

M.Sc. Mathematics

MAL-516

**Programming with FORTRAN
(Theory)**



**Directorate of Distance Education
Guru Jambheshwar University of Science &
Technology. HISAR-125001**



CONTENTS

Chapter No.	Chapter Name	Writer	Vetter	Page No.
1	Computer Fundamentals	Dr.Sandeep Dalal	Prof.Sunita Rani	1-30
2	Steps to Solve a Mathematical Problem with Computer	Dr.Sandeep Dalal	Prof.Sunita Rani	31-55
3	Basic Concepts of Fortran	Dr.Sandeep Dalal	Prof.Sunita Rani	56-85
4	Input and Output Statements	Dr.Sandeep Dalal	Prof.Sunita Rani	86-112
5	Control Structure	Dr.Sandeep Dalal	Prof.Sunita Rani	113-138
6	Operators and expressions	Dr.Sandeep Dalal	Prof.Sunita Rani	139-160
7	Arrays & String	Dr.Sandeep Dalal	Prof.Sunita Rani	161-186
8	Functions & Subroutines	Dr.Sandeep Dalal	Prof.Sunita Rani	187-212
9	Derived Type & Pointers	Dr.Sandeep Dalal	Prof.Sunita Rani	213-239
10	File Processing	Dr.Sandeep Dalal	Prof.Sunita Rani	240-263

Author: Dr.Sandeep Dalal (Assistant Professor)

Dept.of Computer Science & Applications
M.D.University Rohtak.(Haryana)

Vetter: Prof. Sunita Rani,

Department of Mathematics,
GJUS&T Hisar

Coordinator: Dr Vizender Singh

Assistant Professor & Programme Coordinator (M.Sc. Mathematics)
Directorate of Distance Education, GJUS&T Hisar



Class	: M.Sc. (Mathematics)	Course Code	: MAL 516
Subject	: Programming with FORTRAN (Theory)		

CHAPTER-1

INTRODUCTION TO COMPUTER

STRUCTURE:

- 1.0 Objective
- 1.1 Introduction
- 1.2 Definition of Computer
- 1.3 History
- 1.4 Characteristics
- 1.5 Advantages
- 1.6 Disadvantages
- 1.7 Applications
- 1.8 Generations of Computers
- 1.9 Component of Computer
- 1.10 Types of Computers
- 1.11 Software
 - 1.11.1 System Software
 - 1.11.2 Application Software
- 1.12 Hardware
 - 1.12.1 Input Devices



1.12.2 Output Devices

1.13 Summary

1.14 Keyword

1.15 Self-Assessment Question

1.16 Suggested Readings

1.0 Learning Objective:

After reading this chapter, you should be able to:

1. Understand the concept computer.
2. Understand the concept of components of computers.
3. Understand the generations of computers.
4. Understand the types of computers.
5. Understand the concept of software and hardware.

1.1 Introduction:

The term computer has been borrowed from the word compute that means calculate. Initially computers were used to perform arithmetic calculations at fast speed, now they are used in nearly every field. You can use computers for Banking Application, Word Processing, Desktop Publishing, Weather Forecasting, Railway Tickets Reservation, Control of Machine and Robots in Factory, Scientific Research, etc. In brief, a computer may be defined as a device that receives data from outside world, analyze it, and then applies a predefined set of instruction's to it to produce output. For instance, when you go to Railway Ticket Reservation Counter, the operator feeds your request for a Ticket Reservation in the computer. The computer analyze the data feed by the operator and make a reservation. Then it prints a ticket for you. The ticket is the output generated by the computer based on the reservation request (Input) entered by the operator. It is also said that the computer is a Data Processing mahine,



because it can receive, store, process and retrieve any kind of data. For instance, you can store the names and addresses of all employees working in a company in a computer file. Later you can ask the computer to print a list of only those employees who work in the Accounts Department.

1.2 Definition of Computer

Computer is an electronic data processing machine which

1. Accepts data from human world and stores data input,
2. Processes the data input, and
3. Generates the output in a required format

The computer which we see, are digital computers. The digital computer carries out five functions in gross terms:

1. Takes data as input.
2. Stores the data/instructions in its memory and use them when required.
3. Processes the data and converts it into useful information.
4. Generates the output
5. Controls all the above four steps.

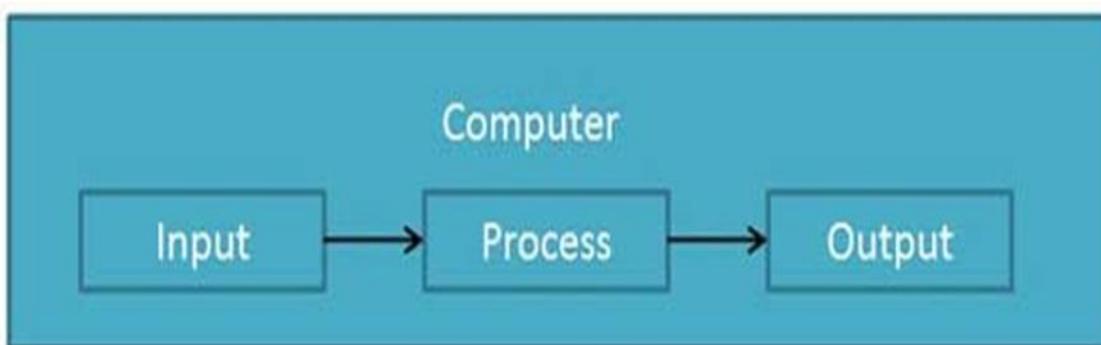


Figure 1.1 IPO (Input-Process-Output) Cycle

Data:



Data can be defined as a representation of facts, concepts or instructions in a formalized manner which should be suitable for communication, interpretation, or processing by human or electronic machine. Data is represented with the help of characters like alphabets (A-Z,a-z), digits (0-9) or special characters(+,-,/,*,<,>= etc.).

Information:

Information is organised or classified data which has some meaningful values for the receiver. Information is the processed data on which decisions and actions are based. For the decision to be meaningful, the processed data must qualify for the following characteristics:

1. Timely - Information should be available when required.
2. Accuracy - Information should be accurate.
3. Completeness - Information should be complete.

Data Processing Cycle:

Data processing is the re-structuring or re-ordering of data by people or machine to increase their usefulness and add values for particular purpose. Data processing consists of basic steps input, processing and output. These three steps constitute the data processing cycle.

1.Input - In this step the input data is prepared in some convenient form for processing. The form will depend on the processing machine. For example, when electronic computers are used, the input data could be recorded on any one of several types of input medium, such as magnetic disks, tapes and so on.

2.Processing - In this step input data is changed to produce data in a more useful form. For example, pay-checks may be calculated from the time cards, or a summary of sales for the month may be calculated from the sales orders.

3.Output - Here the result of the proceeding processing step are collected. The particular form of the output data depends on the use of the data. For example, output data may be pay-checks for employees.

1.3 History



The computer was born not for entertainment or email but out of a need to solve a serious number-crunching crisis. By 1880, the U.S. population had grown so large that it took more than seven years to tabulate the U.S. Census results. The government sought a faster way to get the job done, giving rise to punch-card based computers that took up entire rooms.

Today, we carry more computing power on our smartphones than was available in these early models. The following brief history of computing is a timeline of how computers evolved from their humble beginnings to the machines of today that surf the Internet, play games and stream multimedia in addition to crunching numbers.

1801: In France, Joseph Marie Jacquard invents a loom that uses punched wooden cards to automatically weave fabric designs. Early computers would use similar punch cards.

1822: English mathematician Charles Babbage conceives of a steam-driven calculating machine that would be able to compute tables of numbers. The project, funded by the English government, is a failure. More than a century later, however, the world's first computer was actually built. He is also known as the father of computer.

1890: Herman Hollerith designs a punch card system to calculate the 1880 census, accomplishing the task in just three years and saving the government \$5 million. He establishes a company that would ultimately become IBM.

1936: Alan Turing presents the notion of a universal machine, later called the Turing machine, capable of computing anything that is computable. The central concept of the modern computer was based on his ideas.

1937: J.V. Atanasoff, a professor of physics and mathematics at Iowa State University, attempts to build the first computer without gears, cams, belts or shafts.

1939: Hewlett-Packard is founded by David Packard and Bill Hewlett in a Palo Alto, California, garage, according to the Computer History Museum.

1941: Atanasoff and his graduate student, Clifford Berry, design a computer that can solve 29 equations simultaneously. This marks the first time a computer is able to store information on its main memory.



1943-1944: Two University of Pennsylvania professors, John Mauchly and J. Presper Eckert, build the Electronic Numerical Integrator and Calculator (ENIAC). Considered the grandfather of digital computers, it fills a 20-foot by 40-foot room and has 18,000 vacuum tubes.

1946: Mauchly and Presper leave the University of Pennsylvania and receive funding from the Census Bureau to build the UNIVAC, the first commercial computer for business and government applications.

1947: William Shockley, John Bardeen and Walter Brattain of Bell Laboratories invent the transistor. They discovered how to make an electric switch with solid materials and no need for a vacuum.

1953: Grace Hopper develops the first computer language, which eventually becomes known as COBOL. Thomas Johnson Watson Jr., son of IBM CEO Thomas Johnson Watson Sr., conceives the IBM 701 EDPM to help the United Nations keep tabs on Korea during the war.

1954: The FORTRAN programming language, an acronym for FORMulaTRANslation, is developed by a team of programmers at IBM led by John Backus, according to the University of Michigan.

1958: Jack Kilby and Robert Noyce unveil the integrated circuit, known as the computer chip. Kilby was awarded the Nobel Prize in Physics in 2000 for his work.

1964: Douglas Engelbart shows a prototype of the modern computer, with a mouse and a graphical user interface (GUI). This marks the evolution of the computer from a specialized machine for scientists and mathematicians to technology that is more accessible to the general public.

1969: A group of developers at Bell Labs produce UNIX, an operating system that addressed compatibility issues. Written in the C programming language, UNIX was portable across multiple platforms and became the operating system of choice among mainframes at large companies and government entities. Due to the slow nature of the system, it never quite gained traction among home PC users.

1970: The newly formed Intel unveils the Intel 1103, the first Dynamic Access Memory (DRAM) chip.

1971: Alan Shugart leads a team of IBM engineers who invent the "floppy disk," allowing data to be shared among computers.

1973: Robert Metcalfe, a member of the research staff for Xerox, develops Ethernet for connecting multiple computers and other hardware.



1974-1977: A number of personal computers hit the market, including Scelbi & Mark-8 Altair, IBM 5100, Radio Shack's TRS-80 — affectionately known as the "Trash 80" — and the Commodore PET.

