IISEE lecture for group training (Seismological course)

Fortran programming for beginner seismologists Lesson 6

Lecturer

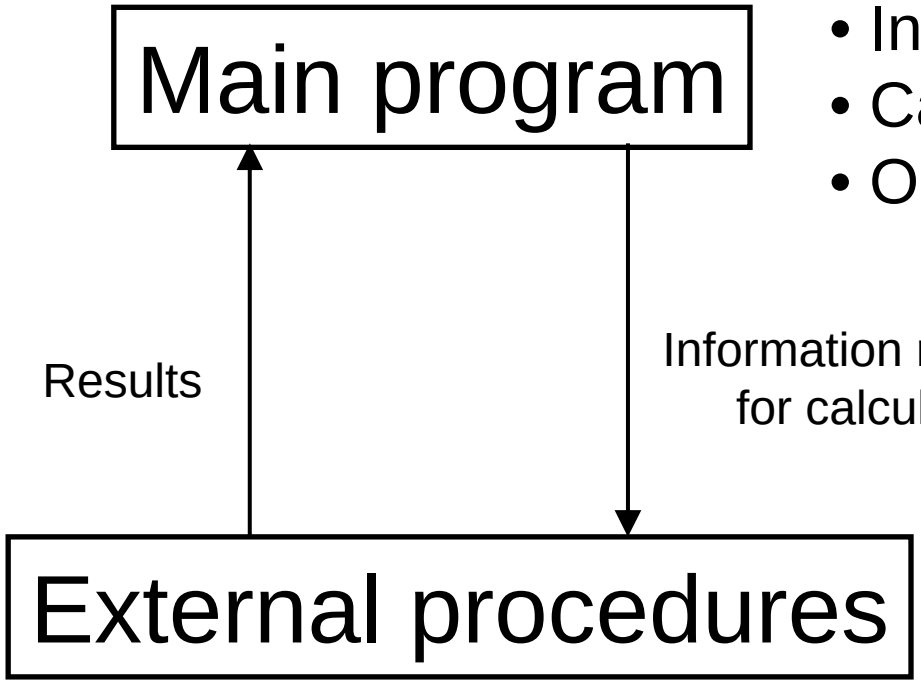
Tatsuhiko Hara

Reference
Introduction to FORTRAN90/95 by S. J. Chapman (New York: McGraw-Hill, 1998)

External procedures

- It is convenient to make an external procedure which has a specific functionality that is frequently used in calculations and analyses.
- In FORTRAN, there are two types of external procedures:
    - Subroutine
    - Function subprogram (or just function)

Main program and external procedures



Main program does:

- Inputs
- Call external procedures
- Outputs

Each external procedure has a certain functionality

Subroutine

A *Subroutine* has a form of:

```
subroutine subr
```

or

```
subroutine subr(arg1 [, arg2, …])
```

where *subr* is a name of a subroutine, *arg1*, *arg2*, … are arguments.

Arguments transfer information from a main program to a subroutine. Also, they transfer information from a subroutine to a main program when computation returns.

Example (1)

- The following is a simple example of subroutine without arguments:

```
subroutine hello
write(*,*) 'Hello.'
end subroutine hello
program main
implicit none
call hello
stop
end program main
```

Example (2)

- The following is a simple example of subroutine with one argument. The type of the argument in the main program should be the same of that in the subroutine.

```
program main
implicit none
real :: a=1.0
call samplesub1(a)
stop
end program main

subroutine samplesub1(b)
implicit none
real :: b
write(*,*) b, 'in samplesub1'
end subroutine samplesub1
```

Example (3)

- This is an example of a wrong program. Note that the type of the argument in the main program is different from that in the subroutine.

```
program main
implicit none
real :: a=1.0
call samplesub1(a)
write(*,*) a, 'in main program'
stop
end program main

subroutine samplesub1(b)
implicit none
integer :: b
write(*,*) b, 'in samplesub1'
end subroutine samplesub1
```

Example (4)

- It is desirable not to use actual arguments as is used in this example:

```
program main
implicit none
real :: a=1.0
call samplesub1(a)
call samplesub1(1.0)
call samplesub1(2.0)
stop
end program main

subroutine samplesub1(b)
implicit none
real :: b
write(*,*) b, 'in samplesub1'
end subroutine samplesub1
```

Example (5)

