Chapter 7
Programming with Subroutines

For a larger project, usually we will divide the project into sub-projects to simplify the design. For computer program, it is also useful to divide the program into sub-programs.

Subroutines in Fortran let the programmer use structured designs.

Form of subroutine:

```
subroutine heading
specification part
execution part
END SUBROUTINE statement
```

Subroutine heading can be:

```
SUBROUTINE subroutine-name(formal-argument-list)
```

or

```
RECURSIVE SUBROUTINE subroutine-name
(formal-argument-list)
```

To reference a subroutine, a CALL statement is used:

```
CALL subroutine-name(actual-argument-list)
```

A subroutine-name is different from a function-name.

A subroutine may have no arguments.

```
PROGRAM Angles_1
IMPLICIT NONE
INTEGER :: NumDegreeS, NumMinites, NumSecounds
CHARACTER(1) :: Response
Do
 WRITE(*, '(1X, A)', ADVANCE = "NO") &
   "Enter degree, minutes, and secondes: "
 READ *, NumDegreeS, NumMinites, NumSecounds
 CALL PrintDegrees(NumDegree, NumMinites, NumSecounds)
  WRITE(*, '(1X, A)', ADVANCE = "NO") &
   "More angles (Y or N )?"
  READ *, Response
  IF (Response /= "Y") EXIT
END DO
CONTAINS
```

```
SUBROUTINE printDegrees(Degrees, Minutes, Seconds)
 INTEGER, INTENT(IN) :: Degrees, Minutes, Seconds
 PRINT 10, Degrees, Minutes, Seconds, &
  REAL(Degrees)+REAL(Minutes)/60.0+REAL(Seconds)/3600.0
 10 FORMAT(1x, I3, "degrees", I3, "minutes",I3, "seconds" &
   / 1x, "is equivalent to" / 1x, F7.3,"degrees")
END SUBROUTINE PrintDegrees
END PROGRAM Angles_1
```

A subroutine can return values.

Example: a subroutine that converting the polar coordinates (r, θ) of a point P to rectangular coordinates (x, y).

$$x = r\cos\theta,\ y = r\sin\theta$$

In this subroutine, the input values are r and θ, the output values are x and y.

```
PROGRAM Polar_to_Rectangular
IMPLICIT NONE
......
CALL Convert_to_Rectangular(RCoord, TCoord, XCood, YCood)
PRINT *, "Rectangular coordinates:", XCood, YCood
......
CONTAINS
 SUBROUTINE Convert_to_Rectangular(R, Theta, X, Y)
 REAL, INTENT(IN):: R, Theta
 REAL, INTENT(OUT) :: X, Y
 X = R * COS(Theta)
 Y = R * SIN(Theta)
 END SUBROUTINE Convert_to_Rectangular
END PROGRAM Polar_to_Rectangular
```

There are three different attributes:

`INTENT(IN)`

`INTENT(OUT)`

`INTENT(INOUT)`

The `OUT` and `INOUT` arguments must be variables. An `INOUT` argument should bring in some values to the subroutine and this value may be changed within the subroutine. Then the variable will return the updated value.

The following subroutine exchanges the values of two integer variables.

```
SUBROUTINE  Swap(a, b)
   IMPLICIT  NONE
   INTEGER, INTENT(INOUT) :: a, b
   INTEGER :: i
   i = a
   a = b
   b = i
END SUBROUTINE  Swap
```

The Heron's formula for computing triangle area with side lengths a, b and c is

$$\sqrt{s(s-a)(s-b)(s-c)}$$

where s is the half of the perimeter length:

$$s = \frac{a+b+c}{2}$$

In order for a, b and c to form a triangle, two conditions must be satisfied.

- $a, b, c > 0$
- $a + b > c, a + c > b, b + c > a$

We use two methods to implement the problem.

- First method uses two functions to calculate the area. One function is used to check if the input values may form a triangle. The other function is used to compute the value.
- Second method uses a subroutine to calculate the area.

In general, any function can be implemented as a subroutine. But a subroutine may not be written as a function.

Usually, a function is simpler than a subroutine.

Sometimes we need to use subprograms as arguments.

For example, we want a general subprogram to calculate

$$\int_a^b f(x)dx$$

where $f(x)$ can be any integrable function. The definite integral used the trapezoidal method.

What kind arguments we need for the subroutine?

$a, b, f()$, and number of subintervals. So in this case we need the function as an argument.

A function or subroutine can used as an argument if:

- it is a module subprogram
- it is an external subprogram
- it is an intrinsic subprogram

An internal subprogram cannot be used as an argument.

