CHAPTER 1

INTRODUCTION TO COMPUTING

1.1 Introduction

FORTRAN, now is more than 40 years old and is a computer language used throughout the world for writing programs for solving problems in science and engineering. Since FORTRAN was created in the 1950’s, it has undergone a number of modifications that have made it a very powerful yet easy-to-use language. The most popular standard FORTRAN version was FORTRAN77. Due to the development in other programming languages, it became clear that some of the new concepts and techniques introduced in other languages should be supported in FORTRAN as well. Thus, another FORTRAN standard was prepared and eventually approved. This FORTRAN standard, called FORTRAN 90, included all of the major features of earlier versions so that the largest investment made in FORTRAN software is protected. Millions of lines of FORTRAN77 code will continue to execute properly in FORTRAN 90. Furthermore, FORTRAN 90 has many features that make it a truly modern language, and they include:

- replacement of the old fixed format for programs with a free form
- longer names for projects, making programs easier to read
- new control constructs for selective and repetitive execution
- new kinds of subprograms to facilitate modular programming
- powerful new array-processing mechanisms
- programmer-defined data types
- dynamic memory allocation and pointers for constructing complex data structures.

1.2 Computer Organization

The main components of a computer machine are:
- **Central Processing Unit** (CPU): it is the heart of the machine, and it controls all the operations and performs all the arithmetic and logic operations.
  - **Memory Unit**: this unit stores instructions and data.
  - **Control Unit**: This unit fetches instructions of the Memory Unit, decodes them and directs the system to execute the operations indicated by the instructions.
  - **Arithmetic-Logic Unit (ALU)**: This unit is part of the CPU and carries out arithmetic and logical operations.

- The **Memory Unit** consists of several components: one of these components is used to store the instructions and data of the programs being executed and has many names, including internal, main, primary, and **random access memory** (RAM). A second component is a set of special high speed memory locations within the CPU, called registers. Values that are stored in registers can typically be accessed thousands of times faster than values stores in RAM. RAM registers is volatile memory components, that is, information stores in these components is lost if the power to the computing system is shut off (either intentionally or accidentally). **Read-only-memory** (ROM) is nonvolatile memory used to store critical information, such as startup instructions, which is too important to lose.

  Figure 1 shows a sketch of the basic computer components.

Other peripherals are used to transmit instructions, data and computed results between the user and the CPU. These are the **input/output devices**, which have several forms, such as terminals, scanners, voice input devices, printers and plotters. The function of these input/output devices is to convert information from an external form understandable to the user to a form that can be processed by the computer system and vice versa.
1.3 Computer Language:

In the very early age of computers, computer programs were written using the *machine language*. The machine language words are some sort of strings having two parts: opcode and operand and each consist of either 0 or 1 numerals. Later, it became possible to write programs in *assembly language*, which uses names in place of numeric opcodes and variable names in place of numeric addresses. Nowadays, most computer programs are written in a *high-level-language* such as FORTRAN, and a compiler translates each statement in the
program into a sequence of basic machine (or assembly language) language instructions.

FORTRAN word is an abbreviation for “FORmula TRANslator” and was one of the first high-level languages to gain acceptance worldwide.

1.4 Programming and Problem Solving

Programming is an art that requires a great deal of creativity and imagination. At the same time, it is a science that uses certain techniques and methodologies to solve problems. A minimum of five steps can be identified in the program development process:

1. Problem analysis and specification
2. data organization and algorithm design
3. program coding
4. execution and testing
5. program maintenance

See an illustrative example (from textbook) in the appendix, showing how to apply these steps.
CHAPTER 2
BASIC FORTRAN

2.1 Introduction

FORTRAN provides five basic data types:

INTEGER
REAL
COMPLEX
CHARACTER
LOGICAL

The first three are numeric types used to store and process various kinds of numbers; the character type is used to store and process strings of characters; and the logical type issued to store and process logical data values (.FALSE. and .TRUE.). In this section we describe the most commonly used forms of the FORTRAN integer, real, and character data types and tell how to declare constants and variables of these types.

2.2 Data Types, Constants, and Variables:

Integers

An integer is a whole number (positive, negative, or zero) and may be represented in FORTRAN by a string of digits that does not contain commas or a decimal point; negative integer constants must be preceded by a minus sign, but a plus sign is optional for nonnegative integers. Thus 0 , 137 , -2516 , +17745 are valid integer constants, whereas the following are invalid for the reasons indicated:
9,999 commas are not allowed in numeric constants
16.0 integer constants may not contain decimal points
--5 only one algebraic sign is allowed
7- the algebraic sign must precede the string of digits.

**Reals**

Real is another data type and it can be represented as ordinary decimal numbers or in exponential notation. When representing a real constant using decimal numbers, a decimal point must be present, but no commas are allowed. Negative real constants must be preceded by a minus sign, but the plus sign is optional for nonnegative reals. Thus,

1.234
-.01536
+.56473

are valid real constants, whereas the following are invalid for the reasons indicated:

12,345 (Commas are not allowed in numeric constants)
63 (Real constants must contain a decimal point)

The exponential representation of a real constant consists of an integer or decimal number representing the mantissa or fractional part, followed by an exponent written as the letter E with an integer constant following. For example, the real constant 337.456 may also be written as

3.37456E2

which means $3.37456 \times 10^2$, or it may be written in a variety of other forms, such as

0.337456E3
337.456E0
Character Strings

Character constants, also called strings, are sequences of symbols from the FORTRAN character set. The ANSI standard character set for FORTRAN is given in the table below:

Table 1: FORTRAN character set

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, …., 9</td>
<td>digits</td>
<td>:</td>
<td>Colon</td>
</tr>
<tr>
<td>A, …., Z</td>
<td>uppercase letters</td>
<td>=</td>
<td>Equal sign</td>
</tr>
<tr>
<td>a, …., z</td>
<td>lowercase letters</td>
<td>!</td>
<td>Exclamation mark</td>
</tr>
<tr>
<td>'</td>
<td>Apostrophe (single quote)</td>
<td>&amp;</td>
<td>Ampersand</td>
</tr>
<tr>
<td>&quot;</td>
<td>double quote</td>
<td>$</td>
<td>Dollar sign</td>
</tr>
<tr>
<td>(</td>
<td>Left parenthesis</td>
<td>;</td>
<td>Semicolon</td>
</tr>
<tr>
<td>)</td>
<td>Right parenthesis</td>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>+</td>
<td>Plus sign</td>
<td>%</td>
<td>Percent symbol</td>
</tr>
<tr>
<td>-</td>
<td>Minus sign</td>
<td>?</td>
<td>Question mark</td>
</tr>
<tr>
<td>/</td>
<td>Slash</td>
<td>,</td>
<td>Comma</td>
</tr>
<tr>
<td>blank</td>
<td>Blank or space</td>
<td>.</td>
<td>Period</td>
</tr>
</tbody>
</table>
Identifiers

Identifiers are names used to identify programs, constant, variables, and other entities in a program. In standard FORTRAN, identifiers must begin with a letter, which may be followed by up to 30 letters, digits, or underscores. Thus,

Mass
Rate
Velocity
Speed_of_Light

Are valid FORTRAN identifiers, but the following are invalid for the reasons indicated:

R2-D2  (Only letter, digits, and underscores are allowed in identifiers)
6Feet  (Identifiers must begin with a letter)

It is a good practice to always use meaningful identifiers that suggest what they represent. FORTRAN90 makes no distinction between upper case and lower case (except in character constants). For instance, Velocity, VELOCITY, VElocity represent the same valid identifier.

Variables

When a variable is used in a FORTAN program, the compiler associated it with a memory location. The value of a variable at any time is the value stored in the associated memory location at that time. Variable names are identifiers; thus, they must follow the rules for identifiers. In FORTRAN programming the type of any variable used has to be declared. This can be done using type statements. The simplest forms of the type statements used to declare integer variables and real variables are:

INTEGER :: list
REAL :: list
A comma must separate the names in the list. For instance, the statements:

 INTEGER :: NumValues, Factorial, Sum
 REAL :: Mass, Velocity

 Declare NumValues, Factorial, and Sum as integer variables and Mass and Velocity as real variables.

 One way to declare a character variable is the use of the following statement:

 CHARACTER (LEN = n ) : : list

 Or in a simpler form:

 CHARACTER(n) : : list

 In which n is an integer constant specifying the length of character constants to be assigned to the variables in the list. The length specifier may be omitted, in which case the length of the values for the variables in the list is 1. For instance, the type statement

 CHARACTER (LEN = 15) : : FirstName, LastName

 Or

 CHARACTER(15) : : FisrtName, LastName

 declares the variables FirstName and LastName to be character variables and specifies that the length of any character value assigned to one of these variables is 15. The statement:

 CHARACTER :: initial

 declares Initial to be a character variable with value of length 1.

 To declare character variables with different lengths on the same statement a length specifier of the form *n may also be attached to any of the individual variables in the list of a CHARACTER statement. In this case, this length specification for that variable overrides the length specification given for the list. The statement:

 CHARACTER(15) : : FirstName, LastName*20, Initial*1, Street, City
declares that the character variables FirstName, LastName, Initial, Street, and City are character variables and that the length of values for FirstName, Street, and City is 15, the length of values for LastName is 20, and the length of values for Initial is 1.

Variables of a given type must be used in a manner that is appropriate to that data type, since the program may fail to execute correctly otherwise. It is therefore important to specify the correct type of each variable used in a program.

**IMPLICIT NONE Statement**

FORTRAN Implicit naming convention is applied to any variable whose type is not explicitly declared in a type statement. The convention says that: any undeclared identifier whose name begins with I, J, K, L, M, or N or their lowercase equivalents will be typed as integer, and all other will be types as real. FORTRAN 90 provides the statement IMPLICIT NONE to cancel this naming convention. It should be used in every program to guard against errors caused by implicit typing of a variable to have a type different than what was intended. When using this statement all the types of all named constants and variables (and functions) must be specified explicitly in type statements. Otherwise, when using any undeclared variable (or function) without declaring it, it will be considered an error by the compiler.

