Chapter 6

# Programming with Functions

Scientific computing usually uses a lot of different functions. The Fortran library only provide a few intrinsic functions. It is useful if we can write our own functions.

Dividing a complex problem into small sub-problems usually simplifies the algorithm.

Some functions or subprograms can be used to solve different problems so the codes can be reused.

## Function subprograms

Basic syntax:

```
FUNCTION function-name (formal-argument-list)
specification part
execution part
END FUNCTION function-name
```

The first line can add `type-identifier` to specify the type of returning value.

If the `type-identifier` is omitted, then the type should be specified inside the function.

The `specification part` include:

- The type of the function value (if the type of function is not specified).
- The types of each argument using `INTENT specifier`. The specifier can be `IN, OUT, INOUT`.
- Other type of variables used in this function and other specifications same as the specifications in Fortran programs.

The execution part of a function subprogram is the same form as that of a Fortran program with the additional stipulation that it should include at least one statement that assigns a value to the identifier of the function:

```
function-name = expression
```

The value of the function will be returned to the program unit that references it when the `END` statement is encountered or when a `RETURN` statement is executed.

A function subprogram can be made accessible to a program in three ways:

1. It is placed in a subprogram section in the main program just before the END PROGRAM statement. In this case it is called an internal subprogram.
2. It is placed in a module from which it can be imported into the program. In this case, it is called module subprogram.
3. It is placed after the END PROGRAM statement of the main program. In this case, it is called external subprogram.

An example of converting temperature from Fahrenheit to Celsius:

```
FUNCTION Fahr_to_Celsius(Temperature)

REAL :: Fahr_to_Celsius
REAL, INTENT(IN) :: Temperature

Fahr_to_Celsius = (Temperature - 32.0) / 1.8

END PROGRAM Fahr_to_Celsius
```

If we use a function as a internal subprogram, then put it after

```
CONTAINS
```

A main program can contain many subprograms.

```
PROGRAM Temperature_Conversion
IMPLICIT NONE
REAL :: FahrenheitTemp, CelsiusTemp
CHARACTER(1) :: Response
DO
  WRITE (*, '(1X, A)', ADVANCE = "NO") "Enter a Fahrenheit &
   & temperature: "
  READ *, FahrenheitTemp
  CelsiusTemp = Fahr_to_CelsiusTemp(FahrenheitTemp)
  PRINT '(1X, 2(F6.2, A))', FahrenheitTemp, &
  " in Fahrenheit is equivalent to ", CelsiusTemp, "in Celsius"
  WRITE(*, '(/ 1X A')', ADVANCE = "NO") &
   "More temperatures to convert (Y or N)?"
  READ *, Response
  IF (Response /= "Y") EXIT
END DO

CONTAINS

FUNCTION Fahr_to_Celsius(Temperature)
REAL :: Fahr_to_Celsius
REAL, INTENT(IN) :: Temperature
Fahr_to_Celsius = (Temperature - 32.0) / 1.8
END PROGRAM Fahr_to_Celsius

END PROGRAM Temperature_Conversion
```

The argument declared in main program is the actual argument while the corresponding argument in the function is a formal argument of the function. The type and number of actual argument should be the same as the formal arguments.

`INTENT(IN)` attribute declaration protects the corresponding actual argument by ensuring that the value of the argument cannot be changed while the function is executed. In this case, any attempt to change the value of the formal argument will cause a compile-time error.

If `INTENT(IN)` is not used and the value of argument is changed in the function, then the value of the actual argument will also be changed accordingly. That might cause some errors.

A function can also declare local identifiers, which are only used within the function.

```
FUNCTION Factorial(N)
 INTEGER :: Factorial
 INTEGER, INTENT(IN) :: N
 INTEGER :: I
 IF(N > 12 .OR. N < 0) STOP "** N should be between 1 to 12 **"
 Factorial = 1
 DO I = 2, N
  Factorial = Factorial * I
 END DO
END FUNCTION Factorial
```

One function can reference other functions. For example, the Poisson probability function is:

$$P(n) = \frac{\lambda^n \cdot e^{-\lambda}}{n!}$$

where $\lambda =$ the average number of occurrences of a phenomenon per time period and $n =$ the number of occurrences in that time period.