1975: The January issue of Popular Electronics magazine features the Altair 8080, described as the "world's first minicomputer kit to rival commercial models." Two "computer geeks," Paul Allen and Bill Gates, offer to write software for the Altair, using the new BASIC language. On April 4, after the success of this first endeavor, the two childhood friends form their own software company, Microsoft.

1976: Steve Jobs and Steve Wozniak start Apple Computers on April Fool's Day and roll out the Apple I, the first computer with a single-circuit board, according to Stanford University.

The TRS-80, introduced in 1977, was one of the first machines whose documentation was intended for non-geeks (Image credit: Radioshack)

1977: Radio Shack's initial production run of the TRS-80 was just 3,000. It sold like crazy. For the first time, non-geeks could write programs and make a computer do what they wished.

1977: Jobs and Wozniak incorporate Apple and show the Apple II at the first West Coast Computer Faire. It offers colour graphics and incorporates an audio cassette drive for storage.

1978: Accountants rejoice at the introduction of VisiCalc, the first computerized spreadsheet program.

1979: Word processing becomes a reality as MicroPro International releases WordStar. "The defining change was to add margins and word wrap," said creator Rob Barnaby in email to Mike Petrie in 2000. "Additional changes included getting rid of command mode and adding a print function. I was the technical brains — I figured out how to do it, and did it, and documented it.

The first IBM personal computer, introduced on Aug. 12, 1981, used the MS-DOS operating system. (Image credit: IBM)

1981: The first IBM personal computer, code-named "Acorn," is introduced. It uses Microsoft's MS-DOS operating system. It has an Intel chip, two floppy disks and an optional color monitor. Sears & Roebuck and Computer and sell the machines, marking the first time a computer is available through outside distributors. It also popularizes the term PC.



1983: Apple's Lisa is the first personal computer with a GUI. It also features a drop-down menu and icons. It flops but eventually evolves into the Macintosh. The Gavilan SC is the first portable computer with the familiar flip form factor and the first to be marketed as a "laptop."

1985: Microsoft announces Windows, according to Encyclopedia Britannica. This was the company's response to Apple's GUI. Commodore unveils the Amiga 1000, which features advanced audio and video capabilities.

1985: The first dot-com domain name is registered on March 15, years before the World Wide Web would mark the formal beginning of Internet history. The Symbolics Computer Company, a small Massachusetts computer manufacturer, registers Symbolics.com. More than two years later, only 100 dot-coms had been registered.

1986: Compaq brings the Deskpro 386 to market. Its 32-bit architecture provides as speed comparable to mainframes.

1990: Tim Berners-Lee, a researcher at CERN, the high-energy physics laboratory in Geneva, develops Hyper Text Markup Language (HTML), giving rise to the World Wide Web.

1993: The Pentium microprocessor advances the use of graphics and music on PCs.

1994: PCs become gaming machines as "Command & Conquer," "Alone in the Dark 2," "Theme Park," "Magic Carpet," "Descent" and "Little Big Adventure" are among the games to hit the market.

1996: Sergey Brin and Larry Page develop the Google search engine at Stanford University.

1997: Microsoft invests \$150 million in Apple, which was struggling at the time, ending Apple's court case against Microsoft in which it alleged that Microsoft copied the "look and feel" of its operating system.

1999: The term Wi-Fi becomes part of the computing language and users begin connecting to the Internet without wires.

2001: Apple unveils the Mac OS X operating system, which provides protected memory architecture and pre-emptive multi-tasking, among other benefits. Not to be outdone, Microsoft rolls out Windows XP, which has a significantly redesigned GUI.

2003: The first 64-bit processor, AMD's Athlon 64, becomes available to the consumer market.



2004: Mozilla's Firefox 1.0 challenges Microsoft's Internet Explorer, the dominant Web browser. Facebook, a social networking site, launches.

2005: YouTube, a video sharing service, is founded. Google acquires Android, a Linux-based mobile phone operating system.

2006: Apple introduces the MacBook Pro, its first Intel-based, dual-core mobile computer, as well as an Intel-based iMac. Nintendo's Wii game console hits the market.

2007: The iPhone brings many computer functions to the smartphone.

2009: Microsoft launches Windows 7, which offers the ability to pin applications to the taskbar and advances in touch and handwriting recognition, among other features.

2010: Apple unveils the iPad, changing the way consumers view media and jumpstarting the dormant tablet computer segment.

2011: Google releases the Chromebook, a laptop that runs the Google Chrome OS.

2012: Facebook gains 1 billion users on October 4.

2015: Apple releases the Apple Watch. Microsoft releases Windows 10.

2016: The first reprogrammable quantum computer was created. "Until now, there hasn't been any quantum-computing platform that had the capability to program new algorithms into their system. They're usually each tailored to attack a particular algorithm," said study lead author Shantanu Debnath, a quantum physicist and optical engineer at the University of Maryland, College Park.

2017: The Defense Advanced Research Projects Agency (DARPA) is developing a new "Molecular Informatics" program that uses molecules as computers. "Chemistry offers a rich set of properties that we may be able to harness for rapid, scalable information storage and processing," Anne Fischer, program manager in DARPA's Defense Sciences Office, said in a statement. "Millions of molecules exist, and each molecule has a unique three-dimensional atomic structure as well as variables such as shape, size, or even color. This richness provides a vast design space for exploring novel and multi-value ways to encode and process data beyond the 0s and 1s of current logic-based, digital architectures.

1.4 Characteristics



The following are the characteristics of a typical computer:

1). Speed

Present day computer operate at very high speed. A computer can perform several million instructions (calculations) in one second. For example, it can add or multiply 2 lakh numbers in a second. There are several different types of computers and they all have different speeds running from high to very-very high. However, even the speed of the slowest personal computer (PC) is very high compare to that of a human being, as far as arithmetic operations are concerned. Typically, the speed of computers is specified in MIP(Million Instructions per Seconds) or MLFOPS(Million Floating-Point Operation Per Seconds).

2). Accuracy

Computers perform with a very high degree of consistent accuracy. Now a days computer technology stabilized, and the chances of a computer giving in accurate results are very rare. If you ask a computer to perform a particular calculation, say, division of two numbers a thousand times, it will perform each division operation with the same accuracy.

Sometimes computers do make mistakes. This may happen if there is an undedicated flaw in the design of the computer (That is very rare now a day). Most of the times, computers make mistakes if they are not programmed correctly. That is, if the programmer who has written the program to do same calculations did not consider all aspects of the data that will be feed into the computer, it can give in-accurate results. Computers can give in-accurate results if the input data is in-accurate, e.g. if you try to divide a number by zero (0).

3). Diligence

When human beings are required to work continuously for a few hours, they become try and start losing concentration. On the other hand, a computer can continue a work for hour (or even days) at the same speed and accuracy. It does not show signs of tiredness or lake of concentration when many to work continuously. Unlike human beings, it does not complain or show lethargy or laziness when made to do the same task repeated. Because of this property, computers are generally used in all such situation where the same or similar task has to be repeated a numbers of times, e.g. preparing the salary slip for



10 thousand employs of a company, or printing divide end checks for ten lakh share holders of a large company.

4). Versatility

Computers are very versatile. The same computer can be used for various applications. For instance, you can use a Personal Computer (PC) to prepare a letter, prepare the balance sheet of a company, store a database of employees, produce a professional-looking advertisement, send or receive fax messages, etc. for a computer to perform a new job, all it needs is a program. (A program is a set of instructions that enables a computer to do a particular task.) Thus, if you want a computer to do perform a new task, all you need to write a new program for that task.

5). Data Storage

A computer can store a huge amount of data in its memory. You can store almost any type of data, such as a letter, Picture, Sound, etc. in a computer. You can recall the stored from the computer whenever you need it. For instance, if you type a letter you can save it. Then, if you want to send a single letter to another person, you can recall that letter from the computers memory, modify it and then print a new letter.

6). No Intelligence

A computer is dumb. It has no intelligence of its own. It cannot think or apply its judgment. It gets its power from the program that it runs. It will do only what it is asked to do. It has to be hold what to do, and in what sequence. Therefore, the program that the computer runs determines what task it will perform. Those, if you run a word processor program on a computer, it becomes a word processor and if you run a Desktop Publishing (DTP) program, It becomes a Desktop publisher. So, a computer does not take its own decisions—it simply follows the programmer or the user.

7). No Emotions

Computers are not living beings. Hence, they do not have any emotions. They do not have any heart or soul. Human beings often take some decisions based on emotions, taste, feelings, etc. in their daily life. On the other hand, computers always take decisions based on a program that they run.



Check your Progress A

Fill in the blanks:

- 1..... is organised or classified data which has some meaningful values for the receiver.
- 2 The first IBM personal computer, introduced on
3. Computers arebeings.

1.5 Advantages

Following list demonstrates the advantages of computers in today's arena.

High Speed

- 1.Space Computer is a very fast device.
- 2.It is capable of performing calculation of very large amount of data.
- 3.The computer has units of speed in microsecond, nanosecond, and even the picosecond.
- 4.It can perform millions of calculations in a few seconds as compared to man who will spend many months for doing the same task.

Accuracy

- 1.In addition to being very fast, computers are very accurate.
- 2.The calculations are 100% error free.
- 3.Computers perform all jobs with 100% accuracy provided that correct input has been given.

Storage Capability

- 1.Memory is a very important characteristic of computers.
- 2.A computer has much more storage capacity than human beings.
- 3.It can store large amount of data.
- 4.It can store any type of data such as images, videos, text, audio and many others.

Diligence



1. Unlike human beings, a computer is free from monotony, tiredness and lack of concentration.
2. It can work continuously without any error and boredom.
3. It can do repeated work with same speed and accuracy.

Versatility

1. A computer is a very versatile machine.
2. A computer is very flexible in performing the jobs to be done.
3. This machine can be used to solve the problems related to various fields.
4. At one instance, it may be solving a complex scientific problem and the very next moment it may be playing a card game.

Reliability

1. A computer is a reliable machine.
2. Modern electronic components have long lives.
3. Computers are designed to make maintenance easy.

Automation

1. Computer is an automatic machine.
2. Automation means ability to perform the given task automatically.
3. Once a program is given to computer i.e., stored in computer memory, the program and instruction can control the program execution without human interaction.

Reduction of Paper Work

1. The use of computers for data processing in an organization leads to reduction in paper work and results in speeding up a process.
2. As data in electronic files can be retrieved as and when required, the problem of maintenance of large number of paper files gets reduced.



Reduction in Cost

1. Though the initial investment for installing a computer is high but it substantially reduces the cost of each of its transaction.

1.6 Disadvantages

Following list demonstrates the disadvantages of computers in today's arena

No I.Q

1. A computer is a machine that has no intelligence to perform any task.
2. Each instruction has to be given to computer.
3. A computer cannot take any decision on its own.

