Introduction to Fortran

for BSc/MSc Physics students

Sandeep K V
Assistant Professor of Physics
Mahatma Gandhi Govt. Arts College, Mahe
U.T. of Puducherry
www.mggacmahe.ac.in

sandeepkv@mggacmahe.ac.in

Visit : phusikos

Physics

Contents

Unit 1

Fortran Basics

1.1 Introduction

Fortran is a general purpose programming language, mainly intended for mathematical computations in e.g. engineering. Fortran is an acronym for **FOR**mula **TRAN**slation, and was originally capitalized as FORTRAN. However, following the current trend to only capitalize the first letter in acronyms, we will call it Fortran. Fortran was the first ever high-level programming language. The work on Fortran started in the 1950's at IBM and there have been many versions since. By convention, a Fortran version is denoted by the last two digits of the year the standard was proposed. Thus we have

- Fortran 66
- Fortran 77
- Fortran 90 (95)

Fortran 77 has been used extensively during the past 20 years. One of the main merits of Fortran 77 is its simplicity and the efficiency of compilers which give a good object code. A major advantage Fortran has is that it is standardized by ANSI and ISO. Consequently, if our program is written in ANSI Fortran 77, using nothing outside the standard, then it will run on any computer that has a Fortran 77 compiler. Thus, Fortran programs are portable across machine platforms.

Fortran 90 has many new features compared to Fortran 77. Even though the standard specifies that all Fortran 77 programs should be executable by Fortran 90 compilers without any change, Fortran 90 differs substantially from Fortran 77.

1.2 Fortran 90 Basics

A program can be regarded as a sequence of statements to solve a particular problem, and it is common to find that this sequence needs to be varied in practice. A Fortran program is

just a sequence of lines of text. The text has to follow a certain structure to be a valid Fortran program. Consider the following simple example:

```
PROGRAM circle
    IMPLICIT NONE
    REAL r, area
    ! This reads a real number r and prints
    ! the area of a circle with radius r.
    PRINT *, 'Give radius r:'
    READ *, r
    area = 3.14159*r*r
    WRITE *, 'Area = ', area
END PROGRAM circle
```

1.2.1 Program organization

A Fortran program generally consists of a *main program* (or driver) and possibly several *subprograms* (procedures or subroutines). The structure of a main program is:

```
program program_name

comments

    declarations

    statements

end program program_name
```


A FORTRAN program consists of the keyword **PROGRAM** followed by a name given to the program. The name can be upto 31 characters long. The first character must be a letter. In Fortran 90 no distinction is made between upper and lower case letters (case-insensitive). The other characters may be any letter or a digit 0, 1, 2 ... or 9 or an underscore character _'.
The program is made up of a number of lines; each line Is called a **statement**. Each statement is made up of variable names, operators, keywordse etc. The statements are executed sequentially. Originally, all Fortran programs had to be written in all upper-case letters. Most people now write lower-case since this is more legible, and so will we. Fortran is **not case-sensitive**, so "X" and "x" are the same variable.

1.2.2 Free Source Format

All source code can be written in *free format* in Fortran 90. The column position is irrelevant. The adopted rule is that all program lines start at column 1, with exceptions of statements

within control constructs such as IF blocks, DO loops and SELECT CASE constructs, which are intended by one TAB position to the right. But to ensure back-compatibility with Fortran 77, Fortran 90 allows *fixed source form also*. In Fortran 90(95) the code layout obeys the following rules, which describe the free format.

- statements can begin in any column
- multiple statements on a line are allowed; they have to be separated by semicolon ";"
- an exclamation mark "!" in any column is the beginning of a comment
- a statement can be continued on the following line by appending a "&" sign on the current line.

So, in the above example all codes can be started from the *first* column itself.

1.2.3 Comments

Comments are indicated by an exclamation mark. All text to the right of an exclamation mark is ignored by the compiler. Programmers use comments to help them remember how a program works. Use of appropriate comments in programs aids understanding and is good practice. A program may have any number of comments written anywhere. Thus the lines that begin with with a "!" are **comments** and have no purpose other than to make the program more readable for humans.

