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

Lesson 3

Types and objects

Paul F. Dubois
X-Division
dubois1@llnl.gov
In our first two lessons, we have concentrated on arrays, modules, and derived types.

It is the combination of types and modules that allows us to work in a style I will call “object-based”.

The homework should have helped you get a feel for creating and working with derived types.
Mini-lesson #3: Give each component in a derived type a default initial value.

```fortran
  type particle
    type(point):: location = point(0.0, 0.0, 0.0)
    type(point):: momentum = point(0.0, 0.0, 0.0)
    real(adequate):: mass = 1.0
    real(adequate):: charge = 0.0
  end type particle
```

The default initial value can be over-ridden by an explicit one.

You must use the “entity-oriented” form of declaration with the two colons.

The default initial value can simplify other coding, and can be used as part of a scheme to implement an “abstract data type”, in which no “illegal” objects of this type can ever be made.
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.

=> The secret is to combine the two.
Mini-lesson #4: Use a module to model singletons.

A singleton is a class which has exactly one instance.

So, if you have some data which should exist exactly once in your application, package it into a module, together with the functions whose exclusive purpose is to manipulate that data.

Then think out what you want the public interface to look like.
The generic interface capability is much more powerful than we have seen so far.

We used it to define a single name that could be applied to objects of varied types. You can also use it to:

- define operators such as + and *;
- add new operators such as .dot., .integrate., or .inverse., of your own choosing;
- take over the assignment operator;
- even extend the meaning of intrinsic names such as sqrt
Here is a module implementing a 3-D point object.

module points
  use precision
  private
  public dot, sqrt, norm
  public operator(+), operator(-), operator(*), operator(/)

  type, public:: point
    real(adequate):: x = 0.0
    real(adequate):: y = 0.0
    real(adequate):: z = 0.0
  end type point

  We’ve defined a new type named “point”. We’ve given every component a default value so there won’t be any uninitialized points.

  Note the alternate syntax used for giving type(point) the attribute “public”.
Now we define the basic arithmetic operators to implement vector arithmetic on points.

We supply three versions of each so that we can do point+point, point+real, real+point, for convenient operations such as translation and scaling.

```fortran
interface operator(+)
    module procedure point_add, point_add_s, point_s_add
end interface

interface operator(-)
    module procedure point_subtract, point_subtract_s, &
        point_s_subtract, point_negate
end interface

interface operator(*)
    module procedure point_multiply, point_multiply_s, &
        point_s_multiply
end interface

interface operator(/)
    module procedure point_divide, point_divide_s, &
        point_s_divide
end interface
```
Module procedures do the real work.

```fortran
function point_add (a, b) result(r)
    type(point), intent(in):: a, b
    type(point):: r
    r%x = a%x + b%x
    r%y = a%y + b%y
    r%z = a%z + b%z
end function point_add

function point_add_s (a, b) result(r)
    type(point), intent(in):: a
    real(adequate), intent(in):: b
    type(point):: r
    r%x = a%x + b
    r%y = a%y + b
    r%z = a%z + b
end function point_add_s

function point_s_add (a, b) result(r)
    type(point), intent(in):: b
    real(adequate), intent(in):: a
    type(point):: r
    r%x = a + b%x
    r%y = a + b%y
    r%z = a + b%z
end function point_s_add

These are all in the procedure part of the module “points”.
```
The compiler chooses an appropriate function based on signature.

type(point) x, y, z

z = x + y  !actually does z = point_add(x, y)
z = x + 1.0 !actually does z = point_add_s(x, 1.0)
z = 1.0 + x !actually does z = point_s_add(1.0, x)
We also define a generic “norm” function, a binary operator for dot product, and an element-wise square root.

Users can write norm(v), v .dot. w, and sqrt(v). And yes, sqrt(2.0) still works!

```fortran
interface norm
   module procedure point_norm
end interface

interface operator(.dot.)
   module procedure point_dot
end interface

interface sqrt
   module procedure point_sqrt
end interface
```
We listed in the interface for each operator the functions that the compiler could use to do the job.

The functions were all in the module itself so all we had to do was list them as “module procedures”.

We could also use functions defined elsewhere by putting interface specifications for them into the interface blocks.

