Read the attached reference : Ref 1

1 15.04.2021 - Prepare your own notes before 25.04.2021

1.1 Go through the following video lectures - Click each title

- ▶ **Input and Output Devices**
- ▶ **Central Processing Unit**
- ▶ **Computer Languages**
- ▶ **System Software and Application Software**
- ▶ **Components of Computer System**

1.2 Find the answers of these questions

1. What are input and output devices of a computer ? Give Examples
2. What is meant by CPU ? What are its parts ?
3. What is meant by a computer program ?
4. What are programming languages ?
5. Differentiate between low level and high level languages. Give Examples
6. What are compilers and interpreters ?
7. What is system software ? Give Example
8. What is an application software ? Give examples
9. What is a software package ? Give examples
10. Write a short note on algorithm and flowchart

2 26.04.2021 - Prepare your own notes before 15.05.2021

2.1 Go through the following video lectures - Click each title

- ♫ **First Programming Language**
- ♫ **Introduction to Fortran - I**
- ♫ **Introduction to Fortran - II**

2.2 Find the answers of these questions

1. What does the word "FORTRAN" imply ?
2. Who developed Fortran ?
3. Give the details of any 5 versions of Fortran ?
4. Give a short note on fields of application of Fortran language
5. What is the general structure of a Fortran programme ?
6. Explain how comments are shown in Fortran programme ?
7. What is continuation mark in Fortran ?
8. Explain in detail variable types and their names and declaration in Fortran
9. Explain *parameter* statement with example
10. What are expressions and assignments ? Explain with examples

3 15.05.2021 - Prepare your own notes before 31.05.2021

3.1 Go through the following video lectures - Click each title

- **Introduction to Fortran - III**
- **Introduction to Fortran - IV**

3.2 Find the answers of these questions

1. What are integer constants and real constants ? How are they declared in Fortran ?
2. What are named constants ?
3. Why do we use IMPLICIT NONE in Fortran ?
4. Pick the incorrect floating point constants from the following list. Explain why they are incorrect:
   (a) 40,943.65
   (b) 425E2.5
   (c) .0045E + 6
   (d) 1/2.2
   (e) 465
   (f) 43
5. Which are the arithmetic operation symbols used in Fortran ? What is the precedence of operators ? Explain with examples

6. Show how $x^{1/4} - y^{-3}$ is coded in Fortran ?
7. What is an assignment expression ?
8. What is meant by type conversion ? Explain
9. What are built-in functions or intrinsic functions in Fortran ? Give examples
10. What are Fortran reserved keywords ?
11. $REAL :: a = 1.5, b = 3.0; INTEGER :: i$ Calculate the following

$$i = b/2.0 + b * 4.0/a - 8.0$$

12. Write a program to convert Fahrenheit temperature to Centigrade
13. Write a program to read the radius of a circle in cms. and compute its circumference and area
14. Explain in detail with examples the input and output statements in Fortran

4 01.06.2021 - Prepare your own notes before 10.06.2021

4.1 Find the answers of these questions

1. What are relational operators ?
2. What is a conditional statement in Fortran ?
3. Explain the IF construct syntax in Fortran
4. Given a 4 digit number representing a year write a program to find out whether it is a leap year
5. Explain various loop constructs in Fortran
6. What are the rules to be followed in setting up DO loops ?
7. Given any decimal number write a program to find its octal equivalent. For example, the octal equivalent of 242 is 362
8. What are logical constants, variables and expressions ?
9. Explain how to declare, read and write strings of characters
10. What are format statements ?

5 11.06.2021 - Prepare your own notes before 18.06.2021

5.1 Find the answers of these questions

1. What are subprograms in Fortran ?
2. Explain how to write and use user defined functions in Fortran
3. Explain how to write and use subroutines in Fortran
4. What are local and global variables ?
5. What are External functions or function subprograms ?
6. What are the syntax rules of function subprogram ?
7. What are the differences between function subprogram and subroutine subprogram ?
8. Write a Fortran program to read an integer from user and to find its multiplication table
9. Write a Fortran program to find the factorial of a number
10. Write a Fortran program to print Fibonacci series
11. Write a Fortran program to find the roots of a quadratic equation

Ref 1 nputer & FORTRAN: Computers: Introduction -input & output devices - CPU, Applications - languages & packages (outline only).

Fortran: Constants, variables, operators - mode of expressions - arithmetic to FORTRAN expression - Hierarchy of operators, Statements- conditional and unconditional - i/p & o/p Statements - executable

Statements - format and go to Statements - computed go to - arithmetic IF - logical IF, Built-in functions, Do statement - simple Do loop - function sub program Subroutine sub program (Introduction)

UNIT -IV

Programming: Algorithm - Flow Chart - Simple programs using FORTRAN: Area and volume of geometrical structures, sum of series, product of 'n' numbers, Straight line, ellipse, parabola and their slope.

Ref 1

A computer is a machine that can be programmed to accept data (input), process it into useful information (output), and store it away (in a secondary storage device) for safekeeping or later reuse. The processing of input to output is directed by the software but performed by the hardware.

To function, a computer system requires four main aspects of data handling: input, processing, output, and storage. The hardware responsible for these four areas operates as follows:

- Input devices accept data in a form that the computer can use; they then send the data to the processing unit.
- The processor, more formally known as the central processing unit (CPU), has the electronic circuitry that manipulates input data into the information people want. The central processing unit executes computer instructions that are specified in the program.
- Output devices show people the processed data-information in a form that they can use.
- Storage usually means *secondary storage*. Secondary storage consists of devices, such as diskettes, which can store data and programs outside the computer itself. These devices supplement the computer's *memory*, which, as we will see, can hold data and programs only temporarily.
- A computer system has three main components: *hardware, software*, and *people*. The equipment associated with a computer system is called *hardware*. *Software* is a set of instructions that tells the hardware what to do. People, however, are the most important component of a computer system - people use the power of the computer for some purpose. In fact, this course will show you that the computer can be a *tool* for just about anyone from a business person, to an artist, to a housekeeper, to a student - an incredibly powerful and flexible tool.

Ref 1e Processor and Memory

In a computer the processor is the center of activity. The processor, as we noted, is also called the central processing unit (CPU). The central processing unit consists of electronic circuits that interpret and execute program instructions, as well as communicate with the input, output, and storage devices.

It is the central processing unit that actually transforms data into information. Data is the raw material to be processed by a computer. Such material can be letters, numbers, or facts like grades in a class, baseball batting averages, or light and dark areas in a photograph. Processed data becomes *information*, data that is organized, meaningful, and useful. In school, for instance, an instructor could enter various student grades (data), which can be processed to produce final grades and perhaps a class average (information). Data that is perhaps uninteresting on its own may become very interesting once it is converted to information. The raw facts (data) about your finances, such as a paycheck or a donation to charity or a medical bill may not be captivating individually, but together, these and other acts can be processed to produce the refund or amount you owe on your income tax return (information).

Computer memory, also known as primary storage, is closely associated with the central processing unit but separate from it. Memory holds the data after it is input to the system and before it is processed; also, memory holds the data after it has been processed but before it has been released to the output device. In addition, memory holds the programs (computer instructions) needed by the central processing unit.

Secondary Storage

Secondary storage provides additional storage separate from memory. Secondary storage has several advantages. For instance, it would be unwise for a college registrar to try to keep the grades of all the students in the college in the computer's memory; if this were done, the computer would probably not have room to store anything else. Also, memory holds

Ref 1a and programs only temporarily. Secondary storage is needed for large volumes of data and also for data that must persist after the computer is turned off.

The two most common secondary storage mediums are magnetic disk and magnetic tape. A magnetic disk can be a diskette or a hard disk.

The hardware devices attached to the computer are called peripheral equipment. Peripheral equipment includes all input, output, and secondary storage devices. In the case of personal computers, some of the input, output, and storage devices are built into the same physical unit. In many personal computers, the CPU and disk drive are all contained in the same housing; the keyboard, mouse, and screen are separate.

In larger computer systems, however, the input, processing, output, and storage functions may be in separate rooms, separate buildings, or even separate countries. For example, data may be input on terminals at a branch bank and then transmitted to the central processing unit at the headquarters bank. The information produced by the central processing unit may then be transmitted to the international offices, where it is printed out. Meanwhile, disks with stored data may be kept in bank headquarters and duplicate data kept on disk or tape in a warehouse across town for safekeeping.

