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

Lesson 1

Arrays and Modules

Paul F. Dubois

X-Division

dubois1@llnl.gov
Fortran 95 is a modern, type-safe language.

- Dynamic memory management
- Type-checked procedural interfaces
- Sophisticated set of intrinsic functions
- Scalable namespace management
- User-defined objects and operations
Migrate to Fortran 95 without pain.

- This 4-lecture series will migrate you.
- You can migrate your existing code a little at a time.
- Change yourself or your code as much or as little as you want.
Your life will get better.

- Define your own types and operations.
- Data alignment problems disappear with modules.
- Dynamic memory management is easy.
- Portable (and easily changeable) precision for real and complex variables is attainable.
Fortran 95 supports object-based programming.

- There is sufficient support for designing your own objects.
- Generic interfaces provide function-signature overloading.
- It isn’t fully OO (yet) but it goes a long way toward the goals of OOP.
Our “upgrade” strategy is to only learn the “right way”.

- No language lawyers allowed.
- Many features omitted.
- Subtle points ignored.
- In short, I’ll try to stick to the stuff that makes a real difference.
Course-related files are available at ftp-icf.llnl.gov/obf95.

These slides

Example sources
Choose your own text

- Digital Fortran Language Ref. Guide
The following example will not be studied in detail. It is just here to frighten you into doing the homework.

Is this Fortran? Yes!

program tryme
  use example
  implicit none
  call init
  write(11, *) report(3) ! This too!
  write(11, *) report() ! Oh, dear...
end program tryme
module example
  implicit none
  private
  public report, init
  integer, dimension(10):: x
contains
  function report (j) result(y)
    integer, intent(in), optional:: j
    integer, dimension(size(x)):: y
    if (present(j)) then
      y = j * x + 2
    else
      y = x + 2
    endif
  end function report
  subroutine init
    integer i
    x = (/ (i, i= 1, size(x)) /)
  end subroutine init
end module example
Fortran 77’s major weakness with respect to large-scale programming is its poor name scoping.

There are only two scopes, local and global.

The global scope contains all function names and common blocks.

Labeled common blocks let you only “use” part of the global variable names but name conflicts occur between common blocks.

The user is required to segregate variables by type in order to achieve correct data alignment.
The second major weakness of F77 is the poor facility for dynamic memory management.

The Cray “pointer” facility is dangerous and prone to memory leaks. The exact syntax and details of its operation are not de facto standards. But without it, no modern program is possible.
Today’s lesson solves both those problems.

- Modules provide scalable namespace control.
- The array facilities include complete support for dynamic memory management.
Mini-lesson #1: Rip out those common blocks

Change:

complex w
real x, z
integer y
common /abcc/ w
common /abcr/ x, z
common /abc1/ y
...
#include “abc.h” or whatever you are doing...

To:

module abc
    complex w
    real x, z
    integer y
end module abc
...
use abc

Gee, that was easy!
The module declaration creates a new name space.

module m1
    implicit none
    real x, y(10)
end module m1

“Use association”

subroutine whatever
    use m1 !makes all names in m1 available here
    write(11, *) x, y
end subroutine whatever

subroutine whomever
    use m1, only: x !makes only the name x available here
    integer y
    write(11, *) x, y
end subroutine whomever

You can even rename:
    use m1, only: x, my_name_for_y => y
    integer y ! no conflict with the y in m1
Arrays are F95’s big forte.

First, the way we were...

```fortran
program main
  implicit none
  open(form="formatted", unit=11, file="lesson1.txt")
  call version77
  call version90
  call part2
end program

subroutine version77
  real y(10)
  real x(10)
  common /mycom/ x
  integer i
  parameter(pi = 3.14159)

  do 100 i = 1, 10
    x(i) = i / 10.0
    y(i) = sin(x(i) * pi)
  100 continue
  write(11, *) "Version 77"
  write(11, *) (y(i), i = 1, 10)
end
```

We use a module to establish the precision of our real variables.

```fortran
module precision
  implicit none
  integer, parameter:: adequate = selected_real_kind (6,35)
  integer, parameter:: precise = selected_real_kind (14,99)
  real(adequate), parameter:: pi = 3.14159_adequate
end module precision
```

The qualifier on the real declaration is called its “kind”.
We could also write real(kind=adequate).

