AMath 483/583 — Lecture 7 — April 11, 2011

Today:

- Array storage in Fortran
- Passing arrays to subroutines
- Fortran modules
- Multi-file Fortran codes

Wednesday:

- Makefile

Read: Class notes and references
There are several new Fortran sections.

Array storage

Rank 1 arrays have a single index, for example:

```
real(kind=8) :: x(3)
real(kind=8), dimension(3) :: x
```

are equivalent ways to define `x` with elements `x(1), x(2), x(3)`.

You can also specify a different starting index:

```
real(kind=8) :: x(0:2), y(4:6), z(-2:0)
```

These are all arrays of length 3 and this would be a valid assignment:

```
y(5) = z(-2)
```


Passing arrays to subroutines

```
! $CLASSHG/codes/fortran/arraypassing1.f90

program arraypassing1

    implicit none
    real(kind=8) :: x,y
    integer :: i,j

    x = 1.
    y = 2.
    i = 3
    j = 4
    call setvals(x)
    print *, "x = ",x
    print *, "y = ",y
    print *, "i = ",i
    print *, "j = ",j

end program arraypassing1

subroutine setvals(a)
    ! subroutine that sets values in an array a of length 3.
    implicit none
    real(kind=8), intent(inout) :: a(3)
    integer i
    do i = 1,3
        a(i) = 5.
        enddo
end subroutine setvals
```

Note: `x` is a scalar, dummy argument `a` is an array.

Passing arrays to subroutines

This produces:

```
x =    5.00000000000000
y =    5.00000000000000
i =   1075052544
j =            0
```

Nasty!!

- The storage location of `x` and the next 2 storage locations were all set to the floating point value 5.0e0
- This messed up the values originally stored in `y, i, j`.
- Integers are stored differently than floats. Two integers take up 8 bytes, the same as one float, so the assignment `a(3) = 5.` overwrites both `i` and `j`.
- The first half of the float 5., when interpreted as an integer, is huge.

Passing arrays to subroutines

```
1  ! $CLASSHG/codes/fortran/arraypassing2.f90
2
3  program arraypassing2
4
5      implicit none
6      real(kind=8) :: x,y
7      integer :: i,j
8
9      x = 1.
10     y = 2.
11     i = 3
12     j = 4
13     call setvals(x)
14     print *, "x = ",x
15     print *, "y = ",y
16     print *, "i = ",i
17     print *, "j = ",j
18
19 end program arraypassing2
20
21 subroutine setvals(a)
22     ! subroutine that sets values in an array a of length 1000.
23     implicit none
24     real(kind=8), intent(inout) :: a(1000)
25     integer i
26     do i = 1,1000
27         a(i) = 5.
28         enddo
29 end subroutine setvals
```

Note: We now try to set 1000 elements in memory!

Passing arrays to subroutines

This compiles fine, but running it gives:

```
Segmentation fault
```

This means that the program tried to change a value of memory it was not allowed to.

Only a small amount of memory is devoted to the variables declared.

The memory we tried to access might be where the program itself is stored, or something related to another program that's running.

Segmentation faults

Debugging seg faults can be difficult.

Tips:

Compile using `-fbounds-check` option.

This catches some cases when you try to access an array out of bounds.

But not the case just shown! The variable was passed to a subroutine that doesn't know how long the array should be.

Segmentation faults

Compile using the `-g` flag and then try running under `gdb`

```
$ gfortran -g arraypassing2.f90
$ gdb ./a.out

 (gdb) run
  Starting program: .../a.out
  Program received signal SIGSEGV, Segmentation fault
  0x080488ce in setvals (a=Cannot access memory at
  address 0xbffff370) at arraypassing2.f90:27
  27          a(i) = 5.

  (gdb) where
  #0  0x080488ce in setvals
  #1  0x080486a8 in arraypassing2 () at arraypassing
```

This tells us that the error occured at line 27, and that the subroutine was called at line 13.

Segmentation faults

You can also probe the value of variables.

To find out what value of `i` it died on:

```
(gdb) print i
$1 = 403
```

This tells us that the error occured when trying to set a(403).

Why not when `i = 4`?

Memory is organized into pages and integer multiples of pages must be devoted to variables in the program.

Apparently a page contains `8 * 402 = 3216` bytes.

Fortran debuggers

gdb does not work very well!!

Unfortunately there's no good open source debugger for Fortran.

Commercial options include totalview.

Rank 2 arrays

An array of rank 2 has two indices, e.g.

```
real(kind=8) :: A(3,4)
```

Compiler must map the 12 array elements to memory locations.

Different languages use different conventions!

In Fortran, arrays are stored by column in memory,
so the 12 consecutive memory locations would correspond to:

```
A(1,1)
A(2,1)
A(3,1)
A(1,2)
A(2,2)
...
A(1,4)
A(2,4)
A(3,4)
```


Rank 2 arrays

```
! $CLASSHG/codes/fortran/rank2.f90

program rank2

    implicit none
    real(kind=8) :: A(3,4), B(12)
    equivalence (A,B)
    integer :: i,j

    A = reshape((/(10*i, i=1,12)/), (/3,4/))

    do i=1,3
        print 20, i, (A(i,j), j=1,4)
 20     format("Row ",i1," of A contains: ", 15x, 4f10.3)
        print 21, i, (3*(j-1)+i, j=1,4)
 21     format("Row ",i1," is in locations ",4i3)
        print 22, (B(3*(j-1)+i), j=1,4)
 22     format("These elements of B contain:", 8x, 4f10.3, /)
        enddo

end program rank2
```

Note: equivalence statement
⟹ same memory locations for `A` and `B`.