**Variable Initialization**

As all variables in FORTRAN are initially undefined, initial values can be assigned to variables (at compile time) in their declarations. For example, to initialize the values for variables W, X, Y, and Z to 1.0, 2.5, 7.73, and -2.956, respectively, one could use the following statement:

```
REAL :: W = 1.0, X = 2.5, Y = 7.73, Z = -2.956
```
Named Constants: the PARAMETER Attribute

When some constants occur so often, they are given names. For instance, the name “pi” is commonly given to the constant 3.14159… and the name “e” to the base 2.71828… of natural logarithms. FORTRAN allows the programmer to specify that an identifier names a constant by including a PARAMETER attribute in the declaration of that identifier using the form:

*Type-specifier, PARAMETER :: list*

Examples:

```
INTEGER, PARAMETER :: Limit = 50

REAL, PARAMETER :: Pi = 3.141593, TwoPi = 2.0 * Pi

CHARACTER(2), PARAMETER :: Units = “cm”
```

The last declaration example can also be written

```
CHARACTER(*), PARAMETER :: Units = “cm”
```

In the last statement, the asterisk (*) is an assumed length specifier indicating that the length of the named constant (Units) being declared is to be the length (2) of the string constant (“cm”) with which it is being associated. The names Limit, Pi, TwoPi, and Units can be used anywhere in the program that the corresponding constant value can be used. For example, a statement such as:

```
XCoordinate = Rate * Cos(TwoPi * Time)
```

is equivalent to

```
XCoordinate = Rate *Cos(6.283186 * Time)
```

However, the first form is more readable and convenient to use.
2.3 Numeric Operations

The following characters are used in FORTRAN as denoted:

*  multiplication,
-  subtraction
+  addition
/  division
**  exponentiation

For example, the equality $B^2 - 4 AC$ is written as

\[ B \times 2 - 4 \times A \times C \]

When two constants or variables of the same type are combined using one of the four basic arithmetic operations (+, -, *, /), the result has the same type as the operands, for example, the sum of the integers 3 and 4 is the integer 7, whereas the sum of the real numbers 3.0 and 4.0 is the real number 7.0. This distinction may seem unimportant until one considers the division operation. Division of the real constant 9.0 by the real constant 4.0

\[ 9.0 / 4.0 \]

produces the real quotient 2.25, whereas dividing the integer 9 by the integer 4,

\[ 9 / 4 \]

produces integer quotient 2, which is the integer part of the real quotient 2.25. Similarly, the operation

\[ 1.0 / 2 \]

yields 0.5, whereas the operation

\[ 1 / 2 \]

yields 0.
**Mixed-Mode Expressions**

It is possible to combine integer and real quantities using arithmetic operations. These expressions are called mixed-mode expressions. When an integer quantity is combined with a real one, the integer quantity is converted to its real equivalent and the result is of real type. The following examples illustrate how the mixed-mode operations are evaluated:

\[
\begin{align*}
1.0 / 4 & \rightarrow 1.0 / 4.0 \rightarrow 0.25 \\
3.0 + 8 / 5 & \rightarrow 3.0 + 1 \rightarrow 4.0 \\
3 + 8.0 / 5 & \rightarrow 3 + 1.6 \rightarrow 4.6
\end{align*}
\]

The last two examples show the use of mixed-mode operations is considered as a poor practice.

Note the following two expressions

\[
\begin{align*}
2.0 ** 3 & \rightarrow 2.0 * 2.0 * 2.0 \rightarrow 8.0 \\
(-4) ** 2 & \rightarrow (-4) * (-4) \rightarrow 16
\end{align*}
\]

However, the expression

\[
2.0 ** 3.0
\]

is equivalent to

\[
e^{3.0 \ln(2.0)}
\]

which will not be equal to 8.0 due to rounding errors associated with storing real numbers and because the exponentiation and logarithm functions produce only approximate values. It is to be noted that raising a negative number to a real power is undefined because the logarithms of negative values are not defined. Thus, \((-4.0) ** 2.0\) is undefined, even though \((-4.0) ** 2\) is evaluated as \((-4) * (-4) = 16.0\). Therefore, a real exponent should never be used in place of an integer exponent.
More Examples

\[ 7.0 ** (1.0 / 2.0) \Rightarrow 7.0 ** (0.5) \]

\[ 7.0 ** (1 / 2) \Rightarrow 7.0 ** 0 = 1.0 \]

**Priority Rules**

1. All exponentiations are performed first; consecutive exponentiations are performed from right to left.
2. All multiplications and divisions are performed next, in the order in which they appear from left to right.
3. The additions and subtractions are performed last, the order in which they appear from left to right.

The following examples illustrate these priorities:

\[ 2 ** 3 ** 2 = 2 ** 9 = 512 \]
\[ 10 – 8 – 2 = 2 – 2 = 0 \]
\[ 10 / 5 * 2 = 2 * 2 = 4 \]
\[ 2 + 4 / 2 = 2 + 2 = 4 \]
\[ 2 + 4 ** 2 / 2 = 2 + 16 / 2 = 2 + 8 = 10 \]

Parentheses can be used to modify the standard order of evaluation, for example:

\[ (5 * (11 – 5) ** 2) * 4 + 9 = (5 * 6 ** 2) * 4 + 9 = 180 * 4 + 9 = 720 + 9 = 729 \]
2.4 The Assignment Statement

Values are assigned to variables using the assignment statement, which has the form:

\[ \text{variable} = \text{expression} \]

Example

REAL :: XCoordinate, YCoordinate
INTEGER :: Number, Tern
XCoordinate = 5.23
YCoordinate = 25.0
Number = 17
Term = Number / 3 + 2
XCoordinate = 2.0 * XCoordinate

If an integer-valued expression is assigned to a real variable, the integer value is converted to a real constant and then assigned to the variable.

Example:

INTEGER :: N
REAL :: Alpha, Beta
N = 9
Alpha = 3
Beta = (N + 3) / 5

The value of Beta in this case would be 2.0.

In case a real-valued expression is assigned to an integer variable, the fractional part of the real value is truncated, and the integer part is assigned to the variable.
Example:

```fortran
INTEGER :: I, Mu, Kappa
REAL :: X
X = 5.75
I = 3.14159
Kappa = X / 2.0
Mu = 1.0 / X
```

The integer values 3, 2 and 0 will be assigned to the variables I, Kappa and Mu, respectively. The following example shows how to truncate the fractional part of a real-valued expression:

```fortran
Kappa = INT( X / 2.0)
```

An assignment statement may also be used to assign a value to a character, as shown in the following example:

```fortran
CHARACTER(5) :: String, Truncated, Padded*10
String = "alpha"
Padded = "particle"
Truncated = "temperature"
```

These assignment statements will assign the value “alpha” to `String`, the value “particle” to the variable `Padded` and the value “tempe” to the variable `Truncated`.

### 2.5 Numeric Functions

FORTRAN provides some useful functions as a convenient mean for dealing with some common functions that are encountered often. The following table shows some FORTRAN functions:
Table 2.2 Some FORTRAN functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS(x)</td>
<td>Absolute value of x</td>
</tr>
<tr>
<td>COS(x)</td>
<td>Cosine of x radians</td>
</tr>
<tr>
<td>EXP(x)</td>
<td>Exponential function</td>
</tr>
<tr>
<td>INT(x)</td>
<td>Integer part of x</td>
</tr>
<tr>
<td>FLOOR(x)</td>
<td>Greatest integer less than or equal to x</td>
</tr>
<tr>
<td>FRACTION(x)</td>
<td>M of x part (mantissa) of x</td>
</tr>
<tr>
<td>LOG(x)</td>
<td>Natural logarithm of x</td>
</tr>
<tr>
<td>MAX(x1, ……, x_n)</td>
<td>Maximum of x1, ……, x_n</td>
</tr>
<tr>
<td>MIN(x1, …. ,x_n)</td>
<td>Minimum of x1, …….., x_n</td>
</tr>
<tr>
<td>MOD(x , y)</td>
<td>x(mod y): x – INT(x/y) * y</td>
</tr>
<tr>
<td>NINT(x)</td>
<td>x rounded to nearest integer</td>
</tr>
<tr>
<td>REAL(x)</td>
<td>Conversion of x to real type</td>
</tr>
<tr>
<td>SIN(x)</td>
<td>Sine of x radians</td>
</tr>
<tr>
<td>SQRT(x)</td>
<td>Square root of x</td>
</tr>
<tr>
<td>TAN(x)</td>
<td>Tangent of x radians</td>
</tr>
</tbody>
</table>

Examples on Numeric Functions

\[
\text{SQRT}(7.0)
\]
\[
\text{SQRT}(B ** 2 – 4 * A * C)
\]
\[
X ** (1.0 / \text{REAL}(N))
\]
\[
\text{REAL}(\text{NUM}) ** (1.0 / \text{REAL}(N))
\]
Character Operations

The concatenation operation // is used to combine two character values. For example:

“centi” // “meters”

produces the string

“centimeters”

and if SquareUnit is a character variable whose value is

“square”

then

SquareUnit // “centi” // “meters”

yields the string

“square centimeters”

Substrings can be extracted from the value of a character variable or character constant by specifying the variable or constant followed by the positions of the first and last characters of the substring, separated by a colon (:) and enclosed in parentheses. For example, if the character variable Units has the value

“centimeters”

then

Units (4:7)

has the value

“time”

If the initial position is not specified, it is assumed to be the last position in the value of the character variable as shown below:

CHARACTER(15) :: Course

Course = “Engineering”
Then

Course (: 6)

has the value

“Engine”

and the value of

Course (8 : )

is

“ringbbbb”

2.6 Input / Output

FORTRAN provided two types of input/output statements: list-directed and formatted. The list directed method is simpler and has the following simplest form:

PRINT * , output-list

Or

WRITE (* , *) output list

Examples:

PRINT *, “At time”, Time, “seconds”

PRINT *, “the vertical velocity is”, Velocity, “m / sec”

PRINT *, “and the height is”, Height, “meters”

Execution of the above statements will produce output similar to the following:

At time 4.5000000 seconds

the vertical velocity is 45.8700752 m/sec

and the height is 4.0570767E+02 meters
Note that each PRINT statement produces a new line of output. The exact format and spacing used to display these values are compiler-dependent.

The PRINT and WRITE statement may be used to produce a blank line of output as shown in the following example:

PRINT *

Or

WRITE (*, *)

The simplest form of list-directed input statement is

READ *, input-list

Or

READ (*, *) input-list

Usually values will be obtained from the keyboard and assigned to the variables in the input list according to the following rules:

1. The entries in each line of input data must be constants and of the same type as the variables to which they are assigned. However, an integer value may be assigned to a real variable, with automatic conversion taking place.

2. Consecutive entries in a line of input data must be separated by commas or by one or more spaces.

3. If the line of input data contains fewer entries than the variables in the input list, successive lines of input are processed until values for all variables in the list have been obtained.

4. If more entries are available in a line of input data than there are variables in the input list, the first data values are used, and all remaining values are ignored.

