Goals for This Lecture:

- Understand the pass-by-reference argument passing mechanism of FORTRAN
- Understand how arrays are passed to a FORTRAN subprogram
- Learn a simple sorting algorithm

The `RETURN` statement

Basic Form:

```
return
```

- Multiple return statements are permitted within a subroutine
- Execution of a `return` statement causes the subroutine to terminate and control returns to the first executable line following the call statement

Using stacks of subroutines

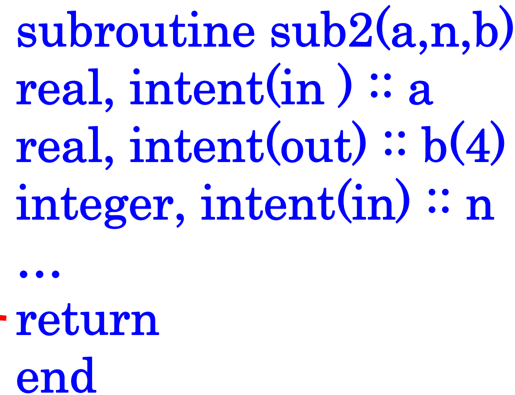
- Fun Factoid # 2:
  - One subroutine can call another
  - But, a subroutine cannot call itself unless it is declared as a recursive subroutine
    - Form:

```
recursive subroutine subroutine_name(parameter_list)
declaration section
…
execution section
…
end subroutine subroutine_name
```

  - Recursive calls are very expensive and to be avoided if at all possible

Argument passing by reference

```
program sub_ex2
real :: x, y(4)
integer :: npts
...
call sub2(x,npts,y)
...
stop
end
```



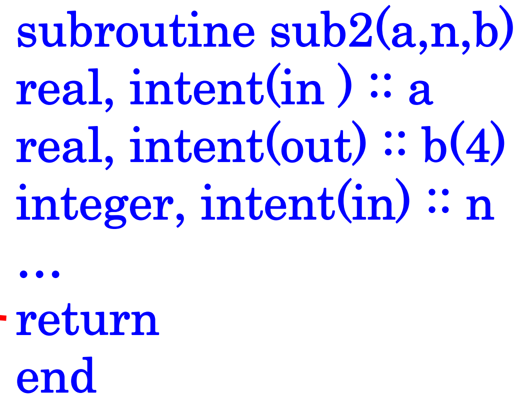
```
subroutine sub2(a,n,b)
real, intent(in ) :: a
real, intent(out) :: b(4)
integer, intent(in) :: n
...
return
end
```

- Fortran associates the memory addresses of the arguments in the main program with the corresponding dummy variables in the subroutine each time it is called.
- Therefore changes to variables in the subroutine are actually changes to the variables in the main program

Memory Address	Main program variable name	Subroutine variable name
41573	x	a
61231	npts	n
02351	y(1)	b(1)
02355	y(2)	b(2)
02359	y(3)	b(3)
02363	y(4)	b(4)

Argument passing by reference

```
program sub_ex2
real :: x, y(4)
integer :: npts
...
call sub2(x,npts,y)
...
stop
end
```



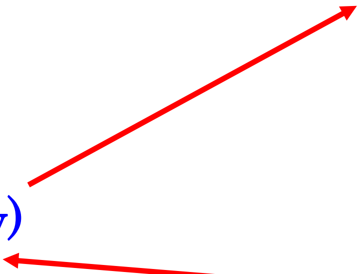
- This argument passing mechanism avoids the copying of one variable into another (a very time and memory consuming operation)

- The address associated with a variable is actually the only thing passed.

Memory Address	Main program variable name	Subroutine variable name
41573	x	a
61231	npts	n
02351	y(1)	b(1)
02355	y(2)	b(2)
02359	y(3)	b(3)
02363	y(4)	b(4)

Argument passing by reference

```
program sub_ex2
real :: x, y(4)
integer :: npts
...
call sub2(x,npts,y)
...
stop
end
```



