4. CONDITIONAL STATEMENTS (IF STRUCTURE)

Example : Derive $\sum_{k=1}^{n} k$ using Fortran (n ≤ 100).

```
$ ./a.out

‘ Type positive integer n (n <= 100)’
```

-> This program returns different messages depending on input values.

case 1

When you type 10,

→ ’Sum total = 55’

case 2

When you type -5,
→ ’Negative integer !!’

case 3

When you type 120,
→ ’Greater than 100 !!’

```fortran
program ex4_1
 implicit none
 integer :: n, k, sum = 0
!
 write(*, *) 'Input positive integer (<= 100)'
 read(*, *) n
 if(n <= 0) stop ': Negative integer!!'
 if(n > 100) stop ': Greater than 100!!'
!
 do k = 1, n
   sum = sum + k
 enddo
 write(*, *) 'Sum total =', sum
 stop
end program ex4_1
```

EXERCISE 4-1
Compile and run the above program (save the file as ex4_1.f90).

The sum of odd integers from 1 to 100.

```
program ex4_2
 implicit none
 integer :: i, sum_odd = 0
!
 do i = 1, 100
   if(mod(i, 2) == 1) then
     sum_odd = sum_odd + i
   endif
 enddo
 write(*, *) 'Sum total =', sum_odd
 stop
end program ex4_2
```

EXERCISE 4-2
Compile and run the above program (save the file as ex4_2.f90).

4.1 Basics of conditional statements

```
if ( logical formula A ) then
     value = value + 1
else if ( logical formula B ) then
     value = value – 1
else
     value = -1 * value
endif
```

value = value + 1	if *value* satisfies the *logical formula A*, perform this procedure
value = value – 1	if *value* does not satisfy the *logical formula A* but satisfies the *formula B*, perform this procedure
value = -1 * value	if *value* does not satisfy any logical formulas, perform this procedure

4.2 Relational operators

		old-fashioned way
`if(a > b)`	$a > b$	`if(a.gt.b)`
`if(a >= b)`	$a \geq b$	`if(a.ge.b)`
`if(a < b)`	$a < b$	`if(a.lt.b)`
`if(a <= b)`	$a \leq b$	`if(a.le.b)`
`if(a == b)`	$a = b$	`if(a.eq.b)`
`if(a /= b)`	$a \neq b$	`if(a.ne.b)`
`if(a>b .and. a<c)`	$b < a < c$	`if(a.gt.b.and.a.lt.c)`
`if(a>b .and. a>c)`	$a > b,c$	`if(a.gt.b.and.a.gt.c)`
`if(a>b .or. a>c)`	$a > b$ or $a > c$	`if(a.gt.b.or.a.gt.c)`

```
False: if (b < a < c), if (a.gt.b.gt.c)
```

EXERCISE 4-3
Derive parameter *a* after reflection of the following IF statements, provided initial parameter *a* of -5, 0 or 12, respectively.

```
a = initial param.
if ( a < 10 ) then
 a = 0
endif
```

a = ?

```
a = initial param
if ( a < 0 ) then
 a = -1 * a + 2
else if ( a > 0 ) then
 a = a + 2
endif
```

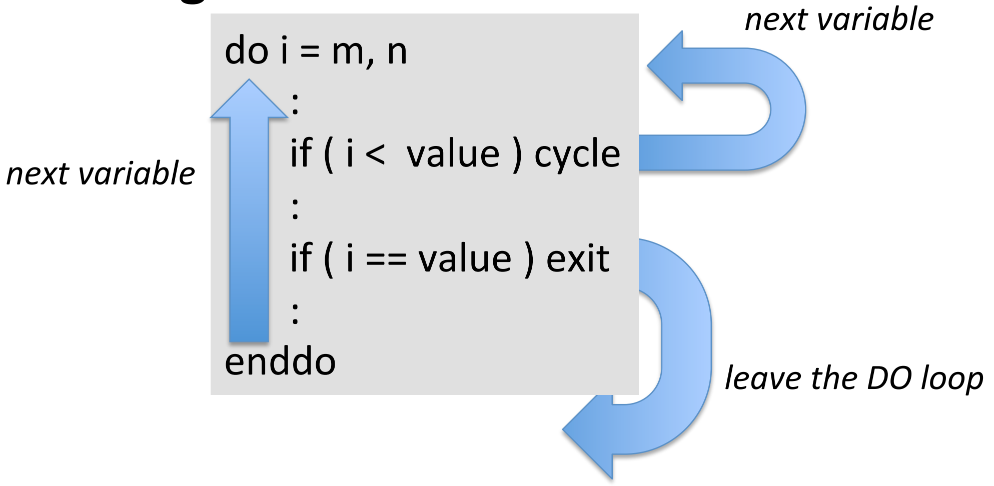
a = ?

```
a = initial param
if ( a > 0 ) then
 a = -1 * a
else
 a = 0
endif
```

a = ?

EXERCISE 4-4
Modify the program ex4_2.f90 to find not only the sum of odd integers (*sum_odd*) but also the sum of even integers (*sum_even*) from 1 to 100 (save the file as ex4_4.f90).

exit, cycle and goto statements



```
do i = m, n
     :
     if ( i <  value ) cycle
     :
     if ( i == value ) exit
     :
enddo
```

cycle:
don't do any more of the statements below this one in the DO loop and cycle back to the beginning of the loop to start the pass with the next value of the index
exit:
leave the DO loop and execute the next statement after the end of the loop

EXERCISE 4-5

Find the minimum integer

n that satisfies $\sum_{k=1}^{n} k \geq 100$

(save the file as ex4_5.f90).

```
program ex4_5
 implicit none
 integer k, sum
 sum = 0
 do k = 1, 100
   sum = sum + k
   if              exit
 enddo
 write(*, *) 'n, sum =', k, sum
 stop
end program ex4_5
```

EXERCISE 4-6

Output 4, 5, 6 and 7 on the screen (save the file as ex4_6.f90).

```
program ex4_6
 implicit none
 integer i
 do i = 1, 10
   if                   cycle
 write(*, *) i
 enddo
 stop
end program ex4_6
```

```
program ex4_7
 implicit none
 integer :: n, k, sum=0
 write(*, *) 'Input positive integer.'
 read(*, *) n
!
  if (n == 0) then
   goto 1
  else if(n < 0) then
    stop ': negative integer !!'
  else
    do k = 1, n
      sum = sum + k
    enddo
  endif
1 continue
  write(*, *) ‘Sum total =', sum
stop
end program ex4_7
```

EXERCISE 4-7
Compile and run the above program that displays the sum of integer from 0 to arbitrary positive integer (save the file as ex4_7.f90).

Supplemental:

Fortran90

```
program over200
 implicit none
 integer i, x
 x = 1
 do i = 1, 10
   x = 2 * x + i
   if (x > 200) exit
   if (x < 100) cycle
   write(*, *) ' x >= 100'
 enddo
 write(*, *) 'x =', x
 stop
end program over200
```

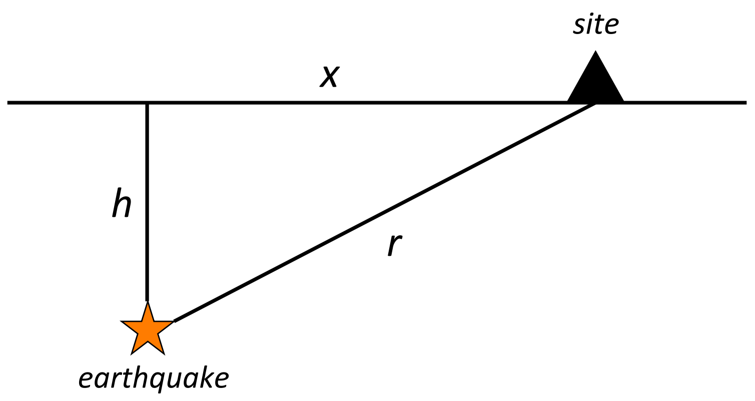
over200.f90

FORTRAN77

```
      program over200
      implicit none
      integer i, x
      x = 1
      do 100 i = 1, 10
       x = 2 * x + i
       if (x.gt.200) go to 200
       if (x.lt.100) go to 100
      write(*, *) ' x >= 100’
100   continue
200   write(*, *) ' x = ',x
      stop
      end program over200
```

over200.f, over200.for

PROJECT II : Seismic Wave Travel Time



Seismic wave speed
V_P: 6.0 km/s
V_S: 3.5 km/s

(a) Write a program ttime.f90 to calculate travel time of *P* and *S* waves for a given focal depth (km: 0 < h < 30) and epicentral distance (km: x > 0) in a half-space with *P*- and *S*-wave speeds (V_P and V_S) of 6.0 km/s and 3.5 km/s, respectively.