5. A new line of data is processed each time a READ statement is executed.
Example:
The following line of input data

100.0, 90.0, 4.5

can be used to respond to the following statement:

READ*, InitialHeight, InitialVelocity, Time

That means 100.0 value will be assigned to the variable InitialHeight, the value 90.0 will be assigned to the variable InitialVelocity and the value 4.5 will be assigned to the variable Time. Spaced can also be used as separators in place of the commas so that the line of input data will be as follows:

100.0 90.0 4.5

or more than one line of data could be used:

100.0, 90

4.5

Character values can also be read using list-directed input, but they must be enclosed in single or double quotes if any of the following is true:

i) The character value extends over more than one line.

ii) The character value contains blanks, commas, or slashes

iii) The character value begins with an apostrophe, a double quote, or a string of digits followed by an asterisk.

2.7 Program Composition and Format

The general form of a FORTRAN computer program is:

1. heading
2. Specification part
3. Execution part
4. Subprogram part
5. END PROGRAM statement

The **program heading** has the form

```
PROGRAM name
```

Where `name` is a legal FORTRAN identifier. This statement marks the beginning of the program and gives it a name. After the heading there should be some **opening documentation** that explains the purpose of the program, clarifies the choice of variable names, and provides other pertinent information about the program. A reclamation mark (!) must precede this documentation comments.

The **specification part** of a program must appear next. The first statement in this part should be:

```
IMPLICIT NONE
```

This is to cancel the naming convention and ensure that all variables and constants are explicitly declared. These declarations are type statements such as:

```
REAL :: InitialHeight, Height, InitialVelocity, Velocity, &
Acceleration = -9.80665
```

whose purpose is to specify the type of each of the variables used in the program.

FORTRAN statements are classified as either executable or non-executable. Non-executable statements provide information that is used during compilation of a program, but they do not cause any specific action to be performed during execution. For example, PROGRAM and type statements are non-executable statements. However, statements that specify actions to be
performed during execution of the program such as assignment statements and input/output statements are placed in the **execution part** of a program.

The last statement in every program must be the **END PROGRAM** statement. This statement indicates to the compiler the end of the program. It also halts execution of the program. Execution can also be terminated with a **STOP statement** of the form

```
STOP
```

Or

```
Stop constant
```

where `constant` is an integer constant or a character constant. The constant will be displayed when execution is terminated by a STOP statement of the second form, but the precise form of the termination message depends on the compiler.

**Program Format**

In FORTRAN90 programming the main rules that must be followed are:

- A line may have a maximum of 132 characters.
- A line may contain more than one statement, provided the statements are separated by semicolons (;). It is good practice, however, to consider at most one statement per line.
- An ampersand (&) must be placed at the end of each line that is to be continued to the next line. At most 39 continuation lines are permitted.
- If a character string must be continued from one line to the next, an ampersand must be placed at the end of the line containing the first part of the string, and another ampersand must be placed before the first character of the continuation of the string: for example:
PRINT *, “Enter the initial height (m) and &
& the initial velocity (m/sec): “

- Any characters following an exclamation mark (!)—except within a string constant—and running to the end of the line form a comment. For example,

\begin{verbatim}
INTEGER :: Number               ! Number of data values read
\end{verbatim}

- If a statement requires a statement label, this label must precede the statement and must be separated from it by at least one blank. Statement labels must be integers in the range 1 through 99999.

2.8 Types of Errors

There are different Errors that a programmer may face during compilation or running of a computer program. Errors in the program’s syntax, such as incorrect punctuation or misspelled key words, will be detected during compilation. Such errors are called \textbf{syntax errors} or \textbf{compile-time errors} and usually make it impossible to complete the compilation and execution of the program. Other errors, such as an attempt to divide by zero in an arithmetic expression, may not be detected until execution of the program has begun. Such errors are called \textbf{run-time errors}.

There are other errors that are more subtle and difficult to identify. These are \textbf{logic errors} that arise in the design of the algorithm or in the coding of the program that implements the algorithm.
CHAPTER 3

INPUT/OUTPUT

There are two types of input/output statements in Fortran: list-directed (free formatted) and formatted.

3.1 List-directed (free formatted) input/output

It was discussed in details in section 2.6.

3.2 Formatted input/output

3.2.1 Formatted input

The formatted input makes it possible to the programmer to read data that has a predetermined form.

General Form: \texttt{read format-specifier, input-list}

Where input-list is a single variable or a list of variables separated by commas; and the format-specifier specifies the format in which the values for the items in the input list are to be entered. The format-specifier may be:

1. * (an asterisk)
   - It indicates list-directed input (free format).
   - \texttt{Example: read *,a,j}

2. A character constant or a character variable (or expression or array) whose value specifies the format for the input.
   - \texttt{Example: read '(*3,i4,i7)',j,k,l}

3. The label of a FORMAT statement
   - In this case, the formatting information is supplied by a FORMAT statement whose label is specified. This statement has the form:
   \begin{tabular}{|l|l|}
   \hline
   label & FORMAT (list of format descriptors) \\
   \hline
   \end{tabular}
Where label is an integer in the range 1 through 99999.
There are many format descriptors that may be used in format specifiers. A list of the most useful
descriptors is given in Table 5.1(page 129 of the text book). The most common of these are
described in the following:

**X- Descriptor**

It skips positions on the data line

General Form: \( nX \)

**I- Descriptor**

It is used when reading a value into an integer variable.

General Form: \( nIw \)

Where \( w \) indicates the width of the field (the number of characters to be read) and \( n \) is the
repetition indicator.

**Example:**

\[
\begin{align*}
\text{read} & \quad 10,m,n \\
10 & \quad \text{format}(5x,i5,i7)
\end{align*}
\]

**F- Descriptor**

It is used when reading a value into a real variable.

General Form \( Fw.d \)

Where \( w \) represents the total number of positions to use from the data line and \( d \) represents the
number of decimal positions.
• If there is a decimal point included in the w positions, the value will be stored as it was entered, regardless of what value has been given to d.

• If there is no decimal point within the specified positions, the value of d is used to position a decimal place before storing the value. If the characters 1246 are read using F4.1, the value stored is 124.6.

Example:

        Read 10, I, r, t
10    format (2x, i5, f7.3, f6.2)

Example:

        read 50, a, c, k
50    format(2x,2f5.1,2x,i4)

For a=80.1, c=36.5 and i = 40, the input line should be as:

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>0</th>
<th>. 1</th>
<th>3</th>
<th>6</th>
<th>. 5</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2F5.1</td>
</tr>
<tr>
<td></td>
<td>2X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I4</td>
</tr>
</tbody>
</table>

**E- Descriptor**

This descriptor is used to read a variable entered in an exponential form.

General Form: \[ E_{w.d} \]

Where w represents the total number of positions to use from the data line and d represents the number of decimal positions.

Example:

        read 30, lter, error
30    format(2x,i5,2x,e7.2)

Data line:
A-Descriptor

It is used to read character variable.

General Form: \[ \text{Aw} \]

Where \( w \) represents the number of positions to be re-read.

Example:
If the line of data

\[
\begin{array}{cccccccccccccccc}
\text{Four} & \text{score} & \text{a} & \text{n} & \text{d} & \text{seven} & \text{e} & \text{a} & \text{r} & \text{s} & \text{ago}
\end{array}
\]

Is read by the following statements

\[
\text{CHARACTER (6) :: Speech1, Speech2}
\]

\[
\text{Read ‘(2A)’, Speech1, Speech2}
\]

The value assigned to Speech1 and Speech2 are:

Speech1: Foursc
Speech2: ore an

If the format specifier ‘(A2,A12)’ were used, the value assigned would be:

Speech1: Fobbbb Where b denotes a blank
Speech2: ebandb Where b denotes a blank

Multiple Input lines

A new line of data is required each time a READ statement is executed or whenever a slash (/) is included in the format specifier.

Example:
The following input lines

Total number of iterations:
845
Maximum Error found:
0.005

can be read by a single READ statement Using
  Integer:: Iterations
  Real::  Max_Error
  READ ‘(/I3/ F6.3)’,Iterations, Max_Error

3.2.2 Formatted output

The formatted output allows the programmer to control the precise format of the output. There are two output statements in Fortran, The print statement and the write statement.

General form:  

Print format-specifier, output list

The format specifier specifies the format in which values of the expressions in the output list are to be displayed. It is one of the following

1.  * (an asterisk)
   It indicates list-directed output. (output format is machine-dependent).
   in which the format for the various types of expressions in the output
Example    print *,A,B,J

2.  A character constant or a character variable (or expression or array) whose value specifies the format for the output.
    Example    print ‘(I3,I4,I7)’,J,K,L

3.  The label of a FORMAT statement
    In this case, the formatting information is supplied by a FORMAT statement whose label is specified. This statement has the form

    label FORMAT (list of format descriptors)

Where label is an integer in the range 1 through 99999.
Each execution of a PRINT statement displays the values of the items in the output list on a new line.

**Example:**

```
PRINT'(1X,F7.2,3X,F5.1)', X,Y
OR
PRINT 2, X,Y
2 FORMAT(1X,F7.2,3X,F5.1)
```

**Definition of Buffer**

The computer uses the specification list to construct each output line internally in memory before actually printing the line. The internal memory region is called a BUFFER. The first character of the buffer is called the carriage control character which determines the vertical spacing for the line.

```
Carriage Control Character
```

**Vertical Spacing control**

For some FORTRAN compilers, the first character of each line of output directed to a printer is used to control the vertical spacing. If this character is one of the following, it is removed from the output line and is used to effect the appropriate printer control:

- Blank: Normal spacing (single spacing)
- 1: New page
- 0: Double-spacing
- +: No vertical spacing
**X, I, F and E specifications**

The general rules of the X, I, F and E specifications are exactly as in formatted input.

**Example (1)**

Print 4

4    Format(‘Fortran course’)  

The buffer output line:

| F o r t r a n c o u r s e |

And the output is:

| o r t r a n c o u r s e |

But if the following format is used:

4    Format(1x,‘Fortran course’)  

The buffer output line:

| F o r t r a n c o u r s e |

And the output is:

| F o r t r a n c o u r s e |

**Example (2)**

Given:  K=20,  L=-30,  M=-450

Print 4, K,L,M

4    Format(1X,I3,2X,I2,3X,I4)  

The buffer output line:

|  2 0 * * - 4 5 0 |
The output is:

\[
\begin{array}{cccc}
2 & 0 & * & * \\
- & 4 & 5 & 0
\end{array}
\]

**Example (3)**

Given: \( T = -0.000234, \ R = 100.68 \)

Print 4, T,R

4 Format(1X,E10.2,3X,F7.3)

The buffer output line:

\[
\begin{array}{ccccc}
- & 0. & 2 & 3 & E - 0.3 \\
1 & 0 & 0 & . & 680
\end{array}
\]

The output is:

\[
\begin{array}{ccccc}
- & 0. & 2 & 3 & E - 0.3 \\
1 & 0 & 0 & . & 680
\end{array}
\]
CHAPTER 4
ALGORITHM

4.1 Introduction
A computer program is a sequence of instructions that must be followed to solve a particular problem. At least four steps or stages can be identified in the program-development process:

1) **Problem analysis and specification**
   Analyse the problem and formulate a precise specification of it. This specification must include a description of the problem’s input and its output.

