Advanced Fortran Programming

National Computational Infrastructure

Outline

Advanced Fortran Programming

Course aims to provide an overview of many of the new features introduced by Fortran90

- Array syntax
- Assumed shape and automatic arrays
- Case
- Intent
- Optional arguments
- Common Blocks
- Interfaces and modules
- Derived Types
- Operators and overloading
- Dynamic memory

Fortran Programming Language

Fortran (FORmula TRANslation) is a high level language aimed at numerical calculations. Evolved over 30-40 years. All compilers should conform to the Fortran standard so that code is completely portable. Recommended textbooks:

- Fortran 90 Explained, Metcalf and Reid, Oxford Science Publications
- Fortran 90/95 Explained, Metcalf and Reid, Oxford University Press
- Fortran 95/2003 Explained, Metcalf, Reid and Cohen, Oxford University Press

Internet resources are

- http://www.fortran.com contains a FAQ and lots of other information.
- news://comp.lang.fortran to speak directly to those who develop Fortran

Free compilers

There are a number of free compilers available

- Fortran 77 - F77 part of the GNU GCC project
- Fortran 95 - g95 based on the GNU GCC project
- Fortran 95 - gfortran part of the GNU GCC 4.0 project
- Fortran 95 - ifort Intel Fortran compilers for linux

Outline

Arrays

Fortran90 introduced the notion of array and array syntax, eg.

```
program main
    real, dimension(100,100) :: a, b

    a = 1.0
    b = 2.0
    a = a+b*3.0+4.0
end program main
```

Equivalent to do-loops

All array operations operate element wise, consquently all array notation can always be represented by equivalent do-loops

```
real, dimension(100,100) :: a. b
integer :: i, j

do i=1,100
    do j = 1, 100
        a(i,j) = 1.0
        b(i,j) = 2.0
        a(i,j) = a(i,j)+b(i,j)*3.0+4.0
    end do
end do
```

Array sections

With the new notation comes the possibility of array sections

```
real, dimension(100,100) :: a
real, dimension(10,10) :: b

a=1.0d0
b=a(11:20,31:40)
```

Arrays and intrinsics

Since all array operations happen element wise, you can apply intrinsics to an array

```
real, dimension(10,10) :: a, b
real :: c

call random_number(a)
b = sin(a)
a = exp(b)
c = sum(a)+product(b)
```

See the Fortran Reference Manual for a complete list.

Array information

There are many intrinsics to query the array:

```
shape(a)                    maxloc(a [,mask])
size(a [,dim])              maxloc(a, dim [,mask])
lbound(a [,dim])            minloc(a [,mask])
ubound(a [,dim])            minloc(a, dim [,mask])
```

and more. See the Fortran Reference Manual for a complete list.

Exercise 1

Using the `random_number` intrinsic to set the initial value of two real arrays, determine and display the root-mean-square distance between the two arrays. `call random_number(harvest)` 0 <= x <= 1 or an array of such numbers. `harvest` may be scalar or an array and must be of type real Try writing the code both explicitly (with do-loops etc) and then using array notation.

Where

Arrays allow a more compact notation for conditional assignment

```
real, dimension(100,100) :: a, b
...
where(a>0.0)
    b = a
elsewhere
    b = 0.0
end where
```

Mask operations

There are also many conditional tests and masked intrinsics

```
if (all(a==0.0)) write(*,*) 'a is zero'
if (any(a==0.0)) write(*,*) 'a has at least 1 zero element'
write(*,*) 'a has', count(a==1.0), ' elements equal to 1'
a = merge(0.000001, a, a==0.0)
b = sum(a, a>0.0)
c = product(a+1.0, a>=-1.0)
```

Assumed shape and automatic

With array notation comes the possibility of assumed shape

```
subroutine foo(a,b)
    real, dimension(:,:) :: a
    real, dimension(0:,-5:) :: b
  ...
end subroutine foo
```

and automatic arrays

```
subroutine foo(a)
    real, dimension(:,:) :: a
    real, dimension(size(a,1), size(a,2)) :: b
  ...
end subroutine foo
```

Exercise 2

Compile and run `array.f90`. Look at the code and understand the output produced.

```
> f90 array.f90 -o array
> ./array
```

