

Goals for This Lecture:

- Understand what modules are
- Understand what module procedures are and how to use them
- Understand explicit and implicit interfaces
- Understand what automatic arrays are and how to use them

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()
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.
- Modules need to be compiled before the routines that access them are compiled

Module Procedures

- Modules can contain subprograms in addition to data
- These are often referred to as module procedures
- Example:

```
module array_sort
  implicit none
  contains
    subroutine bubble_sort(x)
      implicit none
      real :: tmpv, x(:)
      integer :: npnts, i, j
      npnts = size(x,dims=1)
      do i = 1,npnts
        do j = 1,npnts-1,1
          if(x(j) > x(j+1)) then
            tmpv = x(j)
            x(j) = x(j+1)
            x(j+1) = tmpv
          endif
        enddo
      enddo
      return
    end subroutine
end module array_sort
```

Modules

- Form:

```
module module_name
  declaration section
contains
  subroutine sub1(arg_list1)
  ...
  end subroutine sub1
  subroutine sub2(arg_list2)
  ...
  end subroutine sub2
  ...
end module module_name
```

- The contains statement signals that subsequent code defines procedures that are to be contained within the module
- If the module is accessed from the main program via a **use** *module_name* statement then the subroutines can be called as they would normally

Module procedures avoid use of interface blocks

```
! Purpose: Sort a data set
! Author:  F. Douglas Swesty
! Date:    10/26/2005
program data_sort

use array_sort  ! Use the array_sort module (no
  interface
  routine)      ! block needed for bubble_sort

implicit none  ! Turn off implicit typing
integer, parameter :: lun1=11  ! Define file LUN
integer, parameter :: npts=0   ! Number data pts
real, allocatable :: x(:)      ! Array to hold data
integer :: i                   ! Loop index
integer :: ierror              ! I/O error status
real :: tmpv                   ! Temporary var.

! 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
  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
```

Module Procedures & Interfaces

- The process of accessing information contained within a module by means of a **use** statement is called **use association**
- Why employ modules to contain subroutines?
 - When a subprogram is accessed via use association the details of the interface are made known to the compiler when the calling program is compiled.
- A procedure compiled within a module is said to have an explicit interface
 - Detailed information about the interface is available to any program unit accessing the procedure through use association
- Procedures compiled outside a module are said to have an implicit interface
 - Only limited information about the procedure interface is known to the compiler when compiling the calling program unit
 - It just assumes that the number, type, intent, etc. of the arguments is OK
 - Some intrinsic functions such as **size**, **ubound**, **lbound**, etc. may not work correctly in procedures with implicit interfaces

Automatic Arrays

- Occasionally it is desirable to quickly and easily create an array within a subprogram that is dynamic in size but which does not appear in the argument list.
- FORTRAN provides a way to do this through the mechanism of automatic arrays.
- The array is an explicit shape array whose dimensions are specified in terms of an integer variable, or integer expression based on a variable, that is in the argument list or which is accessed by use association.
- Example:

```
subroutine sub_ex(n)
  implicit none
  real :: x(n)
  ...
  return
end subroutine sub_ex
```

Automatic Arrays vs. Allocatable Arrays

- Automatic arrays & allocatable arrays can both be used to create temporary arrays
- Automatic arrays are allocated whenever the routine in which they are defined is entered and deallocated whenever the routine is exited. Allocatable arrays must be allocated & deallocated manually.
- Allocatable arrays are more flexible
 - They can be created in main programs
 - Arrays can be created and destroyed in separate routines
 - Allocatable arrays can be contained within modules
- Allocatable arrays can create **memory leaks**
 - An array allocated in a routine and not deallocated ties up the memory associated with that array until the program terminates.

Reading Assignment

- Sections 7.4, 7.5, 11.1-11.2 of Chapman