1.2.4 IMPLICIT NONE

In early Fortran compilers including FORTRAN 77 it was not mandatory to declare variables and it was implicitly assumed that any variable name starting with *I, J, K, L, M, N* was an integer variable and a variable name starting with any of the other letters of the alphabet real. It was found that not requiring the declaration of variables before their use was a source of serious logical errors in Fortran. As all FORTRAN 77 programs have to be accepted by Fortran 90 compilers to ensure compatibility, if a variable is by mistake, not declared in a Fortran 90 program then the compiler will not detect it as it will assume implicit typing rules. This means that the type of each variable, parameter or function result needs to declared explicitly in the declaration part of a program unit. This part must be preceded by the line: **IMPLICIT NONE**. Then if a variable is undeclared it will be detected by the compiler and an error message will be displayed.

1.2.5 Identifiers

An identifier is a name used to identify a variable, procedure, or any other user-defined item. A name in Fortran must follow the following rules:

- It cannot be longer than 31 characters
- It must be composed of alphanumeric characters (all the letters of the alphabet, and the digits 0 to 9) and underscore _
- First character of a name must be a letter

- Names are case-insensitive

1.2.6 Fortran Keywords

Keywords are special words, reserved for the programming language. These reserved words cannot be used as identifiers or names. Some examples for Fortran reserved keywords are : *program, stop, end, in, if, go to, else, use, save, type, then, print, read, write, close, open, do, public, private, implicit* etc.

1.3 Constants and Variables

A **constant** is a data object that is defined before a program is executed, and that does not change value during the execution of the program. A **variable** is a data object that can change value during the execution of a program.

Each Fortran variable in a program unit must have a unique name. Name being an identifier, all the rules applicable for identifier are also applicable to variable names.

1.3.1 Integer Constants and Variables

The integer data type consists of integer constants and variables. This data type can only store integer values (it cannot represent numbers with fractional parts). An integer constant is any number that does not contain a decimal point. Decimal point and/or comma are not allowed in an integer.

0, -23, +45855, 56, -359 etc are examples for integer constants.

An **integer variable** is a variable containing a value of the integer data type. Almost all Fortran compilers support integers with more than one length. Most PC compilers support 16-bit, 32-bit, and 64-bit integers. These different lengths of integers are known as different **kinds** of integers. Fortran has an explicit mechanism for choosing which kind of integer is used for a given value.

1.3.2 Real Constants and Variables

A real constant may be written in one or two forms called fractional form or the exponent form.
1.0, -5689.5, +356.0, -0.569 etc are examples for real constants.
If used, the exponent consists of the letter E followed by a positive or negative integer, which corresponds to the power of 10 used when the number is written in scientific notation. If the exponent is positive, the + sign may be omitted. The mantissa of the number should contain a decimal point and decimal points are not allowed in exponents.

A real variable is a variable containing a value of the real data type.

+1.0E-3	(1.0×10^{-3})
6.625E-34	(6.625×10^{-34})
0.15E+5	(0.15×10^{5})
-2.58E10	(-2.58×10^{10})

A quantity which may vary during program execution is called a variable and each variable has a specific storage location in memory where its numerical value is stored. The variable is given a name and the variable name is the "name tag" for the storage location. Identifiers used as variable names are explicitly typed as REAL or INTEGER by the following declaration which should appear at the beginning of a program before the variable names are used.

```
type_name :: variable name1, variable name2
```

For example

```
 REAL :: radius, area, mass, g
 INTEGER :: age, mobile
```

In Fortran 90 a value can be assigned to a variable name when it is declared which is the initial value of the variable.

```
 REAL :: radius=8.5, g=9.8
 INTEGER :: age=34, i=5
```


Fortran 90 allows to declare some variables as **double precision** which doubles the number of significant digits. The declaration used in this case is:

```
 REAL(KIND = 2) :: proton_mass
```

The shorter version of the REAL data type on any particular computer is known as **single precision**, and the longer version of the REAL data type on any particular computer is known as **double precision**. On most computers, a single-precision real value is stored in 32 bits and a double-precision real value is stored in 64 bits. Some 64-bit processors use 64 bit for single precision and 128 bits for double precision. The kind of a real value is specified in parentheses after the REAL, either with or without the phrase KIND=. A variable declared with a kind type parameter is called a parameterized variable. If no kind is specified, then the default kind of real value is used. The default kind may vary among different processors.