Dependency

1. It functions as per a user's instruction, so it is fully dependent on human being
2. Environment
3. The operating environment of computer should be dust free and suitable.

No Feeling

1. Computers have no feelings or emotions.
2. It cannot make judgement based on feeling, taste, experience, and knowledge unlike a human being.

1.7 Applications

The use of computers is increasing at such a rate that there is hardly any field where computers are not used. The following list describes some of the applications of computers:

1. In offices and homes for preparing documents and to perform other data processing jobs.
2. To prepare salary slips and salary cheques in office and factories.
3. To maintain accounts and transfer funds in banks.
4. To store and retrieve large amount of information in offices.



5. To send and receive electronic mail / fax.
6. To search and retrieve information from other computers.
7. To reserve tickets in the transportation sectors, eg Railways, Air Lines, etc.
8. To regulate traffic lights on roads and to control machines and robots in factories.
9. To design automobiles, buildings and dams and to forecast weather.
10. To create animation / cartoon movies and compose music.
11. To control modern automobiles, trains, airplanes etc.
12. To control electronic appliances, such as air-conditioner, TVs, VCRs etc.
13. To On-line banking, buy and sell merchandise, shares, bonds, etc.
14. To control and simulate defence equipment.

1.8 Generations of Computers:

The history of the computer goes back several decades however and there are five definable generations of computers.

Each generation is defined by a significant technological development that changes fundamentally how computers operate – leading to more compact, less expensive, but more powerful, efficient and robust machines.

1.8.1 First Generation (1940 – 1956 Vacuum Tubes)

These early computers used vacuum tubes as circuitry and magnetic drums for memory. As a result they were enormous, literally taking up entire rooms and costing a fortune to run. These were inefficient materials which generated a lot of heat, sucked huge electricity and subsequently generated a lot of heat which caused ongoing breakdowns.

These first generation computers relied on ‘machine language’ (which is the most basic programming language that can be understood by computers). These computers were limited to solving one problem at a time. Input was based on punched cards and paper tape. Output came out on print-outs. The two



notable machines of this era were the UNIVAC and ENIAC machines – the UNIVAC is the first every commercial computer which was purchased in 1951 by a business – the US Census Bureau.

1.8.2 Second Generation (1956 – 1963 Transistors)

The replacement of vacuum tubes by transistors saw the advent of the second generation of computing. Although first invented in 1947, transistors weren't used significantly in computers until the end of the 1950s. They were a big improvement over the vacuum tube, despite still subjecting computers to damaging levels of heat. However they were hugely superior to the vacuum tubes, making computers smaller, faster, cheaper and less heavy on electricity use. They still relied on punched card for input/printouts.

The language evolved from cryptic binary language to symbolic ('assembly') languages. This meant programmers could create instructions in words. About the same time high level programming languages were being developed (early versions of COBOL and FORTRAN). Transistor-driven machines were the first computers to store instructions into their memories – moving from magnetic drum to magnetic core 'technology'. The early versions of these machines were developed for the atomic energy industry.

1.8.3 Third Generation (1964 – 1971 Integrated Circuits)

By this phase, transistors were now being miniaturised and put on silicon chips (called semiconductors). This led to a massive increase in speed and efficiency of these machines. These were the first computers where users interacted using keyboards and monitors which interfaced with an operating system, a significant leap up from the punch cards and printouts. This enabled these machines to run several applications at once using a central program which functioned to monitor memory.

As a result of these advances which again made machines cheaper and smaller, a new mass market of users emerged during the '60s.

1.8.4 Fourth Generation (1972 – 1989 Microprocessors)

This revolution can be summed in one word: Intel. The chip-maker developed the Intel 4004 chip in 1971, which positioned all computer components (CPU, memory, input/output controls) onto a single chip. What filled a room in the 1940s now fit in the palm of the hand. The Intel chip housed thousands



of integrated circuits. The year 1981 saw the first ever computer (IBM) specifically designed for home use and 1984 saw the MacIntosh introduced by Apple. Microprocessors even moved beyond the realm of computers and into an increasing number of everyday products.

The increased power of these small computers meant they could be linked, creating networks. Which ultimately led to the development, birth and rapid evolution of the Internet. Other major advances during this period have been the Graphical user interface (GUI), the mouse and more recently the astounding advances in lap-top capability and hand-held devices.

1.8.5 Fifth Generation (1990 to onwards Artificial Intelligence)



Figure 1.2 show the artificial intelligence

Computer devices with artificial intelligence are still in development, but some of these technologies are beginning to emerge and be used such as voice recognition.

AI is a reality made possible by using parallel processing and superconductors. Looking to the future, computers will be radically transformed again by quantum computation, molecular and nano technology.

The essence of fifth generation will be using these technologies to ultimately create machines which can process and respond to natural language, and have capability to learn and organise themselves.

Check your Progress B

Fill in the blanks:



1. GUI stands for.....
2. These early computers usedas circuitry and magnetic drums for memory.
3. Integrated Circuits used ingeneration.
4. A.I stands for.....

1.9 Components of Computer

A computer is a machine operating under the control of instructions stored in its own memory. These operations and instructions enable the computer to receive data from a user (input), transform and manipulate the data according to specified rules (process), produce results (output). Additionally, data, instructions, and information are stored (storage) for future retrieval and use. Many computers are also capable of another task: communicating directly with other machines.

The user performs the input task with a device such as a keyboard, mouse, or digital scanner. These devices allow the user to enter data and instructions into the computer. A secondary storage system stores and retrieves additional data and instructions that may also be used in the input and processing stages. This system might include magnetic or optical devices, such as CD-ROMs, hard disks, floppy disks, and tapes. The central processing system, which manipulates the data, is perhaps the most important part of the computer. This system is the “brain” of the computer in that it enables the computer to transform unorganized inputs into useful information. The central processing system includes the central processing unit (CPU) and the primary memory. The computer’s output system displays the results of the data manipulation. The output system might include a monitor, a printer, a plotter, a voice output device, or microfilm/microfiche equipment. A final element of a computer is the communication system, which passes information from computer to computer over communication media. Each of these systems is discussed in more detail below.

As noted above, computers come in many types. It would be difficult to adequately cover the variations in the components of these different computer types in a brief introduction. Therefore, we will confine the following discussion to personal computers (PCs). However, most of the discussion, especially as relating to basic computer operations, is easily transferable to other computer types. All types of



computers follow a same basic logical structure and perform the following five basic operations for converting raw input data into information useful to their users.

Sr.No.	Operation	Description
1	Take Input	The process of entering data and instructions into the computer system
2	Store Data	Saving data and instructions so that they are available for processing as and when required.
3	Processing Data	Performing arithmetic, and logical operations on data in order to convert them into useful information.
4	Output Information	The process of producing useful information or results for the user, such as a printed report or visual display.
5	Control the workflow	Directs the manner and sequence in which all of the above operations are performed.

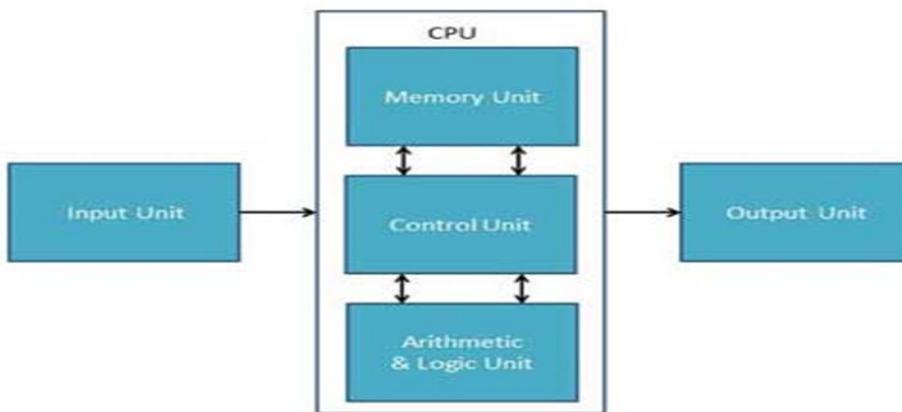


Figure 1.3 Components of CPU

1. Input Unit

This unit contains devices with the help of which we enter data into computer. This unit makes link between user and computer. The input devices translate the information into the form understandable by



computer. The digital computers can understand the binary language made of zeros and ones. The digital computers are mainly used in input devices.

2. CPU (Central Processing Unit)

CPU is considered as the brain of the computer. CPU performs all types of data processing operations. It stores data, intermediate results and instructions(program). It controls the operation of all parts of computer.

CPU itself has following three components

- Memory Unit
- Control Unit
- ALU (Arithmetic and Logical Unit)

Memory or Storage Unit

This unit can store instructions, data and intermediate results. This unit supplies information to the other units of the computer when needed. It is also known as internal storage unit or main memory or primary storage or Random access memory(RAM).

Its size affects speed, power and capability. Primary memory and secondary memory are two types of memories in the computer. Functions of memory unit are:

- It stores all the data and the instructions required for processing.
- It stores intermediate results of processing.
- It stores final results of processing before these results are released to an output device.
- All inputs and outputs are transmitted through main memory.

Storage Unit- Storage devices are the computer hardware used to remember/store data. There are many types of storage devices, each with their own benefits and drawbacks. Below are explanations about different storage devices.

1. Hard Disk Drive (HDD)
2. Solid State Drive (SSD)



3. Random Access Memory (RAM)
4. Static RAM (SRAM)
5. Dynamic RAM (DRAM)
6. ROM
7. USB Flash Memory

Control Unit

This unit controls the operations of all parts of computer but does not carry out any actual data processing operations.

Functions of this unit are:

- It is responsible for controlling the transfer of data and instructions among other units of a computer.
- It manages and coordinates all the units of the computer.
- It obtains the instructions from the memory, interprets them, and directs the operation of the computer.
- It communicates with Input/Output devices for transfer of data or results from storage.
- It does not process or store data.

ALU(Arithmetic Logic Unit)

This unit consists of two subsections namely

- Arithmetic section
- Logic Section

Arithmetic Section

Function of arithmetic section is to perform arithmetic operations like addition, subtraction, multiplication and division. All complex operations are done by making repetitive use of above operations. The mathematical function eg. $\sin_x e^4$ etc are not evaluated in this unit.

Logic Section



Function of logic section is to perform logic operations such as comparing, selecting, matching and merging of data.

3. Output Unit

Output unit consists of devices with the help of which we get the information from computer. This unit is a link between computer and users. Output devices translate the computer's output into the human readable format.

1.10 Types of Computers

When talking about a computer or a "PC," you are usually referring to a desktop computer found in a home or office. Today, however, the lines of what makes a computer are blurring. Below are all the different examples of what is considered a computer today.

