Chapter 1

Supplement No. 1(B): Fundamentals of the FORTRAN 95 Programming Language

LEARNING OBJECTIVES

The objective of this supplement is to
- present a short summary of basic concepts of C++ programming language;
- describe the implementation of basic programming operations such as loops, accumulators, conditional constructs in C language;
- explain how to apply pseudofunction or pseudocodes into functions or subprograms in C++ programming language.

The textbook focuses on computational engineering applications. Thus, supplementary course materials involving programs in C/C++, Fortran, Visual Basic, Python, Matlab® and Wolfram Mathematica® languages or software are used. This supplement is a reference document for Fortran 95 computer language and illustrates how pseudocode statements can be converted to actual programming languages. It is assumed that the reader is familiar with programming concepts in general and may also be familiar with the fortran programming language.

The algorithms presented in this book are presented in a form that requires very little time and effort to digest. The motivations for using pseudocodes have been stated in the preface and Chapter 1. The aim of this appendix is to present the basic syntax and constructions with simple and concise examples.

1.1 FORTRAN PROGRAM STRUCTURE

A fortran program consists of three sections: *declarations*, *execution*, and *termination* sections. *Comments* are inserted freely anywhere before, after or within the program.

1.1.1 Declaration Section

The declaration section is located at the beginning of the main and subprograms. It comprises the non-executable statements defining name, number and the types of variables used in the program.

The first statement the `PROGRAM` statement that specifies the name of the program. Fortran 95 program names may be up to 31 characters long and contain any combination of alphabetic characters, digits, and the underscore (_) character, provided that the first character is alphabetic.

1.1.2 Execution Section

This section consists of one or more executable statements describing the tasks to be carried out by the program. Any statement pertinent to reading, writing, computing, and so on are part of this section.

1.1.3 Termination Section

The termination statements used in this section are the `STOP` and `END PROGRAM`. The `STOP`, which is optional, directs the computer to stop running the program. The `END PROGRAM` statement tells the compiler that the end of program is reached. The compiler automatically generates a `STOP` command when `END PROGRAM` statement is reached. Therefore, in reality, it is rarely used.

1.1.4 Commenting

Commenting is done to allow *human-readable* descriptions detailing the purpose of some of the statement(s) and/or to create *in situ* documentation. All characters following an exclamation mark (!) except in a character string, are commentary, and are ignored by the compiler.

1.1.5 A sample Fortran 95 Example

The basic features of a Fortran 95 program (`example.f95`) is illustrated in **F95 code 1.1**:

Lines 1-4, 6, 9, and 12 are reserved entirely for comments. Note that in this example explanatory comments are dispersed throughout the program. The other lines also include comments at the end of the statements following ! mark.

In line 5, the`PROGRAM example` is executed first.

In line 7, `PI` is declared as constant using `PARAMETER` identifier, and its numerical value is assigned.

In line 8, the variables (`area` and `radius`) are declared as `REAL`.

In line 10, the first executable statement in this program is the `PRINT` statement, which prints out a message prompting the user to enter the radius of the circle.

In line 11, the next executable statement is a `READ` statement, which reads in the `radius` supplied by the user.

In line 12, the third executable statement instructs the computer to calculate the area from `PI*radius`^{2} and store the result in variable `area`.

In line 13, the program execution is stopped with `STOP` statement.

In line 14, the program is terminated with `END PROGRAM`.

1.2 VARIABLES, CONSTANTS, AND INITIALIZATION

1.2.1 Identifiers and Data `Types`

Identifiers (i.e., symbolic names for variables, functions, and so on) are represented symbolically with letters or combinations of letters and numbers (`a`, `b`, `ax`, `xy`, `a1`, `tol`, ...). The first character of an identifier must be a letter, which includes underscore (_). The Fortran95 language keywords (such as `integer`, `do`, `while`, print, etc)) are reserved for specific tasks and cannot be used as identifiers.

Fortran language supports the a wide variety of built-in data types: integer (`integer`), real types (`real`, `double`), `complex`, *logical*, and character type (*see* full listing).

Integer (`int`): Integers are whole numbers that can hold positive, or negative whole values; e. g., 0, -23, or 140. On most compilers, the default kind stores integers as short integers 4 bytes in size (`kind=4`). Long integers are usually 8 bytes (`kind=8`).

F95 Code 1.1

```
! ======================================================================
! Description : A Fortran program to calculate the area of a circle.
! Written by  : Z. Altac
! ======================================================================
PROGRAM example
! Declarationsn section
REAL, PARAMETER :: PI = 3.14159 ! PI declared and defined as constant
REAL :: radius, area            ! Local variables declared
! Execution section
PRINT*, "Enter radius : "       ! Prompt input instruction
READ*, radius                   ! Local data (radius) is supplied
area = PI * radius * radius     ! Calculate the area of circle
PRINT *, "The area of the circle is: ", area  ! Print output (area)
! Termination section
STOP
END PROGRAM
```

Character (`char`): The character data type stores strings of characters. A character constant is a string of characters enclosed in single (') or double (") quotes. The mini- mum number of characters (a single character) in a string is 1.

Real numbers (`real, double precision`): Real numbers (or floats) with a decimal point can be represented; e. g., -2.3, 0.14, 1.2e5, and so on. The default kind for real variables is 4 bytes (32 bits). Most compilers support a double precision data type that uses 8 bytes (64 bits).

