

# Goals for This Lecture:

- Learn a simple sorting algorithm
- Understand how compilation & linking of separate main program and subprogram files are accomplished.
- Understand how to use subprograms to create reusable programs
- Understand the **save** attribute and **save** statement and how it affects local variables
- Introduce the subject of modules

# Example using assumed shape arrays: sorting a data set

```
! Purpose: Sort a data set
! Author:  F. Douglas Swesty
! Date:    10/26/2005
program data_sort
implicit none      ! Turn off implicit typing
integer, parameter :: lun1=11  ! Define an LUN for file
integer, parameter :: npts=0   ! Number of data points
real,allocatable :: x(:)      ! Define array to hold data
integer :: i                ! Loop index
integer :: ierror           ! I/O error status variable

interface          ! Define the explicit interface to the subroutine
  subroutine bubble_sort(x)
    implicit none
    real :: x(:)
  end subroutine bubble_sort
end interface

                                ! Open the file
open(unit=lun1,file='exp1.dat',status='OLD',iostat=ierror)
if(ierror /= 0) then
  write(*,*) ' Could not read in file!'
  stop
endif
```

```
do                                ! Find number of points
  read(lun1,*,iostat=ierror) tmpv ! Read in data into
                                array
  if(ierror /= 0) exit
  npts = npts+1
enddo
allocate(x(npts))                ! Allocate array to correct size

rewind(lun1)                      ! Rewind the file to beginning

do i= 1,npts,1                    ! Read data into the array
  read(lun1,*) x(i)
enddo
close(unit=lun1)                  ! Close the file

call bubble_sort(x)              ! Sort the array

write(*,*) ' sorted list: '
do i=1,npts                       ! Loop over data set
  write(*,*) i,x(i)
enddo

stop
end program data_sort
```

# A simple sorting algorithm: The bubble sort

1. for  $i=1,2,\dots,n$
2.     for  $j=1,2,\dots,n-1$
3.         compare elements  $x(j)$  &  $x(j+1)$
4.         if( $x(j) > x(j+1)$ ) then swap  $x(j)$  &  $x(j+1)$
5.     end of inner loop
6. end of outer loop

- Biggest value “bubbles” to the top of array
- The bubble sort is not the fastest sorting algorithm but it is easy to encode
- Therefore we'll use it for this example

# A bubble sort subroutine

```
! Purpose: Bubble sort an array
! Author:  F. Douglas Swesty
! Date:    10/26/2005
subroutine bubble_sort(x)
implicit none      ! Turn off implicit typing
integer :: npts    ! Number of data points
real :: x(:)       ! Define an assumed size array
real :: tmpv       ! Temporary variable
integer :: i, j    ! Loop indices

npts = size(x,dim=1) ! Get size of assumed size array

                        ! Execute a bubble sort of array
do i=1,npts            ! Repeat bubbling task npts times
  do j=1,npts-1,1     ! Bubble bigger values to the top
    if(x(j) > x(j+1) ) then ! Swap elements j & j+1
      tmpv = x(j)
      x(j) = x(j+1)
      x(j+1) = tmpv
    endif
  enddo
enddo

return                ! Return to calling routine
end subroutine bubble_sort
```

# Separate Compilation of Subprograms

- Subprograms can be placed in a separate file and compiled separately from the main program.
- The compilation of each file produces a object file ending in .o
- The two object files can be linked together with the compiler to produce an executable
- Example:

```
ifort -c main.f90
ifort -c subprogram.f90
ifort -o main main.o subprogram.o
```
- This allows flexibility in combining subprograms with main programs