Although the equipment may vary widely, from the simplest computer to the most powerful, by and large the four elements of a computer system remain the same: input, processing, output, and storage. Now let us look at the way computers have been traditionally classified.

Classification of Computers

Computers come in sizes from tiny to monstrous, in both appearance and power. The size of a computer that a person or an organization needs depends on the computing requirements. Clearly, the National Weather Service, keeping watch on the weather fronts of many continents, has requirements different from those of a car dealer's service department that is trying to keep track of its parts inventory. And the requirements of both of them are different from the needs of a salesperson using a small

Ref 1top computer to record client orders on a sales trip.

Supercomputers

The mightiest computers-and, of course, the most expensive-are known as supercomputers . Supercomputers process billions of instructions per second. Most people do not have a direct need for the speed and power of a supercomputer. In fact, for many years supercomputer customers were an exclusive group: agencies of the federal government. The federal government uses supercomputers for tasks that require mammoth data manipulation, such as worldwide weather forecasting and weapons research. But now supercomputers are moving toward the mainstream, for activities as varied as stock analysis, automobile design, special effects for movies, and even sophisticated artworks

Mainframes

In the jargon of the computer trade, large computers are called mainframes. Mainframes are capable of processing data at very high speeds-millions of instructions per second-and have access to billions of characters of data. The price of these large systems can vary from several hundred thousand to many millions of dollars. With that kind of price tag, you will not buy a mainframe for just any purpose. Their principal use is for processing vast amounts of data quickly, so some of the obvious customers are banks, insurance companies, and manufacturers. But this list is not all-inclusive; other types of customers are large mail-order houses, airlines with sophisticated reservation systems, government accounting services, aerospace companies doing complex aircraft design, and the like.

Personal Computers

Personal computers are often called PCs. They range in price from a few hundred dollars to a few thousand dollars while providing more computing power than mainframes of the 1970s that filled entire rooms. A

Ref 1 usually comes with a *tower* that holds the main circuit boards and disk drives of the computer, and a collection of *peripherals*, such as a keyboard, mouse, and monitor.

Internet and Networking

The Internet is the most widely recognized and used form of *computer network* . Networks connect computers to each other to allow communication and sharing of services. Originally, a computer user kept all the computer hardware in one place; that is, it was centralized in one room. Anyone wanting computer access had to go to where the computer was located. Although this is still sometimes the case, most computer systems are decentralized. That is, the computer itself and some storage devices may be in one place, but the devices to access the computer-terminals or even other computers-are scattered among the users. These devices are usually connected to the computer by telephone lines. For instance, the computer and storage that has the information on your checking account may be located in bank headquarters. but the terminals are located in branch banks all over town so a teller in any branch can find out what your balance is. The subject of decentralization is intimately tied to data communications, the process of exchanging data over communications facilities, such as the telephone.

A network uses communications equipment to connect computers and their resources. In one type of network, a local area network (LAN), personal computers in an office are hooked together so that users can communicate with each other. Users can operate their personal computers independently or in cooperation with other PCs or mainframes to exchange data and share resources. We discuss computer networks in detail in a later chapter.

Input Devices

Input device is any peripheral (piece of computer hardware equipment to provide data and control signals to an information processing system such as a computer or other information appliance.

Ref 1ut device Translate data from form that humans understand to one that the computer can work with. Most common are keyboard and mouse.

Example of Input Devices:-

1. Keyboard	2. Mouse (pointing device)	3. Microphone
4. Touch screen	5. Scanner	6. Webcam
7. Touchpads	8. MIDI keyboard	9.
10.Graphics Tablets	11.Cameras	12.Pen Input
13.Video Capture Hardware	14.Microphone	15.Trackballs
16.Barcode reader	17.Digital camera	18.Joystick
19.Gamepad	20.Electronic Whiteboard	21.

Output devices

An output device is any piece of computer hardware equipment used to communicate the results of data processing carried out by an information processing system (such as a computer) which converts the electronically generated information into human - readable form.

Example on Output Devices:

1. Monitor	2. LCD Projection Panels
3. Printers (all types)	4. Computer Output Microfilm (COM)
5. Plotters	6. Speaker(s)
7. Projector	

Software

Software is actually a computer *program*. To be more specific, a program is a set of step-by-step instructions that directs the computer to do the tasks you want it to do and to produce the results you want. A computer programmer is a person who writes programs. Most of us do not write programs, we use programs written by someone else. This means we are *users* - people who purchase and use computer software.

Software is a generic term for organized collections of computer data and instructions, often broken into two major categories: system software that provides the basic non – task specific functions of the computer, and application software which is used by users to accomplish specific tasks.

Ref 1 tware is a collection of so many programs.

Software Types

A. **System software** is responsible for controlling, integrating, and managing the individual hardware components of a computer system so that other software and the users of the system see it as a functional unit without having to be concerned with the low -level details such as transferring data from memory to disk, or rendering text onto a display. Generally, system software consists of an operating system and some fundamental utilities such as disk formatters, file managers, display managers, text editors, user authentication (login) and management tools, and networking and device control software.

B. **Application software** is used to accomplish specific tasks other than just running the computer system. Application software may consist of a single program, such as an image viewer; a small collection of programs (often called a software package) that work closely together to accomplish a task, such as a spreadsheet or text processing system; a larger collection (often called a software suite) of related but independent programs and packages that have a common user interface or shared data format, such as Microsoft Office, which consists of closely integrated word processor, spreadsheet, database, etc.; or a software system, such as a database management system, which is a collection of fundamental programs that may provide some service to a variety of other independent applications.

Programming Languages

A program is a set of instructions to do a specific task – simple examples are – to find the smallest of n-numbers, to plot some graphs etc.

A **programming language** is really a set of tools that allow us to program at a much higher level than the 0s and 1s that exist at the lowest levels of the computer. There are two types of programming languages-low-level languages and high-level languages.

Programming languages that are machine dependent are called **low level languages**. Loosely speaking, computers can only execute programs written in low-level languages-sometimes referred to as machine

Ref 1guages or assembly languages. Programming a computer by utilizing hex or binary code is known as machine language programming. Programming a microcomputer by writing mnemonics is known as assembly language programming.

On the other hand, programming languages that are machine independent are called **high level languages**. In comparison to low-level programming languages, it may use natural language elements, be easier to use, or be from the specification of the program, making the process of developing a program simpler and more understandable with respect to a low-level language. Python is an example of a high-level language;other high-level languages you might have heard of are C, C++, Perl, and Java,BASIC, FORTRAN, ALGOL, COBOL. It is much easier to program

in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. High-level languages are portable, meaning that they can run on different kinds of computers with a few or no modifications.

Ref 1: 2 Introduction to Fortan-77 language & Module 5 Computer problems using FORTRAN

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

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

The most common Fortran version today is still Fortran 77.A major advantage Fortran has is that it is standardized by ANSI and ISO. Consequently, if our program is written in ANSI Fortran 77, using nothing outside the standard, then it will run on any computer that has a Fortran 77 compiler. Thus, Fortran programs are portable across machine platforms.

1.Fortran 77 Basics

A Fortran program is just a sequence of lines of text. The text has to follow a certain *structure* to be a valid Fortran program. We start by looking at a simple example:

```
program circle
real r, area

c This program reads a real number r and prints
c the area of a circle with radius r.

write (*,*) 'Give radius r:'
read  (*,*) r
area = 3.14159*r*r
write (*,*) 'Area = ', area

stop
end
```

The lines that begin with with a "c" are *comments* and have no purpose other than to make the program more readable for humans. Originally, all Fortran programs had to be written in all upper-case letters. Most people now write lower-case since this is more legible, and so will we. You may wish to mix case, but Fortran is not case-sensitive, so "X" and "x" are the same variable.

2.Program organization

A Fortran program generally consists of a main program (or driver) and possibly several subprograms (procedures or subroutines). For now we will place all the statements in the main program; subprograms will be treated later. The structure of a main program is:

```
program name

declarations

statements

stop
```

Sandeep K V/9447546957

Ref 1

3.Column position rules

Fortran 77 is *not* a free-format language, but has a very strict set of rules for how the source code should be formatted. The most important rules are the column position rules:

Col. 1 : Blank, or a "c" or "*" for comments
Col. 1-5 : Statement label (optional)
Col. 6 : Continuation of previous line (optional)
Col. 7-72 : Statements
Col. 73-80: Sequence number (optional, rarely used today)

Most lines in a Fortran 77 program starts with 6 blanks and ends before column 72, i.e. only the statement field is used.