Outline

Case

Often want to test for more than one condition. An `if-then-elseif-else-endif` tree is not suitable. Fortran 90 provides a `case` statement

```
select case(ch)
    case('a','c','d')
        x = sin(y)
    case('w':)
        x = cos(y)
    case default
        x = y
end select
```

Intent

Fortran90 provides a mechanism to tell the compiler about how variables are used within a sub program. This allows the compiler to provide additional checking and optimisation.

```
subroutine foo(a, b, c)
    real, intent(in) :: a
    real, intent(inout) :: b
    integer, intent(out) :: c
  ...
end subroutine foo
```

now, something like `a=1.0` inside `foo` will generate a compiler error. Also, `c` will be undefined on entry into `foo`

Optional arguments

Subroutines and fuction can have optional arguments in Fortran90

```
function foo(a, b)
    real :: foo
    real, intent(in) :: a
    integer, intent(in), optional :: b

    if (present(b)) then
        foo = a**b
    else
        foo = a
    end if
end function foo
```

Common Blocks

Common blocks are a Fortran 77 construct, providing areas of shared storage between different subroutines. eg.

```
program main
    integer :: a, b
    real :: c
    common /d/ a, b, c
  ...
end program main

subroutine foo()
    integer :: a, b
    real :: c
    common /d/ a, b, c
  ...
end subroutine foo
```

the common storage is called `d` and contains the integers `a, b` and the real `c`. Setting these variables in one routine changes it in all. Common blocks are now replaced by modules in Fortran 90.

Outline

Interfaces

F77 did not have type checking for arguments to subroutines and functions. F90 allows the declaration of an interface to a sub program which the compiler will then enforce.

```
program main
    interface
        real function fun(x)
            real :: x
        end function fun
    end interface

    real :: y

    y = fun(10.0)
end program main
```

Calling `y=fun(1)` will produce a compiler error.

Modules

Modules provide a mechanism to package data types, derived types, function, subroutines and interfaces together. Including a module gains access to all public components within that module and automatically defines interfaces to all functions and subroutines within.

```
module foobar
    real :: a
    integer :: b
contains
    subroutine foo(c, d)
        integer :: c
        real :: d

        d = d * a**c + b
    end subroutine foo
end module foobar
```

Using modules

A module can then be used to gain access to its components

```
program main
    use foobar

    real :: x

    x = 10.0
    b = 3
    call foo(5, x)
end program main
```

Since the subroutine `foo` is within a module its interface is automatic and enforced by the compiler.

Exercise 3

Write a short Fortran90 program to compute the integration of `2*sin(x+y)**2` over the domain `0.25 <= x <= 2.5` and `1 <= y <= 10`. Then, move the function evaluation into a module and `use` that module in your main program.
Assuming the main program is `integration.f90` and the module is `integmod.f90` you would compile and execute the code as

```
> f90 integmod.f90 integration.f90 -o integration
> ./integration
```

NOTE: the generation of the `.mod` file. This contains the interface information for the module sub programs.

Modules cascade

Modules can be cascaded.

```
module aaa
   ...
end module aaa

module bbb
     use module aaa
   ...
end module bbb

module ccc
     use module bbb
   ...
end module ccc
```

Module `ccc` now has access to all public objects within module `bbb` and module `aaa`

Being more specific

You can elect to use only certain objects from a module

```
module bbb
    use aaa, only: foo, bar
  ...
end module bbb
```

which only gains access to `foo` and `bar` within module `aaa`

Public and private

You can state which objects within a module are publically available or private

```
module foobar

    private
    public :: foo

    real :: a
    real, public :: b

contains
    subroutine foo(...)
      ...
    end subroutine foo

    subroutine bar(...)
      ...
    end subroutine bar
end module foobar
```

Outline

Derived types

Ofter more complex data types are required. Fortran90 allows this through derived types.

```
type particle
    real, dimension(3) :: pos, vel, acc
    real :: mass
    integer :: n
end type particle

type(particle) :: p

p%mass = 1.0
p%n = 1
p%pos = (/ 1.0, 2.0, 4.0 /)
p%vel = 0.0
p%acc = 0.0
```

More complex derived types

Derived types may be extremely complex

```
type particlebox
    type(particle), dimension(:), pointer :: particles
    real :: lx, ly, lz
end type particlebox

type(particlebox) :: box

allocate(box%particles(100))
box%particles(10)%mass = 1.0d0
```

Exercise 4

Write a Fortran90 program which uses a derived type to store the details of box of particles

- Length, width and height `real`
- Number of particles `integer`
- Mass of each particle `real`
- Charge of each particle being `-1.6*10**-19`

This program should take input from the keyboard to setup the box of particles.