Logical: The logical data type stores boolean values and can only contain values `true` or `false`.

1.2.2 Type Declaration or Initialization of Variables and Constants

A *variable* is a symbolic representation of the address in memory space where values are stored, accessed, and updated. A *variable declaration* is to specify its *type* of the variable (i.e., the *identifier*) as `char`, `int`, `float` and so on, but a value to the variable has yet to be assigned. The value of a variable changes during the execution of a computer program, while the value of a `constant` does not. All variables (or constants) used in a program or subprogram must be declared before they are used in any statement.

The type of every constant or variable name must be declared in the program before it is used. The type declaration syntax for variables or constants is as follows:

```
type specifier ::  identifier-1, identifier-2, ..., identifier-n ;
type specifier, PARAMETER :: identifier = value ;
```

A *variable definition* or *variable initialization* is carried out by assigning a value to it, typically with the assignment operator "=".

In fortran, variable identifiers that begin with the letters `I` thru `N` are assumed to be of type `INTEGER` by default. identifiers starting with other letters are assumed to be of type `REAL`. Also the identifiers are *not* cases sensitive. For example, `Name`, `NAME`, or `name` denote the same variable.

The type declaration is carried out at the beginning of the main() function. IMPLICIT NONE overrides the default implicit typing rules for the variable (identifiers) names. When IMPLICIT NONE is invoked, all constant and variable names in a program module must be explicitly declared.

A variable can be initialized while being declared. The following are a set of example declarations.

```
INTEGER :: iter, maxit, i, j  ! Declare integer variables
REAL    :: radius, area;      ! Declare float (real) variables
INTEGER, PARAMETER :: m = 99  ! Integer m is declared and initialized
REAL, PARAMETER :: PI=3.14159 ! Real pi is declared and initialized
DOUBLE PRECISION :: yd, zd=2.0*PI ! zd is declared and initialized
CHARACTER(1) :: yes='y', no='n' ! yes and no are declared and initialized
CHARACTER(LEN=72) :: line   ! line is declared as 72-character-long string
```

In this example, m, pi, zd, yes and no identifiers are not only type-declared but also initialized at the same time. The order of declaration can be important. For instance, in order to initialize zd, PI must be declared and initialized beforehand. A string is declared according to its maximum length as CHARACTER(72) or CHARACTER(LEN=72).

In general, a variable does not have a default value. Different compilers may act differently. Using the value of an uninitialized variable in operations may lead to unpredictable results or in some compilers the program may crash.

1.3 INPUT/OUTPUT (I/O) FUNCTIONS

An inevitable element in any program is the communication of the input and output data with the main program or its sub-programs. This is done through READ * and PRINT * statements, which are used to read initial values of variables into the program from a keyboard or print (or write) the intermediate or final values of variables to a screen.

This form of input-output is called *list-directed input/output* and the exchange of I/O data is in *free format*. The term *list-directed input* implies that the types of the variables in the variable list determine the required format of the input data.

The READ * and PRINT * are commonly used to read and display data from and on input and output devices, respectively. The syntax for the READ,PRINT and WRITE statements are as follows:

```
READ  fmt,  variable-1, variable-2, ... , variable-n
PRINT fmt,  variable-1, variable-2, ... , variable-n
```

where fmt is the format specification and the variable's are supplied with a list to be read from keyboard or written on screen.

Consider following code segment:

```
1  INTEGER :: n1, n2;
2  PRINT*, "Enter two integer numbers: "
3  READ *, n1, n2
4  PRINT*, "You entered",n1," and",n2
5  PRINT*, "Their squares are",n1*n1," and",n2*n2
```

Table 1.1: Fortran edit descriptors.

DATA TYPE		EDIT DESCRIPTORS	
INTEGER form		Iw	Iw.m
REAL	Decimal form		Fw.d or fw.d
	Exponential form		Ew.d, ew.d, or Dw.d
	Scientific form		ESw.d or ESw.dEe
LOGICAL form		Lw	
CHARACTER form		A or Aw	
Positioning	Horizontal	nX or nx	
	vetrical	/	

where * denotes free format, but it may also be replaced with *format specifications*.

For the input values of n1=3 and n2=5, the output becomes

```
You entered           3  and           5
Their squares are           9  and          25
```

For the formatted output operations, fmt is used together with the file unit number as (u, fmt). The syntax for reading from and writing data to a file are as follows:

```
READ (u, fmt)  variable-list
PRINT(u, fmt)  variable-list
WRITE(u, fmt)  variable-list
```

The value of u=5 indicates reading data from the terminal device and 6 for writing to the terminal device. The asterisk (*) appearing as a logical unit identifier stands for standard input device (i.e., keyboard) when it appears in a READ statement. It stands for standard output device (i.e., monitor) when it appears in a WRITE or PRINT statement.

1.3.1 Format Specifications

A *format specification* is a placeholder denoting a value to be filled in during the printing. It is a string, containing a list of edit descriptors which specify the exact format (width, digits after decimal point, etc.) of the numbers to be displayed. As it gives the user a control over the appearance of the output, but it can make the I/O statements a bit complicated or hard to read.

Fortran edit descriptors are presented in **Table 1.1**, where w refers to the width of the field, m is the minimum number of characters to be used, d is the number of digits to the right of the decimal point, and e is the number of digits in the exponent.

