, y
  end type curve

  real(adequate), target:: x(4) = (/ (i, i=1, 4) /)
  real(adequate), target:: y(4), z(4)
  real(adequate), pointer:: zp(:)
  type(curve) my_curve

  y = x**2
  zp => z  ! pointer assignment, zp now associated with z
  zp = y   ! assignment, z now has same values as y
  write(11, *) "z ", z

  my_curve = curve(x, y)
  ! due to pointer components, cannot simply write *
  write(11, *) "mycurve%x ", my_curve%x
  write(11, *) "mycurve%y ", my_curve%y
end subroutine example3a
The type name is also used to create new instances of the type at runtime.

So what happened when we said my_curve = curve (x, y)?

1. The curve (x, y) constructs a temporary object T of type curve. It initializes the x component by pointer assignment to x. It initializes the y component by pointer assignment to y.
2. The assignment statement does my_curve%x => T%x. But T%x is also a pointer, so my_curve%x is associated with its target, namely x.
3. Likewise, my_curve%y ends up associated with y.

Object-oriented persons will note that this is the only constructor for the derived type, and that there is no destructor.
Used separately, we see that modules and derived types each have some aspect of classes.

Modules can contain both data and functions, and do information hiding. But there is only one instance of the data in a module.

Derived types have a constructor that can be used as a cookie cutter to produce instances, but they do not contain methods.

Derived types do have the possibility of hiding all or part of their data but you lose some conveniences if you do so.
It is possible to have a public type whose components are private.

module mod4
  use precision
  implicit none
  public

  type atom
    private
      real(adequate) mass
      integer charge
  end type atom

  contains
  ... to be continued

You give up the ability to use the “constructor”

atom (amass, acharge)

outside of the module, since presumably nobody else knows anymore what is inside of an atom.

Likewise, the compiler can no longer write such objects for you using data-directed i/o.
We need to put routines into the module to retrieve and manipulate the components.

This is reminiscent of C++’s common paradigm of private components and public accessors.

```fortran
subroutine atom_new (a, name)
    character*(*), intent(in):: name
    type(atom), intent(out):: a
    select case(name)
      case("H")
        a = atom(1.0, 0)
      case("He")
        a = atom(4.0, 0)
      case("C")
        a = atom(12.0, 0)
      case default
        ! hm...no exception facility
        stop "Illegal atom creation attempted."
    end select
end subroutine atom_new

function atom_mass (a)
    real(adequate) atom_mass
    type(atom), intent(in):: a
    atom_mass = a%mass
end function atom_mass
```
Note the convention of having the first argument be the atom to be operated upon.

```fortran
function atom_charge (a)
    real(adequate) atom_charge
    type(atom), intent(in):: a
    atom_charge = a%charge
end function atom_charge
function atom_as_string (a)
    type(atom), intent(in):: a
    character*(32) atom_as_string
    write (atom_as_string, 100) a%mass, a%charge
100 format("atom( ", e14.4, ", ", i2, ")")
end function atom_as_string
subroutine set_atom_mass (a, m)
    type(atom), intent(inout):: a
    real(adequate), intent(in):: m
    a%mass = m
end subroutine set_atom_mass
subroutine set_atom_charge (a, c)
    type(atom), intent(inout):: a
    integer, intent(in):: c
    a%charge = c
end subroutine set_atom_charge
end module mod4
```
Now we can create and manipulate our atoms with the accessor functions.

subroutine example4
  use mod4
  type(atom) a1, a2, a3
  call atom_new(a1, "H")
  call atom_new(a2, "He")
  call atom_new(a3, "He")
  call set_atom_charge(a3, -1)
  ! write(11, *) a1 (no good, type(a) has private components)
  write(11, *) "atom_mass(a3) ", atom_mass(a3)
  write(11, *) "a3 ", trim(atom_as_string(a3))
end subroutine example4
Generic interfaces can be specified to allow you to treat many different types in a similar manner.

subroutine example5
    use atoms
    use molecules
    type(atom) a1, a2, a3
    type(molecule) m1, m2, m3

    call new(a1, "H")
    call new(a2, "He")
    call new(a3, "He")
    call set_charge(a3, -1)

    call new(m1, "O2")
    call new(m2, "CH4")
    call new(m3, "H2O")
    call set_charge(m3, 0)

    write(11, *) "mass(m2) ", mass(m2)
    write(11, *) "a3 ", trim(as_string(a3))
    write(11, *) "m3 ", trim(as_string(m3))
end subroutine example5
To do this we add interface statements for each new generic function name we want.

module atoms
    use precision
    implicit none
    private

    type atom
        private
        real(adequate) mass
        integer charge
    end type atom

    public atom, mass, charge, set_mass, set_charge, new, &
        as_string

    interface new
        module procedure atom_new
    end interface new

    interface mass
        module procedure atom_mass
    end interface mass
interface charge
   module procedure atom_charge
end interface charge

interface set_mass
   module procedure set_atom_mass
end interface set_mass

interface set_charge
   module procedure set_atom_charge
end interface set_charge

interface as_string
   module procedure atom_as_string
end interface as_string

contains

subroutine atom_new (a, name)
...
... the rest as it was before...
...
end module atoms
Module molecules is very similar.

```fortran
module molecules
  use precision
  implicit none
  private

  type molecule
    private
    real(adequate) mass
    integer charge
    integer number_of_elements, number_of_atoms
  end type molecule

  public molecule, mass, charge, set_mass, set_charge, new, &
    as_string

  interface new
    module procedure molecule_new
  end interface new

  interface mass
    module procedure molecule_mass
  end interface mass

  ...
```
There are pros and cons to using this “private component” style.

Pro: better locality. For example,
- if you have atom \((m, c)\) all over the code, and you get a new component, those references must be all changed. You have more flexibility if there are one or more “constructors” that you write, possibly using keyword or optional arguments.
- OO programmers have learned that if a client can READ a component directly they can WRITE it, possibly creating an object that no longer has a valid value.

In fact, \(a.b\) on the left side of an assignment operator is a red flag to an experienced OO programmer.

Con: takes more work. Besides, writing \(a\%b = c\) is nice and clear and I never make mistakes. And I can do the generic functions whether or not the components are private. Anyway, I don’t trust the compiler to do those accessors efficiently.
Homework

Write a program with a user-defined type “point” that has x, y, z components. Write another user-defined type “quadrilateral” that has a component “vertices” that is an array of 4 points. (For simplicity, make everything public).

Try creating and manipulating some quadrilaterals and arrays of quadrilaterals.

If q is an array of quadrilaterals,
q(1)%vertices(1)%x

is the x’th coordinate of the first vertex of the first quadrilateral in q.

But can you do q(1)%vertices%x, q(1)%vertices(2:3)%x, q(2:4)%vertices%x?

Just how cool is this?

And what about
integer:: iv(3) = (/ 1, 2, 4 /)

real:: integer x(20)

What about:

x(iv)?

q(1)%vertices(iv)?