Fortran - Subprograms

Functions, Subroutines, Interfaces, Modules

Spring 2017

Victor Eijkhout and Charlie Dey

Subprograms

Subroutines and Functions

Our programs need to be organized and modular.

We achieve this through the use of Subroutines and Functions.

Subprograms

Subroutines and Functions

```
program without_fct

integer, parameter :: m = 100
integer            :: n, n2, i, j
real, dimension(m) :: a, a2
real               :: sum, aver, …

! Read data (n,a) from a file

! Calculate Average
sum = 0.
do i=1, n; sum = sum + a(i); enddo
aver = sum / real(n)

! Read more data (n2, a2)
open …; read …; close …

! Calculate Average again
s2 = 0
do j=1, n2
  s2 = s2 + a2(j)
enddo
aver2 = s2 / real(n2)
end program
```

Without using functions/subroutines, a lot of tedious coding.

Subprograms

Function Example

```
real function average(n, x)
integer             :: n, i
real, dimension(n)  :: x
real                :: sum
sum = 0.
do i=1, n
  sum = sum + x(i)
enddo
average = sum / real(n)
end function average

program with_fct
! Declaration of variables
! Read data (n,a)

! Calculate Average
aver = average(n, a)    ! Function
                        !   call
! Read more data (n2, a2)
open …; read …; close …

! Calculate Average again
aver2 = average(n2, a2)
end program
```

Instead, let's invoke a function average()
we now have less code and more reuse.

Subprograms

Subroutines and Functions

Advantages are:

- Reusable code
  - Function can be called multiple times and with different arguments
- Insulation from unintended side effects
  - only variables in the argument list are communicated
  - Local variables (i, sum) do not interfere
- Independent testing of subtasks
  - function compiled and tested separately

NOTE:

- The names in the parameter lists in the function definition and the function call do need not to have the same name but have to be the same type
- All arguments are "passed by reference"
  - if their value of the parameter changes in the function, the corresponding variable within the main program also changes.

Subprograms

Subroutines

```
subroutine average(aver, n, x)
integer             :: n, i
real, dimension(n) :: x
real                :: aver, sum
sum = 0.
do i=1, n
  sum = sum + x(i)
enddo
aver = sum / real(n)
end subroutine average

program with_sub
! Declaration of variables
! Read data (n1,a1)

! Calculate Average
call average(aver1, n1, a1)       ! Subroutine call
! Read more data (n2, a2)
open …; read …; close …

! Calculate Average again
call average(aver2, n2, a2)
end program
```

Since everything is pass by reference, we can rewrite our earlier example using a subroutine instead.

Subprograms

Structure: Main Program

```
program name
   specifications
   execution statements
   [ contains
     internal routines ]
end program [ Name ]
```

Specifications

- include use of modules
- implicit or strong typing
- namelist declaration
- type definitions
- variable declarations

Internal routines are subroutines and/or functions defined inside encapsulating program unit

Subprograms

Structure: Subroutines and Functions

```
subroutine name[ (argument list) ]
   specification statements
   execution statements
   [ contains
     internal routines ]
end subroutine [ name ]

return-type function name[ (argument list) ]
   specification statements
   execution statements
   [ contains
     internal routines ]
end function [ name ]
```

Argument list - a way of passing data in/out of a subroutine or function

Specifications

- include use of modules
- implicit or strong typing
- namelist declaration
- type definitions
- variable declarations

Subroutines/Functions may also have internal routines of other subroutines and/or functions defined inside encapsulating subroutine/function unit

Subprograms

Arguments: Subroutines and Functions

```
real function average(n, x)
integer             :: n, i
real, dimension(n)  :: x
real                :: sum
sum = 0.
do i=1, n
  sum = sum + x(i)
enddo
average = sum / real(n)
end function average

program with_fct
! Declaration of variables
real :: average ! declare your function here.
… other declarations as normal ...
! Read data (n1,a1)

! Calculate Average
aver = average(n1, a1)    ! Function
                        !   call
! Read more data (n2, a2)
open …; read …; close …

! Calculate Average again
aver2 = average(n2, a2)
end program
```

- Arguments passed to routines are alled actual arguments, e.g. `n1, a2, n2` and `a2` in the main program
- Arguments in routines are called dummy arguments, e.g.`n` and `x` in the function
- Actual and dummy arguments must have number and type conformity.

Subprograms

Subroutines and Functions

- Subroutines
  - enables modular programming
  - structured like main program, but with argument list
  - may be internal, i.e. resides in the main program
  - or external, i.e. resides in "modules"
  - does *not* return a value
- Functions
  - enables modular programming
  - similar to subroutines (argument list, structure)
  - may be internal or external
  - returns a value

Subprograms

Summary: Subroutines vs Functions

```
subroutine average(aver, n, x)
implicit none
integer              :: n, i
real, dimension(n)   :: x
real                 :: sum
sum = 0.
do i=1, n
  sum = sum + x(i)
enddo
aver = sum / real(n)
end subroutine average
```

```
real function average(n, x)
implicit none
integer              :: n, i
real, dimension(n)   :: x
real                 :: sum
sum = 0.
do i=1, n
  sum = sum + x(i)
enddo
average = sum / real(n)
end function average
```

What's different vs. C/C++?