Examples and types of computers



Figure 1.4 shows the types of computers

The picture above shows several types of computers and computing devices, and is an example

Analog computer

These systems were the first type to be produced. It is an electronic machine capable of performing arithmetic functions on numbers which are represented by some physical quantities such as temperature, pressure, voltage, etc. Analog refers to circuits or numerical values that have a continuous range. Popular analog computer used in the 20th century was the slide rule.

Digital Computers

Virtually all modern computers are digital. Digital refers to the processes in computers that manipulate binary numbers (0s or 1s), which represent switches that are turned on or off by electrical current. A bit



can have the value 0 or the value 1, but nothing in between 0 and 1. A desk lamp can serve as an example of the difference between analog and digital. If the lamp has a simple on/off switch, then the lamp system is digital, because the lamp either produces light at a given moment or it does not. If a dimmer replaces the on/off switch, then the lamp is analog, because the amount of light can vary continuously from on to off and all intensities in between. Digital computers are more common in use and it will be our focus of discussion.

Hybrid Computer

This is when a computer make is of both Analog and Digital components and techniques. Such computer require analog to digital and digital to analog converter which will make analog and digital data palatable to it.

1.The Desktop :

A computer is referred to a desktop when it is relatively small enough to be positioned on top when a person is working. Such a computer can also be placed on floor or somewhere under, or aside of, the table in which case the monitor would be placed on top of the table. This is the most common type of computer used in office or at room or at home. A desktop computer is made of different parts that are connected with cables.

2. The Laptop

A computer is called laptop when it combines the CPU, the monitor, the keyboard, and the mouse in one unit to be so small that you can carry it on your laps when traveling or commuting. A laptop is also called a notebook. Other parts, such as an external mouse, an external keyboard, or peripherals such as a printer or a projector, can be connected to the laptop. A laptop is only physically smaller than a desktop but, everything considered, it can do anything that a desktop can do.

3.The Server:

A server is a computer that holds information that other computers, called workstations, can retrieve. Such workstations are connected to the server using various means they could be connected using cable, wireless connection, etc. Only computers that maintain a type of connection with the server can get the information that is stored in the server.



Normally, although not particularly recommended, any computer, including a desktop or even a laptop can be used as a server, as long as it can do the job required. A server is more defined by the program (called an operating system) that is installed in it, not how the machine looks.

Any type of computer, including a desktop, a laptop, a CD or DVD machine, etc can be connected to a server.

The person who sets up a server also defines the types of connections it is made for.

3. The Mainframe

A manifest is a computer, usually physically big, that does almost all the jobs for other types of computers that are connected to it. This is a broad definition but other aspects are involved. Like a server, the program (operating system) that runs in the mainframe defines its role.

1.11 Software and Types

Software is a general term used to describe a collection of computer programs, procedures, and documentation that perform some task on a computer system. Practical computer systems divide software systems into two major classes: system software (Operating system) and application software. Software is an ordered sequence of instructions for changing the state of the computer hardware in a particular sequence. Software is typically programmed with a user-friendly interface that allows humans to interact more efficiently with a computer system as the user interacts with the application software.

A computer system needs more than the hardware described above in order to function. The hardware by itself, even when powered-up, is incapable of producing useful output. It must be instructed how to direct its operations in order to transform input into output of value to the user. This is the role of software; i.e., to provide the detailed instructions that control the operation of a computer system. Just as hardware comprises the tangible side of the computer, so software is the intangible side of the computer. If the CPU is the physical brain of the computer, then software is its mind.

Software instructions are programmed in a computer language, translated into machine language, and executed by the computer. Between the user and the hardware (specifically, the memory), generally stand two layers of software: system software and application software.

Types Of Software



1. System software
2. Application Software

1.11.1 System software

System software directly controls the computer's hardware, whereas application software is one level removed from hardware. System software manages the computer's resources, enables the various components of the computer to communicate, runs application software, and makes the hardware respond to the user's needs. When the system software operates efficiently, the difficult operations of controlling the hardware are transparent to the user. System software includes four main types:

The operating system provides an interface between the computer hardware and the user or the application software.

Language translators convert application programs and any other software programs into the machine language (discussed below) that actually controls the computer's operations.

Network and communications software operates the communications hardware in a computer so that it can transmit and receive information from other computers. Network and communications software requires two types of software: software for the PC operating system and software for the network operating system. In some cases, the latter comes built-in the former.

Utility programs perform various specialized "housekeeping" tasks, such as file management, virus protection, disk defragmentation, program installation and uninstallation, file and disk back up, disk formatting, and providing screen saver programs. This list is far from exhaustive. The user directly controls most utility programs, although some utility programs can be set to run automatically (e.g.; screen savers and anti-virus scanning).

The system software is collection of programs designed to operate, control, and extend the processing capabilities of the computer itself. System software are generally prepared by computer manufactures. These software products comprise of programs written in low-level languages which interact with the hardware at a very basic level. System software serves as the interface between hardware and the end users.

Some examples of system software are Operating System, Compilers, Interpreter, Assemblers etc.



Features of system software are as follows:

1. Close to system
2. Fast in speed
3. Difficult to design
4. Difficult to understand
5. Less interactive
6. Smaller in size
7. Difficult to manipulate
8. Generally written in low-level language

Computers of all types require system software to coordinate their resources. The system software for a single-user PC is not nearly as complex as the system software for a multiuser mainframe computer. However, as the PC's system capacity has increased, the sophistication of its system software also has increased. Many of the features once found only in mainframe and minicomputer systems have been incorporated into PCs.

1.11.2 Application Software

Application software enables the user to direct the computer's processing system in the tasks of manipulating and transforming input data into useful output information. Furthermore, it allows the user to alter the information generated by the processing system; e.g., how the information is presented. This is the type of software with which most users interact. It is the usual interface between user and computer. Rarely do users directly manipulate systems software, especially the operating systems software.

Application software can be written for a specific user's application (custom software), or it can be mass-produced for general use (commercial or packaged software). Naturally, custom software is usually far more expensive than commercial software. An accounting package written for a specific company might cost many thousands of dollars, whereas a commercial accounting package might cost only a few hundred dollars at a retail store. The advantage of custom software is that it is tailored to the



user's specific needs and can be seamlessly integrated into the user's existing software. It is not only commercial software, but less costly, and available also immediately, and the package can be evaluated before being purchased.

Application software comes in an incredible variety. It is available for business, personal, educational, communication, and graphic design purposes—to name the more usual categories. There is almost certainly a software package somewhere available to suit any need. If not, there are programmers ready to be hired to build it. For our purposes, we will limit our discussion to the four types of application software most likely to be useful to accounting and business students: word processing, spreadsheet, database, and presentation graphics. These four applications are frequently sold together in a single software package. Three of the most popular packages are Corel WordPerfect Suite, Microsoft Office 2000, and Lotus SmartSuite. In addition to the four "standard" applications, these packages usually include email, Internet, video processing, and desktop publishing applications.

Word processing programs allow the user to quickly and easily create and revise texts on the computer screen. By using word processing applications, the user can format documents with ease, changing font size, margins, color, etc. Different types of documents—e.g., letters, memos, and reports—are often preformatted in the application. PC-based word processing software is so capable and inexpensive that, in most businesses, it has become the usual tool for creating documents, even when more powerful mainframes and minicomputers are available.

Spreadsheet programs are especially useful in business and accounting. The electronic spreadsheet consists of rows and columns of data, which the user can easily edit, copy, move, or print. Using numeric data entered in the spreadsheet, the computer can perform numerous mathematical calculations automatically, many of impressive sophistication (e.g., statistical, logical, and engineering functions). One of the spreadsheet program's most powerful features for business purposes is that it enables the user to do "what-if" analyses on existing data and to input different data for various scenarios. Non-numeric data (e.g., names and dates) may also be entered in a spreadsheet. Spreadsheets can perform some non-mathematical operations (e.g., sorting and filtering) on this data, although this type of analysis is not a spreadsheet's strength.

Database software allows the user to enter, store, maintain, retrieve, and manipulate data. In some ways, databases pickup where spreadsheets leave off, although a fairer assessment is probably that



the relationship between the two types of software is reciprocal. Database software is certainly more efficient and effective at handling non-numeric data than is spreadsheet software. Conversely, numeric data is usually easier to manipulate in a spreadsheet. In most databases, data is entered to tables of rows and columns, similar to spreadsheets. Unlike spreadsheets, these tables can be connected into relationships that allow users incredible versatility in what they can do with that data. For example, data—both numeric and non-numeric—from several individual tables may be retrieved and used together in calculations, with the results presented in a business-style report.

Presentation graphics software enable users to design professional-quality presentations for business and educational purposes. The presentations usually consist of formatted slides for projecting onto a screen from a computer projector or overhead projector, or for display on a large monitor. These presentations may also be used for online meetings and Web broadcasts. The slides can be designed to include backgrounds, graphic images, charts, clipart, shading, animation, and audio effects—and, of course, text, which can sometimes get lost in all of the embellishments.

Application software products are designed to satisfy a particular need of a particular environment. All software applications prepared in the computer lab can come under the category of Application software.

Application software may consist of a single program, such as a Microsoft's notepad for writing and editing simple text. It may also consist of a collection of programs, often called a software package, which work together to accomplish a task, such as a spreadsheet package.

Examples of Application software are following:

1. Payroll Software
2. Student Record Software
3. Inventory Management Software
4. Income Tax Software
5. Railways Reservation Software
6. Microsoft Office Suite Software
7. Microsoft Word



8. Microsoft Excel

9. Microsoft PowerPoint

1.12 Hardware/Fixware

The term hardware refers to mechanical device that makes up computer. Computer hardware consists of interconnected electronic devices that we can use to control computer's operation, input and output. Examples of hardware are CPU, keyboard, mouse, hard disk, etc. Hardware is also known as fixware.

1.12.1 Input Devices:

Following are some of the important input devices which are used in a computer –

- Keyboard
- Mouse
- Joy Stick
- Light pen
- Track Ball
- Scanner
- Graphic Tablet
- Microphone
- Magnetic Ink Card Reader(MICR)
- Optical Character Reader(OCR)
- Bar Code Reader
- Optical Mark Reader(OMR)

1.12.2 Output Devices:

Following are some of the important output devices used in a computer.

- Monitors



1. Cathode-Ray Tube (CRT)
 2. Flat-Panel Display
- Graphic Plotter
 - Printer
 1. Impact Printers
 2. Non-Impact Printers

Check Your Progress C:

Fill in the blanks:

- 1 CRT display is made up of picture elements called
- 2 Printer is used to information on
- 3 Software is a set of

1.13 Summary:

A computer is a programmable device that stores, retrieves, and processes data. The term "computer" was originally given to humans (human computers) who performed numerical calculations using mechanical calculators, such as the abacus and slide rule. The term was later given to a mechanical device as they began replacing the human computers. Today's computers are electronic devices that accept data (input), process that data, produce output, and store (storage) the results.