1.3.3 Character Constants and Variables

The character data type consists of strings of alphanumeric characters. A character constant is a string of characters enclosed in single (') or double (") quotes. The minimum number

of characters in a string is zero, while the maximum number of characters in a string varies from compiler to compiler. The characters between the two single or double quotes are said to be in a character context. Any characters representable on a computer are legal in a character context.

```
"What is your name ?"
'The value of g is'
"mass"
"36.589"
'(values)'
```

A character variable is a variable containing a value of the character data type.

There are no default names associated with the character data type as in REAL or INTEGER, so all character variables must be explicitly typed using the CHARACTER type declaration statement. The syntax is

```
CHARACTER (len=<len>) :: var1, var2
```

(len=<len>) is the number of characters in the variables, which is optional.

```
CHARACTER :: question
CHARACTER(10)::id
```


1.3.4 Named Constants

The best way to achieve consistency and precision throughout a program is to assign a name to a constant, and then to use that name to refer to the constant throughout the program. For example if we assign the name PI to the constant 3.141593, then we can refer to PI by name throughout the program, and be certain that we are getting the same value everywhere.

Named constants are created using the **PARAMETER** attribute of a type declaration statement.

```
type, PARAMETER :: name=value
```

For example

```
REAL, PARAMETER :: PI=3.141593, g=9.8
```

If the named constant is of type character, then it is not necessary to declare the length of the character string.

```
CHARACTER, PARAMETER :: LOGIN_ERROR=''INVALID USER_NAME OR PASSWORD !"
```

1.4 Arithmetic Operations

An arithmetic expression is a series of variable names and constants connected by arithmetic operation symbols, namely, addition, subtraction, multiplication, division and exponentiation. During the execution of the program the actual numerical values stored in variable names are used together with the operation symbols, to calculate the value of the expression.

1.4.1 Assignment Operator

Calculations are specified in Fortran with an assignment statement. Its general form is

```
variable_name = expression
```

The assignment statement calculates the value of the expression to the right of the equal sign, and assigns that value to the variable named on the left of the equal sign.

```
j = j +1
```

Take The current value stored in variable j is taken and 1 is added to it and then it is stored to variable j. Thus if initially j is 5, after executing the above expression, j becomes 6.

1.4.2 Arithmetic Operators

The expression to the right of the assignment operator can be any valid combination of constants, variables, parentheses, and arithmetic or logical operators. The standard arithmetic operators included in Fortran are

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

No two operators may occur side by side. The implied multiplication must be written explicitly in Fortran. Parentheses may be used to group terms whenever desired. When parentheses are used, the expressions inside the parentheses are evaluated before the expressions outside the parentheses.

```
2**((30-10)/2)
```

gives result as 1024.

1.4.3 Integer Arithmetic

Integer arithmetic is arithmetic involving only integer data and always produces an integer result. If the division of two integers is not itself an integer, the computer automatically truncates the fractional part of the answer. Because of this behavior, integers should never be used to calculate real-world quantities that vary continuously and should only be used for things that are intrinsically integer in nature, such as counters and indices.

```
3/4=0
6/2=3
10/3=3
6/4=1
```

1.4.4 Real Arithmetic

Real arithmetic (or floating-point arithmetic) is arithmetic involving real constants and variables. Real arithmetic always produces a real result.

```
3./2.=1.5
8.0/4.0=2.0
```


Fortran 90 allows mixing integers and reals in an expression.

```
1  + 1/4    gives 1
1. + 1/4    gives 1.
1 + 1./4    gives 1.25
```

Automatic type conversion also occurs when the variable to which the expression is assigned is of a different type than the result of the expression. For example,

```
INTEGER :: g_res
g_res = 2.35+8.0/3
```

stores the value to g_res as 4 as g_res is declared as INTEGER

1.4.5 Type Conversions

A variable declared REAL can be set equal to an integer expression and vice-versa. If an integer expression is assigned to a real variable name then the value of the integer expression is converted to a REAL number and is stored in the REAL variable name.