In standard setting a floating-point is displayed to six digits by default. Decimal numbers are printed without a decimal point unless there are digits after the decimal point by default. Very small or large numbers are displayed in exponential form.

```
program main
    implicit none
    integer :: i=1234
    real :: r=123.456789

    print *, "set width to 6 chars"
```

```
        write(*,'(I6)') i
        print *, "set width to 10 chars"
        write(*,'(I10)') i
        print *, "set precision to 2, 4, and 6"
        write(*,'(F10.2,4x,F10.4,4x,f10.6)') r,r,r
    end program main
```

The output looks like this

```
 set width to 6 chars
  1234
 set width to 10 chars
       1234
 set precision to 2, 4, and 6
 bbbb123.46xxxxbb123.4568xxxx123.456787
```

For example, the following code segment reserves 5 characters (w=5) for the integer and 8 characters (w=8 with p=5) for the float variable.

```
    real :: a=14.125, b=12.033, c=2415.1357
    write (*,'(f4.1)') a
    write (*,'(f6.2,f6.3)') a, b
    write (*,'(3f10.4)') a, b, c
    write (*,'(3(2X,E10.4))') a, b, c
    write (*,'(3(2X,ES10.4))') a, b, c
    write (*,'(2X,EN10.1,2X,EN12.2,4X,EN12.4))') a, b, c
```

The output is

```
          1         2         3         4
1234567890123456789012345678901234567890123
-------------------------------------------
14.1212.033
  14.1250   12.0330 2415.1357
 0.1412E+02  0.1203E+02  0.2415E+04
 1.4125E+01  1.2033E+01  2.4151E+03
   14.1E+00     12.03E+00      2.4151E+03
```

1.4 ASSIGNMENT STATEMENT AND ARITHMETIC CALCULATIONS

Expressions involving the arithmetic operators often involve the assignment operator =. The general form of an assignment statement is

```
        Variable-name = expression
```

In fortran, an *expression* on the right-hand side of the "=" sign is evaluated first, and its value is placed at the allocated memory location of the *variable* on the left-hand side. Any recently computed value of *variable* replaces its previous value. An *assignment* denoted by ← (a left-arrow) in pseudocode notation is replaced with '=' sign.

In computer programming, we generally apply `sum = sum + x` or `sum = sum - x` operations, which is illegal in normal algebra. But it makes sense in computer programs in that the values of the expressions are stored in memory. In this regard, the value of `x` is added to the current value stored in variable `sum` and the result is stored back into the memory location of `sum`.

Consider the fortran code segment below:

```
1    integer :: X = 0  ! X is initialized by zero
2    X = X + 1   ! increment X by 1, X becomes 1
3    X = X + 2   ! increment X by 2, X becomes 3
4    X = X + 4;  ! increment X by 4, X becomes 7
5    X = X - 1   ! decrement X by 1, X becomes 6
6    X = X - 3   ! Equivalent to X = X -3, decrement X by 3, X becomes 3
```

In line 1, `X` is declared as integer and initialized at the same time. The memory value of `X` is zero. In line 2, the assignment operation adds "1" to `X`. fix the order In line 3, the memory value of `X` is substituted in the rhs, the addition operation updates the value of `X` as 4. Finally, in line 3, the rhs is evaluated first (4+6=10), and the result is placed in the memory location of `X`.

1.5 SEQUENTIAL STATEMENTS

Frequently, a sequence of statements (with no imposed conditions) are used to perform a specific task. These statements will be executed in the order they are specified in the program. The statements can begin in any column. By using a semicolon (;) as a separator, multiple expressions can be placed in a single line. An exclamation mark "!" in any column is the beginning of a comment and thus the rest of the line is ignored by the compiler. A statement can be continued on the following line by appending a "&" sign on the current line.

For instance, the following fortran code segment illustrates how the first and second statements are presented on a single line with the use of a semicolon.

```
1       a = b * b + c * c ;  d = sqrt(a) ! Statement-1 and Statement-2
2       x1 = d / ( b + c )               ! Statement-3
3       x2 = d / ( b - c )               ! Statement-4
4       y = x1 + x2 + x3    &            ! Statement-5
5           + x4 + x5                    ! continuation of statement4
```

In line 1, statements 1 and 2 are separated with a semicolon. Single arithmetic statements occupy lines 2 and 3. In lines 4 and 5, the statement 5 extending over two lines are connected with "&" sign.

1.6 ARITHMETIC OPERATIONS

Arithmetic operations involve plus (+), minus (-), multiplication (*), and division (/), exponentiation (**) operators. These operations can be used with integer or floating-point types. The division operator involving an integer division truncates any fractional part, e.g., `15/3`, `16/3`, and `17/3` all yield 5.

In fortran, the *exponentiation operation* x^y is carried out as `x**y`. While using the `exp (a * ln(b) )` algebraic formula will certainly work, but it takes longer to carry out the operation and is less accurate than an ordinary series of multiplications. Thus, if we have a choice, we should try to raise real numbers to integer powers instead of real powers.

Table 1.2: Arithmetic, equality/relational, logical and assignment operators in C.