Today, there are two types of computers the PC (IBM compatible) and Apple Mac. Several companies that make and build PCs, and if you get all the necessary parts for a computer, you can even build a custom PC. However, when it comes to Apple, only Apple designs and makes these computers. See our computer companies page for a listing of companies (OEMs) that make and build computers.

1.14 Keywords:

Computer: Computer is an electronic data processing device which accepts and stores data input, processes the input data, and generated the input data.



Data: Data can be defined as a representation of facts, concepts or instructions in a formalized manner which should be suitable for communication, interpretation, or processing by human or electronic machine. Data is represented with the help of characters like alphabets (A-Z, a-z), digits (0-9) or special characters (+,-,/,*,<,>= etc.).

Information: Information is organised or classified data which has some meaningful values for the receiver. Information is the processed data on which decisions and actions are based. For the decision to be meaningful, the processed data must qualify for the following characteristics: timely, accuracy, completeness.

Data Processing: Data processing is the re-structuring or re-ordering of data by people or machine to increase their usefulness and add values for particular purpose. Data processing consists of basic steps input, processing and output. These three steps constitute the data processing cycle i.e input, processing and output.

ANSWERS TO CHECK YOUR PROGRESS

CHECK YOUR PROGRESS A

1. Information
2. Aug. 12, 1981
3. not living

CHECK YOUR PROGRESS B

1. Graphical user interface
2. vacuum tubes
3. Third
4. artificial intelligence

CHECK YOUR PROGRESS C

- 1 small, pixels
- 2 print, paper
- 3 instructions



1.15 Self Assessed Question:

1. What is computer? Explain in brief?
2. What are components of computers?
3. Explain the types of computers?
4. Explain the generations of computers in brief?
5. What is Software? Explain in brief?
6. Explain the Characteristics of Computer.
7. What is Hardware? Explain in brief?
8. What are the storage devices?

1.16 SUGGESTED READINGS

1. Computer Fundamentals by P.K Sinha, BPA Publications
2. computer Fundamentals Architecture and Organization by B Ram Sanjay Kumar, New Age International Publishers
3. Computer Fundamentals Introduction to Computers by Faithe Wempen, Johan Wiley & Sons.



Class : M.Sc. (Mathematics)	Course Code : MAL 516
Subject : Programming with FORTRAN (Theory)	

Chapter-2

STEPS TO SOLVE A MATHEMATICAL PROBLEM WITH COMPUTER

STRUCTURE:

- 2.0 Objective
- 2.1 Introduction
- 2.2 Problem Solving
- 2.3 Problem Solving Process
- 2.4 Structured Analysis
- 2.5 Structured Analysis Tools
 - 2.5.1 DFD
 - 2.5.2 Data Dictionary
 - 2.5.3 Decision Tree
 - 2.5.4 Decision Table
 - 2.5.5 Structured English
 - 2.5.6 Pseudocode
- 2.6 Guidelines for Choosing the Tools
- 2.7 Algorithm Design
 - 4.7.1 Characteristics of Algorithm



4.7.2 Algorithm vs Pseudocode

2.8 Problem Solving

2.9 Summary

2.10 Keyword

2.11 Self Assessment Question

2.12 Suggested Readings

2.0 Objective:

After reading this chapter, you should be able to:

- 1) Understand the concept of problem solving.
- 2) Understand the concept of flow charts.
- 3) Understand the concept of algorithms.
- 4) Understand the concept of structured analysis.

2.1 Introduction

Computers do what we tell them to do, NOT what we want them to do. Computer Programming involves writing instructions and giving them to the computer to complete a task. A computer program or software is a set of instructions written in a computer language in order to be executed by a computer to perform a useful task. Ex: Application software packages, such as word processors, spreadsheets and databases are all computer programs. A Computer Programmer is a person who translates the task you want a computer to do into a form that a computer can understand. A well designed computer program must be:

- Correct and Accurate
- Easy to Understand
- Easy to Maintain and Update



- Efficient
- Reliable
- Flexible

The program development process involves the following steps:

1. Program Documentation: Program documentation is the text or graphics that provide description of the purpose of a function or a particular step
2. The purpose of instruction in a program:
 - ✓ Determine the user needs
 - ✓ Design the program specification
 - ✓ Review the program specifications
 - ✓ Design the algorithm
 - ✓ Steps that will convert the available input into the desired output
 - ✓ Step by step solution for a given problem is called the algorithm
 - ✓ A flowchart graphically details processing steps of a particular program
3. Code the program: Write the program in a programming language using a program editor. A program editor is a program that allows to
 - ✓ Type, edit and save a program code on a disk
 - ✓ Compile, test and debug the program
4. In order to find out possible errors in the program
 - ✓ types of errors may be syntax errors
 - ✓ run-time errors and logic errors
 - ✓ get program to the user
 - ✓ install software on the user's computer and offer training

Types of programming errors

- ✓ Syntax of a programming language is the set of rules to be followed when writing a program
- ✓ Syntax error occurs when these rules are violated
- ✓ Run-time errors occur when invalid data is entered during program execution e.g. program expects numeric data and alphabetic data is entered



- ✓ Program will crash, logic error will not stop the program but results will be inaccurate

The process of finding and correcting errors in a program is called debugging. For successful programming:

- ✓ Give no ambiguity in program instructions
- ✓ Give no possibility of alternative interpretations
- ✓ Make sure there is only one course of action

Programming task can be made easier by breaking large and complex programs into smaller and less complex subprograms (modules). Programming task can be separated into 2 phases

- ✓ Problem solving phase
- ✓ Produce an ordered sequence of steps

that describes the solution of problem. This sequence of steps is called an algorithm. e.g of an algorithm: a recipe, to assemble a brand new computer etc.

Implementation phase:

- ✓ Implement the program in some programming language (Pascal, Basic, C)

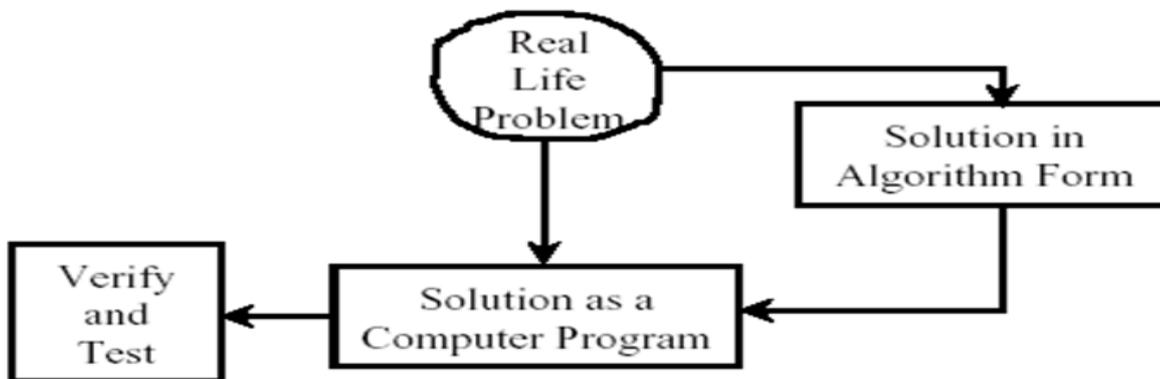


Figure 2.1: Problem Solving and Programming

Structured programming: A programming technique that splits the program into smaller segments (modules) to

1. Decrease program development time



2. Decrease program maintenance cost
3. Improve the quality of software

Structured programming achieves these goals by using

1. Top-down design and use of modules
2. Use of limited control structures (sequence, selection and repetition)
3. Management control

Top-down design starts with major functions involved in a problem and divide them into sub-functions until the problem has been divided as much as possible.

2.2 Problem Solving:

Problem solving is the act of defining a problem; determining the cause of the problem; identifying, prioritizing, and selecting alternatives for a solution; and implementing a solution.

- The problem-solving process

Step	Characteristics
1. Define the problem	Differentiate fact from opinion Specify underlying causes Consult each faction involved for information State the problem specifically Identify what standard or expectation is violated Determine in which process the problem lies Avoid trying to solve the problem without data
2. Generate alternative solutions	Postpone evaluating alternatives initially Include all involved individuals in the generating of alternatives Specify alternatives consistent with organizational goals Specify short- and long-term alternatives



	Brainstorm on others' ideas Seek alternatives that may solve the problem
3. Evaluate and select an alternative	Evaluate alternatives relative to a target standard Evaluate all alternatives without bias Evaluate alternatives relative to established goals Evaluate both proven and possible outcomes State the selected alternative explicitly
4. Implement and follow up on the solution	Plan and implement a pilot test of the chosen alternative Gather feedback from all affected parties Seek acceptance or consensus by all those affected Establish ongoing measures and monitoring Evaluate long-term results based on final solution

Problem solving resources



Figure 2.2 Problem Solving Chart

2.3 Problem Solving Process:

In order to effectively manage and run a successful organization, leadership must guide their employees and develop problem-solving techniques. Finding a suitable solution for issues can be accomplished by following the basic four-step problem-solving process and methodology outlined below.

1. Define the problem



Diagnose the situation so that your focus is on the problem, not just its symptoms. Helpful problem-solving techniques include using flowcharts to identify the expected steps of a process and cause-and-effect diagrams to define and analyze root causes.

The sections below help explain key problem-solving steps. These steps support the involvement of interested parties, the use of factual information, comparison of expectations to reality, and a focus on root causes of a problem. You should begin by:

- Reviewing and documenting how processes currently work (i.e., who does what, with what information, using what tools, communicating with what organizations and individuals, in what time frame, using what format).
- Evaluating the possible impact of new tools and revised policies in the development of your "what should be" model.

2. Generate alternative solutions

Postpone the selection of one solution until several problem-solving alternatives have been proposed. Considering multiple alternatives can significantly enhance the value of your ideal solution. Once you have decided on the "what should be" model, this target standard becomes the basis for developing a road map for investigating alternatives. Brainstorming and team problem-solving techniques are both useful tools in this stage of problem solving.

Many alternative solutions to the problem should be generated before final evaluation. A common mistake in problem solving is that alternatives are evaluated as they are proposed, so the first acceptable solution is chosen, even if it's not the best fit. If we focus on trying to get the results we want, we miss the potential for learning something new that will allow for real improvement in the problem-solving process.

3. Evaluate and select an alternative

Skilled problem solvers use a series of considerations when selecting the best alternative. They consider the extent to which:

- A particular alternative will solve the problem without causing other unanticipated problems.



- All the individuals involved will accept the alternative.
- Implementation of the alternative is likely.
- The alternative fits within the organizational constraints.