When we implement the Poisson function, we use the factorial function.

```
PROGRAM Poisson_probability
IMPLICIT NONE
......
Probability = Poisson(AveOccurs, NumOccurs)
......
CONTAINS
  FUNCTION Poisson(Lambda, N)
   REAL :: Poisson
   REAL, INTENT(IN) :: N, Lambda
   Poisson = (Lambda**N*EXP(-Lambda))/REAL(Factorial(N))
  END FUNCTION Poisson

  FUNCTION Factorial(N)
    ......
  END FUNCTION Factorial
END PROGRAM Poisson_probability
```

The order in which the subprograms are arranged is not important.

An entity (variables, constants etc) declared within a subprogram is called local entity. A local entity is not accessible outside that subprogram.

An entity declared in the main program is called a global entity. A global entity is accessible throughout the main program and in any internal subprogram in which no local entity has the same name as the global item.

Fundamental Scope Principle: The scope of an entity is the program or subprogram in which it is declared.

Although global variables can be used to share data between the main program and internal subprograms, it is usually not wise to do so. It is a good idea to make a function or subprogram as independent as possible.

Statement labels are not global. So the `FORMAT` statement in a main function can not be used in a subprogram.

`IMPLICIT NONE` declaration is global.

The values of local variables in a subprogram are not retained from one execution of the subprogram to the next unless either:

1. They are initialized in their declarations, or
2. They are declared to have the `SAVE` attribute.

Two methods to use the SAVE attribute. One is using

```
type, SAVE ::  list-of-local-variables
```

Or using a SAVE statement:

```
SAVE list-of-local-statements
```

In this case, if the list is omitted, all the local variables will have the SAVE attribute.

## Example: Numerical Integration

Using a program to approximate the area under the graph of a nonnegative function $f(x)$ from $x = a$ to $x = b$, thus obtaining an approximate value for the integral

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

which can be calculated as

$$\Delta x \left( \frac{y_0 + y_n}{2} + \sum_{i=1}^{n-1} y_i \right)$$

Program: pp. 129.

Application: pp. 131-133.

Fortran provides modules which can be used to build libraries of subprograms. A module is a program unit used to package together type declarations, subprograms etc. A module file also use the form of `file-name.f90`.

Simple version:

```
MODULE name
CONTAINS
 subprogram_1
 subprogram_2
 ... ...
 subprogram_n
END MODULE name
```

```
MODULE Temperature_Library
 IMPLICIT NONE
 REAL, PARAMETER :: HeatOfFusion = 79.71 &
                    HeatOfVaporization = 539.55
CONTAINS
  FUNCTION Fahr_to_Celsius(Temperature)
      ... ...
  END FUNCTION Fahr_to_Celsius

  FUNCTION Celsius_to_Fahr(Temperature)
     ... ...
  END FUNCTION Celsius_to_Fahr
END MODULE Temperature_Library
```

Once a module has been written, it can be used by other program unit. Put

```
USE module-name
```

at the beginning of the specification part of a program unit. All identifiers declared in the `module-name` are imported into the program.

If you just want to use parts of the module, then you can use:

```
USE module-name, ONLY: list
```

Each item in the list is of the form:

`new-identifier => identifier` or just `identifier`.

The temperature conversion program now can use `Temperature_Library` as follows

```
PROGRAM Temperature_conversion_2

USE Temperature_Library, ONLY: Fahr_to_Celsius

IMPLICIT NONE
REAL :: FahrenheitTemp, CelsiusTemp
  ... ...
  ... ...
END PROGRAM Temperature_conversion_2
```

Translation of source codes to produce an executable program consists of two steps:

1. Compilation. Create machine-language program.
2. Linking. Create an executable file.

Fortran 90 uses `f90` to do the both jobs:

```
f90 file1.f90 file2.f90 -o file.out
```

If we just want to do compile, then we can use `-c` flag:

```
f90 -c file1.f90
```

It will produce a `file1.o` file. Then `f90` can be used to link the object files.

```
f90 file1.o file2.o -o file.out
```

If `-o file.out` is omitted in linking, the default executable file will be `a.out`.