2) **Data organization and algorithm design**
   Design a plan to solve the problem. This plan has two parts:
   1. Determine how to organize and store the data in the problem.
   2. Develop procedures to process the data and produce the required output. These procedures are called **algorithms**

3) **Program coding**
   The process of implementing data objects and algorithms in some programming language

4) **Execution and testing**
   Check that the algorithm and program are correct. The simple way to do this is to execute the program with input values for which the correct output values are already known (or are easily calculated).

5) **Program maintenance**
   Modification of software is necessary to
   - improve its performance
   - add new features
   - consider the changes in computer and/or the system software
   - etc.
4.2 Algorithm

It is a sequence of logical steps used to solve a problem.

Algorithm of any problem can be described by the following methods:

1. **Pseudocode**: Presents algorithm steps in a series of English-like statements. It uses a mixture of natural language and symbols, terms, and other features commonly used in high-level languages.

2. **Flowchart**: Graphical representation of algorithm steps. Each step of the algorithm is placed in a box of appropriate shape connected with each other according to the order of performance of each step (see Table 3.1).

The two-dimensional nature of the flowcharts compared to the one-dimensional nature of the pseudocode makes it easier to visualize and understand the structure of some algorithms.

The steps in an algorithm must be organized in a logical and clear manner. Structured algorithms and programs are designed using three basic methods of control: sequential, selection, and repetition.
Table 4.1: Flowchart Symbols

<table>
<thead>
<tr>
<th>Shape</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Oval" /></td>
<td>Oval</td>
<td>Beginning and end of an algorithm</td>
</tr>
<tr>
<td><img src="image" alt="Parallelogram" /></td>
<td>Parallelogram</td>
<td>Inputs or outputs information</td>
</tr>
<tr>
<td><img src="image" alt="Rectangle" /></td>
<td>Rectangle</td>
<td>Assignments of values or computation process</td>
</tr>
<tr>
<td><img src="image" alt="Diamond" /></td>
<td>Diamond</td>
<td>Selection point</td>
</tr>
<tr>
<td><img src="image" alt="Hexagon" /></td>
<td>Hexagon</td>
<td>Beginning of a repetition</td>
</tr>
<tr>
<td><img src="image" alt="Double-lined rectangle" /></td>
<td>Double-lined rectangle</td>
<td>Reference to a sub algorithm</td>
</tr>
<tr>
<td><img src="image" alt="Flow lines" /></td>
<td>Flow lines</td>
<td>Order in which the steps are performed</td>
</tr>
<tr>
<td><img src="image" alt="Connector" /></td>
<td>Connector</td>
<td>Joint of flow lines</td>
</tr>
</tbody>
</table>
**Sequential**

Steps are performed in a strictly sequential manner, each step being executed once.

**Example:**
Develop a computer algorithm that computes the area of a circle for a given radius.

**Solution:**

**Fortran Program**

```fortran
program Area_of_a_circle
    Read *,r
    area=3.14*r**2
    print*,R, area
    stop
end program Area_of_a_circle
```

**Flow Chart**

1. **Start**
2. **Read R**
3. **A= \pi R^2**
4. **Print R, A**
5. **Stop**
**Selection**

One of a number of alternative actions is selected and executed.

**Example:**
Write an algorithm (flow chart) for computing the zakat of the total annual saving of a person.

**Solution:**
Repetition

One or more steps are performed repeatedly.

Example:
Write an algorithm (flow chart) for computing and printing the summation of the squares of integer numbers between 2 and 50 \( S = \sum N^2 \)

Solution:
CHAPTER 5

IF STATEMENT

5.1 Relational Operators

Table 5.1 Summarizes the relational operators to be used in Fortran.

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Math symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; or .LT.</td>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;= or .LE.</td>
<td>≤</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&gt; or .GT.</td>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;= or .GE.</td>
<td>≥</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>== or .EQ.</td>
<td>=</td>
<td>equal to</td>
</tr>
<tr>
<td>/= or .NE.</td>
<td>≠</td>
<td>not equal to</td>
</tr>
</tbody>
</table>

Examples:

Write these mathematical expressions in Fortran

<table>
<thead>
<tr>
<th>Math Expression</th>
<th>Math expression in Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x^2+y^2 \leq xy)</td>
<td>((x^2+y^2) \leq (x*y))</td>
</tr>
<tr>
<td>or</td>
<td>( (x^2+y^2) \leq (x*y) )</td>
</tr>
<tr>
<td>(I+5 \neq 2I)</td>
<td>((I+5) \neq (2*I))</td>
</tr>
<tr>
<td>or</td>
<td>((I+5) = (2*I))</td>
</tr>
<tr>
<td>((x+y)/8 = x/y)</td>
<td>( ((x+y)/8) = (x/y) )</td>
</tr>
</tbody>
</table>

5.2 Logical Operators

Table 5.2 Summarizes the logical operators in Fortran.

<table>
<thead>
<tr>
<th>Logical operator</th>
<th>Order of precedence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NOT.</td>
<td>1\textsup{st}</td>
<td>Not</td>
</tr>
<tr>
<td>.AND.</td>
<td>2\textsup{nd}</td>
<td>and</td>
</tr>
<tr>
<td>.OR.</td>
<td>3\textsup{rd}</td>
<td>or</td>
</tr>
<tr>
<td>.EQV.</td>
<td>4\textsup{th}</td>
<td>Equivalence</td>
</tr>
<tr>
<td>.NEQV.</td>
<td>4\textsup{th}</td>
<td>Nonequivalence</td>
</tr>
</tbody>
</table>
.NOT.

<table>
<thead>
<tr>
<th>A</th>
<th>.NOT. A</th>
<th>T = True</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
<td>F = False</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

.AND.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A .AND. B</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

.OR.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A .OR. B</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

.EQV.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A .EQV. B</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

.NEQV.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A .NEQV. B</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>
Examples:

<table>
<thead>
<tr>
<th>Math condition</th>
<th>Math condition in Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L=1) or (m &gt; 5)</td>
<td>L==1.OR.M&gt;5</td>
</tr>
<tr>
<td>(5&lt;x≤15)</td>
<td>X&gt;5.0.AND.X&lt;=15</td>
</tr>
</tbody>
</table>

* The .Not. operator must never immediately precede the .AND. or .OR.
* Two cases for the logical operators to follow each other:
  1. .AND. .NOT.
  2. .OR. .NOT.

5.3 Logical Variable

The general form of declaring the variable as a logical one is

```
LOGICAL :: list of variable(s)
```

Example:

LOGICAL :: Done, Work

The following logical operators that used with the real or integer variables will be modified to be used with the logical variables as:

- .EQ. → .EQV.
- .NE. → .NEQV.

Example:
If we want to compare .Not.Done. to the value .true., we could use this statement

```
IF (.NOT.Done.EQV..TRUE.)  Then
```

5.4 Logical IF Statement

General form:

```
IF (logical expression) executable statement
```
The executable statement above will be executed if the result of the logical expression is true.

Examples:

1. IF (X<=Y**2)  X = Y/2.0
2. IF (Time>1.5) READ*, Distance

Example (1)

Draw flow chart and write a Fortran program to calculate and print \( y \) from the following function:

\[
y = \begin{cases} 
  x^2 - 1 & x \geq 0 \\
  2x + 5 & x < 0 
\end{cases}
\]
Real::X,Y
Read *, X
IF (X.GE. 0.0) Go To 2
Y=2*X+5
Go To 1
2 Y=X**2-1
1 Print *, Y
Stop
End

5.5 Block IF Statement
General form:

IF (logical expression) THEN
    Statement 1
    .
    .
    .
    Statement n
END IF

The statements from 1 to n will be executed only if the result of the logical expression is true.

Example:

IF (Den.EQ.0.0) THEN
PRINT *, ‘Denominator is zero’
STOP
END IF
Fractn=Num/Den
PRINT*, ‘fraction=’, Fractn

Example (2) Repeat the solution of Example (1) using IF……..Then statement:

Read *, X
IF (X .GE. 0.0) Then
Y=X**2-1
Print *, Y
Stop
END IF
Y =2*X+5
Print *, Y
STOP
END
OR

Read *, X
IF (X .GE. 0.) Then
  Y = X**2 - 1.
  Go to 5
  END IF
  Y = 2*X + 5
5 Print *, Y
STOP
END

5.6 Nested IF statement

If statement can be nested as shown in the following general form:

```
IF (logical expression # 1) THEN
  Statement 1
  .
  .
  Statement n
  IF (logical expression #2) THEN
    Statement n+1
    .
    .
    Statement m
    END IF
  Statement m+1
  .
  .
  END IF
Statement q
```

- If the logical expressions are true, all steps from 1 → p will be executed.
- If the logical expression # 1 is true and the logical expression # 2 is false, then the steps n+1 to m will not be executed and the other statements will be executed.
- If the logical expression # 1 is false, all the statements inside the outer “IF block” will not be executed, and the program will jump to execute step q.
5.7 IF-ELSE Statement

General form:

<table>
<thead>
<tr>
<th>IF (logical expression) THEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement 1</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>Statement n</td>
</tr>
<tr>
<td>ELSE</td>
</tr>
<tr>
<td>Statement n+1</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>Statement m</td>
</tr>
<tr>
<td>END IF</td>
</tr>
</tbody>
</table>

- If the logical expression is true the statements from 1 to n will be executed only, then the execution will jump to the END IF statement.
- If the logical expression is false, the statement from n+1 to m will be executed only.