(b) Find arrival times of P and S waves at the observation site when h = 15 km and x = 30 km.

Travel times:

$$t_P = \frac{r}{V_P} = \frac{\sqrt{\boxed{\quad ? \quad}}}{V_P}$$

$$t_S = \frac{r}{V_S} = \frac{\sqrt{\boxed{\quad ? \quad}}}{V_S}$$

5. FILES AND FORMATTED INPUT/OUTPUT

In the previous chapter, we have developed a program to calculate a travel time for a given pair of a focal depth and an epicentral distance.

It is necessary to calculate travel times for a given focal depth for set of epicentral distances to obtain a travel time table.

In this chapter, we are going to extend our program to make a travel time table.

EXERCISE 5-1
Modify the program ttime.f90 to output x, t_P and t_S for an earthquake (depth = 20 km) at intervals of 1 km ($1 \le x \le 50$ km) and save the file as ex5_1.f90.

```
$ gfortran –o ex5_1.exe ex5_1.f90
$ ./ex5_1.exe

  1.00000000    3.33749747    5.72142410
  2.00000000    3.34995842    5.74278593
  3.00000000    3.37062478    5.77821398
  4.00000000    3.39934635    5.82745075
  5.00000000    3.43592143    5.89015102
      :              :             :
  46.0000000    8.35995770    14.3313551
  47.0000000    8.51306248    14.5938206
  48.0000000    8.66666698    14.8571424
  49.0000000    8.82074547    15.1212778
  50.0000000    8.97527409    15.3861847
```

5.1 Output data to a file

(a) Redirection (in terminal)

```
$ ./ex5_1.exe  >  tt1.dat
```

arbitrary file name

EXERCISE 5-2
(a) Create an output file 'tt1.dat' using the redirection option in the xterm and confirm it using the *ls* command.
(b) Type as "cat tt1.dat" in the xterm to see the content.

(b) OPEN and CLOSE statements

```
program ex5_3
 implicit none
 integer i
 real x, tp, ts
 real :: vp=6., vs=3.5, h=20.
!
 open(10, file = 'tt2.dat')
 do i = 1, 50
 x = real(i)
 tp = sqrt(h**2 + x**2) / vp
 ts = sqrt(h**2 + x**2) / vs
 write(10, *) x, tp, ts
 enddo
 close(10)
 stop
end program ex5_3
```

<u>write(*A*, *)</u>

A: device number

"*"
→ screen

"integer"
→ a file assigned in the open statement

Output file "tt2.dat" will be created.

5.2 Input parameters from a data file

```
program ex5_4
 implicit none
 integer i
 real h, x, vp, vs, tp, ts
 open(10, file = 'param1.dat')
 open(11, file = 'tt3.dat’)
 read(10, *) vp
 read(10, *) vs
 read(10, *) h
 do i = 1, 50
   x = real(i)
   tp = sqrt(h**2 + x**2) / vp
   ts = sqrt(h**2 + x**2) / vs
   write(11, *) x, tp, ts
 enddo
 close(10)
 close(11)
 stop
end program ex5_4
```

param1.dat (existing file)

```
6.0
3.5
20.0
```

read(*A*, *)
A: device number

“*” : from keyboard

integer
→ a file assigned in the open statement

Input file “param1.dat” will be read.
Output file “tt3.dat” will be created.

```
program ex5_5
 implicit none
 integer i
 real h, x, vp, vs, tp, ts
 open(10, file = 'param2.dat')
 open(11, file = 'tt4.dat')
!
 read(10, *) vp, vs, h
 do i = 1, 50
   x = real(i)
   tp = sqrt(h**2 + x**2) / vp
   ts = sqrt(h**2 + x**2) / vs
   write(11, *) x, tp, ts
 enddo
 close(10)
 close(11)
 stop
end program ex5_5
```

param2.dat (existing file)

```
6.0  3.5   20.0
```

Input file “param2.dat” will be read.
Output file “tt4.dat” will be created.

EXERCISE 5-3
Compile and run the program ex5_3.f90 (three slides back) to check the new output file “tt2.dat” is created.

EXERCISE 5-4
Create the input file “param1.dat” and compile and run the program ex5_4.f90 (two slides back) to check the new output file “tt3.dat” is created.

EXERCISE 5-5
Create the input file “param2.dat” as shown in the previous slide and compile and run the program ex5_5.f90 to check the new output file “tt4.dat” is created.

EXERCISE 5-6
Check the contents of the output files “tt1.dat”, “tt2.dat”, “tt3.dat” and “tt4.dat” are the same.

5.3 Formatted Input/Output

```
1.00000000    3.33749747    5.72142410
2.00000000    3.34995842    5.74278593
3.00000000    3.37062478    5.77821398
4.00000000    3.39934635    5.82745075
5.00000000    3.43592143    5.89015102
    :              :             :
46.0000000    8.35995770    14.3313551
47.0000000    8.51306248    14.5938206
48.0000000    8.66666698    14.8571424
49.0000000    8.82074547    15.1212778
50.0000000    8.97527409    15.3861847
```

```
  1.00  3.34  5.72
  2.00  3.35  5.74
  3.00  3.37  5.78
  4.00  3.40  5.83
  5.00  3.44  5.89
   :     :     :
 46.00  8.36 14.33
 47.00  8.51 14.59
 48.00  8.67 14.86
 49.00  8.82 15.12
 50.00  8.96 15.39
```

```
write(*,*) real(i), tp, ts
```

```
write(*,'(3f6.2)') real(i), tp, ts
```

EXERCISE 5-7
Modify the program ex5_3.f90 as the following to see the content of the output file “tt5.dat” (save the file as ex5_7.f90).

```
program ex5_7
implicit none
integer i
real x, r, tp, ts
real :: vp=6., vs=3.5, h=20.
!
open(10, file = 'tt5.dat')
do i = 1, 50
  x=real(i)
  tp = sqrt(h**2 + x**2) / vp
  ts = sqrt(h**2 + x**2) / vs
  write(10, '(3f6.2)') x, tp, ts
enddo
close(10)
stop
end program ex5_7
```

Basics of format statements

write(*, '(   )')

integer (123)

i6 (I6)	→	_ _ _ 1 2 3
i6.4 (I6.4)	→	_ _ 0 1 2 3

real (1.23)

f6.2 (F6.2)	→	_ _ 1 . 2 3
f6.0 (F6.0)	→	_ _ _ _ 1 .
e10.2	→	_ _ 0 . 1 2 E + 0 1
e10.2e1	→	_ _ _ 0 . 1 2 E + 1

character (ABC)

a (A)	→	A B C _ _ _ _ _ _ _
a10 (A10)	→	_ _ _ _ _ _ _ A B C

Ex.

3f4.1	→	_ 2 . 1 | _ 3 . 2 | _ 4 . 5
f4.1, 2x, f6.1	→	_ 3 . 1 | _ _ | _ 4 3 2 . 5

5.4 Character statements

The character statement is a type declaration statement for the character data type. (len = <len>) following character specifies its length.

```
program main
 character(len = 10) :: ss1 = '1234567890'
 character(len = 10) :: ss2 = 'abcdefghij'
 write(*, *) ss1
 write(*, *) ss2
 write(*, *) ss1(1:3)
 write(*, *) ss2(1:3)
 write(*, *) ss1(4:6)
 write(*, *) ss2(4:6)
 write(*, *) ss1(7:10)
 write(*, *) ss2(7:10)
end program main
```

```
$ ./a.exe

1234567890
abcdefghij
123
abc
456
def
7890
ghij
```

5.5 Gnuplot

Now we have obtained the file which contains the travel time table (tt5.dat). Let's plot this table using *gnuplot*.

<GNUPLOT>

・Gnuplot is a portable command-line driven interactive data and function plotting utility for UNIX, Linux, MS Windows family, Mac, etc (http://www.gnuplot.info/).

・Gnuplot is included in software packages of Cygwin.