Operator	Description	Example
`+, -`	Addition and subtrations	`a + b` or `a - b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`==`	compares the operands to determine equality	`a == b`
`/=`	compares the operands to determine unequality	`a /= b`
`>`	determines if first operand greater	`a > b`
`<`	determines if first operand smaller	`a > b`
`<=`	determines if first operand smaller than or equal to	`a > b`
`>=`	determines if first operand greater and equal to	`a > b`
`.AND.`	Logical AND operator	`a.AND.b`
`.OR.`	Logical OR operator	`a.OR.b`
`.NOT.`	Logical NOT operator	`.NOT.a`
`.EQV.`	Logical equivalence EQV operator	`a.EQV.b`

1.7 RELATIONAL AND LOGICAL OPERATORS

Branching in a computer program causes a computer to execute a different block of instructions, deviating from its default behavior of executing instructions sequentially.

Logical calculations are carried out with an assignment statement:

```
Logical_variable = Logical_expression ;
```

Logical_expression can be a combination of logical constants, logical variables, and logical operators. A logical operator is defined as an operator on numeric, character, or logical data that yields a logical result. There are two basic types of logical operators: relational operators (`<, >, <=, >=, ==, /=`) and combinational (logical) operators (`AND, OR, EQV, NEQV`). Branching is carried out using and . In **Table 1.2**, the arithmetic, relational, and logical operators are summarized.

Branching structures are controlled by *logical variables* and *logical operations*. Logical operators evaluate relational expressions to either 1 (`True`) or 0 (`False`). Logical operators are typically used with Boolean operands. The logical AND operator (&&) and the logical OR operator (||) are both binary in nature (require two operands). The logical NOT operator (!) negates the value of a Boolean operand, and it is a unary operator.

Logical operators are used in a program together with relational operators to control the flow of the program. The `&&` and `||` operators connect pairs of conditional expressions. Let $\mathbf{L}_1$ and $\mathbf{L}_2$ be two logical prepositions. In order for $\mathbf{L}_1$ `&&` $\mathbf{L}_2$ to be True, both $\mathbf{L}_1$ and $\mathbf{L}_2$ must be True. In order for $\mathbf{L}_1$ `||` $\mathbf{L}_2$ to be True, it is sufficient to have either $\mathbf{L}_1$ or $\mathbf{L}_2$ to be True. When using `!`, a unary operator, in any logical statement, the logic value is changed to True when it is False or changed to False when it is True. These operators can be used to combine multiple expressions. For given `x=5`, `y=9`, `a=18`, and `b=3`, we can construct following logical expressions:

```
1    (x < y).AND.(y < a).AND.(a > x)     ! TRUE
2    (x < y).AND.(y > a).AND.(a >= b)    ! FALSE
3    ((x < y).AND.(y < a)).OR.(a < b)    ! TRUE
4    ((x > y).OR.(y > a)).OR.(a < b)     ! FALSE
```

The order of evaluation of `.AND.` and `.OR.` is from left to right.

1.8 CONDITIONAL CONSTRUCTIONS

In fortran, branching control and conditional structures allow the execution flow to jump to a different part of the program. *Conditional* statements create branches in the execution path based on the evaluation of a condition. When a control statement is reached, the condition is evaluated, and a path is selected according to the result of the condition.

In Fortran language, we use `IF-`, `IF-THEN-ELSE`, and `SELECT CASE` constructions allow one to check a condition and execute certain parts of the code if *condition* (logical expression) is True.

1.8.1 Block `IF` Construction

The block `IF` construct shown below is the most common and simplest form of the conditional constructs. It executes a block of statements *if and only if* a *condition* (i.e., logical expression) is True.

```
IF (condition) THEN
    STATEMENTs       ! if condition is true
ENDIF
```

Note that the *condition* is enclosed with round brackets. The `IF` construct can also command multiple statements. A block of statements is syntactically equivalent to a single statement.

1.8.2 The `ELSE` and `ELSE IF` Clauses

Often we are required to execute a set of statements if one condition is true, and a different set of statements if another condition is true. In reality, there may be several options to consider. The `ELSE` and `ELSE IF` clauses may be attached to the block `IF` construct to cover all possible options.

The simplest `IF-THEN-ELSE` construct (presented below) contains another block for the statements to be executed in the case the *condition* is False.

```
IF (condition) THEN
    STATEMENT(s)      ! if condition is true
ELSE
    STATEMENT(s)      ! if condition is false
ENDIF
```

A more complicated `IF-ELSE IF` construct can be devised as follows:

```
IF (<condition-1>) THEN       ! if <condition-1> is true
    IF (<condition-2>) THEN
        STATEMENT(s)          ! if <condition-2> is true
    ELSE
        STATEMENT(s)          ! if <condition-2> is false
    ENDIF
ELSE IF (<condition-3>) THEN  ! if <condition-1> is false
        STATEMENT(s)          ! if <condition-3> is true
    ELSE
```

```
            STATEMENT(s)          ! if <condition-3> is false
        ENDIF
    ENDIF
```

1.8.3 Logical IF Construction

An alternative form of a block IF construct described above is the logical IF construct, which is a single statement of the form

```
IF (logical_expression) STATEMENT
```

where STATEMENT is a single executable Fortran statement. If the logical expression is true, the program executes the single statement on the same line with it. Otherwise, the program skips to the next executable statement in the program.

```
      integer :: p, maxit
       ...
100   CONTINUE
       ...
      IF (p <= maxit) GOTO 100
```

1.8.4 SELECT CASE Construction

The SELECT CASE construction is an alternative to the IF-ELSE-IF ladder. A SELECT CASE construction allows a multi-decision cases to be executed based on the value of an integer selector variable.

