Chapter 2
Basic elements of Fortran 90/95/2003

- Character set
- Structure of Fortran programming
- Basic elements of programming
  - Identifiers
  - Constants
  - Data types
  - Fortran Operators
  - Fortran expression
- Precedence rules of Operators
- Line Discipline
- Fortran I/O statements
- List Directed Output statements
- Sample Fortran program
  - Formatted Input statements
  - File I/O statements

Character set

- Special alphabet used in FORTRAN programming language is known as FORTRAN character set.
- It consists of the following 86 symbols:
  - ***Alphanumeric characters (52)***
    - ***A … Z,***
    - ***a … z,***
  - ***Digits (10)***
    - ***0 - 9***
  - ***Arithmetic symbols (5)***
    - **+ - * / ****
  - ***Underscore character***
    - **_**
  - ***Special symbols (18)***
    - ***( ) . = , ' $: " % & ; < > ? ! Blank***

Character set

- From the character set, we can build *tokens* which are one of six the types:

Token Type	Example
labels	*100 1234 9999*
constants	*15 30.5 'This is a string' .TRUE.*
keywords	*PROGRAM END IF DO*
names	*MYNAME Manchester_United Chelsea123*
operators	+ - * / ** > =
separators	*( ) : ;*

- From tokens we can build *statements*. e.g.

  X = (-B + SQRT(B ** 2 – 4.0 * A * C)) / (2.0 * A)

- From statements we can build *program units*.

Structure of Fortran programming

Every Fortran program will have the following general structure

PROGRAM *name*

statements

END [**PROGRAM** *name*]

Structure of Fortran programming

- Fortran program structure can be further be seen as a set of the following basic components
  - **Header line,**
  - **Specification part of non-executable statements,**
  - **Execution part and**
  - **Ending part.**

Header Line

- Every Fortran program should start with the PROGRAM statement with the following syntax

  PROGRAM program_name

- This is not executable statement but provides information to the compiler where the program will start execution
- The program names may be up to 31 characters long and any combination of alphabetic characters, digits and underscore.
- The first character of program names should always be alphabetic

Specification statement

- The specification part is a non executable statement and they are used for various purpose:

1. Used in declaration of variables
2. Specify data type (Integer, Real, Double precision, complex, Logic)
3. Specify dimension of an array
4. Specify the variable type (Constant or not)
5. Indicate optional parameters (OPTIONAL)
6. Indicate the intent of an argument (INTENT)
7. Specify possibility of implicit variable usage (IMPLICIT)

Execution statement

- The Execution part consists of a set of instruction that will be executed by the computer. These includes the following:
  - reading from a key board,
  - writing on to a screen,
  - checking status,
  - modifying value, etc

- Execution statements are issues of the course starting from this chapter till the end of the course

Ending statement

- Any type of program unit (main program or sub-programs) ends with a END statement.
- End statement can have optionally the name and type of the program unit
- The form of the end statement is

  END[PROGRAM_UNIT_TYPE[UNIT_NAME]]
- The END statement will terminate the compiler from compilation.

Basic elements of programming

- All programming language involves concepts such as
  - Identifiers
  - Data types
  - Constants
  - Fortran Operators
  - Fortran expression
  - operators and operands
- This chapter will address each of the above concepts from Fortran programming perspective

Identifier

- An identifier is a name that can be used to name
  - Memory locations that store values,
  - function,
  - subroutine,
  - modules, etc
- Identifiers that are used to name memory location that store values can be classified as variable identifiers and constant identifiers

Identifier

- Variable identifiers are also called variable names
  - Used to name memory location whose value can change while the program is running
- Variable identifier naming follow some rule
  - Must be at least one alphabetic character long, up to a maximum of 31 alphanumeric characters.
  - Must start with an alphabetic character.
  - It is case insensitive.
  - Alphanumeric characters are : a-z, 0-9 and the underscore (_).