4. Implement and follow up on the solution

Leaders may be called upon to direct others to implement the solution, "sell" the solution, or facilitate the implementation with the help of others. Involving others in the implementation is an effective way to gain buy-in and support and minimize resistance to subsequent changes.

Regardless of how the solution is rolled out, feedback channels should be built into the implementation. This allows for continuous monitoring and testing of actual events against expectations. Problem solving, and the techniques used to gain clarity, are most effective if the solution remains in place and is updated to respond to future changes.

2.4 Structured Analysis:

Structured Analysis is a development method that allows the analyst to understand the system and its activities in a logical way.

It is a systematic approach, which uses graphical tools that analyze and refine the objectives of an existing system and develop a new system specification which can be easily understandable by user.

It has following attributes –

- It is graphic which specifies the presentation of application.
- It divides the processes so that it gives a clear picture of system flow.
- It is logical rather than physical i.e., the elements of system do not depend on vendor or hardware.
- It is an approach that works from high-level overviews to lower-level details.

Check your progress A

1 Structured programming the development time.

2 Problem solving is the a problem.



3 is a development method that allows the analyst to understand the system.

2.5 Structured Analysis Tools

During Structured Analysis, various tools and techniques are used for system development. They are –

- Data Flow Diagrams
- Data Dictionary
- Decision Trees
- Decision Tables
- Structured English
- Pseudocode

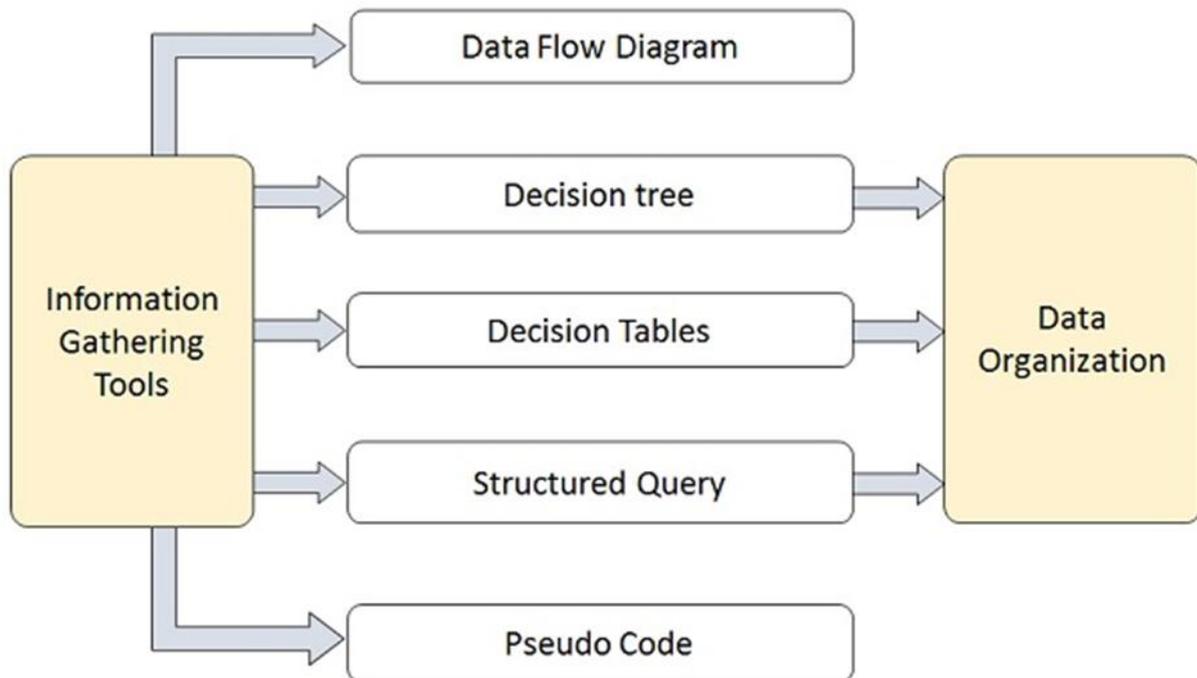


Figure 2.3 shows the structured analysis tool

2.5.1 Data Flow Diagrams (DFD)

It is a technique developed by Larry Constantine to express the requirements of system in a graphical form.



- It shows the flow of data between various functions of system and specifies how the current system is implemented.
- It is an initial stage of design phase that functionally divides the requirement specifications down to the lowest level of detail.
- Its graphical nature makes it a good communication tool between user and analyst or analyst and system designer.
- It gives an overview of what data a system processes, what transformations are performed, what data are stored, what results are produced and where they flow.

Basic Elements of DFD

DFD is easy to understand and quite effective when the required design is not clear and the user wants a notational language for communication. However, it requires a large number of iterations for obtaining the most accurate and complete solution.

The following table shows the symbols used in designing a DFD and their significance –

Symbol Name	Symbol	Meaning
Square		Source or Destination of Data
Arrow		Data flow
Circle		Process transforming data flow
Open Rectangle		Data Store

Types of DFD



DFDs are of two types: Physical DFD and Logical DFD. The following table lists the points that differentiate a physical DFD from a logical DFD.

Physical DFD	Logical DFD
It is implementation dependent. It shows which functions are performed.	It is implementation independent. It focuses only on the flow of data between processes.
It provides low level details of hardware, software, files, and people.	It explains events of systems and data required by each event.
It depicts how the current system operates and how a system will be implemented.	It shows how business operates; not how the system can be implemented.

Context Diagram

A context diagram helps in understanding the entire system by one DFD which gives the overview of a system. It starts with mentioning major processes with little details and then goes onto giving more details of the processes with the top-down approach.

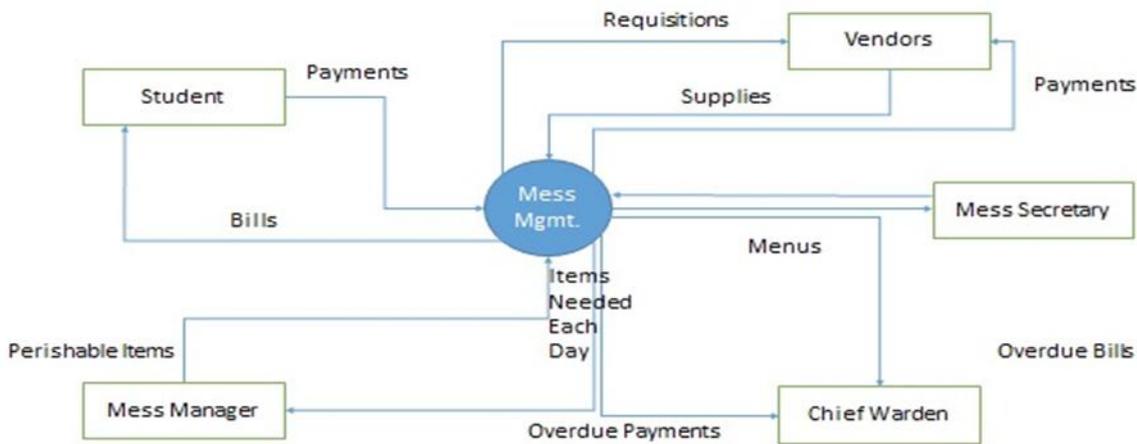


Figure 2.4 Context Diagram to show the mess management system

2.5.2 Data Dictionary

A data dictionary is a structured repository of data elements in the system. It stores the descriptions of all DFD data elements that is, details and definitions of data flows, data stores, data stored in data stores, and the processes.



A data dictionary improves the communication between the analyst and the user. It plays an important role in building a database. Most DBMSs have a data dictionary as a standard feature. For example, refer the following table –

Sr. No.	Data Name	Description	No. of Characters
1	ISBN	ISBN Number	10
2	TITLE	Title	60
3	SUB	Book Subjects	80
4	ANAME	Author Name	15

2.5.3 Decision Trees

Decision trees are a method for defining complex relationships by describing decisions and avoiding the problems in communication. A decision tree is a diagram that shows alternative actions and conditions within horizontal tree framework. Thus, it depicts which conditions to consider first, second, and so on.

Decision trees depict the relationship of each condition and their permissible actions. A square node indicates an action and a circle indicates a condition. It forces analysts to consider the sequence of decisions and identifies the actual decision that must be made.

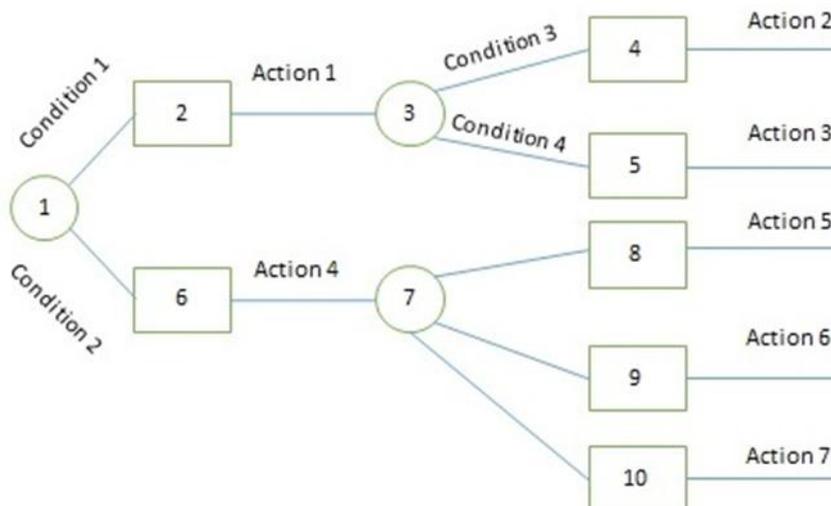


Figure 2.5 Decision tree structure



The major limitation of a decision tree is that it lacks information in its format to describe what other combinations of conditions you can take for testing. It is a single representation of the relationships between conditions and actions.

For example, refer the following decision tree –



Figure 2.6 Decision tree for customer

2.5.4 Decision Tables

Decision tables are a method of describing the complex logical relationship in a precise manner which is easily understandable.

- It is useful in situations where the resulting actions depend on the occurrence of one or several combinations of independent conditions.
- It is a matrix containing row or columns for defining a problem and the actions.

Components of a Decision Table

- Condition Stub – It is in the upper left quadrant which lists all the condition to be checked.
- Action Stub – It is in the lower left quadrant which outlines all the action to be carried out to meet such condition.
- Condition Entry – It is in upper right quadrant which provides answers to questions asked in condition stub quadrant.
- Action Entry – It is in lower right quadrant which indicates the appropriate action resulting from the answers to the conditions in the condition entry quadrant.