・You can start this software by the following command:

```
$ gnuplot
```

Plotting travel times

EXERCISE 5-8
Try the following commands.

```
gnuplot> plot 'tt5.dat’
```
→ *P-wave travel times are plotted with “+”*
```
gnuplot> plot 'tt5.dat' using 1:2
```
→ *P-wave travel times are plotted with “+”*
```
gnuplot> plot 'tt5.dat' using 1:2 with lines
```
→ *P-wave travel times are plotted with line*
```
gnuplot> plot 'tt5.dat' using 1:3 with lines
```
→ *S-wave travel times are plotted with line*
```
gnuplot> plot 'tt5.dat','tt5.dat' using 1:3
```
→ *P- and S-wave travel times are plotted with “+”* in different colors

Gnuplot commands

Title
```
gnuplot> set title “Travel time”
```

Axis
```
gnuplot> set xlabel “Epicentral distance (km)”
gnuplot> set ylabel “Travel time (s)”
```

Legend
```
gnuplot> plot 'tt5.dat' title “P-wave”
gnuplot> plot 'tt5.dat' title “P-wave”, \
>  'tt5.dat' using 1:3 title “S-wave”
```

$\uparrow$ *Automatically displayed, not to necessary to type.*

Command file

It is a tiring job to type the commands in the previous slides each time.

Let's create a file “plotcom” which contains the following commands:

```
set title “Travel time”
set xlabel “Epicentral distance (km)”
set ylabel “Time (sec)”
plot 'tt5.dat' title “P-wave”, \
'tt5.dat' using 1:3 title “S-wave”
```

How to use a command file?

You can “load” a file “plotcom” in the interactive mode.

EXERCISE 5-10
Try the following command after starting gnuplot:

```
   gnuplot> load 'plotcom'
```

You can use a command file as an argument of gnuplot.

EXERCISE 5-11
Add the following two lines at the beginning of “plotcom.”

```
   set terminal postscript
   set output “tt.ps”
```

Then, try the following in the xterm:

```
   $ gnuplot plotcom
   $ ls
```

-> You will find that the postscript file “tt.ps” is created.

6. ARRAYS

In numerical calculations, vectors and matrices are often used.

In Fortran, we use “array” to store values of vectors, matrices and multi dimensional quantities.

Fortran has an advantage in treating “array”.

6.1 Runk-1 array “Inner product of vectors”

Consider position vectors ***u*** and ***v*** on a two-dimensional surface. The inner product (dot product) of ***u*** and ***v*** is expressed as follows.

$$p = \mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{2} u_i v_i = u_1 v_1 + u_2 v_2$$

In Fortran, the vector components are expressed as,

$$u_1, u_2 \rightarrow \text{u(1), u(2)}$$

$$v_1, v_2 \rightarrow \text{v(1), v(2),}$$

where u(n) and v(n) are array elements and the numbers in the brackets “1, 2” are subscripts.

```
program ex6_1
 implicit none
 integer i
 real u(2), v(2), p
 u(1) = 1.2
 u(2) = 3.4
 v(1) = 4.1
 v(2) = 2.6
 !
 p = 0.0
 do i = 1, 2
   p = p + u(i) * v(i)
 enddo
 write(*, *) 'Inner product =', p
 stop
end program ex6_1
```

p = 0.

i=1   p ← <u>p</u> + u(1) * v(1)

i=2   p ← <u>p</u> + u(2) * v(2)

p = (u(1) × v(1)) + (u(2) × v(2))

EXERCISE 6-1
Compile and run the above program (save the file as ex6_1.f90).

6.2 Array declaration

The array **x** declared by

`real x(5)`
`real x(1:5)`
`real, dimension(5) :: x`
`real, dimension(1:5) :: x`

have five *real* elements; `x(1), x(2), x(3), x(4), and x(5).`

`real x(3:5)`
`real, dimension(3:5) :: x`

have three *real* elements; `x(3), x(4), and x(5).`

`real x(-1:2)`
`real, dimension(-1:2) :: x`

have four *real* elements; `x(-1), x(0), x(1), and x(2).`

`real, dimension(-10:10) :: y`   has $\boxed{\quad\quad}$ *real* elements

```
real x(4), y(4)
real, dimension(4) :: x, y
```
have eight *real* elements; `x(1), x(2), x(3), x(4),`
`y(1), y(2), y(3), and y(4).`

```
integer a(3:5)
integer, dimension(3:5) :: a
```
have three *integer* elements; `a(3), a(4), and a(5).`

`integer b(0:4)` has $\boxed{}$ *integer* elements

```
integer, parameter :: nsize = 5
real x(nsize)
```
has five *real* elements; `x(1), x(2), x(3), x(4), and x(5).`

6.3 Value assignment

(a) Direct assignment

```
real u(3)
u(1) = 1.2
u(2) = 3.4
u(3) = 2.7
```

(b) Initialization in type declaration statements

array constants

```
real :: u(3) = (/ 1.2, 3.4, 2.7 /)
real :: u(1:3) = (/ 1.2, 3.4, 2.7 /)
real, dimension(3) :: u = (/ 1.2, 3.4, 2.7 /)
real, dimension(1:3) :: u = (/ 1.2, 3.4, 2.7 /)
```
The number of elements in the constant must match the number of elements!

```
integer, parameter :: n = 3
real, dimension(n) :: u = (/ 1.2, 3.4, 2.7 /)
```

(c) DO loop

```
do i = 1, 3
  x(i) = i
enddo
```

x(1) = 1
x(2) = 2
x(3) = 3

(d) Read statement

```
do i = 1, 3
   read(*, *) x(i)
enddo
```

type manually

```
% ./a.exe
2
3
4
```

x(1) = 2
x(2) = 3
x(3) = 4

(e) Implied DO loop

```
read(*, *) (x(i), i=1,3)
```

(f) File

```
read(10, *) (x(i), i=1,3)
```

file

```
10
20
30
```

$\Rightarrow$ x(1) = 10
x(2) = 20
x(3) = 30

Example:

```
integer m(6)
 m(1:3) = 0
 m(4:6) = (/ 2, 3, 4 /)
 m(1:3) = (/ (i, i = 1, 5, 2) /)
 m(:) = 3
```

m(1) = 0, m(2) = 0, m(3) = 0
m(4) = 2, m(5) = 3, m(6) = 4
m(1) = 1, m(2) = 3, m(3) = 5
m(1) = 3, m(2) = 3, … m(6) = 3

```
integer :: a(1:4) = 0
```

a(1) = 0, a(2) = 0, a(3) = 0, a(4) = 0

```
integer :: b(1:2) = (/2, 3/)
```

b(1) = 2, b(2) = 3

```
integer :: i, c(1:4) = (/ (i, i = 11, 5, -2) /)
```

increment

c(1) = 11, c(2) = 9, c(3) = 7, c(4) = 5

Difference between *a* and *a*(n)

```
program test_noarray
 implicit none
 integer i
 real a
!
 do i = 1, 5
   a = real(i)
   write(*, *) 'a =', a
 enddo
 stop
end program test_noarray
```

```
$ ./a.exe
 a = 1.000000
 a = 2.000000
 a = 3.000000
 a = 4.000000
 a = 5.000000
```

a → 1.0
a → 2.0
a → 3.0
a → 4.0
a → 5.0

```
program test_array
 implicit none
 integer i
 integer, parameter :: n = 5
 real a(n)
 do i = 1, n
  a(i) = real(i)
  write(*, *) 'a =', a(i)
 enddo
 stop
end program test_array
```

```
$ ./a.exe
 a = 1.000000
 a = 2.000000
 a = 3.000000
 a = 4.000000
 a = 5.000000
```

a(1) → 1.0
a(2) → 2.0
a(3) → 3.0
a(4) → 4.0
a(5) → 5.0

```
program test_noarray
 implicit none
 integer i
 real a
!
 do i = 1, 5
   a = real(i)
 enddo
 write(*, *) a
 stop
end program test_noarray
```

```
$ ./a.exe
   5.000000
```

only one value is stored

```
program test_array
 implicit none
 integer i
 integer, parameter :: n = 5
 real a(n)
 do i = 1, n
   a(i) = real(i)
 enddo
 write(*, *) a(1)
 write(*, *) a(2)
 write(*, *) a(3)
 write(*, *) a(4)
 write(*, *) a(5)
 stop
end program test_array
```

```
$ ./a.exe
   1.000000
   2.000000
   3.000000
   4.000000
   5.000000
```

five values can be stored !