Identifier

- Variable names can be defined implicitly or explicitly
- The statement used to define explicit variable name is called declaration statement
- According to implicit variable specification, all variables that start with a letter from I to N inclusive (case insensitive) are integers and the rest are real
- Implicit variable use can be suppressed in Fortran programming language with the help of IMPLICIT NONE statement

Identifier

- **Valid identifier names:**
  - X
  - THEDAY
  - Min_cur
  - Time28
  - Y
  - FIRST
  - LAST
  - GCD
  - X1
  - X2
- **Invalid identifier names**
  - X*Z
  - THE TIME
  - 7YEARS
  - _no_way$
  - 5X
  - X_1
  - $COST
  - X&&^1

Constant identifier

- Used to name memory location whose value is initialized when the program get loaded and remain unchanged while the program is running
- Constant identifier naming follow the same rule as above which is also true for all identifiers
- Any constant identifier must be declared before use (no implicit constant identifier)

Data type

- Data type is a specification of a value that can be stored in a memory location or an identifier
- Data types
  - Define a set of values a variable can assume
  - Define the possible operation that can be performed
  - Defines the memory size an object require to be represented in the memory
- Fortran has
  - numeric data types (INTEGER, REAL, DOUBLE PRECISION and COMPLEX),
  - character data types (CHARACTER) and
  - logical data types (LOGICAL).

Data type

Every identifiers (constant or variable) must have data type.

- Data type can be
  - explicitly specified using declaration statement or
  - implicitly given using Fortran implicit usage
- Declaration statement is used to set the data type information explicitly for variable, constant and some other identifiers
- The form of Fortran explicit declaration statement is

<DataType[*SIZE1]> [::] <variable[*SIZE2]> [,<variable(s)>]

Where **Data Type** is either INTEGER, REAL, DOUBLE PRECISION, COMPLEX or LOGICAL

SIZE1 is valid for all except LOGICAL and **SIZE2** is valid only for CHARACTER

Data type

Examples

Integer, *I = 20*

Real *x = 20.25*

Complex *z = (34,45)*

Integer, *I ,J,K*

Real *x,y,z,pi = 22.0/7.0*

- Various conversions are possible between them.

Data type

The following program compile with out any error but doesn't have any execution statement

```
program test
        IMPLICIT NONE
        character *20 name, fname, department1 *10, gender *5
        character ch
        !valid size for integer are 2 or 4
        Integer num
        integer*1 num1
        integer*2 num2
        integer*4 num4
        !valid size for double precision are 8 or 4
        double precision d1
        double precision d2
        real *4 r4
        real *8 r8
        complex c1
        logical answer
end program
```

Data type

- The range of values that you can store in a variable is machine dependent.
- We can specify the size we need in the declaration statement (*SIZE)

For instance,

- an integer can be 2 or 4 bytes long. (default SIZE may be either 2 or 4)
- a real can be 4, 8 or 16 bytes long. (default SIZE may be either 4 or 8)
- A character can take any positive integer (default SIZE is 1)

Data type

- Integer constants are written as a string of digits, possibly preceded by one unary minus or plus sign.
- Examples:

  555 -678 +9

- Fortran operator rules prohibit more than one unary plus or minus, but writing -(-9) is allowed.
- The main reason for using double precision variables is numerical accuracy:
  - real (i.e. in single precision) is usually around 10-6,
  - double precision is around 10-15.

Data type

- Real numbers can be written as integer plus fractional part (3.141592) or in scientific notation (31.41592E-1) where the part after the E is the exponent denoting the powers of ten OR using a D instead of an E which makes the data type double precision .314152D1
- The complex data type is specified by the keyword COMPLEX in a declaration.
- A complex value consists of a pair of REAL variables.
- It can be written as (1.5,-2.3) that is, using two real constants.
- The first element is the real part and the second element is the imaginary part

Data type

- In character variable declaration, SIZE2 override size1 for that specific variable.
- Example: CHARACTER*20 A, B
  declares A and B to be a string of at most 20 characters.
- In CHARACTER*20 A,B,C*10, D
  the variables A, B and D are 20 characters long, but C is ten long.
- Character constants are written as a string of characters delimited by:
  - a pair of single quotes (apostrophes): as in 'ABC 12D' or
  - a pair of double quote as in “ABC 12D”
- Logical variables has only two possible values, written as .TRUE. .FALSE.