The entries in decision table are given by Decision Rules which define the relationships between combinations of conditions and courses of action. In rules section,

- Y shows the existence of a condition.
- N represents the condition, which is not satisfied.
- A blank - against action states it is to be ignored.
- X (or a check mark will do) against action states it is to be carried out.

For example, refer the following table –

CONDITIONS	Rule 1	Rule 2	Rule 3	Rule 4
Advance payment made	Y	N	N	N
Purchase amount = Rs 10,000/-	-	Y	Y	N
Regular Customer	-	Y	N	-
ACTIONS				
Give 5% discount	X	X	-	-
Give no discount	-	-	X	X

2.5.5 Structured English

Structure English is derived from structured programming language which gives more understandable and precise description of process. It is based on procedural logic that uses construction and imperative sentences designed to perform operation for action.



- It is best used when sequences and loops in a program must be considered and the problem needs sequences of actions with decisions.
- It does not have strict syntax rule. It expresses all logic in terms of sequential decision structures and iterations.

For example, see the following sequence of actions –

if customer pays advance

 then

 Give 5% Discount

 else

 if purchase amount $\geq 10,000$

 then

if the customer is a regular customer

 then Give 5% Discount

else No Discount

 end if

 else No Discount

end if

end if

2.5.6 Pseudocode

A pseudocode does not conform to any programming language and expresses logic in plain English.

- It may specify the physical programming logic without actual coding during and after the physical design.
- It is used in conjunction with structured programming.



- It replaces the flowcharts of a program.

2.6 Guidelines for Selecting Appropriate Tools

Use the following guidelines for selecting the most appropriate tool that would suit your requirements –

- Use DFD at high or low level analysis for providing good system documentations.
- Use data dictionary to simplify the structure for meeting the data requirement of the system.
- Use structured English if there are many loops and actions are complex.
- Use decision tables when there are a large number of conditions to check and logic is complex.
- Use decision trees when sequencing of conditions is important and if there are few conditions to be tested.

2.7 Algorithm Design

The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space.

To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient. However, one has to keep in mind that both time consumption and memory usage cannot be optimized simultaneously. If we require an algorithm to run in lesser time, we have to invest in more memory and if we require an algorithm to run with lesser memory, we need to have more time.

Problem Development Steps

The following steps are involved in solving computational problems.

- Problem definition
- Development of a model
- Specification of an Algorithm
- Designing an Algorithm
- Checking the correctness of an Algorithm



- Analysis of an Algorithm
- Implementation of an Algorithm
- Program testing
- Documentation

2.7.1 Characteristics of Algorithms

The main characteristics of algorithms are as follows –

- Algorithms must have a unique name
- Algorithms should have explicitly defined set of inputs and outputs
- Algorithms are well-ordered with unambiguous operations
- Algorithms halt in a finite amount of time. Algorithms should not run for infinity, i.e., an algorithm must end at some point

2.7.2 Difference between Algorithm and Pseudocode

An algorithm is a formal definition with some specific characteristics that describes a process, which could be executed by a Turing-complete computer machine to perform a specific task. Generally, the word "algorithm" can be used to describe any high level task in computer science.

On the other hand, pseudocode is an informal and (often rudimentary) human readable description of an algorithm leaving many granular details of it. Writing a pseudocode has no restriction of styles and its only objective is to describe the high level steps of algorithm in a much realistic manner in natural language.

For example, following is an algorithm for Insertion Sort.

Algorithm: Insertion-Sort

Input: A list L of integers of length n

Output: A sorted list L1 containing those integers present in L

Step 1: Keep a sorted list L1 which starts off empty



Step 2: Perform Step 3 for each element in the original list L

Step 3: Insert it into the correct position in the sorted list L1.

Step 4: Return the sorted list

Step 5: Stop

Here is a pseudocode which describes how the high level abstract process mentioned above in the algorithm Insertion-Sort could be described in a more realistic way.

for i ← 1 to length(A)

 x ← A[i]

 j ← i

 while j > 0 and A[j-1] > x

 A[j] ← A[j-1]

 j ← j - 1

 A[j] ← x

Example: Write an algorithm to determine the students final grade and indicate weather it is passing or failing. The final grade is calculated as the average of four marks.

Solution:

Pseudocode: *Input a set of 4 marks*

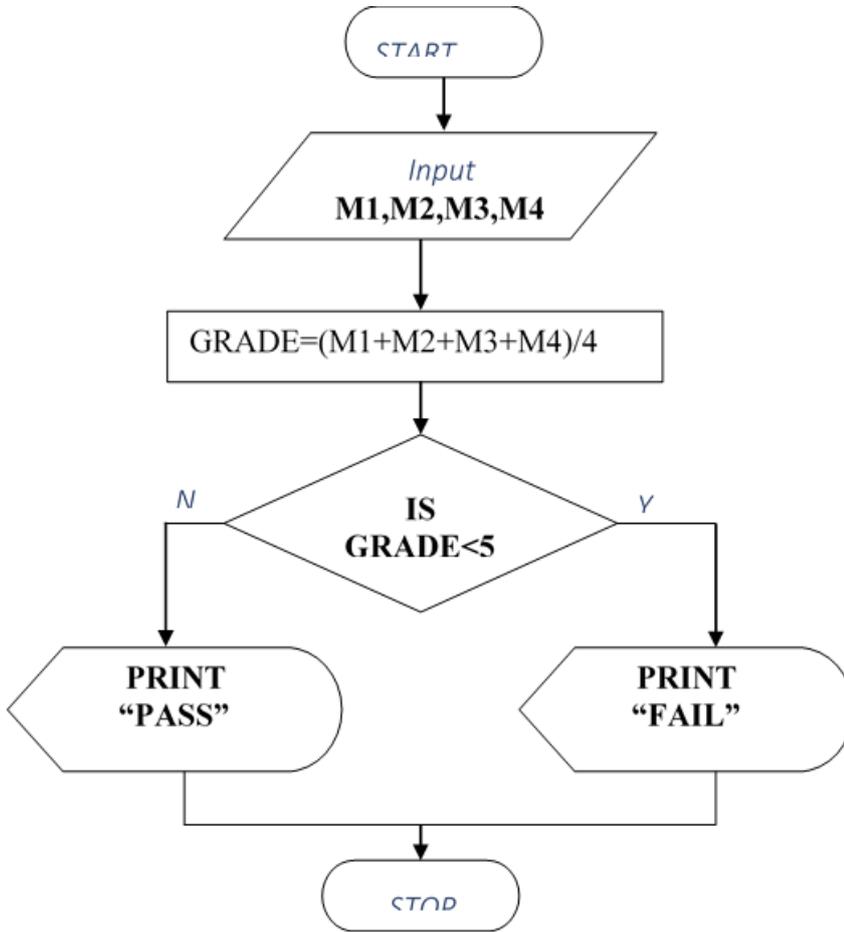
Calculate their average by summing and dividing by 4
 if average is below 50

Print "FAIL"

else

Print "PASS"

Detailed Algorithm: Step 1: Input M1, M2,M3,M4
 Step 2: GRADE = (M1+M2+M3+M4)/4
 Step 3: if (GRADE <50) then
 Print "FAIL"
 else
 Print "PASS"



Flow chart to determine the student grade

TRACE TABLE:

MARK	GRADE	STATUS
M1=80	80	
M2=30	+ 30	
M3=40	+ 40	
M4=30	+ 30 /4	
	= 45	FAIL

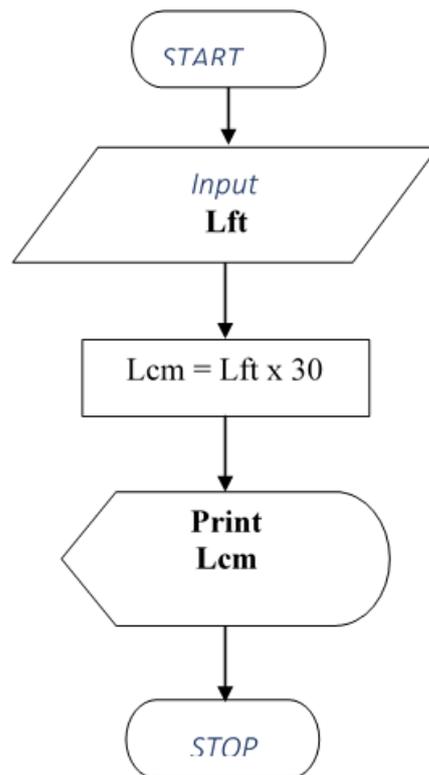


Example: Write an algorithm and draw a flowchart to convert the length in feet to centimeter.

Pseudocode: Input the length in feet (Lft)
 Calculate the length in cm (Lcm) by multiplying LFT with 30
 Print LCM

Algorithm: Step 1: Input Lft
 Step 2: Lcm = Lft x 30
 Step 3: Print Lcm

Flowchart



Example : Write an algorithm and draw a flowchart that will read the two sides of a rectangle and calculate its area.



Pseudocode: Input the width (W) and Length (L) of a rectangle

 Calculate the area (A) by multiplying L with W

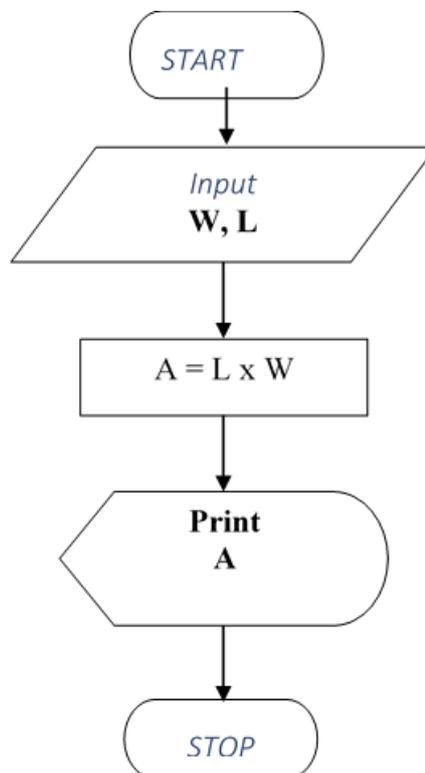
 Print A

Algorithm: Step 1: Input W,L

 Step 2: $A = L \times W$

 Step 3: Print A

Flowchart



Example : Write an algorithm and draw a flowchart that will calculate the roots of a quadratic equation $ax^2 + bx + c = 0$

Hint: $d = \sqrt{b^2 - 4ac}$, and the roots are: $x_1 = \frac{-b + d}{2a}$ and $x_2 = \frac{-b - d}{2a}$