Function	**Type of Argument**	**Type of Result**	Comment
INT(x)	REAL	INTEGER	Integer part of x (x is truncated)
NINT(x)	REAL	INTEGER	Nearest integer to x (x is rounded)
CEILING(x)	REAL	INTEGER	Nearest integer above or equal to the value of x
FLOOR(x)	REAL	INTEGER	Nearest integer below or equal to the value of x
REAL(I)	INTEGER	REAL	Converts integer value to real

1.4.6 Hierarchy of Operations

The order in which operations are performed has a major effect on the final result of an algebraic expression. In a program the value of any expression is calculated by executing one arithmetic operation at a time. The order in which the arithmetic operations are executed in an expression is based on the rules of precedence of operators. The precedence of operators is: unary minus (-) FIRST, exponentiation (**) SECOND, multiplication (*) and Division (/) THIRD, Addition (+) and Subtraction (-) LAST.

The contents of all parentheses are evaluated first, starting from the innermost parentheses and working outward. All exponentials are evaluated, working from right to left.

1.5 Data Types in FORTRAN

Fortran provides **five intrinsic data types**. We can derive our own data types as well which are known as *derived data types*. The five intrinsic types are

1. Integer type
2. Real type
3. Complex type
4. Logical type
5. Character type

1.6 Operators in Fortran

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. Fortran has four types of operators: arithmetic, relational, logical, and character.

1.6.1 Arithmetic Operators

The syntax of using arithmetic operators are included in section 1.4

1.6.2 Relational Operators

Fortran 90 has six relational operators. Each of these six relational operators takes two expressions, compares their values, and yields **.TRUE.** or **.FALSE.**. COMPLEX values can only use $==$ and $/=$. Relational operators have Relational operators have lower priority than priority than arithmetic operators, and $//$.

Operator	Equivalent	Description
$==$	.eq.	Checks if the values of two operands are equal or not, if yes then condition becomes true
$/=$	.ne.	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true
$>$	.gt.	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true
$<$	.lt.	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true
$>=$	.ge.	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true
$<=$	.le.	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true

Table 1.1: Relational Operators

1.6.3 Logical Operators

A LOGIAL variable can only hold either .TRUE. or .FALSE. , and cannot hold values of any other type. There are 5 LOGICAL operators in Fortran 90: .NOT., .OR., .AND., .EQV. and .NEQV. .NOT. is the highest followed by is the highest, followed by .OR. and .AND., .EQV. and .NEQV. are the lowest. .NOT. is evaluated from is evaluated from right to left right to left..NOT. is higher than .NEQV.

Operator	Description
.and.	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.
.or.	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.
.not.	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.
.eqv.	Called Logical EQUIVALENT Operator. Used to check equivalence of two logical values.
.neqv.	Called Logical NON-EQUIVALENT Operator. Used to check non-equivalence of two logical values.

Table 1.2: Logical Operators

1.6.4 Character Operator

Fortran has only one character operator, the **concatenation operator** //. The concatenation operator cannot be used with arithmetic operators. Consider the following example


```
CHARACTER(LEN=8)    :: First = "Sandeep", Last  = "KV"
CHARACTER(LEN=10)   :: Full
Full = First//Last
WRITE(*,*) Full
```

The above code gives the result Sandeep KV

1.6.5 Operators Precedence in Fortran

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator. The precedence of operators is shown in the table. The operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Logical NOT and negative sign	.not. (-)	Left to right
Exponentiation	**	Left to right
Multiplicative	* /	Left to right
Additive	+ -	Left to right
Relational	< <= > >=	Left to right
Equality	== /=	Left to right
Logical AND	.and.	Left to right
Logical OR	.or.	Left to right