Data type

- Various attributes may be specified for variables in their type-declaration statements for instance PARAMETER
- A variable defined with this value, is called constant identifier (PARAMETER)
- Constant identifier’s value can not be changed once it is set during declaration
- Constants referred by name are important
  - to avoid logical error and
  - to ease replacement operation that will result catastrophic problem that couldn’t be debugged easily

Data type

- The syntax for definition of a variable as a parameter is as follow

<**Data Type**[*size]>, PARAMETER [::] <variable> = VALUE

OR

PARAMETER (<variable> = value)

- The second option is possible if implicit data type is allowed or if variable was previously declared

Data type

- It is often used to emphasise key physical or mathematical constants; e.g.

```
REAL, PARAMETER :: PI = 3.14159
REAL, PARAMETER :: GRAVITY = 9.81
```

Other Examples

```
PARAMETER (PI = 3.14159)
PARAMETER (GRAVITY = 9.81)
REAL g,pi
INTEGER days
PARAMETER (days=365)
PARAMETER (g=9.81,pi=3.142)
```

Data type

- To enforce explicit variable declaration in Fortran programming put the following specification statement at the specification section of the program

IMPLICIT NONE

- IMPLICIT NONE statement is to mean implicit declaration is not allowed in this program.

- An IMPLICIT statement also used to allow user defined implicit declaration
- This implicit declaration can be used to change the default setting or confirm the default implied integer and real typing.

IMPLICIT typeName (a [,a]...) [,typeName (a [,a]...)]...

  - Where a is either a single letter or a range of single letters in alphabetical order.
  - A range is denoted by the first and last letter of the range separated by a minus
  - Typename is one of INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER [*len]

- Example

  IMPLICIT LOGICAL (s, v-z), CHARACTER (a-k)

- The Fortran implicit declaration (i-n is integer) and the rest is real will be override by the above implicit statement
- Hence

•A-L	becomes	CHARACTER
•M-N	becomes	INTEGER
•O-R,T-U	becomes	REAL
•S, V-Z	becomes	LOGICAL

- Based on this the following variables will have the stated type

• Bank	character
• tax	REAL
• large	INTEGER
• Set	LOGICAL

Data type

For the all data types stated above, identify the possible range of values that can be assigned to variables declared from the data type

Operators & expression

- Fortran expression is the building block of executable statement.
- Expression consists of a valid sequence of operand and operators
- The operands are either identifiers (constant identifier and/or variable identifiers), functions or constant values.
- Constants can be like

  1, 10, -300 (Integer constant)
  2.3, -30.4 , 1.02E23 (real constants)
  -003D-2 (double constants)
  (3.2, -45.0) (complex constants)
  “hello world”, ‘this is a man’ (character constants)
  .TRUE., .FALSE. (logical constants)

Operators & expression

- There are 4 different categories of basic operators in Fortran.

- These are:
  1. ***Assignment operator***
  2. ***Arithmetic operators***
  3. ***Relational operators***
  4. ***logical operators***

Assignment Operators

- Assignment operator is used to modify the value of a variable identifier with the value of an expression.
- The symbol for the operator is =
- Assignment statement is a statement that involves an assignment operator.
- The syntax of the assignment statement becomes
- Variable = expression
- For example

  X = sin(20*y)

  dept = “computational data science”

  Length = sqrt (x*x + y * y)

Arithmetic Operators

- Are operators that operate on numerical operands and results in a numerical outputs
- There are five different types of arithmetic operators.
- These are:

Operator	**Meaning**	**precedence**
**	Exponentiation as (x**y) to mean x^y	1
*	Multiplication x*y	2
/	division	2
+	Addition as in (x+y) or unary plus	3
-	subtraction as in (x-y) or unary minus	3
//	Character concatenation	Can't be mixed with other operators in an expression

Arithmetic Operators

- An operator that involve two operand is called binary operator

Type Coercion

- When a binary operator has operands of different type, the weaker (usually integer) type is coerced (i.e. converted) to the stronger (usually real) type and the result is of the stronger type.
- This process is called type coercion (type casting).

  3 / 10.0 ⎵ 3.0 /10.0 ⎵ 0.3

- If integer and real values are mixed in an expression then the result will be a real number
- If a real number is assigned to an integer variable, then the numbers after the decimal point will be truncated

Arithmetic Operators

- The biggest source of difficulty is with integer division.
- If an integer is divided by an integer then the result must be an integer and is obtained by truncation towards zero.
- Thus, in the above example, if we had written 3/10 (without a decimal point) the result would have been 0.
- Be careful when mixing reals and integers in mixed mode expressions.
- If you intend a constant to be a real number, use a decimal point! As in 3. 4 or 4.0

Arithmetic Operators Precedence

- Expressions are evaluated in order:
- highest precedence (exponentiation) first, then left to right.
- Brackets ( ), which have highest precedence of all, can be used to override this.
- Assignment operator is evaluated last if it exists in a statement.
- The type conversion in this case is into the type of the variable identifier which is to the left of the assignment operator
- The precedence of arithmetic operators from the highest to the lowest is as follows
- ( ) ≺ ** ≺ * and / ≺ + and -