6.4 Data output

Declaration

```
integer a(1:3)
a(1:3) = (/ 3, 4, 7 /)
```

Write statements

```
do i = 1, 3
 write(*, *) a(i)
enddo
```

```
3
4
7
```

$\Updownarrow$

implied DO loop

```
write(*, *) (a(i), i = 1, 3)
```

```
3    4    7
```

$\Updownarrow$

```
write(*, *) a(1:3)
```

```
3    4    7
```

```
program ex6_2
 implicit none
 integer i, a(4)
 a(1:4) = (/ 1, 2, 3, 4 /)
!
 do i = 1, 4
   write(*, *) a(i)
 enddo
!
 write(*, *) (a(i), i = 1, 4)
!
 write(*, *) a(1:4)
 stop
end program ex6_2
```

```
integer i, a(4)
a(1:4) = (/ 1,2,3,4 /)

integer i
integer :: a(4) = (/ 1,2,3,4 /)

integer i
integer :: a(1:4) = (/ 1,2,3,4 /)

integer i
integer, parameter :: n =4
integer :: a(n) = (/ 1,2,3,4 /)
```

EXERCISE 6-2
Compile and run the above program (save the file as ex6_2.f90).

```
program ex6_3_1
 implicit none
 integer i, a(4)
 a(1:4) = (/ 1, 2, 3, 4 /)
 write(*, *) (a(i), i = 1, 4)
 do i = 2, 4
   a(i) = a(i-1)
 enddo
 write(*, *) (a(i), i = 1, 4)
 stop
end program ex6_3_1
```

```
program ex6_3_2
 implicit none
 integer i, a(4)
 a(1:4) = (/ 1, 2, 3, 4 /)
 write(*, *) (a(i), i = 1, 4)
 a(2:4) = a(1:3)
 write(*, *) (a(i), i = 1, 4)
 stop
end program ex6_3_2
```

EXERCISE 6-3
Compile and run the above programs to see the difference between the output values (save the files as ex6_3_1.f90 and ex6_3_2.f90, respectively).

6.5 Whole array operations

It is possible to perform array operations in Fortran90.
The operation is applied on an element-by-element basis.

Statements in codes	Operations for each element
`a + b`	`a(i) + b(i)`
`a - b`	`a(i) – b(i)`
`a * b`	`a(i) * b(i)`
`a / b`	`a(i) / b(i)`
`a**n`	`a(i)**n`

Both a and b are one-dimensional arrays. n is a variable.

EXERCISE 6-4
Compile and run the following program (save the file as ex6_4.f90).

```
program ex6_4
 implicit none
 integer ia(4), ib(4), i
 ia(1:4) = (/ 1, 2, 3, 4 /)
 do i = 1, 4
   ib(i) = 10 + 2 * ia(i)
 enddo
 write(*, *) (ib(i), i = 1, 4)
 stop
end program ex6_4
```

EXERCISE 6-5
Compile and run the following program (save the file as ex6_5.f90).

```
program ex6_5
 implicit none
 integer, parameter :: nsize = 3
 real a, x(nsize)
 write(*, *) 'Coefficient a?'
 read(*, *) a
 write(*, *) 'array x?'
 read(*, *) x
 x = x + a
 write(*, *) 'array x (= x + a):', x
 x = x * a
 write(*, *) 'array x (= x * a):', x
 stop
end program ex6_5
```

EXERCISE 6-6

Compile and run the following program (save the file as ex6_6.f90).

```
program ex6_6
 implicit none
 integer, parameter :: nsize = 3
 real, dimension(nsize) :: x = (/1., 2., 3./), y, z
 integer i
 write(*, *) 'array x:', x
 write(*, *) 'array y?'
 read(*, *) y
 z = x + y
 write(*, *) 'array z (= x + y):', z
 do i=1, nsize
   z(i) = x(i) + y(i)
 enddo
 write(*, *) 'array z (= x + y):', (z(i), i=1, nsize)
 stop
end program ex6_6
```

EXERCISE 6-7

Compile and run the following program (save the file as ex6_7.f90).

```
program ex6_7
 implicit none
 integer, parameter :: nsize = 10
 integer i
 real, dimension(nsize) :: x=(/(i, i=1,10)/), &
 y=(/(2*i, i=2, 20, 2)/), z
 write(*, *) 'x(i)'
 write(*, *) x
 write(*, *) 'y(i)'
 write(*, *) y
 write(*, *) '-x(i)'
 write(*, *) -x
 write(*, *) 'sqrt(y(i))'
 z = sqrt(y)
 write(*, *) z
 stop
end program ex6_7
```

6.6 Intrinsic function: DOT_PRODUCT

EXERCISE 6-8
Modify the program ex6_1.f90 as below to confirm the same output (save the file as ex6_8.f90).

```
program ex6_8
 implicit none
 integer i
 real u(2), v(2), p
 u(1) = 1.2
 u(2) = 3.4
 v(1) = 4.1
 v(2) = 2.6
!
 p = dot_product(u, v)
 write(*, *) 'InnProduct =', p
 stop
end program ex6_8
```

```
do i = 1, 2
   p = p + u(i) * v(i)
enddo
```

$$\Updownarrow$$

```
p = dot_product(u, v)
```

6.7 Subset of arrays

An array subset is specified with a subscript triplet (vector subscript):

`subscript_1 : subscript_2 : stride`

where

`subscript_1`
is the first subscript to be included in the subset

`subscript_2`
is the last subscript to be included in the subset

`stride`
is the increment for the subscript

EXERCISE 6-9
Compile and run the following program (save the file as ex6_9.f90).

```
program ex6_9
 implicit none
 integer, parameter :: nsize = 10
 integer i
 integer :: x(nsize) = (/(i, i = 1, 10)/)
 write(* ,*) x(:)
 write(* ,*) x(4:6)
 write(* ,*) x(3:)
 write(* ,*) x(:5)
 write(* ,*) x(2:8:2)
 write(* ,*) x(6:3:-1)
 write(* ,*) x(9:3:-2)
 write(* ,*) x(9:2:-2)
 write(* ,*) x(::3)
 write(* ,*) x(10:10)
 stop
end program ex6_9
```

6.8 Runk-2 array

“Representation of a matrix”

The element of a matrix *A* in the *i* th row and *j* th column $a_{i,j}$ is expressed as `a(i,j)` in Fortran.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} \mathrm{a}(1,1) & \mathrm{a}(1,2) \\ \mathrm{a}(2,1) & \mathrm{a}(2,2) \end{pmatrix}$$

EXERCISE 6-10,6-11,6-12
Compile and run the programs on the following pages.

```
program ex6_10_1
 implicit none
 integer i, j
 real a(2, 2)
 a(1, 1:2) = (/ 1.2, 3.4 /)
 a(2, 1:2) = (/ 5.6, 7.8 /)
 do i = 1, 2
   write(*, *) (a(i, j), j = 1, 2)
 enddo
 stop
end program ex6_10_1
```

$$\begin{pmatrix} a(1,1) & a(1,2) \\ a(2,1) & a(2,2) \end{pmatrix} = \begin{pmatrix} 1.2 & 3.4 \\ 5.6 & 7.8 \end{pmatrix}$$

```
$ ./a.exe
1.200000      3.400000
5.600000      7.800000
```

```
program ex6_10_2
 implicit none
 integer i, j
 real a(2, 2)
 a(1, 1:2) = (/ 1.2, 3.4 /)
 a(2, 1:2) = (/ 5.6, 7.8 /)
 write(*, *) a(:, :)
 stop
end program ex6_10_2
```

```
$ ./a.exe
1.200000   5.600000   3.400000   7.800000
```

* double DO loop

```
program ex6_10_3
 implicit none
 integer i, j
 real a(2, 2)
 a(1, 1:2) = (/ 1.2, 3.4 /)
 a(2, 1:2) = (/ 5.6, 7.8 /)
 do i = 1, 2
   do j = 1, 2
     write(*, *) a(i, j)
   enddo
 enddo
 stop
end program ex6_10_3
```

$$\begin{pmatrix} a(1,1) & a(1,2) \\ a(2,1) & a(2,2) \end{pmatrix} = \begin{pmatrix} 1.2 & 3.4 \\ 5.6 & 7.8 \end{pmatrix}$$

```
$ ./a.exe
1.200000
3.400000
5.600000
7.800000
```

```
program ex6_11_1
 implicit none
 integer i, j
 real a(2, 2)
 a(1:2, 1:2) = 0.0
 do i=1, 2
   write(*, *) (a(i, j), j = 1, 2)
 enddo
 stop
end program ex6_11_1
```

```
program ex6_11_2
 implicit none
 integer i, j
 real a(2, 2)
 a(:, :) = 0.0
 do i=1, 2
   write(*, *) (a(i, j), j = 1, 2)
 enddo
 stop
end program ex6_11_2
```

$$\begin{pmatrix} a(1,1) & a(1,2) \\ a(2,1) & a(2,2) \end{pmatrix} = \begin{pmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \end{pmatrix}$$