The general form of the SELECT CASE statement is as follows:

```
SELECT CASE (case-expression)
    CASE (case_selector_1) [name]
        STATEMENT(s)-1
    CASE (case_selector_2) [name]
        STATEMENT(s)-2
    ....
    CASE DEFAULT [name]
        STATEMENT(s)-n
END SELECT
```

where case_selector_1, case_selector_2, and so on are integers. Upon evaluating case-expression, if and when it matches one of the available cases, the construction branches to the matching case and executes the block statements from that point onwards. Otherwise, it branches to default, which is optional, and executes its statements.

The following fortran code segment uses year to execute SELECT CASE construct. For the case of year=1, Freshman is displayed; for year=2, year=3, and year=4, Sophomore, Junior, and senior is displayed, respectively. If year corresponds none of the above, the message Graduated is displayed.

```
INTEGER :: year=3      ! year is initialized
SELECT CASE (year)
    CASE (1)
        PRINT*, "Freshman"
    CASE (2)
        PRINT*, "Sophomore"
    CASE (3)
        PRINT*, "Junior"      ! Output is  Junior
    CASE (4)
        PRINT*, "Senior"
    CASE default
        PRINT*, "Graduated"
END SELECT
```

1.9 CONTROL CONSTRUCTIONS

Control (loop) constructions are used when a program needs to execute a block of instructions repeatedly until a <*condition*> is met, at which time the loop is terminated. There are three control statements in most programming languages that behave in the same way: `while`, `do-while` (equivalent to Repeat-Until loop), and `for`-constructs.

1.9.1 DO WHILE CONSTRUCT

A `DO WHILE`-construct has the following form

```
DO WHILE ( <condition> )
    STATEMENT(s)    ! if <condition> is true
ENDDO
```

In the `DO WHILE` construct, the `<condition>` is evaluated before the statement block. If the `<condition>` is `true`, the block of statement(s) will be executed. If the condition is initially `false`, the statement block will be skipped.

1.9.2 DO- CONSTRUCT

The `DO`-construct is used when a block of STATEMENT(S) is to be executed a specified number of times. The `DO`-construct has the form

```
DO  loop_variable = <l_start>, <l_stop>, <l_step>
     STATEMENT(s)    ! statements in the block are executed
ENDDO
```

where `loop_variable` is an integer loop-control variable, the `l_start` and `l_stop` are the initial and the final value, and the `l_step` denotes the increment (if `l_step`>0 or decrement (`l_step`<0) in the loop variable. If `l_step`> is omitted, then the loop variable is increased by 1.

The total number of iterations is (`l_stop`>-`l_start` + `l_step`>)/`l_step`. To break out of a **DO**-loop (**Exit**) before it reaches `stop`, and also a *condition* within the STATEMENT(S) block needs to be specified.

For instance, consider the following loops:

```
INTEGER :: i, j, k, m
DO i = 2, 9
   STATEMENT(s)    ! The loop-block is executed for i = 2, 3,..., 9
ENDDO
DO j = 1, 8, 2
   STATEMENT(s)    ! The loop-block is executed for i = 1, 3, 5, 7
ENDDO
DO k = 5, 3, -1
   STATEMENT(s)    ! The loop-block is executed for k = 5, 4, 3
ENDDO
DO m = 6, 1, -2
   STATEMENT(s)    ! The block is executedfor k = 6, 4, 2
ENDDO
```

The DO loop over variable-i is executed for every i from 2 to 9 with increments of +1 while the loop over the variable-j is executed for every j from 1 to 8 with increments of +2. On the other hand, the DO loop with loop variable-k is executed for every k from 5 to 3 with decrements of -1 while the loop over the variable-m is executed for every m from 6 to 1 with decrements of -2.

A DO-construct is used when the number of iterations to be performed is known beforehand. It is easy to use in nested loop settings (with arrays) due to having clearly identified loop indexes.

1.9.3 EXIT and CYCLE Statements

It is sometimes required to exit a loop other than by a specified <condition> at the top or bottom. The EXIT statement provides an early exit from a loop construction. In other words, execution of the EXIT statement results in transferring the control to the next executable statement after the END DO statement to which it refers.

With this in mind, a conditional loop with a test <condition> at the bottom (i.e., equivalent to Repeat-Until pseudo-construct) can be can be established with DO- loop. The conditional DO- construct has the form

```
DO
    STATEMENT(s)     ! if <condition> is false
    IF ( <condition> ) EXIT
ENDDO
```

This construction is similar to DO WHILE-construct in that the statement-block is executed as long as the <condition> is False. Note that the block of statements is executed at least once.

In case of nested loops, it terminates execution of the innermost DO construct in which it is enclosed. Consider the following fortran code segment with two loops. The EXIT statement is used inside the innermost loop.

```
INTEGER :: i, d
DO d = 2, 3
    DO i = d, 6
        PRINT *, "d=", d, " i=", i
```

```
            IF (i == 4) EXIT
        ENDDO
    ENDDO
```

The code output is

```
 d=           2  i=           2
 d=           2  i=           3
 d=           2  i=           4
 d=           3  i=           3
 d=           3  i=           4
```

Note that the `PRINT *` statement is before the logical `IF` construct. When `i` becomes 4, the loop for `i=5` and `i=6` is skipped and the loop is transferred to the next loop (over `d`-variable) which continues with `d=3`. The innermost loop over `i`-variable spans from 3 to 6, but the iterations are carried out only for `i=3` and `i=4` until the `EXIT` statement is executed.