Arithmetic Operators

Character concatenation

- Can be considered as adding two strings together
- Can be made by using the concatenation operator // as follow
- Str = “the book” // “ is on the table”
- !will assign the string “the book is on the table” to str

Relational Operators

- The relational operator returns a Boolean value based on the comparison result.
- All are binary operators and the operands must be comparable
- Can be used to compare strings as ‘Abebe’ < ‘abebe’
- The above comparison returns .true.
- All have the same priority but lower than arithmetic operators and higher than logical operators

Operator	**Meaning**
<	Less than (x<y) return true if x <y
<=	Less than or equal to (x<=y) return true if x ≤y
>	Greater than (x>y) return true if x >y
>=	Greater than or equal to (x>y) return true if x ≥y
==	equal to (x>y) return true if x = y
/=	equal to (x/=y) return true if x ≠ y

Logical Operators

- Logical operators are operators that manipulate logical operands.
- The following table shows the list of logical operators and their priorities from highest to lowest
- The operands can be result of the relational expression or the logical constants .TRUE. And .FALSE.

Precedence in decreasing order

Operator	**Meaning**
.NOT.	The unary NOT
.AND.	Binary and operator
.OR.	The binary or
.EQV.	The binary equivalent which return true if both operands have the same value other wise false
.NEQV.	This return true if the operands have different value

Precedence rules of Operators

Precedence in decreasing order

- Parenthesis
- Unary minus
- Arithmetic operators
- Relational Operators
- Logical Operator

Line Discipline

- The usual layout of statements is one-per-line, interspersed with blank lines for clarity.
- This is the recommended form in most instances.
- However, There may be more than one statement per line, separated by a semicolon; e.g.

  A = 1; B = 10; C = 100

- This is only recommended for simple initialization.
- Each statement may run onto one or more continuation lines if there is an ampersand (&) at the end of the line to be continued. e.g.

  RADIANS = DEGREES * PI &
  / 180.0

- is the same as the single-line statement

  RADIANS = DEGREES * PI / 180.0

- There may be up to 132 characters per line.

Fortran I/O statements

- Input statements provide the means of transferring data from external media to internal storage or from an internal file to internal storage.
  - This process is called reading.
- Output statements provide the means of transferring data from internal storage to external media or from internal storage to an internal file.
  - This process is called writing.
- Fortran I/O statements require information about
  - The logical unit
  - Type of I/O statement and others

Fortran I/O statements

- There are three data transfer input/output statements
  1.READ
  2.WRITE
  3.PRINT
- Print and write statements are output statements and read is input statement.

Logical unit

- A logical unit is a channel through which data transfer occurs between the program and device or file.
- You identify each logical unit with a logical unit number, which can be any nonnegative integer from 0 to a 4 byte integer value
- Unit number is linked with the logical number by open statement
- Certain unit numbers are pre-connected to standard devices.
- Unit number 5 is associated with stdin , unit 6 with stdout , and unit 0 with stderr .
- Fortran implicitly opens units 5, 6, and 0 and associates them with their respective operating system standard I/O files

Types of I/O statement

There are 3 types of I/O statement in FORTRAN

1. List-Directed Statements (It is part of formatted I/O but the format is already defined by the compiler)
2. Formatted I/O statements
3. Unformatted I/O statements (Reading Exercise)

List Directed I/O statements

- List-directed I/O, sometimes called "free format" I/O, is the simplest kind of formatted I/O.
- List-directed I/O lets the compiler select a format for the I/O data, depending upon the type and magnitude of the data in the variable list on the I/O statement (therefore the name list-directed).
- List-directed I/O is selected by an asterisk where the format specification normally appears.

List Directed Input statements

- A list-directed READ statement can be written in one of two ways.
- First, the READ statement can specify the input unit number for the source of the input data.

  For example, READ(5,*) i, j, k
- Second, the READ statement does not have to specify the unit number; instead, the input data is taken from the standard input device.

  For example, READ *, i, j, k
- Data items read by a list-directed READ statement can be separated by a comma, blanks, or can be on separate lines.

- This flexibility makes list-directed input convenient for you to enter data.
- **Syntax**

  Read *, var1 [, var12][,….]

  Read(*,*) var1 [, var2][,….]