4.Comments

A line that begins with the letter "c" or an asterisk in the first column is a comment. Comments may appear anywhere in the program. Well-written comments are crucial to program readability. Commercial Fortran codes often contain about 50% comments. You may also encounter Fortran programs that use the exclamation mark (!) for comments. This is not a standard part of Fortran 77, but is supported by several Fortran 77 compilers and is explicitly allowed in Fortran 90. When understood, the exclamation mark may appear anywhere on a line (except in positions 2-6).

5.Continuation

Sometimes, a statement does not fit into the 66 available columns of a single line. One can then break the statement into two or more lines, and use the continuation mark in position 6. Example:

```
c23456789 (This demonstrates column position!)

c The next statement goes over two physical lines
      area = 3.14159265358979
     +       * r * r
```

Any character can be used instead of the plus sign as a continuation character. It is considered good programming style to use either the plus sign, an ampersand, or digits (using 2 for the second line, 3 for the third, and so on).

6.Blank spaces

Blank spaces are ignored in Fortran 77. So if we remove all blanks in a Fortran 77 program, the program is still acceptable to a complier but almost unreadable to humans.

7.Variables, types, and declarations

7.1Variable names

Variable names in Fortran consist of 1-6 characters chosen from the letters a-z and the digits 0-9. The first character must be a letter. Fortran 77 does not distinguish between upper and lower case, in fact, it assumes all input is upper case. However, nearly all Fortran 77 compilers will accept lower case. If you should ever encounter a Fortran 77 compiler that insists on upper case it is usually easy to convert the source code to all upper case. The words which make up the Fortran language are called *reserved words* and cannot be used as names of variable. Some of the reserved words which we have seen so far are "program", "real", "stop" and "end".

Sandeep K V/9447546957

7[illegible]d declarations

Ref 1

Every variable *should* be defined in a *declaration*. This establishes the *type* of the variable. The most common declarations are:

```
integer   list of variables
real      list of variables
double precision  list of variables
complex   list of variables
logical   list of variables
character list of variables
```

The *list of variables* should consist of variable names separated by commas. Each variable should be declared exactly once. If a variable is undeclared, Fortran 77 uses a set of *implicit rules* to establish the type. This means all variables starting with the letters i-n are integers and all others are real. Many old Fortran 77 programs uses these implicit rules, but *you should not!* The probability of errors in your program grows dramatically if you do not consistently declare your variables.

7.3Integers and floating point variables

Fortran 77 has only one type for integer variables. Integers are usually stored as 32 bits (4 bytes) variables. Therefore, all integer variables should take on values in the range [-m,m] where m is approximately 2*10^9.

Fortran 77 has two different types for floating point variables, called real and double precision. While real is often adequate, some numerical calculations need very high precision and double precision should be used. Usually a real is a 4 byte variable and the double precision is 8 bytes, but this is machine dependent. Some non-standard Fortran versions use the syntax real*8 to denote 8 byte floating point variables.

7.4 The **parameter** statement

Some constants appear many times in a program. It is then often desirable to define them only once, in the beginning of the program. This is what the parameter statement is for. It also makes programs more readable. For example, the circle area program should rather have been written like this:

```
program circle
real r, area, pi
parameter (pi = 3.14159)

c This program reads a real number r and prints
c the area of a circle with radius r.

write (*,*) 'Give radius r:'
read  (*,*) r
area = pi*r*r
write (*,*) 'Area = ', area

stop
end
```

The syntax of the parameter statement is

parameter (*name = constant, ... , name = constant*)

The rules for the parameter statement are:

- The *name* defined in the parameter statement is not a variable but rather a constant. (You cannot change its value at a later point in the program.)
- A *name* can appear in at most one parameter statement.

Sandeep K V/9447546957

Ref 1 parameter statement(s) must come before the first executable statement.

Some good reasons to use the parameter statement are:

- It helps reduce the number of typos.
- It makes it easier to change a constant that appears many times in a program.
- It increases the readability of your program.

8.Expressions and assignment

8.1Constants

The simplest form of an expression is a *constant*. There are 6 types of constants, corresponding to the 6 data types. Here are some integer constants:

1
0
-100
32767
+15

Then we have real constants:

1.0
-0.25
2.0E6
3.333E-1

The E-notation means that you should multiply the constant by 10 raised to the power following the "E". Hence, 2.0E6 is two million, while 3.333E-1 is approximately one third.

For constants that are larger than the largest real allowed, or that requires high precision, double precision should be used. The notation is the same as for real constants except the "E" is replaced by a "D". Examples:

2.0D-1
1D99

Here 2.0D-1 is a double precision one-fifth, while 1D99 is a one followed by 99 zeros.

The next type is complex constants. This is designated by a pair of constants (integer or real), separated by a comma and enclosed in parentheses. Examples are:

(2, -3)
(1., 9.9E-1)

The first number denotes the real part and the second the imaginary part. The fifth type is logical constants. These can only have one of two values:

.TRUE.
.FALSE.

Note that the dots enclosing the letters are required.

The last type is character constants. These are most often used as an *array* of characters, called a *string*. These consist of an arbitrary sequence of characters enclosed in apostrophes (single quotes):

'ABC' Sandeep K V/9447546957

Ref 1

'Here we go goes!'
'It is a nice day'

Strings and character constants are case sensitive. A problem arises if you want to have an apostrophe in the string itself. In this case, you should double the apostrophe:

'It''s a nice day'

8.2Expressions

The simplest non-constant expressions are of the form

operand operator operand

and an example is

x + y

The result of an expression is itself an operand, hence we can nest expressions together like

x + 2 * y

This raises the question of precedence: Does the last expression mean x + (2*y) or (x+2)*y? The precedence of arithmetic operators in Fortran 77 are (from highest to lowest):

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

All these operators are calculated left-to-right, except the exponentiation operator **, which has right-to-left precedence. If you want to change the default evaluation order, you can use parentheses.

The above operators are all binary operators. there is also the unary operator - for negation, which takes precedence over the others. Hence an expression like -x+y means what you would expect.

Extreme caution must be taken when using the division operator, which has a quite different meaning for integers and reals. If the operands are both integers, an integer division is performed, otherwise a real arithmetic division is performed. For example, 3/2 equals 1, while 3./2. equals 1.5 (note the decimal points).

8.3Assignment

The assignment has the form

variable_name = expression

The interpretation is as follows: Evaluate the right hand side and assign the resulting value to the variable on the left. The expression on the right may contain other variables, but these never change value! For example,

area = pi * r**2

does not change the value of pi or r, only area.

8.4Type conversion

When different data types occur in the same expression, *type conversion* has to take place, either explicitly or implicitly. Fortran will do some type conversion implicitly. For example,

real x
x = x + 1

will cast the integer one to the real number one, and has the desired effect of incrementing x by one. However, in more complicated expressions, it is good programming practice to force the necessary type conversions explicitly. For numbers, the following functions are available:

```
int
real
dble
ichar
char
```

The first three have the obvious meaning. ichar takes a character and converts it to an integer, while char does exactly the opposite.

Example: How to multiply two real variables x and y using double precision and store the result in the double precision variable w:

```
w = dble(x)*dble(y)
```

Note that this is different from

```
w = dble(x*y)
```

9.Logical expressions

Logical expressions can only have the value .TRUE. or .FALSE.. A logical expression can be formed by comparing arithmetic expressions using the following *relational operators*:

```
.LT.  meaning <.LE. <=".GT.">
.GE.        >=
.EQ.        =
.NE.        /=
```

So you *cannot* use symbols like *logical operators* .AND. .OR. .NOT. which have the obvious meaning.

9.1Logical variables and assignment

Truth values can be stored in *logical variables*. The assignment is analogous to the arithmetic assignment. Example:

```
logical a, b
a = .TRUE.
b = a .AND. 3 .LT. 5/2
```

The order of precedence is important, as the last example shows. The rule is that arithmetic expressions are evaluated first, then relational operators, and finally logical operators. Hence b will be assigned .FALSE. in the example above. Among the logical operators the precedence (in the absence of parenthesis) is that .NOT. is done first, then .AND., then .OR. is done last.

