

# Goals for This Lecture:

- Understand what function subprograms are
- Understand how to use function subprograms
- Understand the various kinds of REAL types
- Understand how to select precision in a processor independent manner

# Function Subprograms

- Thus far we have used **intrinsic functions** which are built into the language
- FORTRAN provides a mechanism to define new functions in the form of **function subprograms**, a.k.a. user-defined functions
- Form:

```
type function func_name(arg_list)
```

```
...
```

```
declaration section
```

```
...
```

```
execution section
```

```
func_name = expr
```

```
return
```

```
end function func_name
```

# Function Example: Midpoint Rule Program

- We can create a version of the midpoint rule program that uses functions instead of subroutines
- Create separate functions for upper & lower limits of integration as well as for integrand
- Function names must be declared with the appropriate type

```
! 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
real :: upper_limit, lower_limit, integrand ! Function subprogs

xhi_ = upper_limit() ! Obtain upper limit
xlow_ = lower_limit() ! Obtain lower limit

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
  fi = integrand(xi) ! 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
```

- Functions `upper_limit` and `lower_limit` supply the limits of integration.
- Function `integrand` evaluates the function at a specified point
- Good programming tip:
  - Use `implicit none` in function declaration section.
  - Declare function type in both definition and calling program
- Functions can be contained in modules and can access data in modules via use association

```

! Purpose: Evaluate f(x) = x+abs(sin(2x)) and
!          supply limits of integration
! Author:  F. Douglas Swesty
! Date:    10/28/2005
real function upper_limit()
implicit none      ! Turn off implicit typing
upper_limit = 2.0*3.14159273 ! Return value
return
end function upper_limit
!-----
real function lower_limit()
implicit none      ! Turn off implicit typing
lower_limit = 0.0 ! Return value
return
end function lower_limit
!-----
real function integrand(x)
implicit none      ! Turn off implicit
                  typing
real, intent(in) :: x      ! Value of x
integrand = x+abs(sin(2.0*x)) ! Evaluate function
return
end function integrand

```

# Side Effects of Function Subprograms

- Input values are passed to function subprograms by the same pass-by-reference mechanism that is used for argument passing in subroutines
- Thus if the function parameters are modified then the corresponding arguments are also changed.
- This can lead to unintended side effects in programs
- It is illegal for a function to modify an argument which appears elsewhere in the same expression
- Good programming tip:
  - Never modify the parameters in a subroutine argument list in a program
  - Always declare arguments in a subroutine with the `intent(in)` attribute
- If you need to modify arguments use a subroutine instead of a subprogram

# The **RESULT** clause

- The return value of the function can be specified through the use of the result clause
- Form:

```
function func_name(arg_list) result(result_var)  
...  
type :: result_var  
...  
result_var = expr  
return  
end function func_name
```
- The type of the result variable listed in the **result** clause must be specified in a type declaration specification

# Vector-valued functions

- Using the result clause vector valued functions can be constructed
- Example:

```
function vec_init(x) result(vf)
  implicit none
  real :: x
  real :: vf(3)
  return
end function vec_init
```

- In order to use this function in a calling routine an explicit interface must exist
- This can be done through an interface block or through the inclusion of the function in a module

# Using vector-valued functions with interface block

- Example:

```
program vfunc_ex1
  implicit none
  real :: x=1.0
  real :: xvector(3)
  interface
    function vec_init(x) result(vf)
      implicit none
      real :: x
      real :: vf(3)
    end function vec_init
  end interface
  vector = vec_init(x)
  write(*,*) xvector
  stop
end function vec_init
```

# Using vector-valued functions with modules

- Example:

```
module vector_funcs
contains
  function vec_init(x) result(vf)
  implicit none
  real :: x
  real :: vf(3)
  end function vec_init
end module vector_funcs
program vfunc_ex1
use vector_funcs
implicit none
real :: x=1.0
real :: xvector(3)
vector = vec_init(x)
write(*,*) xvector
stop
end function vec_init
```

# Kinds of REAL variables

- The default REAL type on most machines is a 32-bit (4 byte) real
- This usually offers only seven digits of precision and can usually only represent numbers between  $10^{38}$  and  $10^{-38}$
- Such 4-byte real values are often referred to as single precision variables
- Most systems can support a 64-bit (8 byte), a.k.a. double precision, real type
- **Always use double precision whenever possible!**
  - **We will do so throughout the remainder of this course!**
- The precision can be specified with a **KIND=** parameter to the **real** declaration statement

```
real(kind=4) :: x_velocity
real(kind=8) :: acceleration
```
- There is no guarantee that either single or double precision types will have the same representation of different compilers or machines
  - On some machines double precision may be specified by **kind=8** while on others it might be specified by **kind=2**

# Parameterizing your REAL declarations

- The kind of REAL type can be parameterized through the use of integer parameters contained in modules
- Makes it trivial to change type declarations by changing one or two lines in a module
- Example:

```
module real_kinds
    integer, parameter :: single_type=4
    integer, parameter :: double_type=8
end module real_kinds
program kind_example
use real_kinds
implicit none
real(kind=single_type) :: x_velocity
real(kind=double_type) :: acceleration
...
stop
end program kind_example
```

# Parameterizing your REALs in a processor independent manner

- The fact the single and double precision real types can change from compiler to compiler is still a pain.
- Fortunately, FORTRAN provides an intrinsic function `kind` to determine the correct kind parameter based on the specification of a constant
- Example:

```
module real_kinds
  integer, parameter :: single_type=kind(1.0)
  integer, parameter :: double_type=kind(1.0d0)
end module real_kinds
program kind_example
use real_kinds
implicit none
real(kind=single_type) :: x_velocity
real(kind=double_type) :: acceleration
...
stop
end program kind_example
```

Always use double precision whenever possible!

# Reading Assignment

- Sections 7.4, 7.5, 11.1-11.2 of Chapman