Example:

```plaintext
LOGICAL Valid
READ*, Volts
IF (Volts >= -5.0 .AND. Volts<=5.0) THEN
  Valid= .TRUE.
ELSE
  Valid= .FALSE.
END IF
IF (.NOT.Valid) PRINT*, 'Error in data'
.
.
STOP
END
```

Example:

```plaintext
IF (Distance<=30.0) THEN
  Velocity=2.45+0.32*Distance-0.0025*Distance**2
ELSE
  Velocity=0.625-0.12*Distance-0.0025*Distance**2
END IF
```
Example (3): Repeat the solution of example (1) using IF…….Then…….Else Statement

Read *, X
IF (X .GE. 0.0) Then
  Y = X**2-1
Else
  Y = 2.*X+5
END IF
Print *, Y
END

Example (4): Draw flow chart and write a Fortran program to calculate and print y from the following function:

\[
\begin{align*}
  y &= \begin{cases} 
  X^2 & X < 0 \\
  0 & X = 0 \\
  \sin(x) & X > 0 
  \end{cases}
\end{align*}
\]

Solution:

Read *, X
IF (X .LT. 0.0) Go To 4
IF (X .EQ.0.0) Then
  Y = 0.0
ELSE
  Y = Sin(X)
END IF
Go to 5
4  Y = X**2
5  Print *, Y
END
5.8 IF-ELSE IF Statement

General form:

IF (logical expression # 1) THEN
Statement 1
.
.
Statement m
ELSE IF (log expression # 2) THEN
Statement m+1
.
.
Statement n
ELSE IF (logical expression # 3) THEN
Statement n+1
.
.
Statement p
ELSE
Statement p+1
END IF
Statement q

- log. exp. 1 → True statements 1 to m will be executed → then END IF.
- log. exp. 2 → False will go to ELSE IF with log. exp. #2
- log. exp. 2 → True statements m+1 to n will be executed → then END IF.
- log. exp. 2 → False will go to ELSELF with log. exp. # 3
- log. exp. 3 → True statements n+1 to p will be executed → then END IF.
- log. exp. 3 → False statements p+1 to q will be executed → then END IF.

Example (5):
Repeat the solution of example (4) using IF…..ELSE IF statement:

Read *, X
IF (X .LT. 0.) Then
Y = X**2
ELSE IF (X==0.) Then
Y = 0.0
ELSE
Y=Sin(X)
END IF
PRINT *, Y
END

END
Example (6) : Write IF statement to determine the category of weight as flow:

<table>
<thead>
<tr>
<th>Category</th>
<th>Weight (kd)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>W ≤ 50.0</td>
</tr>
<tr>
<td>2</td>
<td>50 &lt; W ≤ 125</td>
</tr>
<tr>
<td>3</td>
<td>125 &lt; W ≤ 200</td>
</tr>
<tr>
<td>4</td>
<td>200 &lt; W</td>
</tr>
</tbody>
</table>

1. **Solution using IF-ELSE statement:**

IF (w<=50.0) THEN
    Categr=1
ELSE IF (W<=125.0) THEN
    Categr=2
ELSE IF (W<=200.0) THEN
    Categr=3
ELSE
    Categr=4
END IF
END IF

2. **Solution using IF-ELSE IF statement:**

IF (W<=50.0) THEN
    Categr=1
ELSE IF (W<=125.0) THEN
    Categr=2
ELSE IF (W<=200.0) THEN
    Categr=3
ELSE
    Categr=4
END IF

3. **Solution using IF statement:**

IF (W<=50.0) Categr=1
IF (W>50.0 .AND. W<=125.0) Categr=2
IF (W>125.0 .AND. W<=200.0) Categr=3
IF (W>200.0) Categr=4
Example (7):
Write Fortran program to evaluate the grade of student in math. exam. according to the university scale:

<table>
<thead>
<tr>
<th>Student Score</th>
<th>Student Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 ≥ Score ≥ 90</td>
<td>A</td>
</tr>
<tr>
<td>90 &gt; Score ≥ 80</td>
<td>B</td>
</tr>
<tr>
<td>80 &gt; Score ≥ 70</td>
<td>C</td>
</tr>
<tr>
<td>70 &gt; Score ≥ 60</td>
<td>D</td>
</tr>
<tr>
<td>Score &lt; 60</td>
<td>F</td>
</tr>
</tbody>
</table>

```
Integer::Score
Read *, Score
IF (Score .GE. 90) Then
  Print *, Score, ‘A’
ELSE IF (Score .GE. 80) Then
  Print *, Score, ‘B’
ELSE IF (Score .GE. 70) Then
  Print *, Score, ‘C’
ELSE IF (Score .GE. 60) Then
  Print *, Score, ‘D’
ELSE
  Print *, Score, ‘F’
END IF
END
```

5.9 Case Constructs

Examples

I. Select Case (Grade)
   Case (‘A’, ‘a’)
   Points = 5
   Case (‘B’, ‘b’)
   Points = 4
   Case (‘C’, ‘c’)
   Points = 3
   Case (‘D’, ‘d’)
   Points = 2
   End Select

II. Select Case (X)
   Case (1.0)
   Print X, X+1.0
   Case (2.0)
   Print X, X+2.0
### III. Select Case (Grade-Char)

Case (‘A’, ‘a’)
Print *, ‘Excellent’
Case (‘B’, ‘b’)
Print *, ‘Very good’
Case (‘C’, ‘c’)
Print *, ‘Good’
Case (‘D’, ‘d’)
Print *, ‘pass’
Case (‘F’, ‘f’)
Print *, ‘Fail’
End select

#### Example (8):

Follow the execution of the following program and print out the results:

```plaintext
Logical flag
Parameter (K=2, J=10)
m=3
Flag = J/M .LE. K
IF (.Not. Flag) Then
   N = K * (M-1)
   IF (N. GE. M. OR. .Not (N. EQ.K)) Then
      Print *, ‘IT is Good’
   Else
      Print *, ‘IT is Bad’
   END IF
ELSE
   Print *, ‘Program is Finished’
END IF
END IF
END
```

<table>
<thead>
<tr>
<th>K</th>
<th>J</th>
<th>M</th>
<th>N</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>3</td>
<td>4</td>
<td>False</td>
</tr>
</tbody>
</table>

Print Out

```
IT is good
```
CHAPTER 6
REPETITIVE EXECUTION

6.1 Introduction
A repetition structure or loop makes possible the repeated execution of one or more statements called the body of the loop.

There are two basic types of repetition:
1. Repetition controlled by a counter
2. Repetition controlled by a logical expression

6.2 Counter-Controlled DO Loop

General Form:

\[
\text{DO} \quad \text{Index} = \text{Initial value, Limit, Increment}
\]

- Statement 1
- ...
- Statement n

\text{END DO}

Where:
- Index is a variable that could be real or integer.
- Initial value or limit or increment may be constant, variables, or expressions that could be also real or integer types.
- Increment can be either positive or negative, but it cannot be zero.
- The number of repetitions is calculated as the larger of the value 0 and the value of the following expression:

\[
\left\lfloor \frac{\text{Limit} - \text{Initial value} + \text{Increment}}{\text{Increment}} \right\rfloor
\]

Example:
The following DO-loop

\[
\text{DO} \quad k=5, \quad 83, \quad 4
\]

would be executed as the following number of times:
\[ \int \left[ \frac{83 - 5 + 4}{4} \right] = 20 \]

but for the following DO-loop

\[ \text{DO } K=10, \ 2, \ 4 \]

will not be executed (i.e. the number of repetitions equal to 0) since the substitution in the formula will give

\[ \int \left[ \frac{2 - 10 + 4}{4} \right] = -1 \]

a negative value.

**Example:**

Write a DO-loop to execute the following summation:

\[ \sum_{i=1}^{50} = 1 + 2 + 3 + 4 + \ldots \ldots + 50 \]

**Solution:**

\[
\begin{align*}
\text{Sum} &= 0 \\
\text{DO } N=1, \ 50 & \\
& \quad \text{Sum} = \text{Sum} + N \\
\text{END DO}
\end{align*}
\]

**Example:**

\[
\begin{array}{c|c|c}
\text{Count} & \text{I} & \text{Count} \\
\hline
0 & 1 & 1 \\
1 & 2 & 2 \\
\Rightarrow & 3 & 3 \\
& \ldots & \ldots \\
& 8 & 8 \\
\end{array}
\]

\[ \text{Count=8} \]
Example:

\[
\begin{array}{c|c|c}
N & M & N \\
\hline
5 & 5 & -1 \\
6 & 6 & 0 \\
7 & 7 & -1 \\
8 & 8 & 0 \\
9 & 9 & -1 \\
10 & 10 & 0 \\
\end{array}
\]

\[N=0\]

6.3 Nested DO Loop

1. A nested DO Loop cannot use the same index as a Loop that contains it.

\[
\begin{array}{ccc}
\text{DO } & \text{I}=1,5 & \Rightarrow \\
\text{DO } & \text{I}=1,3 & \Rightarrow \\
\end{array}
\]

2. DO Loops that are independent of each other may have the same index.

3. When one loop in nested with another, the inside loop is completely executed each pass through the outer one.

Example:

Follow the execution of the following nested DO-Loop and find the value of \(N\).

\[
\begin{array}{c|c|c|c}
N=0 \\
\text{DO } & \text{I}=1, 2 & I & J & N \\
\text{DO } & \text{J}=1, 3 & 1 & 1 & 2 \\
N = I*J+1 & 1 & 2 & 3 \\
\text{END DO} & 1 & 3 & 4 \\
\text{END DO} & 2 & 1 & 3 \\
\text{END DO} & 2 & 2 & 5 \\
\text{END DO} & 2 & 3 & 7 \\
\end{array}
\]

\[N=7\]

4. Each DO Loop should be end by separate END DO statement as shown above.
6.4 General DO Loops

General DO loop is used for the problems in which the number of iterations cannot be determined in advance and a more general repetition structure is required.

6.4.1 DO-Exit Construct

General form:

```fortran
DO
  statement-sequence 1
  IF (logical-expression) EXIT
  statement-sequence 2
END DO
  statement-sequence 3  ! to be executed for true condition
```

Example (1):
Write a Fortran program segment to calculate the following summation:

\[
\sum_{i=1}^{50} 1 + 2 + 3 + 4 + \ldots \ldots + 50
\]

Solution:

1. Using DO-Exit construct:

   Sum = 0.0
   Number = 1
   DO
   Sum = Sum + Number
   Number = Number + 1
   IF (Number > 50) EXIT
   END DO

2. Using If statement:

   Sum = 0.0
   Number = 1

10  IF (Number <= 50) THEN
    Sum = Sum + Number
    Number = Number + 1
GO TO 10
END IF
Example (2)

Write a Fortran program to calculate and print the factorial of an integer number K, where

\[ \text{Factorial } K = K \times (K-1) \times (K-2) \times \cdots \times 2 \times 1 \]

\[
\begin{align*}
\text{Integer::Fact, K, I } \\
\text{Read *, K } \\
\text{Fact=1 } \\
\text{DO I=2, K } \\
\text{Fact=Fact * I } \\
\text{END DO } \\
\text{PRINT *, Fact } \\
\text{END}
\end{align*}
\]

Example (3)

Write a Fortran program to calculate and print Y, where:

\[ Y = 5 \times X^m - 3 \times X^{m-1} + 2 \times X - 1 \]