Read and write elements of rank-2 array

```
program ex6_12
 implicit none
 integer, parameter :: msize=3, nsize=2
 integer i, j
 real x(msize, nsize)
 do i = 1, msize
   write(*, *) 'Input', nsize, 'real numbers for row', i
   read(*, *) (x(i,j), j=1, nsize)
 enddo
 do i = 1, msize
   do j = 1, nsize
     write(*, *) 'row', i, 'column', j, x(i,j)
   enddo
 enddo
 write(*, '(a20, 2i10)') 'column',(j, j=1, nsize)
 do i = 1, msize
   write(*, '(a10,i10,2f10.2)') 'row', i, (x(i,j), j=1, nsize)
 enddo
 stop
end program ex6_12
```

6.9 Matrix calculation using rank-2 arrays

Statements in codes	Operations for each element
`a + b`	`a(i,j) + b(i,j)`
`a - b`	`a(i,j) – b(i,j)`
`a * b`	`a(i,j) * b(i,j)`
`a / b`	`a(i,j) / b(i,j)`
`a**n`	`a(i,j)**n`
`alpha*a`	`alpha * a(i,j)`

Both A and B are 2-rank arrays. n and `alpha` are variables.

6.10 Matrix Multiplication

a. Multiplying a matrix by a single number

(scalar multiplication)

Ex.

$$alpha \times \begin{pmatrix} a_{11} \; a_{12} \\ a_{21} \; a_{22} \end{pmatrix} = \begin{pmatrix} alpha \times a_{11} \; alpha \times a_{12} \\ alpha \times a_{21} \; alpha \times a_{22} \end{pmatrix}$$

EXERCISE 6-13
Compile and run the following program (save the file as ex6_13.f90).

$$2 \times \begin{pmatrix} 4 & 0 \\ 1 & -9 \end{pmatrix} = \begin{pmatrix} 8 & 0 \\ 2 & -18 \end{pmatrix}$$

```
program ex6_13
 implicit none
 integer i, j, a(2, 2)
 a(1, 1:2) = (/ 4, 0 /)
 a(2, 1:2) = (/ 1, -9 /)
 write(*, *) (2 * a(1, j), j = 1, 2)
 write(*, *) (2 * a(2, j), j = 1, 2)
 stop
end program ex6_13
```

b. Multiplying a matrix by a vector

$$y_i = \sum_{j=1}^{n} a_{i,j} b_j$$

Ex.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} a_{11} \times b_1 + a_{12} \times b_2 \\ a_{21} \times b_1 + a_{22} \times b_2 \\ a_{31} \times b_1 + a_{32} \times b_2 \end{pmatrix}$$

EXERCISE 6-14
Compile and run the following program (save the file as ex6_14.f90).

```
program ex6_14
 implicit none
 integer i, j, a(3, 2), b(2), y(3)
 a(1, 1:2) = (/ 2, 5 /)
 a(2, 1:2) = (/ 3, 6 /)
 a(3, 1:2) = (/ 4, 7 /)
 b(1:2) = (/ 3, 1 /)
 do i = 1, 3
   y(i) = 0
   do j = 1, 2
     y(i) = y(i) + a(i, j) * b(j)
   enddo
   write(*, *) y(i)
 enddo
 stop
end program ex6_14
```

$$\begin{pmatrix} 2 & 5 \\ 3 & 6 \\ 4 & 7 \end{pmatrix} \times \begin{pmatrix} 3 \\ 1 \end{pmatrix} = \begin{pmatrix} 11 \\ 15 \\ 19 \end{pmatrix}$$

c. Multiplying matrices

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$

Ex.

$$\begin{pmatrix} a_{11} \; a_{12} a_{13} \\ a_{21} \; a_{22} a_{23} \end{pmatrix} \times \begin{pmatrix} b_{11} \; b_{12} \\ b_{21} \; b_{22} \\ b_{31} \; b_{32} \end{pmatrix}$$

$$= \begin{pmatrix} a_{11} \cdot b_{11} + a_{12} \cdot b_{21} + a_{13} \cdot b_{31} & a_{11} \cdot b_{12} + a_{12} \cdot b_{22} + a_{13} \cdot b_{32} \\ a_{21} \cdot b_{11} + a_{22} \cdot b_{21} + a_{23} \cdot b_{31} & a_{21} \cdot b_{12} + a_{22} \cdot b_{22} + a_{23} \cdot b_{32} \end{pmatrix}$$

```
program ex6_15
 implicit none
 integer i, j, k, a(2, 3), b(3, 2), c(2, 2)
 a(1, 1:3) = (/ 1, 2, 3 /)
 a(2, 1:3) = (/ 4, 5, 6 /)
 b(1, 1:2) = (/ 7, 8 /)
 b(2, 1:2) = (/ 9, 10 /)
 b(3, 1:2) = (/ 11, 12 /)
 do j = 1, 2
   do i = 1, 2
     c(i, j) = 0
     do k = 1, 3
       c(i,j) = c(i, j) + a(i, k) * b(k, j)
     enddo
   enddo
 enddo
 write(*, *) c(1,1:2)
 write(*, *) c(2,1:2)
 stop
end program ex6_15
```

6.11 Intrinsic function: MATMUL

Programs ex6_14.f90 can be written as follows, by using the intrinsic function *matmul*.

```
program ex6_16
 implicit none
 integer i, j, a(3, 2), b(2), y(3)
 a(1, 1:2) = (/ 2, 5 /)
 a(2, 1:2) = (/ 3, 6 /)
 a(3, 1:2) = (/ 4, 7 /)
 b(1:2) = (/ 3, 1 /)
 y(:) = matmul(a(:, :), b(:))
 write(*, *) y(1)
 write(*, *) y(2)
 write(*, *) y(3)
 stop
end program ex6_16
```

```
y = matmul(a, b)
```

Programs ex6_15.f90 can be written as follows, by using the intrinsic function *matmul*.

```
program ex6_17
 implicit none
 integer i, j, k, a(2, 3), b(3, 2), c(2, 2)
 a(1, 1:3) = (/ 1, 2, 3 /)
 a(2, 1:3) = (/ 4, 5, 6 /)
 b(1, 1:2) = (/ 7, 8 /)
 b(2, 1:2) = (/ 9, 10 /)
 b(3, 1:2) = (/ 11, 12 /)
 c(:, :) = matmul(a, b)
 write(*, *) c(1,1:2)
 write(*, *) c(2,1:2)
 stop
end program ex6_17
```

```
c = matmul(a, b)
```

```
c(:, :) = matmul(a(:,:), b(:, :))
```

6.12 Dynamic memory allocation

It is possible to allocate memory dynamically at execution time using allocatable arrays. Allocatable arrays can be declared using the ALLOCATABLE attribute.

```
program ex6_18
implicit none
 integer i, m
 integer, allocatable :: a(:)
 write(*, *) 'Input array size.'
 read(*, *) m
 allocate(a(m))
 do i = 1, m
   a(i) = i * i
 enddo
 write(*, *) a
 deallocate(a)
 write(*, *) 'Input array size.'
 read(*, *) m
 allocate(a(1:m))
 do i = 1, m
   a(i) = i * i
 enddo
 write(*, *) a
 stop
end program ex6_18
```

```
program ex6_19
 implicit none
 integer i, j, m, n
 integer, allocatable :: a(:, :), b(:), y(:)
 write(*, *) 'Input i, j (0 < i, j < 10)'
 read(*, *) i, j
 allocate (a(i, j), b(j), y(i))
 do m = 1, i
   do n = 1, j
     write(*, '(a8i1a1i1a1)') "Input a(",m,",",n,")"
     read(*, *) a(m, n)
   enddo
 enddo
 do m = 1, j
   write(*, '(a8i1a1)') "Input b(",m,")"
   read(*, *) b(m)
 enddo
 y(:) = matmul(a, b)
 write(*, *) y(1:i)
 stop
end program ex6_19
```

EXERCISE:

Make a program to perform the following calculation using rank-2 arrays:

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} 7 & 9 \\ 8 & 10 \end{bmatrix}$$

Answer:

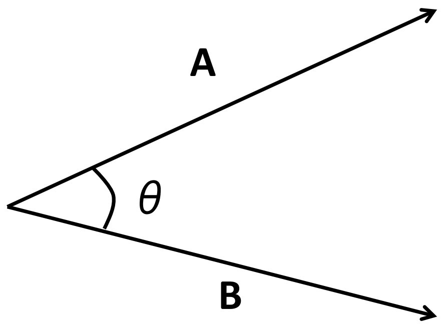
$$\begin{bmatrix} 39 & 49 \\ 54 & 68 \\ 69 & 87 \end{bmatrix}$$

The value of the array elements should be assigned in the program.

