

Fortran - Subprograms

Functions, Subroutines, Modules

Kevin Schmidt, Susan Lindsey, 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

```
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)
contains
  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
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 the value of the parameter changes in the function, the corresponding variable within the main program also changes.

Subprograms

Subroutines

```
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)
contains
  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
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: Main Program + Subroutine

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

end program [ Name ]
```

Specifications also required in the subprogram

- 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

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

Organization

```
program hello
implicit none

call helloWorld
print *, myAdd(1, 2)

contains

subroutine helloWorld
print *, "Hello World"
end subroutine

integer function myAdd(a, b)
implicit none
integer :: a, b

myAdd = a + b
end function

end program
```

Subroutines or Functions should be placed *inside* the program after the your execution section using the `contains` keyword.

Note: In this case, the compiler *will* catch an error between the function call and the function definition i.e. calling `myAdd (1.5, 2.5)` or `myAdd (1, 2, 3)`

Subprograms

Arguments: Subroutines and Functions

```
program with_fct
! Declaration of variables
... 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)
contains
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
end program
```

- Arguments passed to routines are called 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 <value>` statement
- function name is the return *argument* in a function
- all parameters are passed by reference

Subprograms - Practice

Subroutines and Functions

Since all arguments are passed by reference,

write a subroutine called swap that takes two parameters that exchanges the values:

```
integer :: i=2,j=3
print *, i, j ! original values
call swap(i,j)
print *, i, j ! values should be swapped
```

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

```
program with_sub
! Declaration of variables
! Read data (n1,a1)
! Calculate Average
call (aver1, n1, a1)      ! Subroutine

! Read more data (n2, a2)
open ...; read ...; close ...
! Calculate Average again
call average(aver2, n2, a2)
contains
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
end program
```

Let's add some INTENT

Subprograms - in class*

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 - Fortran Project Exercise 1*

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 - Fortran Project Exercise 2*

Subroutines and Functions

Write a subroutine that reads in an argument that represents how many prime numbers you want to print and prints that many successive primes.

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                :: madness_level
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                :: madness_level
end type scientist
end module mad_science

program main
use mad_science
type(scientist) :: you

you%name    = 'some name'
you%madness_level = 4.5
```

Introducing `type`

What does this remind you of now?

Modules

Silly example

```
module mad_science
implicit none
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    = 'Jeaime'
you%mad = .true.
you%madness_level = 8.7

me%name = 'Charlie'
me%mad = .true.
me%madness_level = 9 ! I have kids.
end program
```

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 *, "is crazy mad!"
  end if
end subroutine
end module mad_science

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

you%name    = 'Carrie'
you%mad = .true.
you%madness_level = 8.7
call is_mad(you)
end program
```

Modules as Objects.

Modules - In Class Challenge 6

Within a Module, PointMod:

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

Modules - In Class Challenge

Within your Module, PointMod:

Add another type LinearFunction

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

Add a real function

evaluate_at(line, x), with x being of type real and line being type LinearFunction and returns the Point on the line at x=4.0;

Fortran Project - Preview

Goldbach Conjecture

The Goldbach conjecture says that every even number $2n$ (starting at 4), is the sum of two primes $p + q$:

$$2n = p + q.$$

Equivalently, every number n is equidistant from two primes. In particular this holds for each prime number:

$$\forall_{p \text{ prime}} \exists_{q \text{ prime}} : r \equiv p + (p - q) \text{ is prime.}$$

Write a program in Fortran that tests this. You need two prime number generators, one for the p -sequence and one for the q -sequence. For each p value, when the program finds the *first* q value, print the q, p, r combination and move on to the next p .

Allocate an array where you record all the $p - q$ distances that you found. Print some elementary statistics, for instance: what is the average distance, do the distances increase or decrease with p ?

Fortran Project - Preview (in other words)

Goldbach Conjecture

Find a prime number ' p ', find the next prime number after ' p ' let's call this ' q '

if $p-(q-p)$ is also prime we'll call that ' r ', and you found a triple we're looking for.
Keep doing this for the first 10000 prime numbers

Allocate an array where you record all the $p - q$ distances that you found. Print some elementary statistics, for instance: what is the average distance, do the distances increase or decrease with p ?