For M=5, \( X=0.1, 0.2, 0.3, \ldots, 10 \)
M=4, \( X=0.1, 0.2, 0.3, \ldots, 10 \)
M=3, \( X=0.1, 0.2, 0.3, \ldots, 10 \)

\[
\begin{align*}
\text{DO M=5,3,-1 } \\
\text{DO I=1,100 } \\
\text{X=I/10.0 } \\
\text{Y=5.*X**m-3.*X**(m-1)+2.*X-1.0 } \\
\text{Print 10, M, X, Y } \\
\text{10 Format (5X, 'M=', I2,2X, 'X=', F7.3, 2X, 'Y=', F9.4) } \\
\text{END DO } \\
\text{END DO } \\
\text{END}
\end{align*}
\]

Example (4)

The value of a function \( \log X \) is given below:

\[ \log X = 2 \left[ \frac{X-1}{X+1} + \frac{1}{3} \left( \frac{X-1}{X+1} \right)^3 + \frac{1}{5} \left( \frac{X-1}{X+1} \right)^5 + \frac{1}{n} \left( \frac{X-1}{X+1} \right)^n \right] \]

Write a program to:
1. determine the value of \( \log X \) if \( X > 0 \)
2. print a message “undefined function” followed by stop state if \( X \leq 0 \)
Real::X, logX, Y
Integer:: n, I
Read *, X, n
If (X .GT. 0.) Then
  Y = (X-1) / (X+1)
  Log X = 0.0
  DO I=1,n,2
    log X = log X + 2 * (1.0/I*Y**I)
  END DO
  Print *, X, n, Log X
ELSE
  Print *, ‘Undefined Function’
END IF
STOP
END

**Example (5)**

Write a program to calculate the mean and standard deviation of Lab. measurements samples X1, X2, ……X10 where

\[
\text{Mean} = \frac{\sum_{i=1}^{n} X_i}{n}
\]

\[
\text{Standard deviation} = \sqrt{\frac{\sum_{i=1}^{n} X_i^2}{n} - \text{mean}^2}
\]

Real::Mean, X, SD
Integer::I,n
Read *, n
Sum1=0.0
Sum2=0.0
DO I=1, n
  Read *, X
  Sum1=Sum1+X
  Sum2=Sum2+X**2
END DO
Mean=Sum1/n
SD=SQR(Sum2/n-Mean**2)
Print *, Mean, SD
END
6.4.2 **DO-CYCLE Construct**

This construct is used to terminate only the current repetition and jump ahead to the next one.

**General form:**

```
DO
    statement-sequence 1
    IF (logical-expression) THEN
        statement-sequence 2
    CYCLE
    END IF
    statement-sequence 3
    IF (logica-expression) EXIT
END DO
```

**Example:**
Write a Fortran program to read temperature values in Celsius and convert the positive of them into Fahrenheit scale.

```
PROGRAM Convert
IMPLICIT NONE
REAL :: Celsius, Fahrenheit
CHARACTER (1):: Response
DO
    PRINT *, “Enter temperature in degrees Celsius”
    READ *, Celsius
    IF (Celsius<0.0) THEN
        PRINT *, “***temperature must be 0.0 or above***”
        CYCLE
    END IF
    Fahrenheit=1.8* Celsius+32.0
    PRINT *, Celsius, “degrees Fahrenheit”
    PRINT *, “More temperature to convert (y or n)?”
    READ *, response
    IF (Response==”N”) EXIT
END DO
END PROGRAM Convert
```
6.5  **Named DO Construct**

DO construct can be named in this way:

```
Name: DO               
   .                      
END DO Name
```

**Example:**

```
Outer: Do M=1, last_M   
   Inner: Do N=1, Last_N
   Product=M*N
   PRINT *, M,” “, N,” “, Product
END DO Inner
END DO Outer
```

**Example**

Follow the execution of the following program and show its output:

```
Logical Flag
Real::A,B,M
Integer::Y, N
Parameter (A=10.2, N=7)
DO Y=2,16,7
   B=Y+N/3
   Flag = A .GT. B
   IF (Flag .EQV. True.) Then
      M=A+B
      Print *, B, M
   ELSE
      IF (A+B .LE. 18.0.OR.B.GT.11.0) Then
         M = A-B
         Print *, B, M
      STOP
   ELSE
      Print 40
   END IF
40 Format (1X, ‘Invalid Value for Y’)
END IF
END DO
PRINT 10, Y, N
10 FORMAT (2X, 2I4)
END
```
Tracing

<table>
<thead>
<tr>
<th>A</th>
<th>N</th>
<th>Y</th>
<th>B</th>
<th>M</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.2</td>
<td>7</td>
<td>2</td>
<td>4</td>
<td>14.2</td>
<td>True</td>
</tr>
<tr>
<td>10.2</td>
<td>7</td>
<td>9</td>
<td>11-</td>
<td></td>
<td>False</td>
</tr>
<tr>
<td>10.2</td>
<td>7</td>
<td>16</td>
<td>18-</td>
<td>-7.8</td>
<td>False</td>
</tr>
</tbody>
</table>

Print Out

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4.</td>
<td>14.2</td>
</tr>
<tr>
<td>2.</td>
<td>7.</td>
</tr>
<tr>
<td>9.</td>
<td>7.</td>
</tr>
</tbody>
</table>

Invalid Value for Y

| 18.    | -7.8|
A data file can be built using an editor or another program (like FORTRAN program).

To use data files by a FORTRAN program, some statements need to be used. These statements are described below:

**Open statement**

Before a file can be used for input or output in a FORTRAN program, it must be opened. This is accomplished using an **OPEN** statement.

General form:

```fortran
OPEN (UNIT = unit-number, File = file-name, STATUS = status)
```

Where:

- `unit-number` is an integer that will be used to reference this file in a READ or WRITE statements.
- `file name` refers to the name given to the file.
- `status` could be ‘OLD’ for an exiting file (input file) or ‘NEW’ for a new file (output file).

- Open statement must precede any READ OR WRITE statements that use the file.
- Open statement should be executed only once, for one file, therefore, it should not be inside a loop.

Once a file has been given a unit number, data can be read from or written to that file using special forms of the READ and WRITE statements.

```fortran
READ ( unit-number, * ) input list
```
Example:

OPEN (unit=7, file = 'TEST.DAT', STATUS= 'old')
READ (7,*) A,B

To write information to a date file, we use the write statement instead of print statement as in the following form:

WRITE (unit number, *) expression list

For free formatted output
or

WRITE(unit number, k) expression list

For formatted output.

**Close Statement**

This statement is used to close the date file when it is needed. General form:

CLOSE (UNIT = integer expression)

Example:

CLOSE (UNIT=5)
CHAPTER 8
ARRAYS

FORTRAN 90 provides arrays for storing, structuring and processing a collection of data values that are related in some way such as a list of test scores.

8.1 One Dimensional Array (1-D Array)

An array is a group of storage locations (elements) that have the same name. The element of an array is distinguished by using the common name followed by a subscript or an index in parentheses. Subscript is represented by consecutive integer, usually begins with the number 1.

Examples

! Specification statement
Real, Dimension (50) :: Failure_Time

! Assignment statement
Failure_Time (4) = 177.8

! Output statement
Print *, Failure_Time (10)

! Read statement
Read (10, *) Failure_Time (10)

! DO loop
DO I=1, 50
Read (10, *) Failure_Time (I)
End DO

! The DO loop is equivalent to 50 Read statements
Read (10, *) Failure_Time (1)
Read (10, *) Failure_Time (2)

Read (10, *) Failure_Time (50)

! Input or output containing the array name
Read (10, *) Failure_Time

! Is equivalent to
Read (10, *) Failure_Time (1), Failure_Time (2),..., Failure_Time (50)

! DO loop and logical expression
DO I=1, Num_Times
IF (Failure_Time (I) > Mean_Time_Failure) &
    print '(1x, F9.1)', Failure_Time (I)
End DO

8.2 Array Declaration

Compile Time Array
In this type of arrays, the memory of the array is allocated during the compilation of the program.

The DIMENSION statement, a non-executable statement, is used to reserve memory space or storage for any array.

General form:

```
Type, DIMENSION (size):: list of array names
```

where array sizes must be specified with integer constants, not variable.
Examples:

INTEGER, DIMENSION (5) :: J, Dist
REAL, DIMENSION (20) :: Calc, Dom1, Kito
CHARACTER (20), DIMENSION (20) :: names, IDs

The range of subscript with an array can be specified with a beginning subscript number and an ending subscript number. Both numbers must be integers separated by a colon and must follow the dimension statement.

General form:

```
    type, DIMENSION (1:u) :: list of array names
```

Or

```
    type :: list of array specifiers
```

Where an array specifier has the form

```
    array name (1:u)
```

and l and u are two integer constants represent the lower and the upper limits of the range of values of subscript.

For example, the array declarations

```
    Integer, parameter : lower limit_1 = 1, upper limit_1 = 3,
                        lower limit_2 = 2, upper limit_2 = 5
    Integer, Dimension (lower limit_1 : Upper limit_1) :: Gamma
    Real, Dimension (lower limit_2, upper limit_2) :: Delta
```
The fixed-size arrays suffer from two problems:

- If the size of the array exceeds the number of values to be stored in it, then the memory is wasted by the unused elements.

- If the size of the array is smaller than the number of values to be stored in it, there is the problem of array overflow.

Both of these problems can be solved using allocatable or run-time arrays, for which memory is allocated during execution instead of during compilation.

**Allocatable Arrays (or run time arrays)**

In this type of arrays, the memory is allocated during the execution instead of during compilation. The declaration of an allocatable array must specify that it has the allocatable attribute.

General form:

```
type, DIMENSION (:), ALLOCATABLE :: list of array names
```

**Example:**

```
REAL, DIMENSION (:), ALLOCATABLE :: A, B
```

The actual bands of an allocatable array are specified in an ALLOCATABLE statement.

General form:

```
ALLOCATE ( list, STAT=status-variable)
```

Where list is a list of array specifications of the form

```
Array-name (i:u)
```
Where \( l \) and \( u \) as defined before. When the STAT=clause is present, the integer variable status-variable will be set to zero if allocation is successful but will be assigned some system dependent error value if there is insufficient memory or if the array has already been allocated.