- The `INTENT` attribute specifies a way how a particular argument is used. In this example, a variable "`b`" in the subroutine `samplesub1` is used both to pass its stored value from the calling program to the subroutine and to return its (modified) value from the subroutine to the calling program.

```
program main
implicit none
real :: a=1.0
call samplesub1(a)
write(*,*) a, 'in main program'
stop
end program main

subroutine samplesub1(b)
implicit none
real, intent(inout) :: b
write(*,*) b, 'in samplesub1'
b = b + 1
end subroutine samplesub1
```

Example (6)

- In this example, `a` is used to pass its stored value from the calling program to the subroutine and `b` is used to return its value from the subroutine to the calling program.

```
program main
implicit none
real :: a=1.0, b
call samplesub1(a,b)
write(*,*) a, b, 'in main program'
stop
end program main

subroutine samplesub1(a,b)
implicit none
real, intent(in) :: a
real, intent(out) :: b
b = a*2.
end subroutine samplesub1
```

Function

A *user-defined Function (*or *Function subprogram)* has a form of:

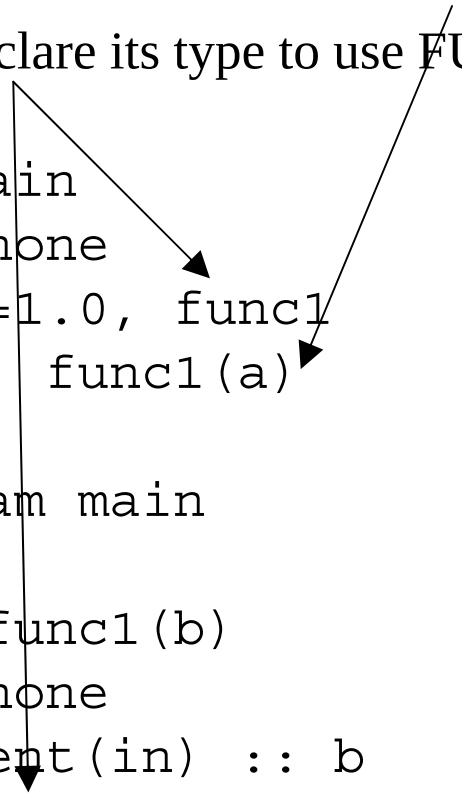
```
function func(arg1 [, arg2, …])
```

where *`func`* is a name of a function, *`arg1`*, *`arg2`*, … are arguments.

Function subprogram is referred as we refer to *intrinsic functions* such as `sin(x)`, `log(x)`.

Example (1)

- The FUNCTION `func1` is referred as `func1(a)`.
- It is necessary to declare its type to use FUNCTION.



```
program main
implicit none
real :: a=1.0, func1
write(*,*) func1(a)
stop
end program main

function func1(b)
implicit none
real, intent(in) :: b
real :: func1
func1 = b*2.
return
end function func1
```

Example (2)

- Compilation of the following program fails, since the type of `func1` is not declared.

```
program main
implicit none
real :: a=1.0, func1
write(*,*) func1(a)
stop
end program main

function func1(b)
implicit none
real, intent(in) :: b
!real :: func1
func1 = b*2.
return
end function func1
```

Internal Procedures (1)

- Subroutines and functions can be contained in a host program unit as internal procedures. Here is an example.

```
program main
implicit none
real :: a=1.0
call add1(a)
write(*,*) a
stop

contains
  subroutine add1(b)
  implicit none
  real, intent(inout) :: b
  b = b + 1
  end subroutine add1

end program main
```

Internal Procedures (2)

- Variables and arrays in a host program unit can be referred from internal procedures without putting them in arguments. Here is an example.

```
program main
  implicit none
  real :: a=2.0, b=5.0, x, y
  write(*,*) 'x?'
  read(*,*) x
  call sub1(x, y)
  write(*,*) a, '*', x ,'+', b, '=', y
  stop
contains
  subroutine sub1(x, y)
    implicit none
    real, intent(in) :: x
    real, intent(out) :: y
    y = a*x + b
  end subroutine sub1
end program main
```

Internal Procedures (3)

- This is another example of a program using a internal procedure.

```
program main
  implicit none
  real :: a=2.0, b=5.0, x !, func1
  write(*,*) 'x?'
  read(*,*) x
  write(*,*) a, '*', x ,'+', b, '=', func1(x)
  stop
contains
  function func1(x)
    implicit none
    real, intent(in) :: x
    real :: func1
    func1 = a*x + b
  end function func1
end program main
```