We are assured that pi will be of a real type that has at least six digits of precision and an exponent range of E-35 to E+35.

The default “real” and “double” still exist but they have unknown properties. These properties also apply to real and double literals.

Note the notation for ensuring that a literal has the desired kind.

Also available in conversion operators: real(24, adequate).
We replace the common block with a module.

module mystuff
    use precision
    implicit none
    real(adequate) x(10)
    ! there is more in this module, but we’ll see that later...
end module mystuff
subroutine version90
  use mystuff
  implicit none
  integer i

  do i = 1, 10
    x(i) = i / 10.0
  enddo
  call version0
  call version1
  call version2
  write(11, *) "Version 3"
  write(11, *) version3(x)
  call version4 (x, 10)
  call version5 (x, 10)
  call version6 (x(1), 10)
end

We don’t need a loop any more. And the compiler can print the array y.
subroutine version0
  use mystuff
  implicit none
  real(adequate) y(10)

  y = sin(x * pi)
  write(11, *) "Version 0"
  write(11, *) y
end subroutine version0
First improvement: no need to hard-wire array sizes.

```
subroutine version1
    use mystuff
    implicit none
    real(adequate) y(size(x))

    y = sin(x * pi)
    write(11, *) "Version 1"
    write(11, *) y
end subroutine version1
```
If you don't know the right size on entry, use allocatable arrays.

```fortran
subroutine version2
    use mystuff
    implicit none
    real(adequate), allocatable:: y(:)

    allocate(y(size(x)))
    y = sin(x * pi)
    write(11, *) "Version 2"
    write(11, *) y
end subroutine version2

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

    y = sin(x * pi)
end function version3
```
Sizes can also be picked up from formal parameters or module variables.

```fortran
subroutine version4 (x, n)
   use precision
   implicit none
   integer, intent(in):: n
   real(adequate), intent(in):: x(n)
   real(adequate) y(n)

   y = sin(x * pi)
   write(11, *) "Version 4"
   write(11, *) y
end subroutine version4
```
The assumed-size array is the old idea we are used to for array arguments.

subroutine version5 (x, n)
    use precision
    implicit none
    integer, intent(in):: n
    real(adequate), intent(in):: x(*) ! assumed-size array
    real(adequate) y(n)

    ! y = sin(x * pi) compiler error, doesn't know the size of x
    y = sin(x(1:n) * pi) ! o.k.
    write(11, *) "Version 5"
    write(11, *) y
end subroutine version5

subroutine version6 (x, n)
    ! Same as version 5, only called with x(1)
    use precision
    implicit none
    integer, intent(in):: n
    real(adequate), intent(in):: x(*) ! assumed-size array
    real(adequate) y(n)

    y = sin(x(1:n) * pi)
    write(11, *) "Version 6"
    write(11, *) y
end subroutine version6
Array pointers are similar to allocatable arrays.

Array pointers have a rank but do not get a shape until they are “pointed” at something with the => operator.

subroutine version7 (x)
    use precision
    implicit none
    real(adequate), intent(in), target:: x(:)
    real(adequate), pointer:: y(:), z(:)
    integer n, m

    n = lbound(x, 1)
    m = ubound(x, 1)
    y => x(n+1: m)
    z => x(n: m - 1)
    write(11, *), "Version 7"
    write(11, *), z - y
end subroutine version7
We have learned there are four kinds of arrays.

1. Explicit-shape:
   - Size evaluable at compile time, such as \( x(100) \). Can be in the main program or a subprogram.
   - Size evaluable on entry to a subprogram, such as \( x(n) \). Shape is determined at once on entry to the subprogram.