During the compiling and linking the compiler will output some message to show the progress it making, such as

```
file1.f90:
file2.f90:
Linking:
```

After a Fortran module file has been compiled, it will also create a `module-name.mod` file. This file provides some information about the module which is for the compiler to use.

External subprograms are put at out of the main program. We can put a function after `END PROGRAM`:

```
PROGRAM Temperature_conversion_3
IMPLICIT NONE
REAL :: Fahr_to_Celsius
REAL :: FahrenheitTemp, CelsiusTemp
 ... ...
END PROGRAM Temperature_conversion_3

FUNCTION Fahr_to_Celsius(Temperature)
IMPLICIT NONE
REAL :: Fahr_to_Celsius
REAL, INTENT(IN) :: Temperature
 ......
END FUNCTION Fahr_to_Celsius
```

For external subprogram, it is desirable to use interface to connect with the main program. In this way, the compiler can perform the necessary consistency checks. Interface block for external function has the form:

```
INTERFACE
  interface-body
END INTERFACE
```

Interface block is put inside the main function.

The interface-body for a external function consists of:

1. The subprogram heading (except that different names may be used for the formal arguments)
2. Declarations of the argument and the result type in the case of a function.
3. An `END FUNCTION ( or END SUBROUTINE)` statement

```
PROGRAM Temperature_conversion_4
IMPLICIT NONE
 INTERFACE
  FUNCTION Fahr_to_Celsius(Temperature)
   REAL :: Fahr_to_Celsius
   REAL, INTENT(IN) :: Temperature
  END FUNCTION Fahr_to_Celsius
 END INTERFACE

REAL :: Fahr_to_Celsius
 ... ...
END PROGRAM Temperature_conversion_4

FUNCTION Fahr_to_Celsius(Temperature)
 ......
END FUNCTION Fahr_to_Celsius
```

A main program can reference subprograms.

A subprogram can reference other subprograms.

A subprogram can reference itself called recursion.

Examples:

- If $f(n) = n!$, then $f(1) = 1$, and $f(n) = n \cdot f(n-1)$ for $n \geq 2$.
- If $f(x, n) = x^n$, then $f(x, 0) = 1, f(x, n) = x \cdot f(x, n-1)$ for $n \geq 2$.

In these examples, the function has a known start point and each step can be easily calculated from the previous step.

In general, a function is said to be defined recursively if its definition consists of two parts:

1. An anchor or base case, in which the value of the function is specified for one or more values of the arguments.
2. An inductive or recursive step, in which the function's value for the current value of the arguments is defined in terms of previous defined function value and/or argument values.

To create a recursive function in Fortran, use the word `RECUSIVE` at the beginning of the subprogram heading and a `RESULT` clause to return the value. The type of the function is specified by declaring the type of the result variable.

```
RECURSIVE FUNCTION Factorial(n) RESULT (Fact)
INTEGER :: Fact
INTEGER, INTENT(IN) :: n
IF (n == 0) THEN
  Fact=1
ELSE
  Fact = n * Factorial(n-1)
END IF
END FUNCTION Factorial
```

Many problems can be solved with either recursive or a non-recursive algorithm. In general, a recursive algorithm is simpler but less efficient. However, some problem is difficult to write a non-recursive algorithm.

Example: greatest common divisor.

Example: Street Network

pp. 147.

The greatest common divisor (GCD) of two integers $a$ and $b$ is the largest positive integer that divides both $a$ and $b$. The Euclidean algorithm for finding GCD$(a, b)$ is as follows: if $b = 0$, GCD$(a, b) = a$. Otherwise, let $a = bq + r$, where $q$ is quotient and $r$ is remainder. Then GCD$(a, b) =$ GCD$(b, r)$.

Write a function subprogram to calculate the GCD of two integers.

One uses non-recursive algorithm and one used recursive algorithm.