If the `CYCLE` statement is executed inside a `DO` loop, the statements after the point at which CYCLE is invoked in the loop are not executed, and the loop control is returned to the top of the loop. Then the loop index is incremented, and iteration continues until the index reaches `l_stop`.

An example of the `CYCLE` statement in a `DO` loop is presented below:

```
1       INTEGER :: i
2       DO i=1, 6
3           IF ( i == 3 .OR. i==5 ) CYCLE
4           PRINT *, i
5       END DO
6       PRINT *, "End of loop!"
```

Here, the `DO` loop spans from 1 to 6, but when the loop variable becomes `i=3` or `i=5` the statements following the logical `IF` statement is skipped. In the code output below, it is noted that the loop statements following line 3 are skipped.

```
           1
           2
           4
           6
 End of loop!
```

1.10 ARRAYS

An array is a special case of a variable representing a set of data (variables) under one group name. Arrays can have one, two or more dimensions. The subscripts are always *integers*.

1.10.1 Declaring Arrays

Arrays must always be explicitly declared at the beginning of each program because the range and length of an array are critical in programming. The `DIMENSION` attribute in the type declaration statement declares the size of the array being defined.

The following code segment includes one-, tw0- and three-dimensional array declarations. The integer-type `no` variable is declared as one-dimensional array of length 16. The real-type `arra`, `arrb`, and `arrc`

variables are all declared as one-dimensional arrays of length 100. The 14-character-long string variables `name` and `last_name` variables are declared as one-dimensional array of length 999. The double precision-type `arr2d` and integer-type `arr3d` variables are defined as two- and three-dimensional arrays, respectively.

```
INTEGER, DIMENSION(16) :: no
REAL, DIMENSION(0:99)  :: arra, arrb, arrc
CHARACTER(len=14), DIMENSION(999) :: name, last_name
DOUBLE PRECISION, DIMENSION(10,10)  :: arr2d
INTEGER, DIMENSION(5,2,2)  :: arr3d
```

Most compilers also allow mixture of declarations. For example, in the following declarations, arrays (involving different sizes) and non-array variables of the same type are declared with a single statement.

```
INTEGER:: i, j, no(16), arr3d(5,2,2)
REAL   :: arra(0:99), arr3d(5,2,2), pi
CHARACTER(len=14) :: name(99), last_name(99), chrindx
```

The first statement defines integer variables `i` and `j` and one- and three-dimensional array variables `no` and `arr3d`. Besides `pi` variable, the second statement creates real array variables of `arra` and `arr3d`. The character variables of 14 characters long are declared as arrays of size 99.

1.10.2 Changing the Subscript Range of an Array

The elements of a one- or multi-dimensional arrays are subscripted, by default, starting from 1. However, in some algorithms, it is more preferable to index the array elements with subscripts that start with zero or some other value. Fortran allows the declaration and use of user-defined subscript ranges as shown below:

```
INTEGER, DIMENSION(60) :: no    ! no(0), no(1), ...., no(60)
REAL, DIMENSION(-2:9)  :: a )   !  a(-2), a(-1), a(0), ..., a(9)
INTEGER, DIMENSION(3,7):: indx  ! indx(3), indx(4), ..., indx(7)
```

1.10.3 Declaring Variable Size Arrays

In many programs, size of the arrays and memory requirements are subjected to the size of the input arrays. Fortran language allows declaration of the array sizes by using named constants, which make it easy to resize the arrays in the program. For example,

```
INTEGER, PARAMETER :: MAKS=99, ROW=5, COL=5  ! no(0), no(1), ...., no(60)
REAL, DIMENSION(M, N)   :: matrixA  ! a variable size matrix declaration
INTEGER, DIMENSION(MAKS):: indx     ! a variable size array declaration
```

1.10.4 Initializing Arrays with Assignment Statements

Initial values may be assigned to an array using assignment statements by either element by element in a `DO` loop or all at once with an array constructor.

The following fortran code segment initialize the elements of integer arrays. In line 1, all elements of `narr` are set to 3. In line 2, all elements of `arr2d` are set to 0. The `DO` construct in line 4-6 allows an initialization process that can be formulated as in line 5. Lastly, in line 7, the elements of an array can be initialized as a list.

```
1       INTEGER :: i, narr(100), arr(5), arr2d(5,5), arr1d(5)
2       narr = 3   ! yields narr(1) = narr(2) = ... narr(100) = 3
3       arr2d = 0  ! yields arr2d(1,1) = arr2d(1,2) = ... =arr2d(5,5) = 0
4       DO i = 1, 5
5          arr(i) = i  ! yields arr(1) = 1, arr(2) = 2, ..., arr(5) = 5
6       ENDDO
7       arr1d = (/ 1, 2, 3, 4, 5 /)
```

Initial values can be assigned to an array at compilation time by declaring their values in a type declaration statement. Initializing large arrays can be carried out using implied DO loop. An implied DO loop has the general form

```
(arg-1, arg-2, ..., index = istart, istop, inc)
```

where arg-1, arg-2, ... are values evaluated each time the loop is executed, and index, istart, istop, and inc behave in the same way as they do for ordinary counting DO loops.

```
1       INTEGER, DIMENSION(5) :: arr1d = (/ 1, 2, 3, 4, 5 /)
2       REAL, DIMENSION(10) :: arr1d = (/ ( 0.1*i, i = 1, 10, 2) /)
3       INTEGER, DIMENSION(30) :: nn = (/ ((1, i = 1, 8), 2*j, 3*j, j=1, 3) /)
```

Implied DO loops can nested or mixed with constants to produce complex patterns. For example, the following statements initialize the all elements of nn to 1 except for i=9, 10, 19, 20, 29 and 30 which are initialized to 2, 3, 4, 6, 6, and 9, respectively.