No `INTENT` attribute used for a subprogram argument.

```
SUBROUTINE Integrate(F, A, B, N)
 REAL, INTENT(IN) :: A,B
 INTEGER, INTENT(IN) :: N
 REAL :: F, DeltaX, X, Y, Sum
 INTEGER :: I
 DeltaX = (B - A)/ REAL(N)
 X = A
 Sum = 0.0
  DO I = 1, N - 1
     X = X + DeltaX
     Y = F(X)
     Sum = Sum + Y
  END DO
  Sum = DeltaX * ((F(A) + F(B)) / 2.0 + Sum)
... ...
END SUBROUTINE Integrate
```

```
MODULE Integrand_Function
CONTAINS
  FUNCTION Integrand(X)
   REAL :: Integrand
   REAL, INTENT(IN) :: X
   Integrand = EXP(X**2)
  END FUNCTION Integrand
END MODULE Integrand_Function
```

```
PROGRAM Definite_Integral_2
USE Integrand_Function
IMPLICIT NONE
 ... ...
 ... ...
CALL Integrate(Integrand, A, B, Number_of_Subinternals)
CONTAINS
!---------------------------------------
!put subroutine Integrate(F, A, B, N) here
!---------------------------------------
END PROGRAM Definite_Integral_2
```

Use intrinsic subprogram as arguments

- `INTRINSIC list of intrinsic subprogram names`
- `type, INTRINSIC ::  function-name`

```
PROGRAM Definite_Integral_4
IMPLICIT NONE
REAL :: A, B
INTRINSIC SIN
... ...
CALL Integrate(SIN, A, B, Number_of_Subintervals)
CONTAINS
 ... ...
END PROGRAM Definite_Integral_4
```

Use external subprogram as arguments

- `type, EXTERNAL :: function-name`
- `EXTERNAL list of external subprogram names`
- use `INTERFACE` block

Using interface block is recommended.

Interface block and `EXTERNAL` specifiers cannot both be used, nor can interface blocks be used for module subprograms or intrinsic subprograms.

```
PROGRAM Definite_Integral_5
IMPLICIT NONE
REAL :: A, B
INTERFACE
  FUNCTION Integrand(X)
   REAL :: Integrand
   REAL, INTENT(IN) :: X
  END FUNCTION Integrand
END INTERFACE
INTEGER :: Number_of_Subintervals
WRITE(*, '(1X, A)', ADVANCE = "NO") &
 ......
CALL Integrate(Integrand, A, B, Number_of_Subintervals)
CONTAINS
... ...
END PROGRAM Definite_Integral_5
```

Fortran 90 provides some intrinsic subroutines

- Time routines
- Bit copy routines
- Random number routines:

```
RANDOM_NUMBER(HARVEST)
!produces a real number between 0 and 1


RANDOM_SEED  ! initialize the generator
```

Random number generator is useful for simulation program and other programs.

When the enriched uranium fuel of a nuclear reactor is burned, high-energy neutrons are produced. Some of these are retained in the reactor core, but most of them escape. Since this radiation is dangerous, the reactor must be shielded. We want to use a program to simulate neutrons entering the shield and to determine what percentage of them get through it.

To simulate neutrons, we assume that neutrons entering the shield follow random paths by moving forward, backward, left or right with equal likelihood, in jumps of one unit. Also assume that losses of energy occur only when there is a change of direction.

Specification:

- Input: Thickness of the shield, limit of the number of direction changes, number of neutrons.
- Output: Percentage of neutrons that reach the outside.
- Use random numbers 1, 2, 3 or 4 to denote the movement forward, backward, to the left or to the right, respectively.

Variables: `Thickness`, `DirectionChangeLimit`, `NewDirection`, `oldDirection`, `NumDirectionChanges`, `Forward`, `NumNeutrons`, `NumEscaped`

Algorithm: pp 162.

```
DO
 CALL RANDOM_NUMBER(RandonReal)
 NewDirection = 1 + INT(4 * RandomReal)
 IF (NewDirection /= oldDirection) THEN
  NumDirectionChange = NumDirectionChanges + 1
  OldDirection = NewDirection
 END IF
 IF (NewDirection == 1) THEN
  Forward = Forward + 1
 ELSE IF (NewDirection == 2) THEN
  Forward = Forward - 1
 END IF
 IF ((Forward >= Thickness) .OR. (Forward <= 0) .OR. &
   (NumDirectionChanges >= DirectionChangeLimit)) EXIT
END DO
```

Recursive subroutine

The Towers of Hanoi problem is to solve the puzzle: move the disks from the left peg to the right peg according to:

1. when a disk is moved, it must be placed on one of the three pegs
2. Only one disk may be moved at a time, and it must be the top disk on one of the pegs
3. A larger disk may never be placed on top of a smaller one

To solve the Towers of Hanoi problem, we can first consider the case of one or two disks and then use a recursive method.

The inductive steps:

1. Move the topmost $N-1$ disks from Peg A to Peg B, using C as an auxiliary peg.
2. Move the large disks remaining on Peg A to Peg C.
3. Move the $N-1$ disks from Peg B to Peg C, using Peg A as an auxiliary peg.

```
RECURSIVE SUBROUTINE Move(N, StartPeg, AuxPeg, EndPeg)
 INTEGER, INTENT(IN) :: N
 CHARACTER(*), INTENT(IN) :: StartPeg, AuxPeg, EndPeg
 IF(N == 1) THEN
  PRINT *, "Move disk from ",StartPeg, "to ", EndPeg
 ELSE
  CALL Move(N-1, StartPeg, EndPeg, AuxPeg)
  CALL Move(1, StartPeg, " ", EndPeg) ! or use the next statement
!   PRINT *, "Move disk from ",StartPeg, "to ", EndPeg
  CALL Move(N-1, AuxPeg, StartPeg, EndPag)
 END IF
END SUBROUTINE Move
```