**Example:**

```fortran
REAL, DIMENSION ( : ), ALLOCATABLE :: A, B
PRINT *, "Enter size of arrays A and B:"
READ *, N
ALLOCATE (A(N), B(0:N+1), STAT=AllocateStatus)
IF (AllocateStatus/=0) STOP "****Not enough memory***"
```

Memory that is no longer needed can be released so that it can be reallocated. The general form for Deallocate Statement is

```
Deallocate (list-of-arrays, stat=status-variable)
```

Storing Data In 1-D Array

**Examples:**

1. \( J(1)=0 \)
2. \( J(5)=\text{NUM}\ast\text{COUNT} \)
3. \( \text{DIST}(2)=46.2+\sin(x) \)

Subscripts can also be integer expression:

**Examples:**

\[
\begin{align*}
J(2*I) &= 3 \\
R(J) &= R(J-1) \\
B &= TR(2*I)+TR(2*I+1)
\end{align*}
\]
Initializing Of 1-D Array

Example:

    INTEGER, DIMENSION (5) :: J
    DO I=1,5
        J(I)=10
    END DO

Input And Output

1. An array constant may be constructed as a list of values enclosed between (/ and /).

Example:

    INTEGER, DIMENSION (8) :: A
    A= (/11, 22, 33, 44, 55, 66, 77, 88/)

2. We may use the read statement to store data into an array.

Example:

If the array A contains 8 elements, then

    REAL, DIMENSION (8) :: A
    READ *, A

will read values for all 8 elements. And if we have

    REAL, DIMENSION (8) :: A
    READ *, A(1), A(2), A(3)

then the read statement will read values for only the first three elements.

3. Array values may also be read using an implied DO Loop.

Example:
If we wish to read the first 10 elements of the array R, we can use the following implied DO-Loop in the read statement.

```plaintext
REAL, DIMENSION (20) :: R
READ *, (R(I), I=1, 10)
```

**Example:**
If we wish to read the first 10 elements of the array R *with each value must be on a separate line*, we can use the following DO-Loop.

```plaintext
REAL, DIMENSION (20) :: R
DO I=1, 10
   READ *, R(I)
END DO
```

5. Assignment statements may be used for array variables:

```
array variable = expression
```

Where the expression must be either:

1. An array of the same size as the array variable, or
2. A simple value

**Example:**

```
A=0
```

Assign 0 to each element of A.

Same as reading data to an array, we can print them by using the print statement.

**Example:**
A group of 30 mass measurements is stored in real array mass. Write a Fortran program to print this array in the following form:

MASS( 1)=xxx.x
MASS( 2)=xxx.x
.
MASS(30)=xxx.x

**Solution:**

```fortran
REAL :: MASS(30)
READ*, MASS
DO I=1, 30
   PRINT 15, I, MASS(I)
15 FORMAT (1X, 'MASS(', I2, ')=', F5.1)
END DO
END
```

### 8.3 Two-Dimensional Array (2-D Array)

Two-dimensional array is a Table that consists of row and columns.

**Example:**

\[
A = \begin{bmatrix}
1 & 5 & 11 & -3 \\
0 & 17 & -5 & 2 \\
-33 & 100 & 11 & -15 \\
1 & 3 & 0 & 6 \\
\end{bmatrix}
\rightarrow \text{two dimensional array with 4 rows and 4 columns}
\]

In the two-dimensional array, the elements are distinguished by using the array name and subscripts. The first subscript represents the row number and the second subscript represents the column number.

\[
M(2,3)
\]

This element refers to an integer element in second row and third column of array M.
**Array Declaration**

General form:


\[
\text{type, DIMENSION (number of rows, number of columns) :: array name(s)}
\]

**Examples:**

REAL, DIMENSION (2,3) :: Course

Also in two-dimensional array the range of the subscript can be specified with a beginning subscript number and an ending subscript number.

General form:


\[
\text{type, DIMENSION (l_r,u_r, l_c,u_c) :: array name(s)}
\]

**Example:**

REAL, DIMENSION (-1:2, 3:10) :: Course

**Example:**

REAL, DIMENSION (0:2, -1:1) :: A

Then A has three rows and three columns. If A having the following values,

\[
A = \begin{bmatrix}
0.0 & 1.1 & 2.2 \\
-10.0 & 7.7 & 3.3 \\
3.5 & 5.3 & 0.0
\end{bmatrix}
\]

then

\[
A(0,0)=1.1 \\
A(+1, -1)=-10.0
\]
**Initialization**

**Example:**

Define an array `Area` with 5 rows and four columns, fill it with the values shown

\[
\begin{bmatrix}
1.0 & 1.0 & 2.0 & 2.0 \\
1.0 & 1.0 & 2.0 & 2.0 \\
1.0 & 1.0 & 2.0 & 2.0 \\
1.0 & 1.0 & 2.0 & 2.0 \\
1.0 & 1.0 & 2.0 & 2.0 \\
\end{bmatrix}
\]

**Solution:**

```
REAL, DIMENSION (5,4):: AREA
- - -
- - -
DO  I=1,5
    AREA (I,1)=1
    AREA (I,2)=1
    AREA (I,3)=2
    AREA (I,4)=2
END DO
```

The general form of a compile-time array declaration for an array with multi dimensions is:

```
type, Dimension (l_1:u_1, l_2:u_2, ...., l_k, u_k) :: list-of-array-names
or
type :: list-of-array-specifiers
```

where each array specifier has the form `array-name (l_1:u_1, l_2:u_2, ....l_m, u_m)`
Example:
Real, Dimension (0:2, 0:3, 1:2) : : Beta

The general form of an allocatable (or run-time) array declaration in multidimension is

type, Dimension (:, :, ....), Allocatable : : list

Example:

Real, Dimension (:, :, :), Allocatable : : Beta
Real, Dimension (:, :), Allocatable : : Gama
---------
---------
---------

Allocate (Beta (0:2, 0:3, 1:2), Gama (1:2, -1:3), & status = Allocate Status)

Example:
Write a FORTRAN program that does the following:

a. Reads from a data file 'Exam.dat', number of students (N), the names of students and their IDs into a character 2-D array number of students (N), and their test scores for four tests into a real 2-D array Test in row order. (Each input data line will have the name of the student, his ID number, and his four test scores).

b. Calculates the average score for the scores for each student and store the result in a real 1-D array Ave.

c. Find the maximum average for the students.
d. Writes the results into output data file called **Output.dat** where each line should contains the name of the student, his ID number, and his four test scores, and the average value. Also, let the program write the name and the ID for the student who got the maximum average, and his average value.

**Solution:**

```fortran
PROGRAM Students_Exam
CHARACTER (20), dimension (50, 2) :: NAME
REAL :: Test (50, 4), Ave (50)
OPEN (UNIT =5, FILE = 'Exam.dat', STATUS = 'OLD')
OPEN (UNIT =6, FILE='Output.dat', STATUS='New')
READ (5, *) Num_Students

DO I=1, Num_students
    READ (5, *) (Name (I, J), J=1,2), (Test (I, K), K=1,4)
END DO

DO I=1, Num_students
    Sum=0
    DO J=1,4
        Sum = Sum +Test (I, J)
    END DO
    Ave (I) = Sum / 4.0
    END DO
    AveMax=Ave(1)
    M=1
    DO I=2, Num_Students
        IF (AveMax < Ave(I) ) THEN
            AveMax=Ave(I)
            M=I
        END IF
    END DO
```

74
END IF
END DO
DO I=1, Num_students
WRITE (6,2) (Name (I,J), J=1,2), (Test (I,K), K=1,4), Ave (I)
2 FORMAT (1X, 2A, 2X, 5F6.2)
END DO
WRITE (6,5) (NAME (M,I), I=1,2), Ave (M)
5 FORMAT (1X, 2A, F6.2)
CLOSE (UNIT=5)
CLOSE (UNIT=6)
STOP
END PROGRAM Students_Exam
CHAPTER 9
SUBPROGRAMS

9.1 Introduction
Subprograms are useful in the following cases:

- When the program became longer and more complicated. Subprograms are used to make these programs readable and simple by dividing them into smaller programs.

- When the same set of operations are performed at more than one location in one program. Subprograms are used to perform these operations each time they are needed without writing the whole set of operations many times.

FORTRAN language has two types of subprograms:
1. FUNCTIONS return single value.
2. SUBROUTINES return more than one value.

Both function and subroutine subprograms can be written as:
- Internal subprograms
- External subprograms

9.2 Functions:
Functions compute a single value and return it. There are two types of functions:
- INTRINSIC functions
- USER-DEFINED functions

9.2.1. Intrinsic Functions
It was given in details in Chapter 2. The following remarks are repeated here:

- The function name and its input values (or arguments) collectively represent a single value.
A function can never be used on the left-hand side of an equal sign in an assignment statement.

The name of the intrinsic function typically determines the type of output from the function. (For example, if the name begins with one of the letters I through N, its value is an integer).

The arguments of a function are generally of the same type as the function itself.

The arguments of a function must be enclosed in parentheses.

The arguments of a function may be constants, variables, expressions, or other functions. For example:

- Constant: SIN (10.0)
- Variable: SIN (theta)
- Expression: SIN (2.0*alfa + theta/3.0)
- Other function: SIN (ABS (a-b))

### 9.2.2 User-defined functions

If a programmer wants to use a function that it is not included in the intrinsic function list, he can define his own function in two ways:

- Statement functions (internal functions)
- Function subprograms (may be internal or external functions).

### Statement Functions

Statement function can be used if the computations can be written in a single statement. It is written inside the program. The following rules should be taken into consideration when writing and using a statement function:

1. The statement function is defined at the beginning of the program along with type statements and array definitions. It is a non-executable statement; thus, it should precede any executable statement.

2. The definition of the statement function contains the name of the function, followed by its arguments in parentheses, on the left-hand side of an equal sign; the expression or computing the function value is on the right-hand side of the equal sign as follows:
3. The function name should be included in a type statement; otherwise, implicit typing will determine the function type.

**EXAMPLE**

Follow the execution of the following program and show its output?

```
PROGRAM STATMENT_FUNCTION
IMPLICIT NONE
INTEGER,DIMENSION(3,4)::RENT
INTEGER :: SET,I,J,L,K
SET(K,L) = K +L/3
DO I=1,3
   DO J=1,4
      RENT(I,J)=SET(I,J)
   END DO
END DO
PRINT*,'ELEMENTS OF ARRAY RENT ARE:'
DO I=1,3
   PRINT*,(RENT(I,J),J=1,4)
END DO
STOP
END PROGRAM STATMENT_FUNCTION
```

**OUTPUT**

ELEMENTS OF ARRAY RENT ARE:

1   1   2   2
2   2   3   3
3   3   4   4
**Function Subprogram**

It can be either internal or external subprogram.

**Internal Function Subprogram**

Which is a separate program itself, and contained by the main program. The function subprogram has the following form:

1. Begins with CONTAINS statement that identify that the main program contains the following subprograms.

2. The beginning of the function subprogram is identified by the function statement with name and argument list, according to the following form:

   ```
   FUNCTION function-name (dummy argument list)
   ```

   or

   ```
   type FUNCTION function-name (dummy argument list)
   ```

4. Execution part
5. Could contain a RETURN statement to return the control to the main program when needed. The return statement is written as:

   ```
   RETURN
   ```

6. The function program must end with END FUNCTION statement:

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

**EXAMPLE (1)**

Write a main program to read three values and prints their average. The average value should be calculated using an internal function subprogram.