1.10.5 Whole Array Operations and Array Subsets

Two or more conformable arrays may be used together in arithmetic operations and assignment statements. If the arrays are conformable, then the desired arithmetic operation will be performed on an element-by-element basis.

The whole array arithmetic expressions do not require DO-loops and operate with the array names as if they were scalars. Consider following segment of a fortran code, where aa = arr1+arr2 and bb =3 × arr2 are computed:

```
1     INTEGER, DIMENSION(5) :: arr1 = (/ 3,-2, 1, 5, 4 /)
2     INTEGER, DIMENSION(5) :: arr2 = (/ 4, 1, 3, 1,-4 /), aa, bb
3     INTEGER :: i, j
4     ! element by element operations
5     DO i = 1, 5
6        aa(i) = arr1(i) + arr2(i)  ! array addition
7        bb(i) = 3 * arr2(i)        ! scalar multiplication
8     ENDDO
9     ! whole array operations
10    aa = arr1 + arr2              ! array addition
11    bb = 3 * arr2                 ! scalar multiplication
```

The two arrays have the same shape, i.e., they are conformable. The element by element operations in lines 5-8 can be replaced by the equivalent operations using the array names, as illustrated in lines 10 and 11.

Consider following 3×4 matrix.

$$\texttt{arr} = \begin{bmatrix} 2 & -3 & 4 & 1 \\ 5 & 0 & 1 & -3 \\ -1 & 2 & 2 & 7 \end{bmatrix}$$

The array subset corresponding to the second row and second column of the array is selected as `arr(2,:)` and `arr(:,2)`, respectively.

$$\texttt{arr(2,:)} = \begin{bmatrix} 5 & 0 & 1 & -3 \end{bmatrix} \quad \text{and} \quad \texttt{arr(2,:)} = \begin{bmatrix} -3 \\ 0 \\ 2 \end{bmatrix}$$

1.11 FUNCTION AND SUBROUTINE CONSTRUCTIONS

A fortran main program is delimited with `PROGRAM-END PROGRAM` statements. In general, it is impractical to write a complete *program* from A to Z that includes everything. Such programs would not only be too long but also too complicated to be practical. When writing a program, it is often necessary to repeat the same set of tasks multiple times within the same program. In fortran, to avoid repetition in coding, `FUNCTION` and `SUBROUTINE`s are utilized to perform specific tasks.

```
PROGRAM name
      Specification statements
      Executable Statements
CONTAINS
       internal-subprograms

END PROGRAM name
```

1.11.1 SUBROUTINE Procedures

`SUBROUTINE`s are can have a number of inputs and output variables, or may not produce any output at all (void subprogram). For example, a subroutine might print a warning or error message and terminate the program.

A subroutine in Fortran 90/95 has the following syntax:

```
SUBROUTINE subroutine-name (p1, p2, ..., pn)
   Argument and local variable declarations
   Executable statements
RETURN
END SUBROUTINE subroutine-name
```

In many programming languages, the arguments are passed to subprograms either *value* or *reference*. When arguments are passed by value, changes to the argument do not affect an original value of the variable in the caller. When an argument is passed by reference, the called function can modify the original value of variable. In Fortran, all variables are passed (by reference) to the arguments of the subroutines. Modifying an argument (input variable) accidentally in the subroutine will modify the values of the arguments (or variables)

in the calling subroutine or program. Thus, the local variables and all arguments with indended uses should be declared in the subroutine before they are used. When the task in the subroutine is completed, the control is returned to the calling program or subroutine using the `RETURN` statement. A subroutine may have several `RETURN`s.

1.11.1.1 Argument Declaration

Type-declaration of the Variables in the argument list are required as usual, but may include an additional information (`intent`), which indicates whether an argument is an *input*, `output`, or both.

```
INTENT(IN)     ! variable is not modified
INTENT(OUT)    ! variable cannot be referenced before a value is set
INTENT(INOUT)  ! an input variable can be modified
```

Fortran passes all arguments to subprograms by reference, so the variables passed as arguments to subprograms that are not intended to be modified may be accidentally changed.

The following subroutine `example` has two input arguments (real `x` and `y` intended as input), four output arguments (real `sums`, `diff`, `produ` and `divv` intended as output), and an integer input argument `n` is modified (intended as input as well as output) in the program.

```
SUBROUTINE example(n, x, y, sums, diff, prod, divv)
IMPLICIT none
REAL, INTENT(IN) :: x, y
REAL, INTENT(OUT):: sums, diff, prod, divv
INTEGER, INTENT(INOUT) :: n
sums = x + y
diff = x - y
prod = x * y
divv = x / y
n = n + 2
END SUBROUTINE
```

The `RETURN` statement in one-path exit procedures is optional.

1.11.2 FUNCTION Procedures

A function in fortran receives a number of values or arguments, uses them to perform calculations, and then return a *single result*. In Fortran, functions can be viewed in two categories: (i) *intrinsic functions*, those provided with the standard fortran library and (ii) *user-defined functions*, those prepared by the user. Functions in both categories are similar to subroutines, but they can be invoked within an expression and return a value that is used within the expression.