Table 1.3: Precedence of Operators

In the hierarchy of operations, **combinational logic operators are evaluated after all arithmetic operations and all relational operators have been evaluated.**

1.7 Decisions, Branches and Loops

The order in which the statements are written in a program is very important. Normally the statements are executed sequentially as written in the program. In other words when all the operations specified in a particular statement are executed, the statement appearing on the next line of the program is taken up for execution. This is known as normal flow of control. There are two broad categories of control statements: branches, which select specific sections of the code to execute, and loops, which cause specific sections of the code to be repeated.
Older Fortran versions had a fairly simple DO loop statement. Fortran 90 has enhanced these capabilities with DO WHILE constructs, the WHERE, CYCLE and EXIT statements. WHERE is a standalone loop type structure for use with arrays, and CYCLE and EXIT are statements that are used in conjunction with DO and DO WHILE loops. IF THEN combined with GO TO statements can do many operations. The DO WHILE and other DO constructs allow us to loop through certain portions of code many times without ever writing GO TO statements. Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed, if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

1.7.1 Decisions

An important part of any programming language are the conditional statements. The most common such statement in Fortran is the IF statement, which actually has several forms. Fortran provides the IF THEN ... END IF construct for decision making. Fortran 90 has three if-then-else forms.

IF-THEN - ELSE IF - END IF

```
IF (logical-expression-1) THEN
    statement sequence 1
ELSE IF (logical-expression-2) THEN
    statement sequence 2
ELSE IF (logical-expression-3) THEN
    statement sequence statement sequence 3
ELSE IF ((logical-expression-3) THEN
    statement sequence statement sequence 4
ELSE
    statement sequence ELSE
END IF
```

Logical expressions are evaluated sequentially (i.e., top-down). The statement sequence that corresponds to the expression evaluated to .TRUE. will be executed. Otherwise, the ELSE sequence is executed.

IF-THEN - END IF

An IF-THEN statement consists of a logical expression followed by one or more statements and terminated by an END IF statement. If the logical expression evaluates to true, then the block of code inside the IF-THEN statement will be executed. If logical expression evaluates to false, then the first set of code after the END IF statement will be executed.

```
IF (logical expression) THEN
   statement
END IF
```


IF-THEN ELSE END-IF

An IF-THEN statement can be followed by an optional else statement, which executes when the logical expression is false.

```
IF (logical expression) THEN
   statement(s)
ELSE
   other statement(s)
END IF
```

For example, check the code given below

```
PROGRAM QuadraticEquation
IMPLICIT NONE
REAL :: a, b, c, d, sd
COMPLEX ::cd
REAL :: root1, root2, rootu
```

```
PRINT *, 'Enter the value of a'
READ *, a
PRINT *, 'Enter the value of b'
READ *, b
PRINT *, 'Enter the value of c'
READ *, c
WRITE (*,*) "The equation is", a,"x^2 +", b, "x+",c,"=0"
d = b*b - 4.0*a*c
IF (d==0.0) THEN
    rootu = (-b)/(2.0*a)
    WRITE (*,*) 'The equation has one unique root', rootu
ELSE IF (d > 0.0) THEN
    sd = SQRT(d)
    root1 = (-b + sd)/(2.0*a) ! first root
    root2 = (-b - sd)/(2.0*a) ! second root
    WRITE (*,*) 'Roots are ', root1, ' and ', root2
ELSE
    sd = SQRT(-1*d)
    WRITE (*,*)  'The equation has no real roots!'
    WRITE (*,*) 'The complex Roots are', (-b)/(2*a),  "+",sd, " j/", 2*a, "and", &
    & (-b)/(2*a), "- ",sd,"j/", 2*a
END IF
END PROGRAM QuadraticEquation
```


1.7.2 Nested IF

We can use one if or else if statement inside another if or else if statement(s).

```
IF ( logical_expression 1) THEN
    STATEMENTS
    IF (logical_expression 2) THEN
        STATEMENTS
    ELSE
        STATEMENTS
    END IF
ELSE
    STATEMENTS
END IF
```

