Array storage

Rank 1 arrays have a single index, for example:

```fortran
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:

```fortran
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:

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

Passing arrays to subroutines

```fortran
program arraypassing1
  implicit none
  real(kind=8) :: x
  integer :: i
  x = 1
  y = 2
  j = 3
  call setvals(x)
  print *, 'x=', x
  print *, 'y=', y
  print *, 'j=', j
end program arraypassing1
```

Note: `x` is a scalar, dummy argument `a` is an array.
Passing arrays to subroutines

This produces:

\[
\begin{align*}
x &= 5.00000000000000 \\
y &= 5.00000000000000 \\
i &= 1075052544 \\
j &= 0
\end{align*}
\]

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.

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

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.
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.

---

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

```bash
$ 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 arraypassing2.f90.
```

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

---

You can also probe the value of variables.

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

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

This tells us that the error occurred 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 $8 \times 402 = 3216$ 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.

```fortran
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)

Note: equivalence statement

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

⇒ 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
```

---

**Notes:**

R.J. LeVeque, University of Washington
AMath 483/583, Lecture 7, April 11, 2011

---

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
```

---

**Notes:**

R.J. LeVeque, University of Washington
AMath 483/583, Lecture 7, April 11, 2011

---

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

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

module circle_mod
    implicit none
    real(kind=8) :: pi = 3.1415926535897938
    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 = 2.00 * pi * r
        end function circumference
end module circle_mod
```

Another module example

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

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

    ! test the area function from module:
    a = area(2.00)
    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: 12.5663706143592
```