Logical variables are seldom used in Fortran. But logical expressions are frequently used in conditional statements like the if statement.

10.The if statements

An important part of any programming language are the *conditional statements*. The most common such statement in Fortran is the if statement, which actually has several forms. The simplest one is the logical if statement:

if (*logical expression*) *executable statement* Sandeep K V/9447546957

Ref 1

The if can be written on one line. This example finds the absolute value of x:

```
if (x .LT. 0) x = -x
```

If more than one statement should be executed inside the if, then the following syntax should be used:

```
if (logical expression) then
  statements
endif
```

The most general form of the if statement has the following form:

```
if (logical expression) then
  statements
elseif (logical expression) then
  statements
 :
 :
else
  statements
endif
```

The execution flow is from top to bottom. The conditional expressions are evaluated in sequence until one is found to be true. Then the associated statements are executed and the control resumes after the endif.

10.1Nested if statements

if statements can be nested in several levels. To ensure readability, it is important to use proper indentation. Here is an example:

```
if (x .GT. 0) then
  if (x .GE. y) then
    write(*,*) 'x is positive and x >= y'
  else
    write(*,*) 'x is positive but x
```

You should avoid nesting many levels of if statements
since things get hard to follow.

11.Loops

For repeated execution of similar things, *loops* are used. If you are familiar with other programming languages you have probably heard about *for*-loops, *while*-loops, and *until*-loops. Fortran 77 has only one loop construct, called the do-loop. The do-loop corresponds to what is known as a *for*-loop in other languages. Other loop constructs have to be built using the if and goto statements.

11.1do-loops

The do-loop is used for simple counting. Here is a simple example that prints the cumulative sums of the integers from 1 through n (assume n has been assigned a value elsewhere):

```
integer i, n, sum

sum = 0
do 10 i = 1, n
  sum = sum + i
  write(*,*) 'i =', i
```

Sandeep K V/9447546957

```
Ref 1*,*) 'sum =', sum
10  continue
```

The number 10 is a statement *label*. Typically, there will be many loops and other statements in a single program that require a statement label. The programmer is responsible for assigning a unique number to each label in each program (or subprogram). Recall that column positions 1-5 are reserved for statement labels. The numerical value of statement labels have no significance, so any integers can be used, in any order. Typically, most programmers use consecutive multiples of 10.

The variable defined in the do-statement is incremented by 1 by default. However, you can define the *step* to be any number but zero. This program segment prints the even numbers between 1 and 10 in decreasing order:

```
    integer i

    do 20 i = 10, 1, -2
      write(*,*) 'i =', i
20  continue
```

The general form of the do loop is as follows:

```
    do label  var =  expr1, expr2, expr3
      statements
label continue
```

var is the loop variable (often called the *loop index*) which must be integer. *expr1* specifies the initial value of *var*, *expr2* is the terminating bound, and *expr3* is the increment (step).

Note: The do-loop variable must never be changed by other statements within the loop! This will cause great confusion.

The loop index can be of type real, but due to round off errors may not take on exactly the expected sequence of values.

Many Fortran 77 compilers allow do-loops to be closed by the enddo statement. The advantage of this is that the statement label can then be omitted since it is assumed that an enddo closes the nearest previous do statement. The enddo construct is widely used, but it is not a part of ANSI Fortran 77.

It should be noted that unlike some programming languages, Fortran only evaluates the start, end, and step expressions once, before the first pass thought the body of the loop. This means that the following do-loop will multiply a non-negative j by two (the hard way), rather than running forever as the equivalent loop might in another language.

```
    integer i,j

    read (*,*) j
    do 20 i = 1, j
      j = j + 1
20  continue
    write (*,*) j
```

11.2while-loops

The most intuitive way to write a while-loop is

```
    while (logical expr) do
      statements
    enddo
```

Sandeep K V/9447546957

Ref 1

ely,

```
do while (logical expr)
  statements
enddo
```

The program will alternate testing the condition and executing the statements in the body as long as the condition in the while statement is true. Even though this syntax is accepted by many compilers, it is not ANSI Fortran 77. The correct way is to use if and goto:

```
label if (logical expr) then
      statements
      goto label
    endif
```

Here is an example that calculates and prints all the powers of two that are less than or equal to 100:

```
  integer n

  n = 1
10 if (n .le. 100) then
     write (*,*) n
     n = 2*n
     goto 10
   endif
```

11.3until-loops

If the termination criterion is at the end instead of the beginning, it is often called an until-loop. The pseudocode looks like this:

```
do
  statements
until (logical expr)
```

Again, this should be implemented in Fortran 77 by using if and goto:

```
label continue
      statements
    if (logical expr) goto label
```

Note that the logical expression in the latter version should be the negation of the expression given in the pseudocode!

12.Arrays

Many scientific computations use vectors and matrices. The data type Fortran uses for representing such objects is the *array*. A one-dimensional array corresponds to a vector, while a two-dimensional array corresponds to a matrix. To fully understand how this works in Fortran 77, you will have to know not only the syntax for usage, but also how these objects are stored in memory in Fortran 77.

12.1One-dimensional arrays

The simplest array is the one-dimensional array, which is just a sequence of elements stored consecutively in memory. For example, the declaration

```
real a(20)
```

Sandeep K V/9447546957

Ref 1

d is a real array of length 20. That is, a consists of 20 real numbers stored contiguously in memory. By convention, Fortran arrays are indexed from 1 and up. Thus the first number in the array is denoted by a(1) and the last by a(20). However, you may define an arbitrary index range for your arrays using the following syntax:

```
real b(0:19), weird(-162:237)
```

Here, b is exactly similar to a from the previous example, except the index runs from 0 through 19. weird is an array of length 237-(-162)+1 = 400.

The type of an array element can be any of the basic data types. Examples:

```
integer i(10)
logical aa(0:1)
double precision x(100)
```

Each element of an array can be thought of as a separate variable. You reference the i'th element of array a by a(i). Here is a code segment that stores the 10 first square numbers in the array sq:

```
integer i, sq(10)

do 100 i = 1, 10
  sq(i) = i**2
100 continue
```

A common bug in Fortran is that the program tries to access array elements that are out of bounds or undefined. This is the responsibility of the programmer, and the Fortran compiler will not detect any such bugs!

12.2Two-dimensional arrays

Matrices are very important in linear algebra. Matrices are usually represented by two-dimensional arrays. For example, the declaration

```
real A(3,5)
```

defines a two-dimensional array of 3*5=15 real numbers. It is useful to think of the first index as the row index, and the second as the column index. Hence we get the graphical picture:

```
(1,1)  (1,2)  (1,3)  (1,4)  (1,5)
(2,1)  (2,2)  (2,3)  (2,4)  (2,5)
(3,1)  (3,2)  (3,3)  (3,4)  (3,5)
```

Two-dimensional arrays may also have indices in an arbitrary defined range. The general syntax for declarations is:

name (low_index1 : hi_index1, low_index2 : hi_index2)

The total size of the array is then

size = (hi_index1-low_index1+1)(hi_index2-low_index2+1)*

It is quite common in Fortran to declare arrays that are larger than the matrix we want to store. (This is because Fortran does not have dynamic storage allocation.) This is perfectly legal. Example:

```
      real A(3,5)
      integer i,j
c
c     We will only use the upper 3 by 3 part of this array.
```

```
c Ref 1
      do 20 j = 1, 3
        do 10 i = 1, 3
          a(i,j) = real(i)/real(j)
   10   continue
   20 continue
```

The elements in the submatrix A(1:3,4:5) are undefined. Do not assume these elements are initialized to zero by the compiler (some compilers will do this, but not all).

12.3Storage format for 2-dimensional arrays

Fortran stores higher dimensional arrays as a contiguous sequence of elements. It is important to know that 2-dimensional arrays are stored *by column*. So in the above example, array element (1,2) will follow element (3,1). Then follows the rest of the second column, thereafter the third column, and so on.

Consider again the example where we only use the upper 3 by 3 submatrix of the 3 by 5 array A(3,5). The 9 interesting elements will then be stored in the first nine memory locations, while the last six are not used. This works out neatly because the *leading dimension* is the same for both the array and the matrix we store in the array. However, frequently the leading dimension of the array will be larger than the first dimension of the matrix. Then the matrix will *not* be stored contiguously in memory, even if the array is contiguous. For example, suppose the declaration was A(5,3) instead. Then there would be two "unused" memory cells between the end of one column and the beginning of the next column (again we are assuming the matrix is 3 by 3).