EXERCISE:

Make a program to calculate the angle (in degrees) for two vectors **a** (2,1) and **b** (1,3) in the x-y coordinate system.

(Hint 1)



$$\cos\theta = \frac{\mathbf{AB}}{|\mathbf{A}||\mathbf{B}|}$$

You can use the intrinsic functions “acos” to derive an angle.

EXERCISE:

Make a program to calculate the angle (in degrees) for two vectors **a** (2,1) and **b** (1,3) in the x-y coordinate system.

(Hint 2)
You can use the intrinsic functions “sqrt” and “dot_product” to calculate the following problems;

$$\mathbf{ab} = \text{dot\_product(a,b)}$$

$$|\mathbf{a}| = \text{sqrt(dot\_product(a,a))} \quad .$$

Answer : 45°

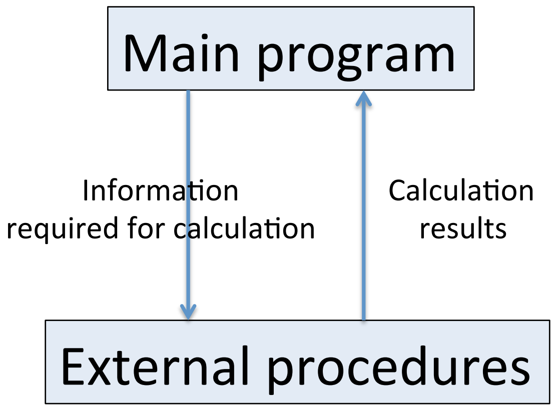
7. FUNCTIONS AND SUBROUTINES

We have learned ways of writing a program that consists of declaration, calculation and termination sections.

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)

7.1 External procedures



Main program does:
・Inputs
・Call external procedures
・Outputs

Each external procedure has a certain functionality

Example:

```
program  sample1
implicit none
integer n, ...
real, dimension(n) :: a, b, c, d
```

calculation A for a(i)

calculation A for b(i)

calculation A for c(i)

calculation A for d(i)

```
write(*, *) ...
end program sample1
```

```
program  sample2
implicit none
integer n, ...
real, dimension(n) :: a, b, c
call subl(a, n)
call subl(b, n)
call subl(c, n)
call subl(d, n)
write(*, *) ...
end program sample2
!
subroutine subl(x, max)
implicit none
integer max
real, dimension(max) :: x
```

Calculation A

```
end subroutine
```

7.2 Subroutine

A *subroutine* has a form of:

`subroutine` *`subr`*

or

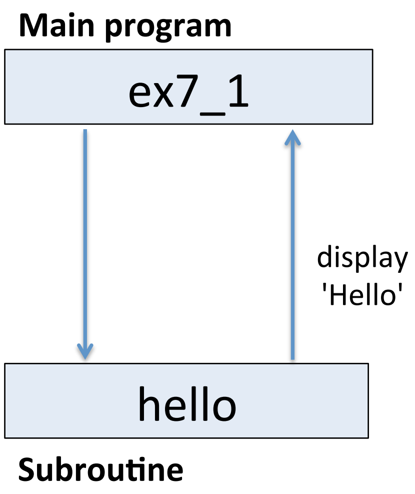
`subroutine` *`subr(arg1 [, arg2, ...])`*

where *subr* is a name of subroutine (arbitrary name) and *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.

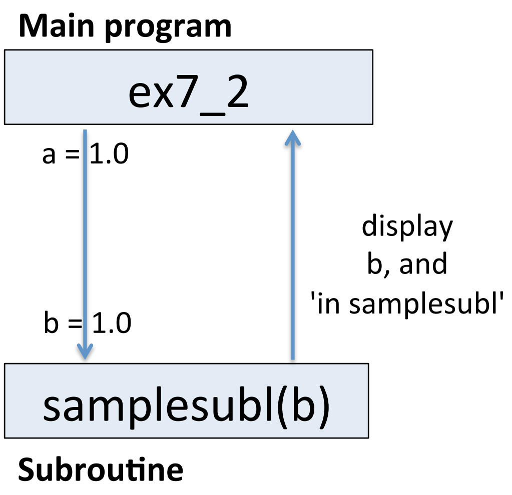
The following is a simple example of subroutine without arguments:

```
program ex7_1
  implicit none
  call hello
  stop
end program ex7_1
!
subroutine hello
  write(*, *) 'Hello.'
end subroutine hello
```



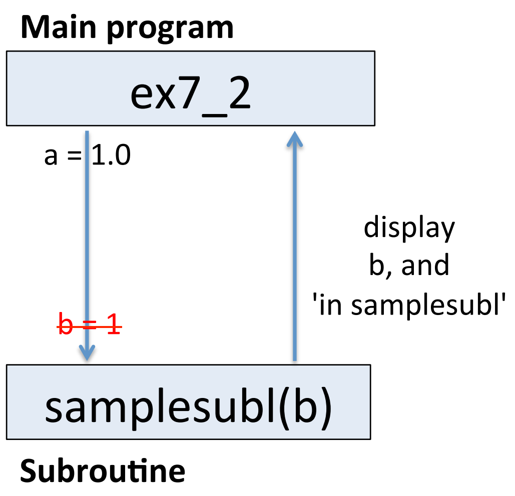
The following is a simple example of subroutine with one argument.
The type of the argument in the main program should be the same of that in the subroutine.

```
program ex7_2
  implicit none
  real :: a = 1.0
  call samplesubl(a)
  stop
end program ex7_2
!
subroutine samplesubl(b)
  implicit none
  real b
  write(*, *) b, 'in samplesubl'
end subroutine samplesubl
```



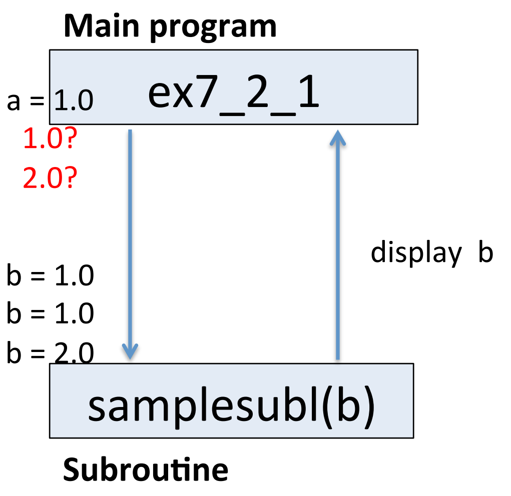
This is an example of a *wrong program*.
Note that the type of the argument in the main program is different from that in the subroutine

```
program ex7_2
 implicit none
 real :: a = 1.0
 call samplesubl(a)
 stop
end program ex7_2
!
subroutine samplesubl(b)
 implicit none
 integer b
 write(*, *) b, 'in samplesubl'
end subroutine samplesubl
```



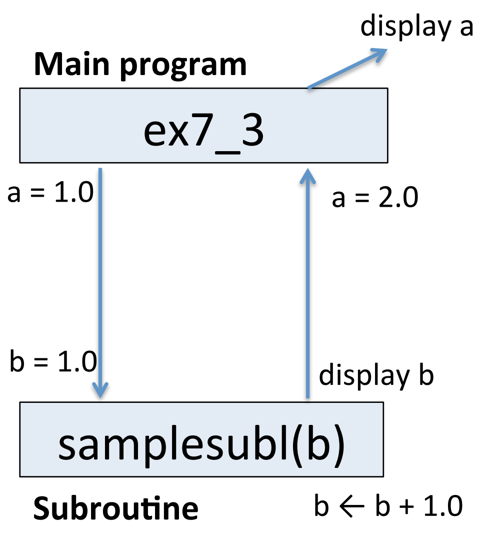
This is an example of an *undesired program*.
It is desirable not to use actual arguments as is used in this example:

```
program ex7_2_1
 implicit none
 real :: a = 1.0
 call samplesubl(a)
 call samplesubl(1.0)
 call samplesubl(2.0)
 stop
end program ex7_2_1
!
subroutine samplesubl(b)
 implicit none
 real b
 write(*, *) b, 'in samplesubl'
end subroutine samplesubl
```