1.7.3 SELECT CASE

A **select case** statement allows a variable to be tested for equality against a list of values. Each value is called a **case**, and the variable being selected on is checked for each select case.

```
[name:] SELECT CASE (expression)
   CASE (selector1)
```

```
    STATEMENTS
    CASE (selector2)
    STATEMENTS
    CASE DEFAULT
    STATEMENTS
END SELECT [name]
```

We can specify a range for the selector, by specifying a lower and upper limit separated by a colon as **CASE (low:high)**. The logical expression used in a select statement could be logical, character, or integer (but not real) expression. When the variable being selected on, is equal to a case, the statements following that case will execute until the next case statement is reached. The case default block is executed if the expression in select case (expression) does not match any of the selectors.

```
PROGRAM TESTRESULT
IMPLICIT NONE

   ! local variable declaration
   integer :: marks
   PRINT *, "What is your marks in the paper ?"
   READ *, marks

   select case (marks)

      case (91:100)
         print*, "Excellent!"

      case (81:90)
         print*, "Very good!"

      case (71:80)
         print*, "Good!"

      case (61:70)
         print*, "First Class Marks Only"

      case (41:60)
         print*, "Try to Improve"

      case (:40)
         print*, "Failed"

      case default
         print*, "Invalid marks"

   end select
   print*, "Your marks in the paper is ", marks

```

```
END PROGRAM TESTRESULT
```

1.7.4 Loops

There may be a situation, when you need to execute a block of code several number of times. **Loops** are Fortran constructs that permit us to execute a sequence of statements more than once. There are two basic forms of loop constructs: while loops and iterative loops (or counting loops). *var* is the loop variable (often called the loop index) which must be integer.

Simple DO loop

The DO loop is used for simple counting.

```
DO label var = start, stop [,step]
   statements
label CONTINUE
END PROGRAM program_name
```


Here is a simple example that prints the cumulative sums of the integers from 1 through n (n is taken from user). The number 10 is a statement **label**. Typically, there will be many loops and other statements in a single program that require a statement label. Recall that column positions 1-5 are reserved for statement labels. The numerical value of statement labels have no significance, so any integers can be used, in any order. Typically, most programmers use consecutive multiples of 10. The variable defined in the do-statement is incremented by 1 by default. However, we can define the **step** to be any number but zero.

```
PROGRAM sumninteger
! Program to find the sum first n counting numbers
IMPLICIT NONE
INTEGER i, n, sum
PRINT*, "Enter the value of n"
READ *, n ! Ask n from user
sum = 0
DO 10 i = 1, n
    sum = sum + i
    WRITE(*,*) 'The sum of first', i, "counting numbers is", sum
10 CONTINUE
END PROGRAM sumninteger
```

Many Fortran compilers allow DO loops to be closed by the END DO statement. The advantage of this is that the statement label can then be omitted since it is assumed that an END DO closes the nearest previous do statement.

```
DO var = start, stop [,step]
   statements
END DO
END PROGRAM program_name
```

Try this code

```
PROGRAM FACTORIAL
IMPLICIT NONE
INTEGER :: n,i,nfact
! compute n factorial
WRITE (*,*) "Enter the value of n"
READ (*,*) n
nfact=1
DO i = 1, n
   nfact = nfact * i
   ! printing the value of n and its factorial
   print*, i, " ", nfact
END DO
END PROGRAM FACTORIAL
```

WHILE-DO or DO-WHILE loop

The syntax of WHILE-DO loop is

```
WHILE (logical expression)DO
    statements
END DO
```

or alternatively

```
DO WHILE (logical expression)
    statements
END DO
```

The program will alternate testing the condition and executing the statements in the body as long as the condition in the while statement is true.

```
program factorial
implicit none
integer :: nfact = 1
integer :: n, i=1
write (*,*) "Enter the value of n"
read (*,*) n
! compute factorials
print*, "The Number        Factorial"
```

```
do while (i <= n)
    nfact = nfact * i
    i = i + 1
    print*,  i, nfact
end do
end program factorial
```