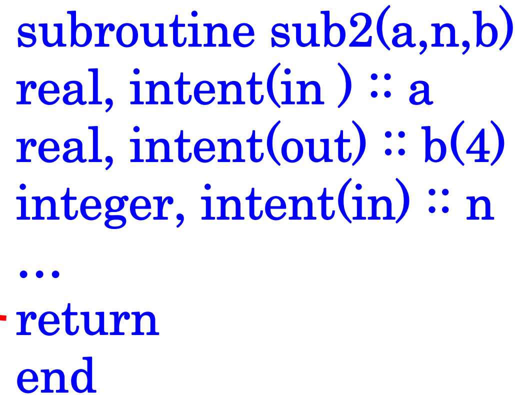
```
subroutine sub2(a,n,b)
real, intent(in ) :: a
real, intent(out) :: b(4)
integer, intent(in) :: n
...
return
end
```

- Note that the array element memory locations are located in sequence at addresses that are 4 bytes apart
- Once the address of y(1) is associated with the address of b(1) the addresses of b(2),b(3), and b(4) are known (they are spaced 4 bytes apart).
- There is no need to pass addresses for the entire array

Memory Address	Main program variable name	Subroutine variable name
41573	x	a
61231	npts	n
02351	y(1)	b(1)
02355	y(2)	b(2)
02359	y(3)	b(3)
02363	y(4)	b(4)

Argument passing by reference

```
program sub_ex2
real :: x, y(4)
integer :: npts
...
call sub2(x,npts,y)
...
stop
end
```



```
subroutine sub2(a,n,b)
real, intent(in ) :: a
real, intent(out) :: b(4)
integer, intent(in) :: n
...
return
end
```

- This association of a sequence of elements based on passing one address is called array element sequence (AES) association
- AES association is very efficient for passing long arrays compared to passing the address of every element or copying the entire array into a dummy variable array.

Memory Address	Main program variable name	Subroutine variable name
41573	x	a
61231	npts	n
02351	y(1)	b(1)
02355	y(2)	b(2)
02359	y(3)	b(3)
02363	y(4)	b(4)

Passing arrays into subroutines. method 1: Explicit shape arrays

- Declare array to have same shape in both calling routine and subroutine
- Least flexible way of doing things
- Requires changing subroutine every time array length in calling routine is changed.

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

subroutine arr_sub(a)
implicit none
real :: a(10)
a = 2.0
return
end subroutine arr_sub
```

Passing arrays into subroutines. method 2: Adjustable arrays

- Declare array to have a variable size where the size is determined by integer expression passed as an argument
- Allows for variable size arrays
- Array size variable must be an argument in the subroutine
- Your book incorrectly confuses the terms explicit shape array with adjustable arrays

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

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

Passing arrays into subroutines. method 3: Assumed shape arrays

- Declare array to have a variable size using a colon as a place holder
- Array in subroutine assumes the size of array in calling routine
- Array size or bounds can be obtained by using the SIZE, LBOUND, or UBOUND intrinsic functions
- Must use an interface block (or put the subroutine in a module) to describe the subroutine interface to the calling program (more on this later)
- In general, this is the best method of passing arrays
  - Allows for very flexible subroutines

```
program sub_ex3
implicit none
real :: x(10)=1.0

interface
  subroutine arr_sub(x)
  implicit none
  real :: x(:)
  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
```

Passing arrays into subroutines. method 4: Assumed size arrays

- Declare array to have a variable size using a * as a place holder in last dimension
- Array in subroutine assumes the size of array in calling routine
- Array extent in the last dimension is unknown
  - No way to get this info from array
- Old-fashioned way of doing things
  - Avoid if possible

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

subroutine arr_sub(a)
implicit none
integer :: n
real :: a(*)
return
end subroutine arr_sub
```

Passing arrays into subroutines. method 5: Pseudo-variable dimensioning

- Declare array to have rank-1 and extent-1 in subroutine
- Array Element Sequencing forces a(1) to have same address as x(1)
  - Perfectly legal
- Array size & extent is unknown
  - No way to get this info from array
- Can't use compiler bounds checking
- Can be used to create a rank-free pointer
- Avoid if possible

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

subroutine arr_sub(a)
implicit none
real :: a(1)
integer :: i
do i=1,10
  a(i) = 1.0
enddo
return
end subroutine arr_sub
```

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
real :: tmpv                     ! Temporary 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,…,n
2. for j=1,2,…,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
integer :: i, j                              ! Loop indices
real :: tmpv                                 ! Temporary variable

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
```