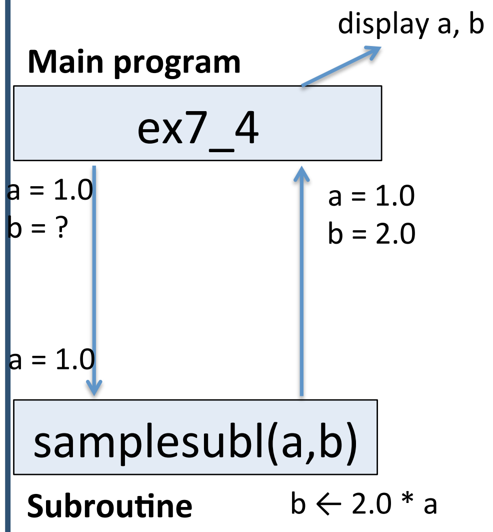
The INTENT attribute specifies a way how a particular argument is used.
In this example, a variable "**b**" in the subroutine **samplesub1** is used both to pass its stored value from the calling program to the subroutine and to return its (modified) value from the subroutine to the calling program.

```fortran
program ex7_3
  implicit none
  real :: a = 1.0
  call samplesubl(a)
  write(*, *) a, 'in main program'
  stop
end program ex7_3
!
subroutine samplesubl(b)
  implicit none
  real, intent(inout) :: b
  write(*, *) b, 'in samplesubl'
  b = b + 1.0
end subroutine samplesubl
```



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.

```fortran
program ex7_4
  implicit none
  real :: a = 1.0, b
  call samplesubl(a, b)
  write(*, *) a, b, 'in main program'
  stop
end program ex7_4
!
subroutine samplesubl(a, b)
  implicit none
  real, intent(in) :: a
  real, intent(out) :: b
  b = 2.0 * a
end subroutine samplesubl
```



7.3 Function

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

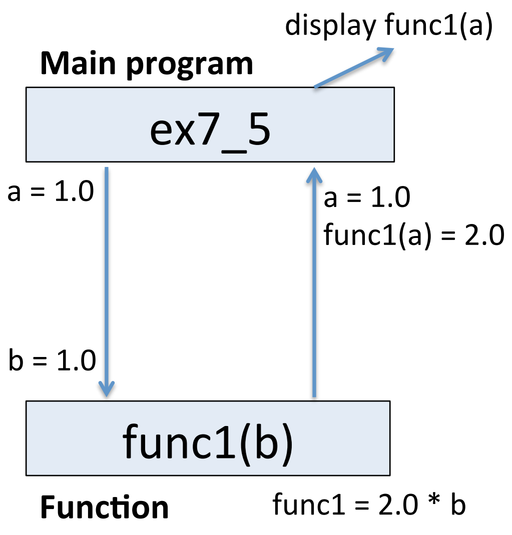
```
function func(arg1 [, arg2, ...])
```

where *func* is a name of function (arbitrary name) and *arg1*, *arg2*, ... are arguments.

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

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

```
program ex7_5
 implicit none
 real :: a = 1.0, func1
 write(*, *) func1(a)
 stop
end program ex7_5
!
function func1(b)
 implicit none
 real, intent(in) :: b
 real :: func1
 func1 = 2.0 * b
 return
end function func1
```



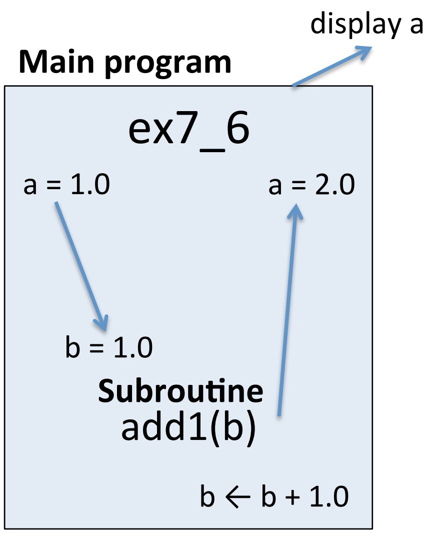
Compilation of the following program fails, since the type of func1 is not declared.

```
program ex7_5
 implicit none
 real :: a = 1.0, func1
 write(*, *) func1(a)
 stop
end program ex7_5
!
function func1(b)
 implicit none
 real, intent(in) :: b
 !real :: func1
 func1 = 2.0 * b
 return
end function func1
```

7.4 Internal procedures

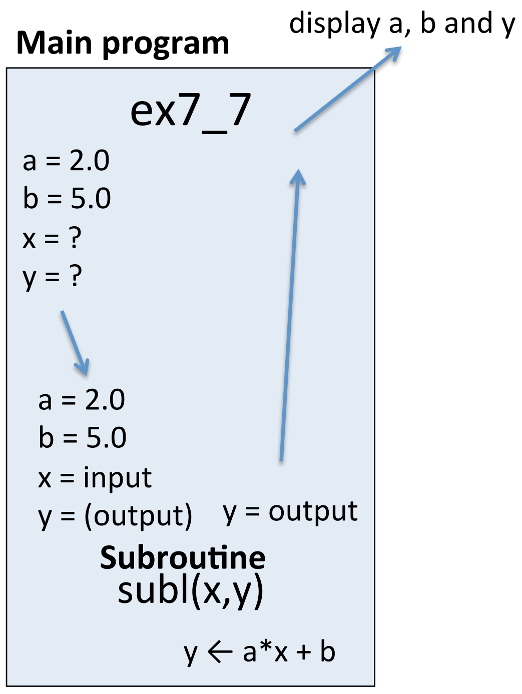
Subroutine and functions can be contained in a host program unit as internal procedures (no 'implicit none' in the subroutine/function).

```
program ex7_6
 implicit none
 real :: a = 1.0
 call add1(a)
 write(*, *) a
 stop
contains
 subroutine add1(b)
  real, intent(inout) :: b
 b = b + 1.0
 end subroutine add1
end program ex7_6
```



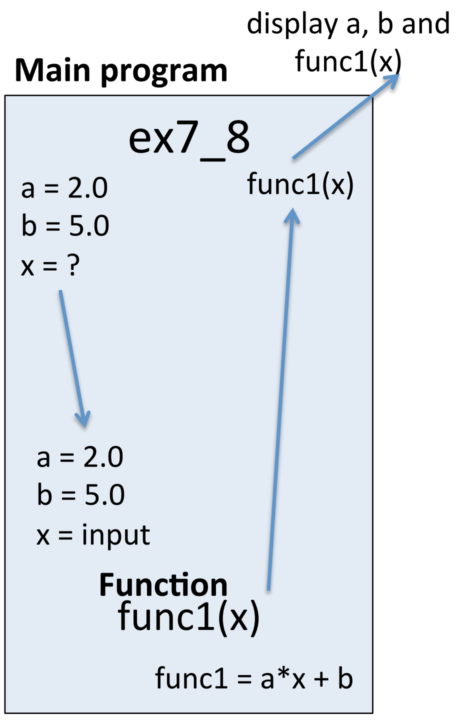
Values and arrays in a host program unit can be referred from internal procedures without putting them in arguments.

```
program ex7_7
 implicit none
 real :: a = 2.0, b = 5.0, x, y
 write(*, *) 'x?'
 read(*, *) x
 call subl(x, y)
 write(*, *) a,'*',x,'+',b,'=',y
 stop
contains
 subroutine subl(x, y)
 real, intent(in) :: x
 real, intent(out) :: y
 y = a * x + b
 end subroutine subl
end program ex7_7
```



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

```
program ex7_8
 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)
 real, intent(in) :: x
 real func1
 func1 = a * x + b
 end function func1
end program ex7_8
```



7.5 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 ex7_9
 implicit none
 integer m
 real s
 write(*, *) 'm?'
 read(*, *) m
 call sub2(s, m)
 write(*, *) m, s
 stop
end program ex7_9
```

```
subroutine sub2(x, n)
 implicit none
 integer i
 integer, intent(in) :: n
 real, intent(out) :: x
 real, allocatable :: temp(:)
 allocate(temp(1:n))
 do i = 1, n
  temp(i) = i
 enddo
 x = 0.0
 do i = 1, n
    x = x + sqrt(temp(i))
 enddo
end subroutine sub2
```

n = input integer
s = from calculated x
in the subroutine

$$x = \sum_{i=1}^{n} \sqrt{real(i)}$$

7.6 Dummy array
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 ex7_10
 implicit none
 integer i, j, l, m
 integer, allocatable:: iarray(:,:)
 write(*, *) 'Input 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
   enddo
 enddo
 stop
end program ex7_10
```

```
subroutine sub2(karray,l,m)
 implicit none
 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
   enddo
 enddo
end subroutine sub2
```