In the “contains” section of the module we put the actual “worker” routines. These are all private to the module.
It is also helpful to add unary minus to the subtract interface.

```fortran
function point_negate (a) result(r)
  type(point), intent(in):: a
  type(point):: r

  r%x = -a%x
  r%y = -a%y
  r%z = -a%z
end function point_negate
```
Now we do the norm, .dot., and sqrt.

function point_norm(self) result(r)
   type(point), intent(in):: self
   real(adequate) r
   r = sqrt(self%x**2 + self%y**2 + self%z**2)
end function point_norm

function point_dot(a, b) result(r)
   type(point), intent(in):: a, b
   real(adequate):: r
   real(precise):: s
   s = real(a%x, precise) * b%x + real(a%y, precise) * b%y + &
      real(a%z, precise) * b%z
   r = real(s, adequate)
end function point_dot

function point_sqrt(a) result(r)
   type(point), intent(in):: a
   type(point):: r
   r%x = sqrt(a%x)
   r%y = sqrt(a%y)
   r%z = sqrt(a%z)
end function point_sqrt
end module points
Let’s test-drive module points.

subroutine try_points
  use precision
  use points
  implicit none
  type(point), parameter:: origin = point(0.0, 0.0, 0.0)
  type(point), parameter:: e1 = point(1.0, 0.0, 0.0)
  type(point), parameter:: e2 = point(0.0, 1.0, 0.0)
  type(point), parameter:: e3 = point(0.0, 0.0, 1.0)
  type(point) x1, x2
  x1 = 3.0 * e1 + 4.0 * e2 + 5.0 * e3
  x2 = -4.0 * x1 - (2.0_adequate * e2) + e3
  write(11, *) "x1 ", x1
  write(11, *) "x2 ", x2
  write(11, *) "norm(x1)", norm(x1)
  write(11, *) "sqrt(x1)", sqrt(x1)
  write(11, *) "dot product", x1 .dot. x2
end subroutine try_points
Note that kind conversion was performed, but type conversion won’t be.

\[
x_1 = 3.0 \times e1 + 4.0 \times e2 + 5.0 \times e3 \quad \text{!ok}
\]
\[
x_1 = 3.0 \times e1 + 4.0 \times e2 + 5 \times e3 \quad \text{!doesn’t compile}
\]

Naturally, you could define two more “handlers” so that each operation would work with integers.
Suppose we want to represent points on the unit circle, and to be able to ask for their x, y, and theta coordinates.

However, we always want $0.0 \leq \theta < 2.0 \times \pi$

We also want to be able to add and subtract angles from such points, and to create the points from real numbers representing the angle.
Here is how we are going to use entities of type ucpoint.

```fortran
subroutine try_ucpoints
  use precision
  use ucpoints
  use points
  type(point) a
  type(ucpoint) r1, r2

  r1 = pi / 4.0_adequate
  r2 = -r1 + pi / 12.0_adequate
  write (11, *) "r1 ", x (r1), y (r1), theta (r1)
  write (11, *) "r2 ", x (r2), y (r2), theta (r2)
  write (11, *) "r3 ", theta (as_ucpoint (-0.3 * pi))
  a = point(x(r1), y(r1), 0.0)
  write (11, *) "a ", a
end subroutine try_ucpoints
```
module ucpoints !points on the unit circle
    use precision
    private
    public ucpoint, as_ucpoint
    public operator(+), operator(-), assignment(=)
    public x, y, theta

type ucpoint
    private
    real(adequate):: theta = 0.0
end type ucpoint

interface assignment(=)
    module procedure ucpoint_set
end interface

interface operator(+)
    module procedure ucpoint_add_s, ucpoint_s_add
end interface

interface operator(-)
    module procedure ucpoint_subtract_s, &
    ucpoint_s_subtract, ucpoint_negate
end interface
Now in the procedure part of the module we define these operators.

contains

function as_ucpoint(r) result(p)
   real(adequate), intent(in):: r
   type(ucpoint) p
   p = r
end function as_ucpoint

function x(p)
   type(ucpoint), intent(in):: p
   real(adequate) x
   x = cos(p%theta)
end function x

function y(p)
   type(ucpoint), intent(in):: p
   real(adequate) y
   y = sin(p%theta)
end function y

function theta(p)
   type(ucpoint), intent(in):: p
   real(adequate) r
   theta = p%theta
end function theta
To define the assignment operator we supply a subroutine.

You can have multiple assignment handlers with different signatures for the second argument. The compiler will pick the right one.

```fortran
subroutine ucpoint_set (p, r)
  type(ucpoint), intent(out):: p
  real(adequate), intent(in):: r
  p%theta = modulo(r, 2.0_adequate * pi)
end subroutine ucpoint_set
```

Note that the intent of p must be out or inout. The intent of r must be in.

