Formatted I/O, Derived Types and Makefiles

22.901 Introduction to Computer Programming for Nuclear Engineers

January 26, 2012

Massachusetts
Institute of
Technology

- Formatted I/O

- Derived Types

- Makefiles

What Have we Done so Far?

- `print *,` - used for printing to screen
- `read *,` - used for reading from screen

These are historical, in modern Fortran we use:

- `write(*,*)<stuff>` - write to screen
- `read(*,*)<stuff>` - read from screen

Really, this is shorthand notation for:

$$write(unit=*,fmt=*)$$

- `unit=*` stdin(unit=6) or stdout=(unit=5)
- `fmt=*` list directed input/output

Using Units

- A unit is an integer identifying a connection to a file

- Generally use values bewteen 10 and 99

- `unit=` can be omitted if the unit is first in the write list

```
write(10,*) 'Hello'
```

- This will write the string Hello to unit 10

- By default, unit 10 directs to the file 'fort.10'

Specifying a Format

- `fmt=` can be omitted if the format is second and the first is unit

- `fmt=*` indicates list directed I/O

- `fmt=<fmt>` indicates a fixed format

- If `<fmt>` is a number, a format card is needed

Implied DO Loops

- There is an alternative to array expressions

- Often more convinient

- The format: (<list>,<indexed loop control>)

For example:

```
((A(i,j),j=1,3),B(i),i=6,2,-2)
```

This prints:
A(6,1),A(6,2),A(6,3),B(6),A(4,1),A(4,2),A(4,3),B(4),...

Integer Descriptor

- In displays an integer

- Integers are printed **right-justified**

- In.m displays at least m digits

- A '-' sign will take up one spot in the field

For example:

> write(*,'(I7)') 123 will give 'ˆˆˆˆ123'
> write(*,'(I7,5)') 123 will give 'ˆˆ00123'

Note: The character ˆ does not actually appear, it is just used to denote a space in this presentation.

Fixed-Formal Real

- `Fn.m` displays to `m` decimal places

- Printed **right-justified** in a field of width `n`

- A '-' and the decimal '.' take up one spot in the filed

- Numbers should be correctly rounded

For example:

```
    write(*,'(F9.3)') 1.23 will give 'ˆˆˆˆ1.230'
  write(*,'(F9.5)') 0.123e-4 will give 'ˆˆ0.00001'
```

Note: The character ˆ does not actually appear, it is just used to denote a space in this presentation.

Field Overflow and Zero Widths

- A field overflow happens when you try to fit a value in a field that is too small

- the whole field is then replaced by asterisks

- putting 12345 into I4 gives ****

- for integer and fixed-format reals using a width of 0 will get rid of leading spaces

For example:
```
write(*,'(I7)') 123 will give '^^^^123'
write(*,'(I0)') 123 will give '123'
write(*,'(''/'',I0,''/'',F0.3)') 12345,987.654321 will
give '/12345/987.654'
```

Exponential Format

- Use the `ESn.m` for `m` decimal places

- `n` is the total field with including '-' and '.' and exponential part (usually 4 spots)

- General Rule: $n \geq m + 7$

For example:

`write(*,'(ES12.5)')` 233323.2324234 will give 2.33323E+05

Character Descriptor

- `An` displays a field width of `n`

- `A` uses the exact width of the output item

- If the output field is too small, the left most characters are used

- If the output field is too large, **right-justified**

For example:
`write(*,'(A3)')` 'abcdefgh' will give abc

Other Descriptors

- `Ln` displays either T or F

- `Gn` or `Gn.m` is a generalized descriptor

- You can do string contants by surrouding in quotes

- Spacing descriptor - `nX`. 4X puts 4 spaces

- Newline descriptor - just do a single /

- Tab descriptor - `Tn` where n is the column number to tab to

Opening a file

- Use the open command

```
open(unit=11,file='fred')
```

- Use unit 11 now in all write or read statements

- Use numbers 10-99

- To close a file use close(unit#)

```fortran
program matmultiply

implicit none

  integer :: n  ! # of rows in first matrix
  integer :: m  ! # of cols in first matrix
  integer :: p  ! # of rows in second matrix
  integer :: q  ! # of cols in second matrix
  integer :: i  ! iteration counter
  integer :: j  ! iteration counter
  real, allocatable :: A(:,:) ! first matrix
  real, allocatable :: B(:,:) ! second matrix
  real, allocatable :: C(:,:) ! result matrix

  ! ask user to enter dimensions of first matrix
  print *,'Enter # of row then columns of first matrix:'
  read *,n,m
```