IF THEN - GO TO - END IF

The above program can be written using **IF THEN - GO TO - END IF**

```
program factorial
implicit none
integer :: nfact = 1
integer :: n, i=1
write (*,*) "Enter the value of n"
read (*,*) n
! compute factorials
print*,  "The Number        Factorial"
10 IF (i <= n)  THEN
    nfact = nfact * i
    print*,  i, nfact
    i = i + 1
GO TO 10
END IF
end program factorial
```


We can use one or more loop construct inside any another loop construct and thus can form nested loops. It is possible to assign a name to a loop.

```
[name:] DO
Statement
Statement
Statement
IF ( logical_expr ) THEN
 CYCLE [name]
...
IF ( logical_expr ) THEN
 EXIT [name]
END DO [name]
```

1.7.5 Loop Control

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. These are two

additional statements that can be used to control the operation of while loops and counting DO loops: CYCLE and EXIT

Cycle

If the CYCLE statement is executed in the body of a DO loop, the execution of the current iteration of the loop will stop, and control will be returned to the top of the loop. The loop index will be incremented, and execution will resume again if the index has not reached its limit.

Exit

If the EXIT statement is executed in the body of a loop, the execution of the loop will stop and control will be transferred to the first executable statement after the loop. Both the CYCLE and EXIT statements work with both while loops and counting DO loops.

Stop

If we wish execution of the program to cease, we can insert a **stop** statement.

1.8 Character Data

Character data can be manipulated using character expressions. A character expression can be any combination of valid character constants, character variables, character operators, and character functions. There are two basic types of character operators: substring specifications and concatenation. Character functions are functions that yield a character result. The intrinsic data type **character** stores characters and strings. The length of the string can be specified by len specifier. If no length is specified, it is 1. Uppercase and lowercase letters are different inside strings.
A substring specification selects a portion of a character variable, and treats that portion as if it were an independent character variable. For example, if the variable str1 is a six-character variable containing the string 'abcdef', then the substring str1(2:4) would be a three-character variable containing the string 'bcd'. The concatenation operation has been discussed in section 1.6.4.
Declaring a string is same as other variables

```
program stringstest
implicit none
CHARACTER (len=10) :: first_name, last_name
character (len=20) :: full_name
first_name = "Sandeep   "
last_name = " K V     "
full_name = trim(first_name)//trim(last_name)
print *, full_name
print *, first_name(2:4)
```

```
10 end program stringstest
```

Character strings can be compared in logical expressions using the relational operators $==, /=, <, <=, >,$ and $>=$. The result of the comparison is a logical value that is either true or false. For instance, the expression 'abc' == 'abc' is true, while the expression 'abc' == 'abcd' is false.

LEN(str1) : Returns length of str1 in characters

LEN_TRIM(str1) : Returns length of str1, excluding any trailing blanks

TRIM(str1) : Returns str1 with trailing blanks removed

The function adjustl takes a string and returns it by removing the leading blanks and appending them as trailing blanks. The function adjustr takes a string and returns it by removing the trailing blanks and appending them as leading blanks.

1.9 Basic & Formatted I/O

We can read data from keyboard using the **read*** statement, and display output to the screen using the **print*** statement, respectively. This form of input-output is **free format** I/O. The basic syntax is

```
1 READ(*,*) var1, var2
2 PRINT *, var1, var2
3 WRITE(*,*) var1, var2
```


A **format** may be used to specify the exact manner in which variables are to be printed out by a program. In general, a format can specify both the horizontal and the vertical position of the variables on the paper, and also the number of significant digits to be printed out. Format specification defines the way in which formatted data is displayed. It consists of a string, containing a list of edit descriptors in parentheses.

```
1 program formattedpi
2 implicit none
3 real pi
4 pi = 3.141592653589793238
5 print "(f6.3)", pi
6 print "(f10.7)", pi
7 print "(f20.12)", pi
8 print "(e16.4)", pi/100
9 end program formattedpi
```

There are many different format descriptors. They fall into four basic categories:

- Format descriptors that describe the vertical position of a line of text.