Operators

You can define your own operators with an interface block

```
module a
    type point
        real :: x, y
    end type point

    interface operator(.dist.)
        module procedure calcdist
    end interface

contains

    function calcdist(p1, p2)
        type(point) :: p1, p2
        real :: calcdist

        calcdist = sqrt((p1%x-p2%x)**2 + (p1%y-p2%y)**2)
    end function calcdist

end module a
```

which allows `d=p1.dist.p2`

Overloading

It is also possible to over load existing operators

```
module aaa
    type point
        real :: x, y
    end type point

    interface operator(+)
        module procedure addpoints
    end interface

contains

    function addpoints(p1, p2)
        type(point) :: p1, p2, addpoints

        addpoints%x = p1%x+p2%x
        addpoints%y = p1%y+p2%y
    end function addpoints

end module aaa
```

which allows `p1+p2`

Outline

Dynamic memory

Fortran90 introduced the concept of dynamic memory allocation and pointers. To dynamically allocate memory, an array must be declared `allocatable`

```
real, dimension(:,:), allocatable :: a

allocate(a(100,100))
...
deallocate(a)
```

Allocatable arrays can not be used within a derived data type. You need to use a pointer.

Pointers

Unlike C style pointers, Fortran pointers point to a specific object. That object must be another pointer of the same type or a `target`.

```
real, dimension(100,100), target :: a
real, dimension(:,:), pointer :: b

b=>a
```

Now, both `a` and `b` refer to the same piece of memory.
NOTE: `b` is NOT an array of pointers. It is a pointer to an array

More pointers

You can allocate memory directly to a pointer as with an `allocatable` array

```
real, dimension(:,:), pointer :: a

nullify(a)
allocate(a(100,100))
...
deallocate(a)
```

A call to `nullify(a)` after an allocate will cause a memory leak.

Exercise 5

Modify your previous *box of particles* code to build a more complex dynamic derived data type to store an array of particles. Upon entry of the number of particles, `n` particles are allocated, each containing an `x-y` position and velocity.
Setup random initial conditions and compute the root-mean-square velocity, average kinetic energy and temperature of the particles (assuming they are an ideal gas). Assume the kinetic energy per particle is `KE = 1/2 m v**2` and the temperature for a 2D gas is given by `T = KE/k` where `k = 1.380658E-23`.

Pointers to subsections

You can use pointers to reference a subsection of an array. Be aware that this may have a serious performance impact.

```
real, dimension(100,100), target :: a
real, dimension(:,:), pointer :: b

b=>a(20:30,40:50)
```

Now, indexing `b` effectively strides through the array `a`

Array of pointers

If you want an array of pointers you need to use a derived type

```
program main
    type pp
        real, dimension(:), pointer :: p
    end type pp

    type(pp), dimension(:), allocatable :: array
    integer :: i

    allocate(array(100))

    do i = 1, size(array)
        allocate(array(i)%p(i))
    end do

  ...
end program main
```

The array is now a lower triagular matrix.

The linked list

One of the most useful features of derived types and pointers is the ability to create a dynamic structure called a link list.

```
type node
    type(node), pointer :: next, prev
    real, dimension(100,100) :: a
end type node

type(node), pointer :: ll, cur
integer :: i

allocate(ll)
ll%next => ll
ll%prev => ll

cur => ll

do i = 1, 10
    allocate(cur%next)
    cur%next%prev => cur
    cur%next%next => ll
    cur => cur%next
end do
```

This creates a dynamic list of arrays.

Linked list operations

Link lists can be grown

```
allocate(newnode)
newnode%prev => cur
newnode%next => cur%next
cur%next%prev => newnode
cur%%next => newnode
```

and shrunk

```
oldnode => cur
cur%prev%next => cur%next
cur%next%prev => cur%prev
cur => cur%next
deallocate(oldnode)
```

Conclusion

Fortran90 extends the functionality of Fortran77, predominantly allowing dynamic data structures and stricter type checking on external sub-programs. Fortran continues to be developed, with the most recent version Fortran2003 being released in 2005. It adds object oriented programming and a well defined system and C interface. Look out for Fortran2008.