Matrix Multiply II

```fortran
! allocate first matrix
allocate(A(n,m))

! read in first matrix
print *,'Enter first matrix (column-major):'
read *, A

! ask user to enter dimensions of first matrix
print *,'Enter # of row then columns of first matrix:'
read *,p,q

! allocate second matrix
allocate(B(p,q))

! read in first matrix
print *,'Enter first matrix (column-major):'
read *, B

! check for mismatch
```

```fortran
if (m /= q) then
  print *,'Fatal ==> Array dimension mismatch!'
  stop
end if

! allocate result matrix
allocate(C(n,q))

! send to subroutine
call matmultiply_sub(A,B,C,n,m,q)

! print out result
do i = 1,n
  write(*,'(20(F0.3,4X))') (C(i,j),j=1,q)
end do

! terminate the program
stop
```

```fortran
end program matmultiply

subroutine matmultiply_sub(A,B,C,n,m,q)

implicit none

  ! formal vars
  integer :: n        ! row dimension of first mat
  integer :: m        ! col dimension of first mat
  integer :: q        ! col dimension of second mat
  real      :: A(n,m) ! first matrix
  real      :: B(m,q) ! second matrix
  real      :: C(n,q) ! result matrix

  ! local vars
  integer :: i ! result row counter
  integer :: j ! internal col counter

  ! set C to 0
```

Matrix Multiply V

```fortran
   C = 0.0

   ! begin loop
   COL:  do  j = 1,q

     ROW:  do  i = 1,n

        ! perform summation
        C(i,j) = sum(A(i,1:m)*B(1:m,j))

     end do ROW

   end do COL

end subroutine matmultiply_sub
```

What is a derived type

- Basically is it a grouping of variables
- This is sometimes referred to as a structure
- These are very useful in organizing data
- Think of the underlying data as attributes

For example:
```
type ::  particle
 real ::  xcoord !  particle location
 real ::  energy !  particle energy
 real ::  angle !  particle angle
 real ::  weight !  statistical weight
 logical ::  alive !  particle alive?
end type particle
```

Component Selection

- The selector '%' is used for this

- It is then followed by a component of the derived type

```
For example:
type(particle) ::  neutron
neutron%energy = 1.e6
neutron%xcoord = start_neutron()
```

Array of Types

- You can have an array of types

```
For example: type ::  tally
 real ::  s1
 real ::  s2
 real ::  mean
 real ::  var
end type tally
type(tally), allocatable ::  tallies(:)
allocate(tallies(n_slabs))
tallies(1)%mean = 0.0
```

Assignment

- You can assign complete derived types

- Or simply just components of types

- Note: They have to be of the **same** type

```
type(bike) ::  mine, yours
yours = mine
mine%front = yours%back
```

Makefiles

- You need to set up rules to compile the modules

- You need to add dependencies to ensure they are rebuilt

Here is a rule to compile objects:
```
program:  <objects>
<tab> $(FC) $(FFLAGS) $(LDFLAGS) -o $@ $^
```

- $(var) just represents a variable that was set already in the makefile

- `<objects>` is a list of objects in order of dependencies

Please search online for more advanced features

End of Class Assignment

- Solve for pi via Monte Carlo

- Read in number of particles to simulate

- Set 1cm bounds for 1/4 dartboard

- Randomly throw a dart with `rand` intrinsic function

- Record how many darts are within quarter circle

- Set up ratio between 1/4 circle and square areas and darts

- Solve for the mean of pi and report that and variance