- no return statement
- all parameters are passed by reference
- function name is the return argument in a function

Subprograms

Organization

```
subroutine helloWorld
print *, "Hello World"
end subroutine

program hello
implicit none

call helloWorld

end program
```

Subroutines or Functions can appear at the top, before the program.

Subprograms

Organization

```
program hello
implicit none

call helloWorld

end program

subroutine helloWorld
print *, "Hello World"
end subroutine
```

Subroutines or Functions can appear at the bottom, after the program.

Subprograms

Organization

```
program hello
implicit none

call helloWorld

contains

subroutine helloWorld
   print *, "Hello World"
end subroutine

end program
```

Or the preferred method: Subroutines or Functions can *inside* the program after the your `execution` section using the `contains` keyword.

Subprograms - Exercise 1

Subroutines and Functions

Since all arguments are passed by reference,

write a subroutine swap of two parameters that exchanges the input values:

```
integer :: i=2,j=3
call swap(i,j)
```

Subprograms

Subroutines and Functions - Safeguarding your arguments

INTENT allows us to declare the intended behaviour of an argument.

INTENT(IN)	- the argument is for input only
INTENT(OUT)	- the argument is for output only
INTENT(INOUT)	- the argument is for input and/or output

Subprograms

Subroutines

```
subroutine average(aver, n, x)
integer, intent(in):: n
integer            :: i
real, dimension(n), intent(in) :: x
real, intent(out)    :: aver
real               :: sum
sum = 0.
do i=1, n
  sum = sum + x(i)
enddo
aver = sum / real(n)
end subroutine average

program with_sub
! Declaration of variables
! Read data (n1,a1)
! Calculate Average
call (aver1, n1, a1)       ! Subroutine
call
! Read more data (n2, a2)
open …; read …; close …

! Calculate Average again
call average(aver2, n2, a2)
end program
```

Since everything is pass by reference, we can rewrite our earlier example using a subroutine instead.

Subprograms - Exercise 1a

Subroutines and Functions

Rewrite Exercise 1 so that the subroutine swaps the values around, but also returns the old values with the proper intent.

```
subroutine swap(i, j, i_old, j_old)
{
…
}
```

Subprograms - Project Exercise 2

Subroutines and Functions

Write a function that takes an integer input and returns a logical corresponding to whether the input was prime.

```
logical :: isprime
isprime = prime_test_function(13)
```

Read the number in, and print the value of the logical.

Subprograms - Project Exercise 3

Subroutines and Functions

Take the prime number testing program, and modify it to read in how many prime numbers you want to print.

Print that many successive primes.

Keep a variable `number_of_primes_found` that is increased whenever a new prime is found.

Interfaces

A Definition

Recall:

- **Function** subprograms in Fortran have an *explicit* type and are intended to return *one* value.
- **Subroutine** subprograms have no *explicit* type and return *multiple* or no values through the **parameter call list.**

Interfaces

A Definition

Introducing the INTERFACE block.

- This block is safety feature which allows main programs and external subprograms to *interface* appropriately.
- An interface block ensures that the calling program and the subprogram have the correct number and type of arguments.
- This helps the compiler to detect incorrect usage of a subprogram at compile time. An interface block consists of:
  - The number of arguments
  - The type of each argument
  - The type of the value(s) returned by the subprogram

Interfaces

A Definition

Introducing the INTERFACE block.

- This block is safety feature which allows main programs and external subprograms to *interface* appropriately.
- An interface block ensures that the calling program and the subprogram have the correct number and type of arguments.
- This helps the compiler to detect incorrect usage of a subprogram at compile time. An interface block consists of:
    - The number of arguments
    - The type of each argument
    - The type of the value(s) returned by the subprogram

Interfaces

example

```
program Area

implicit none

interface
   real function Area_Circle (r)
     real, intent(in)  :: r
   end function Area_Circle
end interface

real :: radius

print *, "enter a radius:"
read*, radius

print *, "Area of circle with radius", radius, " is", &
            Area_Circle(radius)

end program

real function Area_Circle(r)

implicit none
real, intent(in)  :: r

! Declare local constant Pi
real, parameter :: Pi = 3.1415927

Area_Circle = Pi * r * r

end function Area_Circle
```

This program makes use of the function Area_Circle to compute the area of a circle of radius r.

The function appears after the `end program` of the main program Area, so it is classified as an external subprogram.

Because it is an external routine, the main program makes use of an interface block to define all of the parameters required by the function Area_Circle.

Interfaces

Polymorphism i.e. overloading functions

Polymorphism refers to a programming language's ability to process objects differently depending on their data type or class.

Fortran doesn't really do *Polymorphism* or *function overloading*, but we can achieve this with an **`Interface`**

Subprograms

Silly example: Interface

```
program Demo
implicit none
integer :: i, j
real :: x, y
i = 1
j = 2
x = 1.5
y= 2.5

call printvalues_real(x, y)
call printvalues_integers(i, j)

contains

   subroutine printvalues_integers(a, b)
      integer :: a, b
      print *, a, b
   end subroutine printvalues_integers

   subroutine printvalues_real(a, b)
      real :: a, b
      print *, a, b
   end subroutine printvalues_real

end program Demo
```

We have a program and 2 subroutines
One subroutine prints integers.
One subroutine prints reals.