Assumed-shape dummy array

```
program ex7_11
 implicit none
 interface
  subroutine sub2(karray)
   implicit none
   integer, intent(out) :: karray(:,:)
  end subroutine sub2
 end interface
 integer i, j, l, m
 integer, allocatable :: iarray(:,:)
 write(*, *) 'Input 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
   enddo
 enddo
 stop
end program ex7_11
```

```
subroutine sub2(karray)
 implicit none
 integer i, j, 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
   enddo
 enddo
end subroutine sub2
```

EXERCISE:

Derive the surface area (*S*) and volume (*V*) of a sphere with arbitrary given radius *r*. Make a main program to input the value of the radius *r* (from the keyboard) and to output the results, and a subroutine program to calculate *S* and *V*.

$$S = 4\pi r^2$$

$$V = \frac{4}{3}\pi r^3$$

Please fill in the blank spaces in the following program.

```
program sphere
 implicit none
 real
 write(*, *) 'Radius of a sphere?'
 read(*, *) r
 call sub_sphere(      )
 write(*, *) 'Surface area    =', s
 write(*, *) 'Volume          =', v
 stop
end program sphere
!
subroutine
 implicit none
 real, intent(   ) :: x
 real, intent(   ) :: sa, vo
 real pi
 pi =
 sa =
 vo =
end subroutine
```

Supplemental: Modules

A module is a separately compiled program unit which is used for the following purposes.

- To share definitions and initial values among program files.
- To define subroutines and functions as internal procedures with declarations and initializations of variables and arrays.
- To define user-defined operations.
- To define user-defined assignments.

Here is an example of programs with a module for sharing definitions and initial values among program units.

common_variables.f90

```
module common_variables
   integer, parameter :: imax = 10
end module common_variables
```

main.f90

```
program main
use common_variables
implicit none
real a(imax)
integer i
do i = 1, imax
  a(i) = i * i
  write(*, *) i, a(i)
enddo
end program main
```

Here is an example of programs with a module to use a subrutine.

sub2.f90

```
subroutine sub2(karray)
 implicit none
 integer i, j, 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
   enddo
 enddo
 end subroutine sub2
```

sub2_module.f90

```
module sub2_module
 interface
   subroutine sub2(karray)
    implicit none
    integer, intent(out) :: karray(:,:)
   end subroutine sub2
 end interface
end module sub2_module
```

main.f90

```
program main
 use sub2_module
 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)
 do i = 1, l
   do j = 1, m
     write(*, *) i,j,iarray(i,j),i+j
   enddo
 enddo
 stop
end program main
```

Compilation(Example)

(a) Single program

```
gfortran program.f90
gfortran -o prog program.f90
```

$\rightarrow$ an execution file 'a.exe' is created

$\rightarrow$ an execution file 'prog' is created

(b) Main program, a subroutine program and a module

```
gfortran sub_module.f90 sub.f90 main.f90
```

an execution file 'a.exe' is created

```
gfortran -o prog sub_module.f90 sub.f90 main.f90
```

an execution file 'prog' is created

```
gfortran -c sub_module.f90      an object file 'sub_module.o' is created
gfortran -c sub.f90             an object file 'sub.o' is created
gfortran -c main.f90            an object file 'main.o' is created
gfortran sub_module.o sub.o main.o
(gfortran -o prog sub_module.o sub.o main.o)
```

Advanced Examples 1: Newton's method

Newton's method is a method for finding successively better approximations to the roots of a real-valued function.

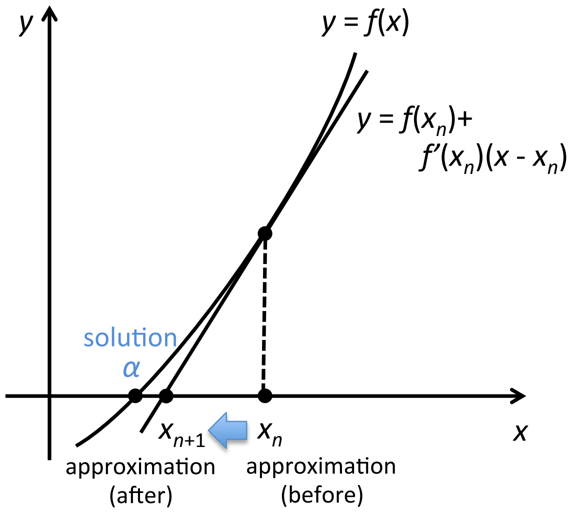
Consider a function *f* defined over the reals *x* and its derivative *f'*. Provided the solution of the function is α [$f(\alpha) = 0$] and its approximate solution is x_n, the function $f(\alpha)$ can be written using the Taylor's series.

$$f(\alpha) \approx f(x_n) + f'(\alpha - x_n) \qquad (1)$$

Since $f(\alpha) = 0$, we can obtain a better approximation solution x_{n+1} as follows;

$$\alpha \approx x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad (2)$$

Repeating this procedure, we can obtain a sufficiently accurate solution that satisfies $f(x) \approx 0$.



EXERCISE: Write a program to obtain $\sqrt{a}$, for real *a* (*a* > 0).

Provided $f(x) = x^2 - a$, we can obtain the following relationship using eq.(2);

$$x_{n+1} = x_n - \frac{x_n^2 - a}{2x_n} \ .$$

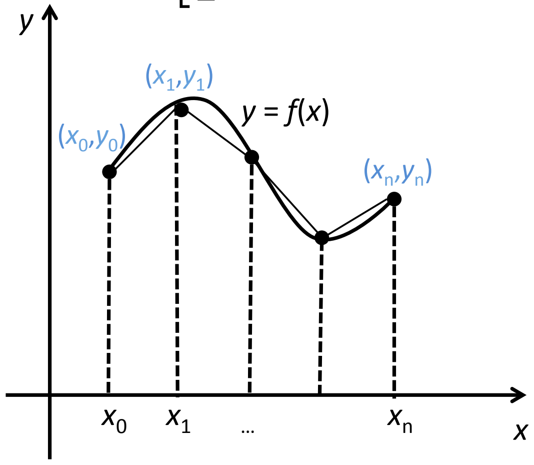
If we assume the initial value of x_n as *a* and perform iterative calculations, we can obtain a sufficiently accurate solution for *x* (i.e.,$\sqrt{a}$).

```
program newton_method
implicit none
real    :: x1, x2, a, er, er0 = 1.0e-6   ! er0: acceptable value for error
integer :: k, kmax = 100                 ! kmax: number of iterations
 write(*, *) 'Input a :'
 read(*, *) a
 if (a <= 0.0) stop 'a <= 0.0'
 x1 = a                                  ! x1: initial value of xn
 do k = 1, kmax
   x2 = x1 - (x1**2 - a) / (2.0 * x1)    ! x2: upgraded solution
   er = abs(x2 - x1)                     ! er: difference between x1 and x2
   if (er < er0) exit                    ! Termination of iterations
   x1 = x2                               ! Upgrade initial value
 enddo
 write(*, *) 'solution, k, er =', x2, k, er
end program newton_method
```

Advanced Examples 2: Numerical integral using trapezoid formula

The definite integral of a function *f* is approximately derived by adding trapezoids that are formed by dividing the integral interval into *n* equal parts, as drawn in figure. As a result, the following formula is derived.

$$s = \frac{1}{2}\Delta x \sum_{k=0}^{n-1}\left(y_k - y_{k+1}\right) = \left[\frac{1}{2}\left(y_0 + y_n\right) + y_1 + y_2 + \cdots y_{n-1}\right]\Delta x$$



EXERCISE: Write a program to obtain an approximation of the definite integral $S = 6\int_0^1 x\left(1-x\right)dx$.

```
program trapezoid_formula
implicit none
real    dx, x, y, s
integer i, n
 write(*, *) ‘Input n :’
 read(*, *) n
 if (n < 1) stop ‘n < 1’
 dx = 1.0 / real(n)
 s = 0.0
 do i = 0, n
    x = dx * real(i)
    y = x * (1.0 – x)
    if (i == 0 .or. i == n) then
       s = s + 0.5 * y
    else
       s = s + y
    endif
 enddo
 s = 6.0 * s * dx
 write(*, *) ‘s =’, s
 stop
end program trapezoid_formula
```