*Solution:*

```fortran
PROGRAM Test
    REAL ::  Test 1,  Test 2,  Test 3, Ave
    READ *, Test 1,  Test 2, Test 3
    PRINT 5, Ave (Test 1,  Test 2, Test 3)
5 FORMAT (1X, ‘Average=’, F5.2)

CONTAINS

REAL  FUNCTION  Ave  (X, Y, Z)
    REAL ::  X, Y, Z
    Ave=(X+Y+Z)/3.0
    RETURN
END FUNCTION Ave

STOP
END PROGRAM Test
```

**Notes:**

1. The function argument referenced in the main program must match in type, number, and order the dummy arguments used in the function statement.

2. A function internal subprogram is usually placed immediately before the END PROGRAM statement.
EXAMPLE (2)

Write a function subprogram that receives an array of 20 real values, compute the square of the average of the array and return it as a function value.

Solution:

```fortran
PROGRAM Average_array
  INTEGER :: I
  REAL :: Scores (20), Save
  READ *, (SCORES (i), I=1,20)
  PRINT 50, Save (Scores)

50 FORMAT (4X, "The square of the average=", F10.3)

CONTAINS
  REAL FUNCTION Save (X)
    INTEGER :: I
    REAL :: X(20), Sum
    Sum=0
    DO I=1,20
       Sum = Sum +X(I)
    END DO
    Save = (Sum/20.0)**2
    RETURN
  END FUNCTION Save

END PROGRAM Average_array
```

External Function Subprogram

External function subprogram can be made accessible to a program unit by attaching it after the END statement of the main program.
Several rules must be observed in writing a function subprogram:

1. The function arguments referenced in the main program are the actual arguments. They must match in type, number, and order the dummy arguments used in the FUNCTION statement. In the example, the actual arguments are TEST1, TEST2, TEST3; the dummy arguments are X, Y, Z. TEST1 corresponds to X, TEST2 corresponds to Y, and TEST3 corresponds to Z.

2. If one of the arguments is an array, its dimensions must be specified in both the main program and the function subprogram.

3. The value to be returned to the main program is stored in the function name using an assignment statement.

4. When the function is ready to return control to the statement in the main program that referenced it, a RETURN statement is executed. A function may contain more than one RETURN statement.

5. A function can contain references to other functions, but it cannot contain a reference to itself.

6. A function subprogram is usually placed immediately after the main program, but it also may appear before the main program. If you have more than one function, the order of the functions does not matter as long as each function is completely separate from the other function.

7. A main program and its subprograms can be stored in the same program file or in separate files. If they are stored in separate files, it is necessary to link them before the main program can be executed. The statements required to perform the linking depend on the compiler and the operating system.

8. The same statement numbers may be used in both a function and the main program. No confusion occurs as to which statement is referenced because the function and the main program are completely separate. Similarly, a function and a main program can use the same variable
name, such as SUM, to store different sums as long as the variable SUM is not an argument of the function.

9. The name of the function should appear in a type statement in the main program as well as in the function statement itself.

**EXAMPLE (1)**
Write a main program to read three values and prints their average. The average values should be calculated using an external function subprogram.

*Solution:*

```fortran
PROGRAM Test
   REAL :: Test 1, Test 2, Test 3, Ave
   READ *, Test 1, Test 2, Test 3
   PRINT 5, Ave (Test 1, Test 2, Test 3)
5 FORMAT (1X, ‘Average=’, F5.2)
END PROGRAM Test

REAL FUNCTION Ave (X, Y, Z)
   REAL :: X, Y, Z
   Ave=(X+Y+Z)/3.0
   RETURN
END FUNCTION Ave
```

**EXAMPLE (2)**
Follow the execution of the following program and show its output?

```fortran
PROGRAM FUNCTION_SUBPROGRAMS
   IMPLICIT NONE
```

83
REAL :: A,B,C,TIME,ADD
A = 1.0
B = 2.0
C = 3.0
TIME = 2.0*ADD(A,B,C)
PRINT 10, TIME
10 FORMAT (1X, 'TIME=', F6.2)
STOP
END PROGRAM FUNCTION_SUBPROGRAMS

! FUNCTION SUBPROGRAM
REAL FUNCTION ADD(X,Y,Z)
REAL :: X,Y,Z
Y = X +Y
Z = Y + Z
X = Z + X
ADD = X + Y + Z
RETURN
END FUNCTION ADD

OUTPUT
TIME= 32.00

9.3 Subroutine Subprogram

Subroutines are modules written to perform operations that cannot be performed by a function. For example, if several values need to be returned from a module, a sub-routine is used. A subroutine is also used for operations that do not compute values, such as reading the values in a data file. Subroutine subprograms, like function subprograms, may be either internal or external.
**Internal Subroutines Subprogram**

The form of an internal subroutine subprogram is:

1. Begins with CONTAINS statement that identify that the main program contains the following subprogram.
2. The beginning of the subroutine subprogram is identified by the subroutine statement with name and argument list, according to the following form:

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

4. Execution part
5. Could contain a RETURN statement to return the control to the main program when needed. The return statement is written as:

   ```
   RETURN
   ```

6. The subroutine program must end with END SUBROUTINE statement:

   ```
   END SUBROUTINE subroutine-name
   ```

   The value of the control will be returned to the main program that references it when this statement is encountered or when the RETURN statement is executed.

7. A subroutine is referenced by a CALL statement of the form

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

Write a Fortran program which uses a subroutine program to calculate the average, the minimum value and the maximum value of an array X with number of elements N represents the scores of students grade.

Solution:

```fortran
PROGRAM Scores
INTEGER :: N, I
REAL :: Tests (100), Ave, Min, Max
READ*, N
READ*, (Tests(I), I=I,N)
CALL Stat (Tests, N, Ave, Min, Max)
Print 5, Ave, Min, Max
5 Format (1X, 3F10.3)

CONTAINS
SUBROUTINE Stat (X, M, Xave, Xmin, Xmax)
INTEGER :: M, I
REAL :: X(M), Xave, Xmin, Xmax, Sum
Sum=X(1)
Xmax=X(1)
Xmin=X(1)
DO I=2, M
Sum=Sum+X(I)
IF (X(I) < Xmin) Xmin=X(I)
IF (X(I) > Xmax) Xmax=X(I)
END DO
Xave=Sum/M
RETURN
END SUBROUTINE Stat

END PROGRAM Scores
```
**External Subroutines Subprogram**

External subroutine subprogram can be made accessible to a program unit by attaching it after the `END` statement of the main program.

Many of the rules for writing and using subroutines are similar to those for functions. The following list of rules outlines the differences between subroutines and functions.

1. A subroutine does not represent a value; thus, its name should be chosen for documentation purposes and not to specify a real or integer value.

2. A subroutine is referenced with an executable statement whose general form is

   \[
   \text{CALL subroutine name (argument list)}
   \]

3. The first line in a subroutine identifies it as a subroutine and includes the name of the subroutine and the argument list, as shown in this general form:

   \[
   \text{SUBROUTINE name (argument list)}
   \]

4. A subroutine uses the argument list not only for inputs to the subroutine but also for all values returned to the calling program. The subroutine arguments used in the CALL statement are the actual arguments, and the arguments used in the SUBROUTINE statement are the dummy arguments. The arguments in the CALL statement must match in type, number, and order those used in the subroutine definition.

5. A subroutine may return one value, many values, or no value. Similarly, a subroutine may have one input value, many input values, or no input value.

6. Because the subroutine is a separate program, the arguments are the only link between the main program and the subroutine. Thus, the choice of subroutine statement numbers and variable names is independent of those in the main program. The variables used in the subroutine that are not subroutine arguments are local variables, and their values are not accessible from the main program.
7. The subroutine, like the function, requires a RETURN statement to return control to the main program or to the subprogram that called it. It also requires an END statement because it is a complete program module.

8. A subroutine may reference other functions or call other subroutines, but it cannot call itself.

**Similarity and Differences:**
Subroutine subprograms have many futures in common with function subprograms:
1. They are program units designed to perform particular tasks under the control of some other program unit.
2. They have the same basic form: each consists of a heading, specification part, an execution part, and END statement.
3. They may be internal or external subprograms.
4. They may be used as arguments of other subprograms.
5. They may be recursive.

They differ, however, in the following respects:
1. Functions are designed to return a single value to the main program that references them. Subroutine often return more than one value, or they may return no value at all but simply perform some task such as displaying a list of instructions to the user.
2. Functions return values via function names, subroutines return values via arguments.
3. A function is referenced by using its name in an expression, whereas a subroutines is referenced by a CALL statement.
EXAMPLE (1)
Repeat the example above using external subroutine.

Solution:

PROGRAM Scores
INTEGER :: N, I
REAL :: Tests (100), Ave, Min, Max
READ*, N
READ*, (Tests(I), I=I,N)
CALL Stat (Tests, N, Ave, Min, Max)
Print 5, Ave, Min, Max
5 Format (1X, 3F10.3)
END PROGRAM Scores

SUBROUTINE Stat (X, M, Xave, Xmin, Xmax)
INTEGER :: M, I
REAL :: X(M), Xave, Xmin, Xmax, Sum
Sum=X(1)
Xmax=X(1)
Xmin=X(1)
DO I=2,M
Sum=Sum+X(I)
IF (X(I) < Xmin) Xmin=X(I)
IF (X(I) > Xmax) Xmax=X(I)
END DO
Xave=Sum/M
RETURN
END SUBROUTINE Stat
EXAMPLE (2)
Follow the execution of the following program and show its output?

PROGRAM SUBROUTINE_EXAMPLE
IMPLICIT NONE
INTEGER,DIMENSION (10):: K
INTEGER::I
DO I=1,10
    K(I)=I
END DO
CALL MODIFY (K,10)
PRINT*,”NEW VALUES OF K ARE:”
PRINT 15, (K(I),I=1,10)
15 FORMAT(1X,5I5)
END PROGRAM SUBROUTINE_EXAMPLE

! Subroutine Modify
! Accepts: Elements of K, No. of elements
! Returns: Modified values of elements of K
SUBROUTINE MODIFY(L,N)
INTEGER :: I,N
INTEGER, DIMENSION (N):: L
DO I=1,N
    L(I)=MOD(L(I),4)
END DO
RETURN
END SUBROUTINE MODIFY

OUTPUT:
NEW VALUES OF K ARE:
bbbb1 bbbb 2 bbbb 3 bbbb 0 bbbb 1
bbbb 2 bbbb 3 bbbb 0 bbbb 1 bbbb 2