- Format descriptors that describe the horizontal position of data in a line.
- Format descriptors that describe the output format of a particular value.
- Format descriptors that control the repetition of portions of a format.

Descriptor	**Description**	**Example**
I	This is used for integer output. This takes the form 'rIw.m' . Integer values are right justified in their fields. If the field width is not large enough to accommodate an integer then the field is filled with asterisks.	print "(3i5)", n,m
F	This is used for real number output. This takes the form 'rFw.d'. Real values are right justified in their fields. If the field width is not large enough to accommodate the real number then the field is filled with asterisks.	print "(f10.4)",pi
E	This is used for real output in exponential notation. The 'E' descriptor statement takes the form 'rEw.d' . Real values are right justified in their fields. If the field width is not large enough to accommodate the real number then the field is filled with asterisks. To print out a real number with three decimal places a field width of at least ten is needed. One for the sign of the mantissa, two for the zero, four for the mantissa and two for the exponent itself. In general, w $\geq$ d + 7.	"print "(e10.5)",12345678.0 gives 0 .12346E+08 print "(e10.3)",12345678.0 gives 0.123E+08 (Note the rounding off of figure)
ES	This is used for real output (scientific notation). This takes the form 'rESw.d'. Scientific notation has the mantissa in the range 1.0 to 10.0 unlike the E descriptor which has the mantissa in the range 0.1 to 1.0. Real values are right justified in their fields.	print "(es10.3)",12345678.0 gives 1.235E+07
A	This is used for character output. This takes the form 'rAw'	print "(a10)", str where str is a character
X	This is used for space output. This takes the form 'nX' where 'n' is the number of desired spaces.	print "(6x, a10)", str where str is a character
/	Slash descriptor – used to insert blank lines. This takes the form '/' and forces the next data output to be on a new line.	print "(/,6x, a10)", str where str is a character

Table 1.4: Descriptors

c	Column number
d	Number of digits to right of the decimal place for real input or output”
m	Minimum number of digits to be displayed”
n	Number of spaces to skip
r	Repeat count - the number of times to use a descriptor or group of descriptors
w	Field width - the number of characters to use for the input or output

Table 1.5: Symbols & description used in descriptors

```
program formattedprint
implicit none

   real :: a = 1.2786456e-9, b = 0.1234567e3
   integer :: n = 300789, l = 45, i = 2
   character (len=25) :: str="Physics is Beutiful"

   print "(i6)", l
   print "(i6.3)", l
   print "(3i10)", n, l, i
   print "(i10,i3,i5)", n, l, i
   print "(a10)",str
   print "(f12.3)", b
   print "(e12.4)", a
   print '(/,3x,"n = ",i6, 3x, "d = ",f7.4)', n, b

end program formattedprint
```


gives the following result

```
45
045
300789 45  2
300789 45  2
Physics is
123.457
0.1279E-08

n = 300789 d = *******
```

The **format** statement allows us to mix and match character, integer and real output in one statement.

```
program StudentDetails
implicit none
```

```
3
4     character (len = 20) :: name
5     integer :: id
6     real :: rank
7     name = 'Sandeep K V'
8     id = 210001
9     rank = 93.8758
10
11    print *,' The rank details are'
12
13    print 50
14    50 format (7x,'Name:', 7x, 'Id:', 1x, 'Rank:')
15
16    print 100, name, id, rank
17    100 format(1x, a, 2x, i6, 2x, f5.2)
18
19 end program StudentDetails
```

gives the following result

```
1 The rank details are
2 Name:              Id:       Rank:
3 Sandeep K V        210001    93.88
```


1.10 References

1. Stephen J. Chapman, Fortran for Scientists and Engineers, Mc Graw Hill Education, 2018
2. M. Metcalf and J. Reid, Fortran 90/95 Explained, Oxford University Press, Oxford U.K., 1996
3. V. Rajaraman, Computer Programming in Fortran 90 and 95, Prentice-Hall of India, 2013