The intrinsic functions provides a collection of functions to perform common mathematical calculations, string manipulations, character manipulations, input/output, etc. On the other hand, the programmer can also write user-defined functions to do specific tasks, such as inverting a matrix, finding the real roots of a polynomial, and so on, that may be used numerous times in a program.

1.11.2.1 Function Declaration

The general structure of a `FUNCTION` in fortran 95 is given below:

```
return-type FUNCTION func_name (p1, p2, ..., pn)
       Argument and local variable declarations
       Executable statements
END FUNCTION func_name
```

where p1, p2,... , pn are the input argument list. return-type indicates whether or not the return value of the function is integer, real, double precision type, etc. The func_name is the a valid FUNCTION identifier. The *return-type* is the data type of the result returned to the caller. The return-type, function-name, and argument-list together are often referred to as the function header.

A function can also have its own internal (*local*) variables that are accessible only internally, i.e., its content is invisible to other functions. A function is only prepared once, and it is accessed and executed from the main function or any other functions whenever needed. Once a return-type function completes the specific task it is supposed to perform, at least one output value is returned to the calling function. There is no restriction on the number of functions. A function can be used with any other relevant programs and can be invoked or accessed many times in the same program.

Consider the following function subprogram, named avg, which computes the average of an array of real numbers of length n.

```
REAL FUNCTION avg(n, arr)
INTEGER, INTENT(IN) :: n
REAL, INTENT(IN), DIMENSION(n) :: arr
INTEGER :: i
REAL :: sums
sums = 0.0
DO i=1, n
   sums = sums + arr(i)
ENDDO
avg = sums/float(n)  ! convert integer n to real value
! float is an intrinsic function converting an integer to a real number
END FUNCTION avg
```

The input argument lists (indicated with INTENT) as well as the local variables (i and sums, are properly declared at the beginning of the function before they can be used. The sum of the array values, sum, are calculated with the aid of a DO loop and divided to n. The result is save on the function name avg.

The following (return-value) function finds the maximum of two real numbers a and b.

```
FUNCTION findmax(a, b) RESULT(max_value)
    implicit none
    REAL, INTENT(IN) :: a, b
    REAL :: max_value
    IF (a > b) THEN
        max_value = a
    ELSE
        max_value = b
    END IF
END FUNCTION findmax
```

Note that this float-type function returns a float value under any circumstances.

Functions are called by naming the function along with their entire argument list (i.e., referred to as function call) to which information is passed. Consider the following main program involving the use of `findmax`.

```
1  PROGRAM findmax_example
2      implicit none
3      REAL :: a, b, c, d, s1, s2
4
5      PRINT*, "Enter four numbers"
6      READ *, a, b, c, d
7      s1 = findmax(a, b)
8      s2 = findmax(c, d)
9      PRINT *, "The max. value is ", findmax(s1, s2)
10
11 CONTAINS
12
13     FUNCTION findmax(x, y) RESULT(max_value)
14       ...
15     END FUNCTION findmax
16
17 END PROGRAM findmax_example
```

In this program `findmax` is invoked in lines 6, 7, and 8. In line 6, `s1` is set to the maximum of `(a,b)`. In line 7, `s2` is set to the maximum of `(c,d)`. In line 8, the maximum of (`s1`,`s2`) is directly evaluated inside `printf` function.

1.11.3 Recursive FUNCTION

In Fortran language, a function in a program is allowed to call itself multiple times. A `RECURSIVE FUNCTION` is a function that can call itself repeatedly until a certain condition or task is met. That is why, a conditional construction is required to prevent the call go into an infinite loop to terminate the task.

Consider the recursive function $f_n(x) \leftarrow n + x * f_{n-1}(x)$ with $f_0(x) = x$. This function can be made a recursive function as follows:

```
1  REAL RECURSIVE FUNCTION func(n, x) RESULT(f)
2  IMPLICIT NONE
3  INTEGER, INTENT(IN) :: n
4  REAL, INTENT(IN) :: x
5  IF (n==0) THEN
6     f = x
7  ELSE
8     f = float(n) + x * func( n - 1, x)
9  ENDIF
10 RETURN
11 END FUNCTION func
```

In line 1, the type of the function is stated as `RECURSIVE` and the local intermediate variable (`f`) assigned for the result is `real`. The function has two arguments intended as input: integer `n` in line 1 and real `x` in line 3. The self-calling takes place in line 8 as `func( n - 1, x)`. The recursive function can be executed anywhere in the program or subprograms simply by invoking `func(n,x)`, which takes the value of `f` on return.

Bibliography

[1] CHAPMAN, S. J., *Introduction To Fortran 90/95*. McGraw-Hill's BEST–basic engineering series and tools, 1998.

[2] CHIVERS, I., SLEIGHTHOLME, Jane. *Introduction to Programming with Fortran*. Springer International Publishing, 2018.

[3] CLERMAN, N. S., SPECTOR, W. *Modern Fortran: Style and Usage*. Cambridge University Press, 2012.

[4] COUNIHAN, M. *Fortran 95*. Taylor & Francis, 1996.

[5] HAHN, B. H. *FORTRAN 90 for Scientists and Engineers.* Elsevier Science, 1994.

[6] METCALF, M., REID, J., COHEN, M. *Modern Fortran Explained*. Oxford Univ. Press, 2011.