This may seem complicated, but actually it is quite simple when you get used to it. If you are in doubt, it can be useful to look at how the *address* of an array element is computed. Each array will have some memory address assigned to the beginning of the array, that is element (1,1). The address of element (i,j) is then given by

addr[A(i,j)] = *addr*[A(1,1)] + (j-1)*lda + (i-1)

where lda is the leading (i.e. row) dimension of A. Note that lda is in general different from the actual matrix dimension. Many Fortran errors are caused by this, so it is very important you understand the distinction!

12.4Multi-dimensional arrays

Fortran 77 allows arrays of up to seven dimensions. The syntax and storage format are analogous to the two-dimensional case, so we will not spend time on this.

12.5The **dimension** statement

There is an alternate way to declare arrays in Fortran 77. The statements

```
real A, x
dimension x(50)
dimension A(10,20)
```

are equivalent to

```
real A(10,20), x(50)
```

This dimension statement is considered old-fashioned style today.

13.Subprograms

When a program is more than a few hundred lines long, it gets hard to follow. Fortran codes that solve real engineering problems often have tens of thousands of lines. The only way to handle such big codes, is to use a *modular* approach and split the program into many separate smaller units called *subprograms*.

A [illegible]am is a (small) piece of code that solves a well defined subproblem. In a large program, one often has to solve the same subproblems with many different data. Instead of replicating code, these tasks should be solved by subprograms . The same subprogram can be invoked many times with different input data.

Fortran has two different types of subprograms, called *functions* and *subroutines*.

13.1Functions

Fortran functions are quite similar to mathematical functions: They both take a set of input arguments (parameters) and return a value of some type. In the preceding discussion we talked about *user defined* subprograms. Fortran 77 also has some *intrinsic* (built-in) functions.

A simple example illustrates how to use a function:

```
x = cos(pi/3.0)
```

Here cos is the cosine function, so x will be assigned the value 0.5 (if pi has been correctly defined; Fortran 77 has no built-in constants). There are many intrinsic functions in Fortran 77. Some of the most common are:

abs	*absolute value*
min	*minimum value*
max	*maximum value*
sqrt	*square root*
sin	*sine*
cos	*cosine*
tan	*tangent*
atan	*arctangent*
exp	*exponential (natural)*
log	*logarithm (natural)*

In general, a function always has a *type*. Most of the built-in functions mentioned above, however, are *generic*. So in the example above, pi and x could be either of type real or double precision. The compiler would check the types and use the correct version of cos (real or double precision). Unfortunately, Fortran is not really a *polymorphic* language so in general you have to be careful to match the types of your variables and your functions!

Now we turn to the user-written functions. Consider the following problem: A meteorologist has studied the precipitation levels in the Bay Area and has come up with a model *r(m,t)* where *r* is the amount of rain, *m* is the month, and *t* is a scalar parameter that depends on the location. Given the formula for *r* and the value of *t*, compute the annual rainfall.

The obvious way to solve the problem is to write a loop that runs over all the months and sums up the values of *r*. Since computing the value of *r* is an independent subproblem, it is convenient to implement it as a function. The following main program can be used:

```
      program rain
      real r, t, sum
      integer m

      read (*,*) t
      sum = 0.0
      do 10 m = 1, 12
        sum = sum + r(m, t)
10    continue
      write (*,*) 'Annual rainfall is ', sum, 'inches'
```

Sandeep K V/9447546957

Ref 1

```
end
```

Note that we have declared 'r' to be 'real' just as we would a variable. In addition, the function *r* has to be defined as a Fortran function. The formula the meteorologist came up with was

```
r(m,t) = t/10 * (m**2 + 14*m + 46) if this is positive
r(m,t) = 0                         otherwise
```

The corresponding Fortran function is

```
real function r(m,t)
integer m
real t

r = 0.1*t * (m**2 + 14*m + 46)
if (r .LT. 0) r = 0.0

return
end
```

We see that the structure of a function closely resembles that of the main program. The main differences are:

- Functions have a type. This type must also be declared in the calling program.
- The return value should be stored in a variable with the same name as the function.
- Functions are terminated by the *return* statement instead of *stop*.

To sum up, the general syntax of a Fortran 77 function is:

```
type function name (list-of-variables)
declarations
statements
return
end
```

The function has to be declared with the correct type in the calling program unit. If you use a function which has not been declared, Fortran will try to use the same implicit typing used for variables, probably getting it wrong. The function is called by simply using the function name and listing the parameters in parenthesis.

It should be noted that strictly speaking Fortran 77 doesn't permit recursion (functions which call themselves). However, it is not uncommon for a compiler to allow recursion.

13.2Subroutines

A Fortran function can essentially only return one value. Often we want to return two or more values (or sometimes none!). For this purpose we use the subroutine construct. The syntax is as follows:

```
subroutine name (list-of-arguments)
declarations
statements
return
end
```

Note that subroutines have no type and consequently should not (cannot) be declared in the calling program unit. They are also invoked differently than functions, using the word *call* before their names and parameters.

We give an example of a very simple subroutine. The purpose of the subroutine is to swap two integers.

Ref 1

```
      subroutine iswap (a, b)
      integer a, b
c Local variables
      integer tmp

      tmp = a
      a = b
      b = tmp

      return
      end
```

Note that there are two blocks of variable declarations here. First, we declare the input/output parameters, i.e. the variables that are common to both the caller and the callee. Afterwards, we declare the *local variables*, i.e. the variables that can only be used within this subprogram. We can use the same variable names in different subprograms and the compiler will know that they are different variables that just happen to have the same names.

13.3Call-by-reference

Fortran 77 uses the so-called *call-by-reference* paradigm. This means that instead of just passing the values of the function/subroutine arguments (*call-by-value*), the memory address of the arguments (pointers) are passed instead. A small example should show the difference:

```
      program callex
      integer m, n
c
      m = 1
      n = 2

      call iswap(m, n)
      write(*,*) m, n

      stop
      end
```

The output from this program is "2 1", just as one would expect. However, if Fortran 77 had been using call-by-value then the output would have been "1 2", i.e. the variables m and n were unchanged! The reason for this is that only the values of m and n had been copied to the subroutine iswap, and even if a and b were swapped inside the subroutine the new values would not have been passed back to the main program.

In the above example, call-by-reference was exactly what we wanted. But you have to be careful about this when writing Fortran code, because it is easy to introduce undesired *side effects*. For example, sometimes it is tempting to use an input parameter in a subprogram as a local variable and change its value. Since the new value will then propagate back to the calling program with an unexpected value, you should *never* do this unless (like our iswap subroutine) the change is part of the purpose of the subroutine.

14.Simple I/O

An important part of any computer program is to handle input and output. In our examples so far, we have already used the two most common Fortran constructs for this: read and write. Fortran I/O can be quite complicated, so we will only describe some simpler cases in this tutorial.

Sandeep K V/9447546957

1. Read and write

Read is used for input, while write is used for output. A simple form is

read (*unit no, format no*) *list-of-variables*
write(*unit no, format no*) *list-of-variables*

The unit number can refer to either standard input, standard output, or a file. The format number refers to a label for a format statement, which will be described in the next lesson.

It is possible to simplify these statements further by using asterisks (*) for some arguments, like we have done in most of our examples so far. This is sometimes called *list directed* read/write.

read (*,*) *list-of-variables*
write(*,*) *list-of-variables*

The first statement will read values from the standard input and assign the values to the variables in the variable list, while the second one writes to the standard output.

Examples

Here is a code segment from a Fortran program:

```
integer m, n
real x, y, z(10)

read(*,*) m, n
read(*,*) x, y
read(*,*) z
```

We give the input through standard input (possibly through a data file directed to standard input). A data file consists of *records* according to traditional Fortran terminology. In our example, each record contains a number (either integer or real). Records are separated by either blanks or commas. Hence a legal input to the program above would be:

```
 -1  100
-1.0 1e+2
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0
```

Or, we could add commas as separators:

```
 -1, 100
-1.0, 1e+2
1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0
```

Note that Fortran 77 input is line sensitive, so it is important not to have extra input elements (fields) on a line (record). For example, if we gave the first four inputs all on one line as

```
 -1, 100, -1.0, 1e+2
1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0
```

then m and n would be assigned the values -1 and 100 respectively, but the last two values would be discarded, x and y would be assigned the values 1.0 and 2.0, ignoring the rest of the second line. This would leave the elements of z all undefined.