List Directed Input statements

- The input field can be terminated abruptly by entering a slash (/) in the input field, causing any remaining items in the I/O list to be skipped.
- For example, if the input line contains the character sequence 5 / is read by one of these statements:

```
READ(5,*) i, j, k
READ *, i, j, k
```

- the variable i is assigned the value 5, and the read terminates.
- The variables j and k are unchanged.
- Entering a slash is useful when you want to enter only the first few values of a long input list.

List Directed Output statements

- You can include a comma before the list of data items in the WRITE statement, as shown below:

  WRITE(6,*),a,b,c

- List-directed output prints numeric data with a leading blank.
- Character data is printed without any leading blanks.
- Numeric data can be printed in scientific notation, depending upon the magnitude.
- However, you should not use list-directed output for applications where the exact format of the output data is critical.

Sample Fortran program

- In this section we will see some sample Fortran programs that demonstrates some basic concepts in Fortran programming language

  - Some of the concepts include
  - List directed I/O statements and sample programs
  - Assignment statements and sample programs
  - Line disciplines sample programs
  - Data statement for simple initialization

Sample Fortran program (I/O)

Simple Syntax

Read *, var1 [, var12][,….]
Read(*,*) var1 [, var2][,….]

Print *, exp1 [, exp2][,….]
Write(*,*) exp1 [, exp2][,….]

- The meaning of the asterisk will be clear later

Sample Fortran program (I/O)

```
!This program display Hello world on to the screen
program test
      IMPLICIT NONE
      print *, "Hello world!"
end program
```

Sample Fortran program (I/O)

```
PROGRAM test_hello_world
    IMPLICIT NONE
    REAL :: x
    PRINT *, “hello world”
    PRINT *, “I am from Addis Ababa university”
    READ *, x
    PRINT *, “The value of X is ", x
    STOP
END test_hello_world
```

Sample Fortran program (I/O)

```
!This program shows how to use relational operators
program test
IMPLICIT NONE
    integer x, y;
    print *, "Enter two numbers "
    read *, x,y
    print *, "X >= y ", x >= y
    print *, "X > y ", x > y
    print *, "X == y ", x == y
    print *, "X < y ", x < y
    print *, "X <= y ", x <= y
    print *, "X /= y ", x /= y
end program
```

Sample Fortran program (I/O)

```
!This program shows how to use logical operators
program test
IMPLICIT NONE
      print *, ".true. .and. .true.      ", .true. .and. .true.
      print *, ".true. .or. .true.       ", .true. .or. .true.
      print *, ".true. .eqv. .true.      ", .true. .eqv. .true.
      print *, ".true. .eqv. .false.     ", .true. .eqv. .false.
      print *, ".true. .neqv. .false.        ", .true. .neqv. .false.
      print *, ".false. .neqv. .false.  ", .false. .neqv. .false.
      print *, ".not. .false.            ", .not. .false.
end program
```

Sample Fortran program (Assignment Statements)

- Assignment statements are statements that involves assignment operator (=)

  Syntax

  Variable = expression

- The effect of an assignment operator is to assign the value of the expression at the right side of the assignment operator to the left side identifier

Sample Fortran program (I/O)

```
!This program shows how we can use arithmetic operators and
assignment operators in a program
program test
    IMPLICIT NONE
    integer x, y, o1,o2,o3,o4
    real l,m,n1,n2,n3,n4
    character *20 name1, fname, fullName*42
    x = 10
    y = 2
    name1 = "Tesfaye"
    o1 = x ** y
    n1 = l ** m
    print *, o1, x, n1, name
end program
```

Sample Fortran program (Line Discipline)

- The following sample programs demonstrate how to use one line for multiple statements and multiple lines for one statement

Sample Fortran program (I/O)

```
!This program shows how we can use one line for multiple statements
program test
    IMPLICIT NONE
    integer x, y, o1,o2,o3,o4
    real l,m,n1,n2,n3,n4
    character *20 name1, fname, fullName*42
    x = 10;       y = 2;         l = 10.0;     m = 2.0
    name1 = "Tesfaye";    fname = "Bekele"
    o1 = x ** y;     o2 = x * y; o3 = x / y
    n1 = l ** m;     n2 = l * m; n3 = l / m
    fullName = name1 // "  "// fname
    print *, o1, o2, o3, n1, n2, n3, fullName
end program
```