2. Assumed-shape: \( x(:) \), where \( x \) is a formal argument, assumes the shape of the actual argument. Shape determined on entry to the subprogram. Use this where you used to use \( x(*) \). OK if more than one dimension, such as \( y(:, :) \).

3. Deferred-shape: \( x(:) \), where \( x \) is allocatable or has the pointer attribute. The rank is determined, but the shape will be determined later. OK if more than one dimension. F95, but not F90, allows a lower bound, such as \( x(2:) \).

4. Assumed size: \( x(*) \), where \( x \) is an argument, and the * is on the last dimension only. Only an address is actually passed. Can be used for compatibility with Fortran 77.

In all cases the rank is explicit and fixed.
Routines that have the “modern” array arguments must have explicit interfaces.

In other words, the compiler has to know, at the place where it makes a call, what the signature of the callee looks like.

If an interface is explicit:

- Arguments and return values can be array-valued.
- The compiler checks the numbers and types of the arguments in each call.
- The compiler performs type conversion on arguments where possible.
- If an argument has intent(in), it cannot be written.
We need to add interfaces to module mystuff.

module mystuff
  use precision
  implicit none
  real(adequate) x(10)
  interface
    function version3(x) result(y)
      use precision
      real(adequate), intent(in):: x(:)
      real(adequate) y(size(x))
    end function version3
    subroutine version7(x)
      use precision
      real(adequate), intent(in):: x(:)
    end subroutine version7
  end interface
end module mystuff

Routines version3 and version7 had “modern” array arguments, while version0, version1, and version2 had no arguments, and the rest took assumed-size x(*) arguments.
There are three ways to make an subprogram’s interface explicit.

1. By containing it inside the routine that calls it.
2. By containing it in a module.
module myfuns
  use precision
  implicit none
! interface to a function defined elsewhere
interface
  function manipulate4(x)
    use precision
    real(adequate), intent(in): x(:)
    real(adequate) manipulate4(size(x))
  end function manipulate4
end interface
contains

function manipulate3(x) result(s)
  use precision
  real(adequate) x(:)
  real(adequate) s

  s = sum(x)
  end function manipulate3
end module myfuns
subroutine part2
  use mystuff
  use myfuns
  implicit none
  real(adequate) manipulate1, manipulate5
  external manipulate1, manipulate5

  write(11, *), "Manipulations"
  write(11, *), manipulate1(x, size(x)), &
    manipulate2(x), &
    manipulate3(x)
  write(11, *), manipulate4(x)
  write(11, *), manipulate5()

contains

  function manipulate2(x)
    ! explicit since contained in the calling routine
    use precision
    real(adequate), intent(in):: x(:)
    integer manipulate2

    manipulate2 = count(x > 0.3_adequate)
  end function manipulate2

end subroutine part2
function manipulate1(x, n) result(w)
   ! implicit interface, new intrinsic dot_product
   use precision
   real(adequate), intent(in):: x(*)
   real(adequate) w

   w = dot_product(x(1:n), x(1:n))
end function manipulate1
function manipulate4(x) result(p_times_y)
! matrix multiply, etc.
  use precision
  real(adequate), intent(in):: x(:)
  real(adequate) y(size(x)), z(size(x)), p(size(x), size(x))
  real(adequate) p_times_y(size(x))
  integer i, j, n
! array expressions
  y = x**2 + 1.0
  z = x / y
! scalar broadcast
  p = 0.0
! sections
  n = size(x)
  do i = 1, n
    p(i,:) = z * (1.0 / (1.0 + i))
  enddo

  p_times_y = matmul (p, y)
end function manipulate4
Mini-lesson #2: You can add explicit interfaces to your existing program now.

Add a module to each “physics package” and in it declare explicit interfaces to the functions in that package.

In each routine that implements or calls one of the functions, add a use statement.

Compile everything with f90 or f95.
Homework

If you’re going to just sit there, you’ll have more fun at a movie.

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?