Interface

Silly example

```
program Demo
implicit none

interface printValues
   subroutine printValues_integer(a, b)
      integer :: a, b
   end subroutine printValues_integer

   subroutine printValues_real(a, b)
      real :: a, b
   end subroutine printValues_real
end interface printValues

integer :: i = 1, j = 2
real :: x = 1.5, y = 2.5

call printValues(x, y)
call printValues(i, j)

end program Demo
```

```
   subroutine printvalues_integers(a, b)
      integer :: a, b
      print *, a, b
   end subroutine printvalues_integers

   subroutine printvalues_real(a, b)
      real :: a, b
      print *, a, b
   end subroutine printvalues_real
```

Now we have one Interface that contains multiple subroutine. The interface is called with any parameter type and the proper subroutine is invoked.

Interface - Exercise 4

Write a interface, `calculate_circumference()` that calculates the circumference of an assumed shape depending on the number of arguments passed.

3 arguments, `calculate_circumference()` assumes it's a triangle

2 arguments, `calculate_circumference()` assumes it's a rectangle

1 arguments, `calculate_circumference()` assumes it's a circle

Modules

- Modules provide a flexible mechanism to organize content
- Modules may contain all kinds of things
  - Declaration of:
    - Parameters (named constants)
    - Variables
    - Arrays
    - Derived Types
    - Structures
  - Subprograms
    - Subroutines
    - Functions
    - Other modules

Modules

Be forewarned…

Silly Example Ahead.

Modules

Silly example

```
module mad_science
implicit none
real, parameter :: pi = 3.  ,c   = 3.e8 ,e   = 2.7
real             :: r
end module mad_science

program go_mad
! make the content of module available
use mad_science
implicit none

r = 2.
print *, 'Area = ', pi * r**2
end program
```

Our module has a few parameters defined:

- pi
- c
- e

and a real variable defined

- r

Modules

Silly example

```
module mad_science
implicit none
real, parameter :: pi = 3.14159 ,c  = 3.e8 ,e  = 2.7
real            :: r
contains
   real function Area_Circle(r)
      real :: r
      Area_Circle = r*r*pi
   end function
end module mad_science

program go_mad
! make the content of module available
use mad_science
implicit none
real :: area

r = 2.
area = Area_Circle(r)
print *, 'Area = ', area
end program
```

Our module has a few parameters defined:

- pi
- c
- e

and a real variable defined

- r

and a function

- Area_Circle

What does this remind you of now?

Modules

Silly example

```
module mad_science
real, parameter :: pi = 3.,   &
                   c  = 3.e8, &
                   e  = 2.7
real              :: r
type scientist
  character(len=10) :: name
  logical           :: mad
  real              :: height
end type scientist
end module mad_science
```

Introducing `type`

What does this remind you of now?

Modules

Silly example

```
module mad_science
real, parameter :: pi = 3.,    &
                   c  = 3.e8,  &
                   e  = 2.7
real            :: r
type scientist
  character(len=10) :: name
  logical           :: mad
  real              :: height
end type scientist
end module mad_science

program main
use mad_science
type(scientist) :: you

you%name   = 'some name'
you%height = 1.78
```

Introducing `type`

What does this remind you of now?

Modules

Silly example

```
module mad_science
real, parameter :: pi = 3.,    &
                   c  = 3.e8,  &
                   e  = 2.7
real            :: r
type scientist
  character(len=10) :: name
  logical           :: mad
  real              :: madness_level
end type scientist
end module mad_science

program main
use mad_science
type(scientist) :: you, me

you%name    = 'Victor'
you%mad = .true.
you%madness_level = 8.7

me%name = 'Charlie'
me%mad = .true.
me%madness_level = 9
```

Modules as Objects.

Modules

Silly example

```
module mad_science
real, parameter :: pi = 3.,   &
                   c  = 3.e8, &
                   e  = 2.7
real            :: r
type scientist
  character(len=10) :: name
  logical           :: mad
  real              :: madness_level
end type scientist
contains
subroutine is_mad(s)
   type(scientist) :: s
   if (s%mad .and. s%madness_level > 8) then
      print *, "He's crazy mad!"
   end if
end subroutine
end module mad_science

program main
use mad_science
type(scientist) :: you, me

you%name   = 'Victor'
you%mad = .true.
you%madness_level = 8.7
```

Modules as Objects.

Modules - Exercise 5

Within a Module, `Plane`:

Make type `Point(x, y)` where x and y are both real numbers.
`Point(x, y)`
and a function `distance` so that if p,q are Point "objects", calling distance(p,q) computes the distance.

Modules - Exercise 5

Within a Module, `Plane`:

Add another type `LinearFunction`

LinearFunction that is defined with 2 points, `Point input_p1,Point input_p2`

and a real function `evaluate_at(line, x )`, with x being of type `real` and `line` being type `LinearFunction`

return the point on the line at x=4.0;