Sample Fortran program (I/O)

```
!This program allows two write single instruction on multiple line
and more than one instruction on one line
program test
    IMPLICIT NONE
    real area, circumference
    real PI, radius
    parameter(PI=3.14174)
    print *, "I am going to compute the area and the ", &
        "circumference of a given circle", &
        "Please enter the radius"; read *, radius
    area = PI * radius ** 2; circumference = 2 * PI * radius
    print *, "Area = ", area, "  circumference = ", circumference
end program
```

Sample Fortran program (Data Statement)

- Is used to initialize variables and arrays at the time of declaration.
- The form of the data statement is

  DATA list-of-variables / list-of-values/, ...
- Examples

  data m/10/, n/20/, x/2.5/, y/2.5/

  data m,n/10,20/,x,y/2*2.5/ !set m, n, x, y to 10,20,2.5,2.5 respectively
- Use of DATA statement for initializing array will be discussed on chapter four

Sample Fortran program (Data Statement)

```
PROGRAM SQUARE_ROOTS
IMPLICIT NONE
INTEGER :: X,Y,Z;
REAL :: X1, Y1, Z1;

    DATA X,Y,Z /10, 34, 55/ !initialization

    !assignment
    X1 = SQRT(real(X))
    !multiple assignment on a single line
    Y1 = SQRT(real(Y)); Z1 = SQRT(real(Z))
    !output statement
    WRITE(*,*) "the square root of ",x, " is ", x1
    PRINT *,"the square root of ",y, " is ", y1
    PRINT *,"the square root of ",z, " is ", z1

    PRINT *,"the square root of ",x, " is ", x1, “the square root of ",y, " is ", y1, &
     “the square root of ",z, " is ", z1 !does the same output as above
    READ(*,*) x!input statement just to pause the output screen
    stop
end program square_roots
```

Formatted I/O statements

- Formatted I/O statements use format descriptors that supplement the READ, WRITE, or PRINT statements to exactly define the format of the data.
- The descriptors can be specified on the READ, WRITE, or PRINT statements, or can appear in a FORMAT statement.
- The FORMAT statement is a non-executable statement that describes how the data listed in a READ, WRITE, or PRINT statement is to be arranged.
- The FORMAT statement can appear anywhere in a program after a PROGRAM, FUNCTION, or SUBROUTINE statement.
- The type of conversion indicated by the format descriptors should correspond to the data type of the variable in the I/O list.

Formatted Input statements

- The formatted READ statement transfers data from an external device to internal storage, and converts the ASCII data to internal representation according to the format descriptor.
- One way of specifying a formatted READ statement is to place the descriptors on the READ statement itself.
- For example, the statement:

  READ (5, '(I5, F5.1)') int, value

- reads data from unit 5 into int and value according to the format descriptors I5 and F5.1, respectively.

Formatted Input statements

- If there are many format descriptors or if the same descriptors are used repeatedly, place the descriptors in a FORMAT statement.
- The statements:

```
READ (5,100) a, b, c
100 FORMAT (F7.1, F8.1, F9.1)
```

- read data from unit 5 into the variables a, b, and c according to the format descriptors F7.1, F8.1, and F9.1, respectively.
- The above read statement is the same as

```
READ 100, a, b, c
```

Formatted Output statements

- The formatted WRITE and PRINT statements transfer data from the storage location of the variables or expressions named in the output list to the file associated with the specified unit.
- The data is converted to a string of ASCII characters according to the format descriptors.
- One way of specifying a formatted WRITE statement is to include the format descriptors on the WRITE statement itself.
- For example, the statement:

```
WRITE (6, '(1X, I5, F3.1)') int_value, real_value
```

- Writes the values of int_value and real_value to unit 6 (the pre-connected output unit) according to the format descriptors I5 and F3.1, respectively.

Formatted I/O statements

- The statement:

```
PRINT '(1X, I5, F3.1)', int_value, real_value
```

  - also writes data from int_value and real_value to the standard output device according to the format descriptors I5 and F3.1.
- To use a FORMAT statement, specify its label in the corresponding WRITE or PRINT statements.
- For example, the statements:

```
WRITE (6,100) int_value, real_value
100 FORMAT (1X, I5, F3.1)
```

  - write data from variables int_value and real_value to the pre-connected output unit according to the descriptors in the FORMAT statement labeled 100.