If there are too few inputs on a line then the next line will be read. For example

Ref 1

```
 100
-1.0
 1e+2
 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0
```

would produce the same results as the first two examples.

15.Format statements

So far we have mostly used *free format* input/output. This uses a set of default rules for how to input and output values of different types (integers, reals, characters, etc.). Often the programmer wants to specify some particular input or output format, e.g., how many decimals places in real numbers. For this purpose Fortran 77 has the *format* statement. The same format statements are used for both input and output.

Syntax

```
    write(*, label) list-of-variables
label format format-code
```

A simple example demonstrates how this works. Say you have an integer variable you want to print in a field 4 characters wide and a real number you want to print in fixed point notation with 3 decimal places.

```
    write(*, 900) i, x
900 format (I4,F8.3)
```

The format label 900 is chosen somewhat arbitrarily, but it is common practice to number format statements with higher numbers than the control flow labels. After the keyword format follows the format codes enclosed in parenthesis. The code I4 stands for an integer with width four, while F8.3 means that the number should be printed using fixed point notation with field width 8 and 3 decimal places.

The format statement may be located anywhere within the program unit. There are two programming styles: Either the format statement follows directly after the read/write statement, or all the format statements are grouped together at the end of the (sub-)program.

Common format codes

The most common format code letters are:

A - text string
D - double precision numbers, exponent notation
E - real numbers, exponent notation
F - real numbers, fixed point format
I - integer
X - horizontal skip (space)
/ - vertical skip (newline)

The format code F (and similarly D, E) has the general form F*w.d* where *w* is an integer constant denoting the field width and *d* is an integer constant denoting the number of significant digits.

For integers only the field width is specified, so the syntax is I*w*. Similarly, character strings can be specified as A*w* but the field width is often dropped.

If a number or string does not fill up the entire field width, spaces will be added. Usually the text will be adjusted to the right, but the exact rules vary among the different format codes.

For horizontal spacing, the *n*X code is often used. This means *n* horizontal spaces. If *n* is omitted, *n=1* is assumed. For vertical spacing (newlines), use the code /. Each slash corresponds to one newline. Note that each read or write statement by default ends with a newline (here Fortran differs from C).

Some examples

This piece of Fortran code

```
    x = 0.025
    write(*,100) 'x=', x
100 format (A,F)
    write(*,110) 'x=', x
110 format (A,F5.3)
    write(*,120) 'x=', x
120 format (A,E)
    write(*,130) 'x=', x
130 format (A,E8.1)
```

produces the following output when we run it:

```
x=      0.0250000
x=0.025
x=  0.2500000E-01
x= 0.3E-01
```

Format strings in read/write statements

Instead of specifying the format code in a separate format statement, one can give the format code in the read/write statement directly. For example, the statement

```
    write (*,'(A, F8.3)') 'The answer is x = ', x
```

is equivalent to

```
    write (*,990) 'The answer is x = ', x
990 format (A, F8.3)
```

Sometimes text strings are given in the format statements, e.g. the following version is also equivalent:

```
    write (*,999) x
999 format ('The answer is x = ', F8.3)
```

16.Examples

16.1. area of a circle

```
program circle
    real r, area

c This program reads a real number r and prints
c the area of a circle with radius r.

    write (*,*) 'Give radius r:'
```

Ref 1

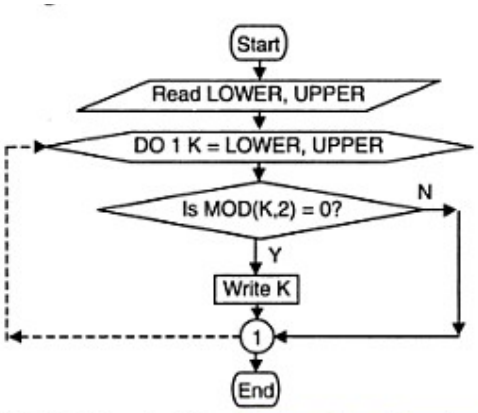
```
,*) r
area = 3.14159*r*r
write (*,*) 'Area = ', area

stop
end
```

16.2 roots of quadratic equation

```
READ *, A,B,C
DISC = B*B-4*A*C
IF (DISC .LT. 0.0) THEN
  R = 0.0 - 0.5 * B/A
  AI = 0.5 * SQRT(0.0-DISC)/A
  PRINT *, 'TWO COMPLEX ROOTS: ',R,' PLUS OR MINUS ',AI,' I'
ELSE IF (DISC .EQ. 0.0) THEN
  R = 0.0 - 0.5 * B/A
  PRINT *, 'ONE REAL ROOT: ',R
ELSE
  SD = SQRT(DISC)
  R1 = 0.5*(SD-B)/A
  R2 = 0.5*(0.0-(B+SD))/A
  PRINT *, 'TWO REAL ROOTS: ',R2,R1
END IF
STOP
END
```

16.3 Finding even numbers in given limits



FC 4.1.1: Flowchart for Finding Even Numbers.

```
INTEGER LOWER,UPPER

WRITE (*,*) 'FEED LOWER LIMIT'

READ(*,*) LOWER

WRITE (*,*) 'FEED UPPER LIMIT'

READ(*,*) UPPER

WRITE (*,*) 'EVEN NUMBERS BETWEEN' , LOWER , 'TO' , UPPER

DO 910 K = LOWER ,UPPER

        IF (MOD (K,2) .EQ. 0) WRITE (*,*) K

910 CONTINUE

STOP
```

Sandeep K V/9447546957

```
END
```

Ref 1

This program can be modiefied to find odd numbers , just by checking equal to 1 instead of 0

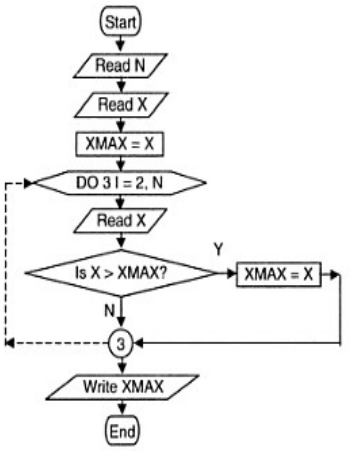
PROG 4.2.1: Locating Minimum of a given set of Numbers

```
      WRITE(*,*)' FEED NUMBER OF DATA POINTS:'
      READ(*,*) NMAX
      WRITE(*,*) NMAX
      I = 1
      WRITE(*,*) ' FEED THE',I,' DATA VALUE : '
      READ(*,*) X
      WRITE(*,*) X
      XMIN = X
      DO 100  K = 2, NMAX
          WRITE(*,*)' FEED THE',K,' DATA VALUE : '
          READ(*,*) X
          WRITE(*,*) X
          IF (X .LT. XMIN) XMIN = X
100   CONTINUE
      WRITE(*,*) 'SMALLEST VALUE = ', XMIN
      STOP
      END
```

PROG 4.2.2: Locating Maximum of a given Set of Numbers

```
      WRITE(*,*)' FEED NUMBER OF DATA POINTS: '
      READ(*,*) N
      WRITE(*,*) N
      I = 1
      WRITE(*,*)' FEED THE',I,' DATA VALUE :'
      READ(*,*) X
      WRITE(*,*) X
      XMAX = X
      DO 100  I = 2, N

      WRITE(*,*) ' FEED THE',I,' DATA VALUE :'
      READ(*,*) X
      WRITE(*,*) X
      IF (X .GT. XMAX) XMAX = X
100   CONTINUE
      WRITE(*,*) ' LARGEST VALUE = ', XMAX
      STOP
      END
```