The statement p = r will be handled by a call ucpoint_set(p, r).
By using our own assignment operator, we can insure that theta is always in the desired range.

function ucpoint_add_s (a, b) result(p)
  type(ucpoint), intent(in):: a
  real(adequate), intent(in):: b
  type(ucpoint):: p

  p = a%theta + b
end function ucpoint_add_s

Note that the assignment is really ucpoint_set (p, a%theta + b).

Thus, ucpoint_set becomes the “gatekeeper” that makes sure only legal ucpoints can be created.
Fortran 95’s derived types compare unfavorably to C++’s.

Comparison of derived types in F95 and C++

<table>
<thead>
<tr>
<th>Net Effect for F90 vs. C++</th>
<th>Area of concern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minus</td>
<td>There are no provisions for destructors.</td>
</tr>
<tr>
<td>Minus</td>
<td>There is only the one “official” constructor.</td>
</tr>
<tr>
<td>Slight Plus</td>
<td>There is no automatic signature matching or implicit conversions.</td>
</tr>
<tr>
<td>Huge Minus</td>
<td>No inheritance in Fortran 95</td>
</tr>
<tr>
<td>Huge Minus</td>
<td>No parameterized types in F95</td>
</tr>
<tr>
<td>Huge Plus</td>
<td>Type safety</td>
</tr>
<tr>
<td>Huge Plus</td>
<td>Interaction of array facility with derived types.</td>
</tr>
<tr>
<td>Huge Plus</td>
<td>Ease of learning to use derived types correctly.</td>
</tr>
</tbody>
</table>
The job of a destructor is to release any assets acquired by an object during its lifetime.

module curves !points on the unit circle
    use precision
    private
    public curve
    public length, segment, create, destroy
    public get_x, get_y

    type curve
        private
        real(adequate), pointer:: x(:) => Null()
        real(adequate), pointer:: y(:) => Null()
    end type curve

    interface create
        module procedure curve_create
    end interface create

    interface destroy
        module procedure curve_destroy
    end interface destroy

    interface length
        module procedure curve_length
    end interface length
Define assignment of one curve to another so that it makes a copy.

interface assignment(=)
    module procedure curve_set_curve
end interface
The constructor and destructor and assignment operator work together to avoid memory leaks.

subroutine curve_create(c, x, y)
   real(adequate), intent(in):: x(:), y(:)
   type(curve), intent(inout):: c
   if(size(x) /= size(y)) then
      stop "curve assignment: incorrect sizes"
   endif
   call curve_destroy(c)
   allocate(c%x(size(x))
   allocate(c%y(size(y)))
   c%x = x
   c%y = y
end subroutine curve_create

subroutine curve_set_curve(p, q)
   type(curve), intent(inout):: p
   type(curve), intent(in):: q
   call curve_create (p, q%x, q%y)
end subroutine curve_set_curve
The destructor uses the “associated” function to see if memory has been allocated in this curve.

```fortran
subroutine curve_destroy(p)
type(curve), intent(inout):: p
if(associated(p%x)) then
  deallocate(p%x)
  deallocate(p%y)
endif
end subroutine curve_destroy
end module curves
```
Memory integrity then depends on a programmer convention to destroy each item that is created.

```fortran
subroutine try_curves
    use precision
    use curves
    implicit none
    type(curve) c1, c2, c3
    real(adequate) xx(4), yy(4)
    xx = (/ 1., 2., 3., 4. /)
    yy = xx**2 + 1
    call create(c1, xx, yy)
    call create(c2, xx, yy / 2.)
    write(11, *) "y(c2) ", y(c2)
    ... some coding which may or may not use c3....
    call destroy(c1)
    call destroy(c2)
    call destroy(c3)
end subroutine try_curves
```
Homework

A tic-tac-toe board consists of a 3 by 3 matrix. Each entry can either be empty, an X, or an O. Create a tic-tac-toe board object which is sufficiently rich to support writing a tic-tac-toe game. In the game, users can “undo” moves right back to the start. The board class should refuse to make illegal moves and return an error flag if so requested.

There is no “right” answer.

How robust you make the input procedure is up to you.

If there is some other more inspiring subject for you, feel free. But a prize for the best tic-tac-toe game will be awarded. Email your program to me by Friday noon.