Formatted I/O statements: Format Descriptors

- In the following table we use the following characters with the following definition:

  - w is the field width (the character space)
  - d is the digits to the right of the decimal point
  - m is the minimum digits to be output (if omitted, m = 1)
  - e is the number of digits of the exponent (if omitted, e = 2)

Formatted I/O statements: Format Descriptors

Table1: Summary of the Format Descriptors

Data Conversion type	**Format descriptors**	**Forms**	**Data declaration Allowed**
Character	**A**	**A[w]**	All data types
	R	**R[w]**	All data types
Logical	**L**	**L[w]**	LOGICAL
Real	**D**	D[*w.d*]	All numeric data types
	E	E[*w.d*[E*e*]]	All numeric data types
	F	F[*w.d*]	All numeric data types
	G	G[*w.d*[E*e*]]	All numeric data types
Integer	**I**	I[*w*[*.m*]]	All numeric data types

Formatted I/O statements: Format Descriptors

Table1: Summary of the Format Descriptors

Data Conversion type	**Format descriptors**	**Forms**	**Data declaration Allowed**
Monetary	**M**	M[*w.d*]	All numeric data types
Numeration	**N**	N[*w.d*]	All numeric data types
Octal	**O**	O[*w*[.*m*]]	Input: INTEGER
	K	K*w*[.*m*]]	Output: All data types
	@	@*w*[.*m*]]	
Hexadecimal	**Z**	Z[*w*[.*m*]]	Input: INTEGER Output: All data types

File I/O statements

- In the previous exercise all input and output was performed to the default devices namely the screen and the keyboard.
- In many circumstances this is not the most appropriate action
- This is mainly because
  1. temporary storage of large amounts of intermediate results;
  2. large amounts of input or output;
  3. output from one program used as the input of another;
  4. a set of input data which is used many times.

File I/O statements

- A mechanism is required which permits a programmer to direct input to be performed on data from a source other than the keyboard (during execution time) and to store output in a more "permanent" and capacious form.
- This is generally achieved by utilizing the computer's file store which is a managed collection of files.

File I/O statements

- A file such as the source program or a set of I/O data is normally formatted, which means it consists of an ordered set of character strings separated by an end of record marker.
- A formatted file may be viewed using an editor or printed on a printer.
- An unformatted file has no discernable structure and should be regarded as single stream of bytes of raw data.
- An unformatted file is normally only viewed using a suitable user written program.

File I/O statements

- Three basic steps are worth knowing to use file I/O

1. Opening the file (creating a channel through which the program can communicate with the physical device).
2. Managing the file
3. Closing the file

File I/O statements

- While opening a file, you may need to know one or more of the following
  - Free logical unit number
  - The name and type of the file
  - The action that you need to perform on the file
  - The status of the file
  - Error recovery means and error trapping mechanism
  - The format of the file
  - The file access method
  - The record length
  - Where to point the file pointer while opening the file
  - Weather you need to pad blank space on a formatted file or not
  - Weather you need to delimit lists on a formatted file or not

File I/O statements: Opening file

- The syntax to open a file is

  OPEN(U, [Options])

- Where U is the unit number which can be

  UNIT=INTEGER or * or the name of an internal file

- Options can be
  - FILE = filename !the keyword FILE can be omitted if it is the second argument
  - ACTION=act, where act may be 'READ', 'WRITE' or 'READWRITE' specifying the permitted modes of operation on the file. Default is processor dependent.
  - FORM=fm, where fm may be 'FORMATTED' or 'UNFORMATTED', the default is 'FORMATTED' for sequential files and 'UNFORMATTED' for direct access files.
  - ACCESS=acc, where acc may be 'SEQUENTIAL' or 'DIRECT’

File I/O statements: Opening file

- STATUS=st, where st may be 'OLD', 'NEW', 'REPLACE', 'SCRATCH' or 'UNKNOWN'. If 'OLD' is specified the file must exist; if 'NEW' the file must not exist; if 'REPLACE' and the file exists it will be deleted before a new file is created; and if 'SCRATCH' the file will be deleted when closed. In general use 'OLD' for input and 'NEW' for output.
- ERR=label, Where label is GOTO label if an error occurs while opening the file.
- IOSTAT=ios, where ios is an integer variable which is set to zero if the statement is executed successfully or to an implementation dependent constant otherwise.
- RECL=rl, where rl is the maximum record length (positive integer) for a direct access file. For formatted files this is the number of characters and for unformatted it is usually the number of bytes or words (system dependent).