FC 4.2.2: Flowchart for Maximum Value in given Data

Sandeep K V/9447546957

Ref 1

PROG 4.2.3: Finding Maximum, Minimum and Range of given Numbers

```
      WRITE(*,*)' FEED NUMBER OF DATA POINTS :'
      READ(*,*) N
      WRITE(*,*) N
      I = 1
      WRITE(*,*) ' FEED THE', I,' DATA VALUE : '
      READ(*,*) X
      WRITE(*,*) X
      XMIN = X
      XMAX = X
      DO 100  I = 2, N
          WRITE(*,*) ' FEED THE', I,' DATA VALUE : '
          READ(*,*) X
          WRITE(*,*) X
          IF (X .LT. XMIN) XMIN = X
          IF (X .GT. XMAX) XMAX = X

100   CONTINUE
      RANGE = XMAX - XMIN
      WRITE(*,*)' SMALLEST VALUE  = ', XMIN
      WRITE(*,*)' LARGEST VALUE   = ', XMAX
      WRITE(*,*)' RANGE OF VALUES = ', RANGE
      STOP
      END
```

PROG 4.1.2: Finding Odd Numbers in given Limits

```
      INTEGER LOWER, UPPER
      WRITE (*,*) ' FEED LOWER LIMIT '
      READ (*,*) LOWER
      WRITE (*,*) ' FEED UPPER LIMIT '
      READ (*,*) UPPER
      WRITE (*,*) ' ODD NUMBERS BETWEEN', LOWER,'  TO', UPPER
      DO 90 K = LOWER, UPPER
          IF(MOD (K, 2) .EQ. 1) WRITE(*,*) K
90    CONTINUE
      STOP
      END
```

PROG 4.3.2: Finding Mean and Standard Deviation of Data

```
      DIMENSION A(100)
      open (unit = 5, file = 'prog432.inp', status = 'old')
      WRITE(*,*)'FEED NO.OF DATA VALUES,N<=100'
      READ(*,*)N
      WRITE(*,*)' FEED DATA VALUES'
      READ(5,*) (A(K), K = 1, N)
      SUMA = 0
      SUMSD = 0
```

Ref 1

```
      DO 2 K = 1, N
          SUMA = SUMA + A(K)
2     CONTINUE
      XBAR = SUMA/ N
      DO 3 K = 1, N
          SUMSD = SUMSD + (A(K) - XBAR) ** 2
3     CONTINUE
      STDDEV = SQRT(SUMSD/N)
      WRITE(*,*) ' ARITHMETIC MEAN = ', XBAR
      WRITE(*,*)' STANDARD DEVIATION = ', STDDEV
      STOP
      END
```

4.3.2 MEAN AND STANDARD DEVIATION OF DATA

For N data points, a_1, a_2,....., a_N, the mean, $\bar{a}$, is defined as

$$\bar{a} = \frac{1}{N}\sum_{1}^{N} a_i \qquad (4.2)$$

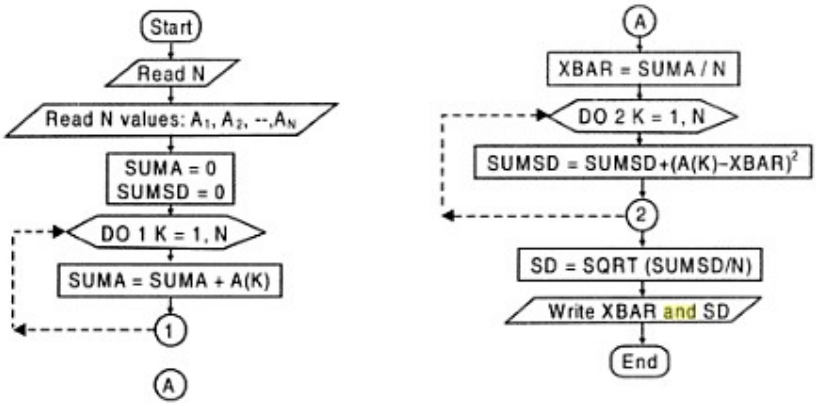
and standard deviation is given by

$$\alpha = \sqrt{\frac{1}{N}\sum_{1}^{N}(a_i - \bar{a})^2} \qquad (4.3)$$

Analysis of the Problem

To find mean ($\bar{a}$) and standard deviation (σ), we initialize two variables *SUMX* and *SUMSD* as 0; to which we add the quantities same as and $(a_i - a)^{-2}$. Finally using the formulas given above, the mean and standard deviation are calculated.

A flowchart for this problem is as under:



FC 4.3.2: Flowchart for Mean and Standard deviation.

Sandeep K V/9447546957

Ref 1

4.4 SUM OF A FINITE SERIES

In this section, we find sum of a series of finite number of terms and how that can be used to construct certain functions. Finally this method is extended to find area under a curve defined by some function.

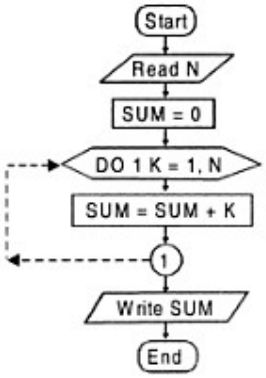
4.4.1 SUM OF NATURAL NUMBERS

Let us find sum of the series given by ***N* natural numbers,**

$$1 + 2 + 3 + 4 + ... + N. \qquad (4.6)$$

Analysis of the Problem

First we read *N*, the total number of terms up to which sum is required. A variable *SUM* is initialized as 0. Then a loop over I is started varying from 1 to *N*. Each value of I is added to *SUM*. Finally when the loop is over, the variable *SUM* yields the desired sum.



FC 4.4.1: Flowchart for Sum of Natural Numbers

4.6.1 ADDITION/ SUBTRACTION OF TWO MATRICES

Sum of two matrices *a* and *b* of order $m \times n$ can be written as

$$C = A + B$$

In matrix element form, it can be expressed as

$$C_{i,j} = A_{i,j} + B_{i,j} \qquad (4.15)$$

where $i = 1, 2, ..., m$, and $j = 1, 2, ..., n$.

Analysis of the Problems

Two matrices *AMAT* and *BMAT* can be added only if these have the same dimensions, which are also shared by their resultant sum matrix *SUM*. Let *AMAT* be a *row*1 × *row*2 matrix and *BMAT* be a *row*2 × *col*2 matrix, which are to be added. First we check that the addition or subtraction is possible only if *row*1 = *row*2 and *col*1 = *col*2. Then the matrix elements of *AMAT* and *BMAT* are read. Finally, using the loop structure, the matrix elements of the sum matrix *SUM* are obtained as:

$$SUM\ (I, J) = AMAT\ (I, J) + BMAT\ (I, J) \qquad (4.16)$$

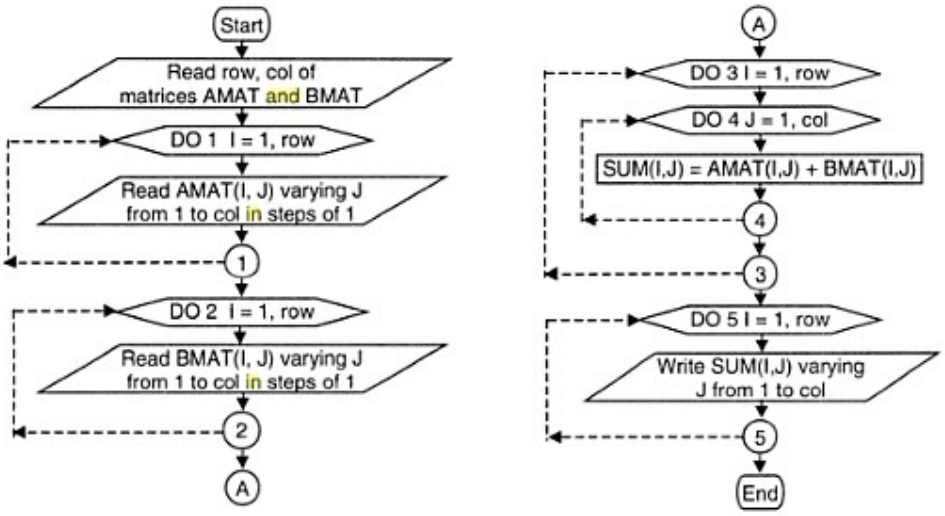
where $I = 1, 2, ..., row1$ and $j = 1, 2, ..., col1$.