# Subroutine Example 2: Midpoint Rule Program

- Subprograms can be used to evaluate a function needed for a numerical method.
- Having the function placed in a separate file and compiled separately from the main program allows the development of generic numerical method codes.
- Example: Midpoint Rule
  - Subroutines to supply integration limits and evaluate function
  - Place subroutines in separate, problem-specific file

```
! Purpose: Integrate a function via midpoint rule
! Author:  F. Douglas Swesty
! Date:    10/28/2005
program midpoint
implicit none      ! Turn off implicit typing
integer, parameter :: n=100 ! Number of subintervals
integer :: i       ! Loop index
real :: xlow, xhi ! Bounds of integral
real :: dx        ! Variable to hold width of subinterval
real :: sum=0.0   ! Variable to hold sum
real :: xi        ! Variable to hold location of ith
                  ! subinterval
real :: fi        ! Variable to value of function at ith
                  ! subinterval

call limits(xhi,xlow) ! Obtain limits
dx = (xhi-xlow)/(1.0*n) ! Calculate width of subinterval

xi = xlow+0.5*dx      ! Initialize value of xi
do i = 1,n,1         ! Initiate loop
  call integrand(xi,fi) ! Evaluate function at ith
                       ! point
  sum = sum+fi*dx      ! Accumulate sum
  xi = xi+dx           ! Increment location of ith
                       ! point
enddo                ! Terminate loop
write(*, '(t2,a13,1x,1es12.5)') ' integral = ',integral
stop                 ! Stop execution of the program
end program midpoint
```

- Subroutine **limits** supplies the limits of integration.
- Subroutine **integrand** evaluates the function at a specified point
- Placing these two routines in the same file allows all problem-specific information to be contained in a single file that is separate from the problem-independent midpoint rule code.
- Solving a different problem means creating a new set of **limits** & **integrand** subroutines

```

! Purpose: Evaluate  $f(x) = x + \text{abs}(\sin(2x))$  and
!           supply limits of integration
! Author:   F. Douglas Swesty
! Date:    10/28/2005
subroutine limits(xlow,xhi)
implicit none      ! Turn off implicit typing
real, intent(out) :: xlow, xhi ! Bounds of integral
xlow = 0.0          ! Lower limit of integration
xhi = 2.0*3.14159273 ! Upper limit of integration
return
end subroutine limits
!-----
subroutine integrand(x,fofx)
implicit none      ! Turn off implicit typing
real, intent(in)  :: x          ! Value of x
real, intent(out) :: fofx       ! Value of function
fofx = x+abs(sin(2.0*x))       ! Evaluate function
return
end subroutine integrand

```

# A Fact About Local Variables

- According to the FORTRAN 90, 95, and 2003 standards the values of local variables in subroutines become undefined when the procedure in which they are defined is exited.
- FORTRAN provides a way to preserve the values with the `save` attribute and `save` statement
- Example of save attribute use:  
`real, save :: x_position`
- Example of save statement use:  
`save :: x_position, y_position`
  - Values of variables `x_position` & `y_position` are preserved between accesses of the procedure in which they are defined
  - If no variables are listed in the `save` statement then all local variables have their values preserved between accesses of the procedure in which they are defined

# Sharing Data Using Modules

- Argument lists are one way of sharing data objects between calling programs and subprograms
- Another mechanism that FORTRAN provides to accomplish data sharing is the **module**
- Modules are external containers where variables defined outside of subprograms and main programs
- Program units can **use** the modules to gain access to data
- Modules are a way to share data between procedures without passing data via argument lists

# A Module Example

## W/ Module

```
module example_1
save
real :: x(10)
end module example_1
```

```
program sub_ex3
use example_1
implicit none
call arr_sub()
stop
end program sub_ex3
```

```
subroutine arr_sub()
use example_1
implicit none
x = 2.0
return
end subroutine arr_sub
```

## W/O Module

```
program sub_ex3
implicit none
real :: x(10)=1.0
interface
  subroutine arr_sub(a)
    implicit none
    real :: a(:)
  end subroutine arr_sub
end interface
call arr_sub(x)
stop
end program sub_ex3
```

```
subroutine arr_sub(a)
implicit none
integer :: n
real :: a(:)
n = size(a,dim=1)
a(1:n) = 2.0
return
end subroutine arr_sub
```

# Modules

- Form:

```
module module_name  
  declaration section  
  ...  
end module module_name
```

- Module is accessed from a procedure via the **use** statement
- Form:  

```
use module_name
```

  - Any **use** statements must precede any implicit none statements
- The use of the **save** statement in the module guarantees that data values are preserved between references to the module by different routines. Alternatively the variables in the module could have been declared with the **save** attribute.

# Reading Assignment

- Note: Mid-Term Examination will be closed book
- Sections 7.4-7.6, 9.2-9.3 of Chapman