File I/O statements: Opening file

- BLANK=bl, where bl is either 'NULL' or 'ZERO' and determines how blanks in a numeric field are interpreted.
- POSITION=pos, where pos may be 'ASIS', 'REWIND' or 'APPEND' which are interpreted as positioning the file at the position it was previously accessed (ASIS), positioning the file at the start (REWIND); and positioning the file after the previously end of the file (APEND). Defaults to ASIS.
- PAD=pad, where pad may be 'YES' or 'NO'. If 'YES' formatted input is padded out with blank characters; if 'NO' the length of the input record should not exceeded the format specified for the input variable.
- DELIM=del, where del may be 'APOSTROPHE' or 'QUOTE' or 'NONE' indicating which character used when delimiting character expressions in list-directed or NAMELIST output. Defaults to 'NONE'.

File I/O statements: Opening file

- Examples
  - OPEN (UNIT=10,FILE='fibonacci.out')
  - OPEN(2, FILE='myfile', STATUS='NEW')
  - OPEN(9, FILE='datafile', STATUS='NEW', ERR=180)
  - OPEN (UNIT=11,FILE='fibonacci.out',STATUS='NEW',ERR=10)
  - OPEN (UNIT=12, FILE='student.records', STATUS='OLD', &
  - ACCESS='DIRECT',RECL=200, FORM='FORMATTED',&
  - ERR=20, IOSTAT=IOS)

File I/O statements: Managing file

- Managing the file
- This refers to reading or writing from/on the file
- Syntax for Reading
  - READ (clist) [Input list]
  - where clist is defined as
  - [UNIT=] unit-number, !from which device
  - [,FMT=] format-spec !format for formatted input
  - [,REC= record-number] !which record
  - [,IOSTAT=ios] !flag to check success/failure
  - [,ADVANCE=adv] !yes or no; no is the default to mea
    !put the pointer on the same line
  - [,SIZE=integer-variable] !the size of input information
  - [,EOR=label] !labels to go when end of Record
  - [,END=label] !labels to go when end of File
  - [,ERR=label] !labels to go when error occur

File I/O statements : Managing file

- Syntax for Writing

  Write (clist) [Output list]

- where clist is defined as
- [UNIT=] unit-number, !from which device
- [FMT=] format-spec !format for formatted input
- [,REC= record-number] !which record
- [,IOSTAT=ios] !flag to check success/failure
- [,ADVANCE=adv] !No means don't insert EOLine
  !default is yes
- [,SIZE=integer-variable] !the size of the output data
- [,EOR=label] !labels to go when end of Record
- [,END=label] !labels to go when end of File
- [,ERR=label] !labels to go when error occur

File I/O statements: Closing File

- This statement permits the orderly disconnection of a file from a unit either at the completion of the program, or so that a connection may be made to a different file or to alter a property of the file.
- The CLOSE statement has the general form

CLOSE ([UNIT=]u [Options])

Where Options are

- [,IOSTAT=ios] !to get success or failure status
- [,ERR=label] !label to go on error
- [,STATUS=st]) !st can be 'KEEP' or 'DELETE'.
- !The value 'KEEP' cannot be applied to a
- !file opened as 'SCRATCH'.

For example:

```
CLOSE (10)
CLOSE (UNIT=10, ERR=10)
CLOSE (UNIT=NUNIT, STATUS='DELETE',ERR=10)
```

File I/O statements: Exercise

1. Complete the following statement, which would open an unformatted direct access file with a record length of 100 bytes
   OPEN(UNIT=10,.....)
2. Write a section of code which would open 10 files on the unit numbers from 20 to 29. The default values should be used for all keywords.
3. Write sections of code to perform the following
   (a) test for the existence of a file called TEMP.DAT
   (b) test if a file has been opened on unit 10
   (c) test to see if the file opened on unit 15 is a direct access file and if so what the record length is.

   The program fragments should output the results in a suitable form.

File I/O statements: Exercise

4. Write a Fortran program which will prompt the user for a file name, open that file and then read the file line by line outputting each line to the screen prefixed with a line number.
   Use the file which contains the source of the program as a test file.
5. Write a Fortran program which will prompt for the name of an existing file and read that file sequentially writing each line to the next record of the temporary direct access file.
   The program should then repeatedly prompt the user for a number representing the number of a line in the input file and display that line on the screen.
   The program should halt when the number 0 is entered. The program should handle all possible error conditions such as the file does not exist or a line number out of range is specified and inform the user accordingly.