Similarly difference of matrices *AMAT* and *BMAT* is given as

$$DIFF\ (I, J) = AMAT\ (I, J) - BMAT\ (I, J)$$

Sandeep K V/9447546957

Ref 1 A flowchart for addition of two matrices is given below:



FC 4.6.1: Flowchart for Addition of Two Matrices

PROG 4.6.1: Addition of Two Matrices

```
      DIMENSION AMAT(10, 10), BMAT(10, 10), SUM(10, 10)
      INTEGER ROW, COL
      WRITE(*,*) ' FEED NO. OF ROWS AND COLUMNS'

      READ(*,*) ROW, COL
      WRITE(*,*) ' FEED THE MATRIX A ROW-WISE  '
      DO 1 I = 1, ROW
          READ(*,*)(AMAT(I, J), J = 1, COL)
1     CONTINUE
      WRITE(*,*) ' FEED THE MATRIX B ROW-WISE  '
      DO 2 I = 1, ROW
          READ(*,*)(BMAT(I, J), J = 1, COL)
2     CONTINUE
      DO 3 I = 1, ROW
          DO 4 J = 1, COL
              SUM(I, J) = AMAT(I, J) + BMAT(I, J)
4         CONTINUE
3     CONTINUE
       WRITE(*,*) ' SUM MATRIX  = '
       DO 5 I = 1, ROW
           WRITE(*,*) (SUM(I, J), J = 1, COL)
 5     CONTINUE
       STOP
       END
```

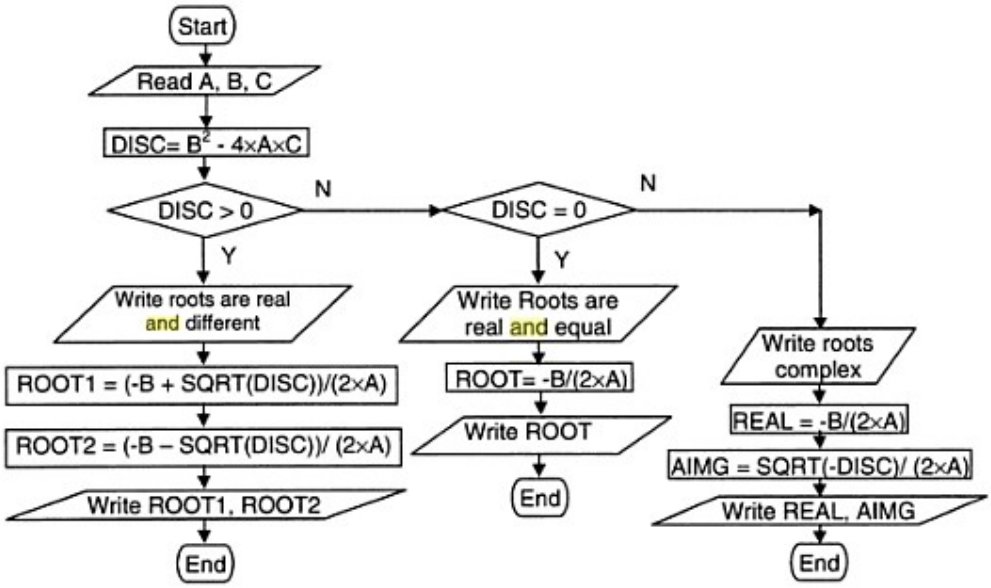
Sandeep K V/9447546957

Ref 1

PROG 4.8: Roots of Quadratic Equation

```
WRITE(*,*)'FEED COEFFICIENTS A, B AND C '
READ(*,*)A, B, C
WRITE(*,*)'THE COEFFICIENTS A, B AND C ARE :'
WRITE(*,*)A, B, C
DISC = B*B - 4*A*C
IF(DISC .GT.0) THEN
    ROOT1 = (-B + SQRT(DISC))/(2*A)
    ROOT2 = (-B - SQRT(DISC))/(2*A)
    WRITE(*,*)'ROOTS ARE REAL, DIFFERENT'
    WRITE(*,*) ' FIRST ROOT  = ', ROOT1
    WRITE(*,*) ' SECOND ROOT  = ', ROOT2
ELSE IF(DISC .EQ. 0) THEN
    ROOT = -B/(2 * A)
    WRITE(*,*) ' ROOTS ARE EQUAL = ',  ROOT
ELSE
    WRITE(*,*) ' ROOTS ARE COMPLEX'
    REAL = -B/(2 * A)
    AIMAG = SQRT(-DISC)/(2*A)
    WRITE(*,*)'REAL PART  = ', REAL
    WRITE(*,*)'IMAGINARY PART = ', AIMAG
ENDIF
STOP
END
```

A flowchart for finding roots of a quadratic equation is as under:



FC 4.8: Flowchart for Roots of Quadratic Equation

Sandeep K V/9447546957

Algorithms and flowcharts are two different tools used for creating new Ref 1grams, especially in computer programming. An algorithm is a step-by-step analysis of the process, while a flowchart explains the steps of a program in a graphical way.

Definition of Algorithm

To write a logical step-by-step method to solve the problem is called algorithm, in other words, an algorithm is a procedure for solving problems. In order to solve a mathematical or computer problem, this is the first step of the procedure. An algorithm includes calculations, reasoning and data processing. Algorithms can be presented by natural languages, pseudo code and flowcharts, etc.

Definition of Flowchart

A flowchart is the graphical or pictorial representation of an algorithm with the help of different symbols, shapes and arrows in order to demonstrate a process or a program. With algorithms, we can easily understand a program. The main purpose of a flowchart is to analyze different processes. Several standard graphics are applied in a flowchart:

- Terminal Box - Start / End

- Input / Output

- Process / Instruction

- Decision

Sandeep K V/9447546957

- Connector / Arrow

Ref 1

The graphics above represent different part of a flowchart. The process in a flowchart can be expressed through boxes and arrows with different sizes and colors. In a flowchart, we can easily highlight a certain element and the relationships between each part.

How to Use Flowcharts to Represent Algorithms

Now that we have the definitions of algorithm and flowchart, how do we use a flowchart to represent an algorithm?

Algorithms are mainly used for mathematical and computer programs, whilst flowcharts can be used to describe all sorts of processes: business, educational, personal and of course algorithms. So flowcharts are often used as a program planning tool to visually organize the step-by-step process of a program. a flowchart is pictorial representation of an algorithm, an algorithm can be expressed and analyzed through a flowchart.

An algorithm shows you every step of reaching the final solution, while a flowchart shows you how to carry out the process by connecting each step. An algorithm uses mainly words to describe the steps while a flowchart uses the help of symbols, shapes and arrows to make the process more logical.

Convert Temperature from Fahrenheit (°F) to Celsius (°C)

Algorithm:

Step 1: Read temperature in Fahrenheit,

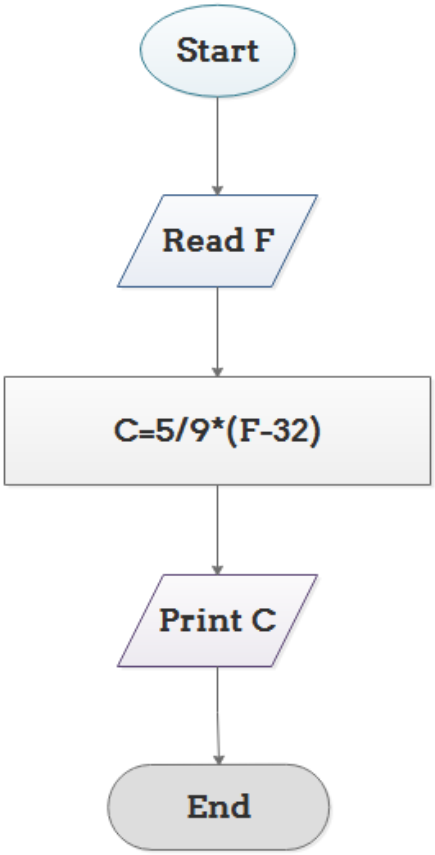
Step 2: Calculate temperature with formula C=5/9*(F-32),

Step 3: Print C,

Flowchart:

Sandeep K V/9447546957

Ref 1



Sandeep K V/9447546957

Equation of a straight line

Ref 1

y =mx + c
slope m = (y2-y2)/x2-x1)
y-intercept = c

Equation of an ellipse

(x^2 / a^2) – (y^2 / b^2) = 1
with horizontal major axis 2a and vertical minor axis 2b
area = pi*a*b

Sandeep K V/9447546957