Dynamic memory allocation: allocatable arrays

- It is possible to allocate memory dynamically at execution time using *allocatable arrays*. Allocatable arrays can be declared using the `ALLOCATABLE` attribute. Here is an example.

```
program main
  implicit none
  integer :: i, m
  integer, allocatable:: a(:)
  write(*,*) 'Array size?'
  read(*,*) m
  allocate(a(1:m))
  do i=1, m
    a(i) = i*i
  end do
  write(*,*) a
  deallocate(a)
  write(*,*) 'Array size?'
  read(*,*) m
  allocate(a(1:m))
  do i=1, m
    a(i) = i*i
  end do
  write(*,*) a
end program main
```

Automatic array

- It is possible to automatically create temporary arrays when a procedure is executing using ALLOCATE attribute and statement.
- These arrays are called automatic arrays.

```
program main
  implicit none
  integer :: m
  real :: s
  write(*,*) 'm?'
  read(*,*) m
  call sub2(s, m)
  write(*,*) m, s
  stop
end program main
```

```
subroutine sub2(x, n)
  real, intent(out) :: x
  integer, intent(in) :: n
  real, allocatable :: temp(:)
  allocate(temp(1:n))
  do i=1, n
    temp(i) = i
  end do
  x = 0.
  do i=1, n
    x = x + sqrt(temp(i))
  end do
end subroutine sub2
```

Explicit-shape dummy array

- It is possible to pass bounds of an array to a subroutine as arguments in the subroutine call and to set the bounds of the corresponding dummy array using those arguments in the subroutine (an *explicit-shape dummy array*).

```
program main
  implicit none
  integer :: i, j, l, m
  integer, allocatable:: iarray(:,:)
  write(*,*) 'Array size?'
  read(*,*) l, m
  allocate(iarray(1:l,1:m))
  call sub2(iarray, l, m)
  do i=1, l
    do j=1, m
      write(*,*) i, j, iarray(i,j), i+j
    end do
  end do
  stop
end program main
```

```
subroutine sub2(karray, l, m)
  integer :: i, j
  integer, intent(in):: l, m
  integer, intent(out):: karray(l,m)
  do i=1, l
    do j=1, m
      karray(i,j) = i+j
    end do
  end do
end subroutine sub2
```

Assumed-shape dummy array

- It is possible to declare array arguments as assumed-shape dummy arrays.
- The actual array shape is set when the procedure is invoked.
- The types and ranks are specified in the calling program unit using `INTERFACE` statements.

```
program main
  implicit none
  interface
    subroutine sub2(karray)
      implicit none
      integer, intent(out):: karray(:,:)
    end subroutine sub2
  end interface
  integer :: i, j
  integer :: l, m
  integer, allocatable:: iarray(:,:)
  write(*,*) 'Array size?'
  read(*,*) l, m
  allocate(iarray(1:l,1:m))
  call sub2(iarray)
  do i=1, l
    do j=1, m
      write(*,*) i, j, iarray(i,j), i+j
    end do
  end do
  stop
end program main
```

```
subroutine sub2(karray)
  integer :: i, j
  integer :: l, m
  integer, intent(out):: karray(:,:)
  l = size(karray,1)
  m = size(karray,2)
  write(*,*) 'l, m:', l, m
  do i=1, l
    do j=1, m
      karray(i,j) = i+j
    end do
  end do
end subroutine sub2
```

`size(array,dim)`

`Size` is an array inquiry function to return the extent of `array` for a particular dimension, `dim`. When `dim` is not present, the total number of elements is returned.

Array in internal procedure

- It is possible to set a shape of an array in an internal procedure using integer variables in the host program unit.

```
program main
  implicit none
  integer :: n, i
  real, allocatable :: x(:), y(:)
  write(*,*) 'size?'
  read(*,*) n
  allocate(x(n), y(n))
  write(*,*) 'Elements for x?'
  read(*,*) (x(i), i=1, n)
  write(*,*) 'Elements for y?'
  read(*,*) (y(i), i=1, n)
  write(*,*) func1(x,y)
  stop
  contains
    function func1(x,y)
      implicit none
      integer :: i
      real, intent(in):: x(n), y(n)
      real :: func1
      func1 = 0.
      do i=1, n
        func1 = func1 + x(i)/y(i)
        write(*,*) i, x(i)/y(i), func1
      end do
    end function func1
end program main
```