Also note implied do loops and format statements.

Rank 2 arrays

Output:

```
Row 1 of A contains:            10.0   40.0   70.0 100.0
Row 1 is in locations   1   4   7 10
These elements of B contain:    10.0   40.0   70.0 100.0

Row 2 of A contains:            20.0   50.0   80.0 110.0
Row 2 is in locations   2   5   8 11
These elements of B contain:    20.0   50.0   80.0 110.0

Row 3 of A contains:            30.0   60.0   90.0 120.0
Row 3 is in locations   3   6   9 12
These elements of B contain:    30.0   60.0   90.0 120.0
```


Splitting Fortran codes into files

Single file program with 2 subroutines:

```
! $CLASSHG/codes/fortran/multifile1/fullcode.f90
program demo
    print *, "In main program"
    call sub1()
    call sub2()
end program demo

subroutine sub1()
    print *, "In sub1"
end subroutine sub1

subroutine sub2()
    print *, "In sub2"
end subroutine sub2
```


Splitting Fortran codes into files

Split into 3 files:

Main program...

```
! $CLASSHG/codes/fortran/multifile1/main.f90
program demo
    print *, "In main program"
    call sub1()
    call sub2()
end program demo
```

and two separate files (for $N = 1, 2$):

```
! $CLASSHG/codes/fortran/multifile1/subN.f90
subroutine subN()
    print *, "In subN"
end subroutine subN
```


Splitting Fortran codes into files

Compile all three and link together into single executable:

```
$ gfortran main.f90 sub1.f90 sub2.f90 \
    -o fullcode.exe
```

Run the executable:

```
$ ./fullcode.exe
 In main program
 In sub1
 In sub2
```

Splitting Fortran codes into files

Can split into separate compile....

```
$ gfortran -c main.f90 sub1.f90 sub2.f90

$ ls *.o
main.o  sub1.o  sub2.o
```

... and link steps:

```
$ gfortran main.o sub1.o sub2.o -o fullcode.exe


$ ./fullcode.exe
 In main program
 In sub1
 In sub2
```


Splitting Fortran codes into files

Advantage: If we modify sub2.f90 to print "Now in sub2" we only need to recompile this piece:

```
$ gfortran -c sub2.f90

$ gfortran main.o sub1.o sub2.o -o fullcode.exe

$ ./fullcode.exe
 In main program
 In sub1
 Now in sub2
```

When working on a big code (e.g. 100,000 lines split between 200 subroutines) this can make a big difference!

Next lecture: Make this easier with Makefiles.

Fortran modules

General structure of a module:

```
module <MODULE-NAME>
    ! Declare variables
contains
    ! Define subroutines or functions
end module <MODULE-NAME>
```

A program or subroutine can use this module:

```
program <NAME>
    use <MODULE-NAME>
    ! Declare variables
    ! Executable statements
end program <NAME>
```


Fortran module example

```
! $CLASSHG/codes/fortran/multifile2/sub1m.f90
module sub1m
contains
subroutine sub1()
    print *, "In sub1"
end subroutine sub1
end module sub1m
```

```
! $CLASSHG/codes/fortran/multifile2/main.f90
program demo
    use sub1m
    print *, "In main program"
    call sub1()
end program demo
```

Fortran modules

Some uses:

- Can define global variables in modules to be used in several different routines.

  In Fortran 77 this had to be done with common blocks — much less elegant.

- Subroutine/function interface information is generated to aid in checking that proper arguments are passed.

  It's often best to put all subroutines and functions in modules for this reason.

- Can define new data types to be used in several routines.

Compiling Fortran modules

If `sub1m.f90` is a module, then compiling it creates `sub1m.o` and also `sub1m.mod`:

```
$ gfortran -c sub1m.f90


$ ls
main.f90    sub1m.f90    sub1m.mod    sub1m.o
```

the module must be compiled before any subroutine or program that uses it!

```
$ rm -f sub1m.mod
$ gfortran main.f90 sub1m.f90
main.f90:5.13:

    use sub1m
            1
Fatal Error: Can't open module file 'sub1m.mod'
for reading at (1): No such file or directory
```


Another module example

```
! $CLASSHG/codes/fortran/circles/circle_mod.f90

module circle_mod

    implicit none
    real(kind=8) :: pi = 3.141592653589793d0

contains

    real(kind=8) function area(r)
        real(kind=8), intent(in) :: r
        area = pi * r**2
    end function area

    real(kind=8) function circumference(r)
        real(kind=8), intent(in) :: r
        circumference = 2.d0 * pi * r
    end function circumference

end module circle_mod
```


Another module example

```
! $CLASSHG/codes/fortran/circles/main.f90

program main

    use circle_mod
    implicit none
    real(kind=8) :: a

    ! print parameter pi defined in module:
    print *, 'pi = ', pi

    ! test the area function from module:
    a = area(2.d0)
    print *, 'area for a circle of radius 2: ', a

end program main
```

Running this gives:

```
pi =    3.14159265358979
area for a circle of radius 2:    12.566370614
```