Pseudocode: Input the coefficients (a, b, c) of the quadratic equation

 Calculate the discriminant d



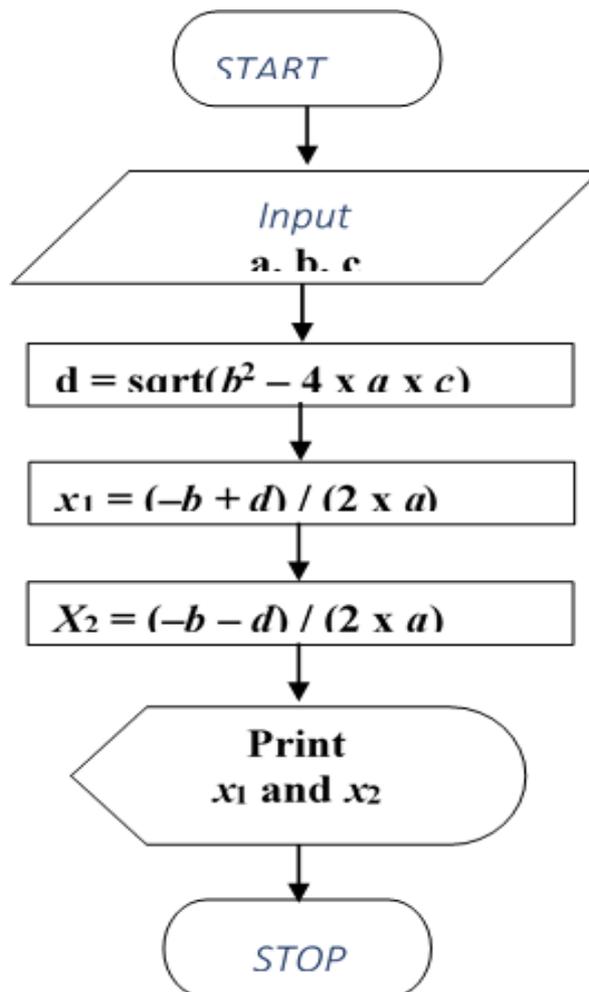
Calculate x_1

Calculate x_2

Print x_1 and x_2

- Algorithm:
- Step 1: Input a, b, c
 - Step 2: $d = \text{sqrt}(b^2 - 4 \times a \times c)$
 - Step 3: $x_1 = (-b + d) / (2 \times a)$
 - Step 4: $x_2 = (-b - d) / (2 \times a)$
 - Step 5: Print x_1 and x_2

Flowchart:





Example : Write an algorithm to

- a) read an employee name (NAME), overtime hours worked (OVERTIME), hours absent (ABSENT) and
- b) determine the bonus payment (PAYMENT).

Bonus Schedule	
OVERTIME – (2/3)*ABSENT	Bonus Paid
>40 hours	\$50
>30 but ≤ 40 hours	\$40
>20 but ≤ 30 hours	\$30
>10 but ≤ 20 hours	\$20
≤ 10 hours	\$10

Algorithm: Step 1: Input NAME,OVERTIME,ABSENT

```

Step 2:   if OVERTIME-(2/3)*ABSENT > 40
           then PAYMENT = 50
           else   if OVERTIME-(2/3)*ABSENT > 30
           then PAYMENT = 40
           else if OVERTIME-(2/3)*ABSENT > 20
           then PAYMENT = 30
           else if OVERTIME-(2/3)*ABSENT > 10
           then PAYMENT = 20
           else PAYMENT = 10
           endif
       endif
   endif
endif

```



endif

Step 3: Print “Bonus for”, NAME “is \$”, PAYMENT

Check your progress B

1 Structured English is derived from

2 Decision tables are a method of describing the in a precise manner which is easily understandable.

3 is a diagram that shows alternative actions.

2.8 Types of Problems

The problems that can be solved on computers are:

- 1- Computational Problems (involve mathematical processing)
- 2- Logical Problems (involve relational or logical processing, decision making on computer)
- 3- Repetitive Problems (involve repeating a set of mathematical or/and logical instructions)

Types of Solutions

1. Algorithmic Solutions: solutions that can be reached through a direct set of steps (series of steps) are called algorithmic solutions.
2. Heuristic Solutions: solutions that cannot be reached through a direct set of steps are called heuristic solutions. (trial and error)
3. Combination of these two kinds of solutions.

2.9 Summary:

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks. An algorithm is an efficient method that can be expressed within finite amount of time and space.



An algorithm is the best way to represent the solution of a particular problem in a very simple and efficient way. If we have an algorithm for a specific problem, then we can implement it in any programming language, meaning that the algorithm is independent from any programming languages

Structured Analysis is a development method that allows the analyst to understand the system and its activities in a logical way.

It is a systematic approach, which uses graphical tools that analysis and refine the objectives of an existing system and develop a new system specification which can be easily understandable by user.

Flow charts are easy-to-understand diagrams that show how the steps of a process fit together. American engineer Frank Gilbreth is widely believed to be the first person to document a process flow, having introduced the concept of a “Process Chart” to the American Society of Mechanical Engineers in 1921.

2.10 Keywords:

Structured Analysis: Structured Analysis is a development method that allows the analyst to understand the system and its activities in a logical way.

Flow chart: Flow charts are easy-to-understand diagrams that show how the steps of a process fit together.

Algorithm: An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks.

Answer to check your Progress

Check your Progress A

- 1 decrease
- 2 act of defining
- 3 Structured Analysis

Check your Progress B

- 1 Structured programming language



2 complex logical relationship

3 Decision tree

2.11 Self Assessed Questions:

1. What is problem solving?
2. What are the program development strategies?
3. What is flow chat? Explain in detail?
4. What is algorithm? Explain in detail?

2.12 SUGGESTED READINGS

1. Decision Making and Problem Solving Strategies by Johan Adiar
2. Strategies for Creative Problem Solving by H.Scott Fogler, Steven. LeBLANC with Benjamin Rizzo
3. Problem Solving Strategies by Arthur Engel, Springer.



Class : M.Sc. (Mathematics) Course Code : MAL 516
Subject : Programming with FORTRAN (Theory)

Chapter-3

BASIC CONCEPTS OF FORTRAN

STRUCTURE

- 3.0 Objective
- 3.1 Introduction
- 3.2 History
- 3.3 Features
- 3.4 Basic Syntax of Fortran Program
- 3.5 Basic character Set
- 3.6 Identifiers
- 3.7 Keywords
- 3.8 Data Types
 - 3.8.1 Integer Type.....
 - 3.8.2 Real Type.....
 - 3.8.3 Complex Type.....
 - 3.8.4 Logical Type.....
 - 3.8.5 Character Type.....
- 3.9 Evolution of Fortran



- 3.10 Constants
 - 3.10.1 Integer Constant
 - 3.10.2 Real Constant
 - 3.10.3 Keeping Constant Consistent
- 3.11 Named Constant & Literals
- 3.12 Variables
 - 3.12.1 Integer Constant & Variable
 - 3.12.2 Character Constant & Variable
 - 3.12.3 Default & Explicit Variable Typing
- 3.13 Declaration of Variables
- 3.14 Implicit Declaration of Variables
- 3.15 Summary
- 3.16 Keyword
- 3.17 Self Assessment Question
- 3.18 Suggested Readings

3.0 Objective:

After reading this chapter, you should be able to:

- 1) Understand why FORTRAN is invented.
- 2) Understand the syntax of the FORTRAN.
- 3) Understand the which keywords are used in FORTRAN.
- 4) Understand the Data Types



3.1 Introduction:

“FORTRAN stands for FORMula TRANslation”.FORTRAN is the first high level programming language developed by John Backus in 1957. It was basically designed to write programs for high-performance computing and is mainly suited for numeric computing scientific computing and scientific applications that involve extensive mathematical computations. The main motive behind its designing was to translate math formulas into code.

Fortran used in numerical, scientific computing. While out with the scientific community, Fortran has declined in popularity over the years, it still has a strong user base with scientific programmers, and is also used in organisations such as weather forecasters, financial trading, and in engineering simulations. Fortran programs can be highly optimised to run on high performance computers, and in general the language is suited to producing code where performance is important.

Fortran is a compiled language, or more specifically it is compiled ahead-of-time. In other words, you must perform a special step called compilation of your written code before you are able to run it on a computer. This is where Fortran differs to interpreted languages such as Python and R which run through an interpreter which executes the instructions directly, but at the cost of compute speed.

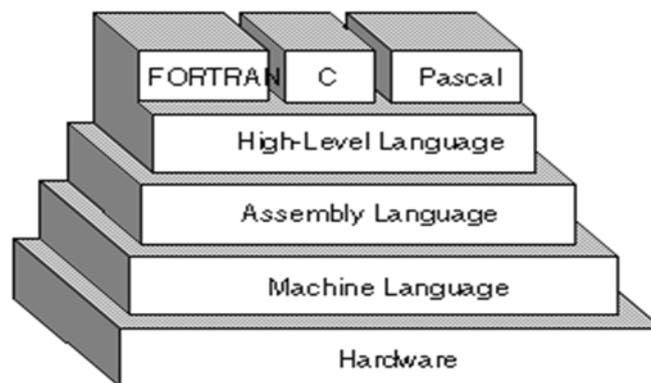


Figure 3.1 Language hierarchy

It supports –

- 1) Numerical analysis and scientific computation
- 2) Structured programming



- 3) Array programming
- 4) Modular programming
- 5) Generic programming
- 6) High performance computing on supercomputers
- 7) Object oriented programming
- 8) Concurrent programming
- 9) Reasonable degree of portability between computer systems.

The structured program consists of well structured and separated modules. But the entry and exit in a Structured program is a single-time event. It means that the program uses single-entry and single-exit elements. Therefore a structured program is well maintained, neat and clean program. The fundamental idea behind array programming is that operations apply at once to an entire set of values. It is a [software design](#) technique that separates the functionality of a [program](#) into independent, interchangeable **modules**, such that each contains everything necessary to execute only one aspect of the desired functionality. Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or [objects](#), rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.

3.2 History

Before the invention of FORTRAN, programmers used assembly/machine code to develop a program which was excessively difficult and time consuming. This led to the invention of FORTRAN which was simple to learn, machine independent, makes mathematical calculations easy, and is suitable for all type of applications. Since it was so easier to code, programmers were able to write their programs 500% faster in FORTRAN than the earlier ones.

Fortran was originally named after the contraction of Formula Translation, highlighting Fortran's origins as a language designed specifically for mathematical calculations. Fortran was developed in the early 1950s and the first ever Fortran program ran in 1954 - making Fortran fairly unusual among programming languages in that it predates the modern transistor computer - the first Fortran program ran on the IBM 704 vacuum tube computer! Fortran has outlived several nation states since its



conception, and still is in wide use today in a number of specialised scientific communities. Unfortunately Fortran is often referred to as an ‘outdated’ or ‘legacy’ programming language. I disagree with this description, as although Fortran has a long history, the language continues to be updated, new features are developed and added to the Fortran language standard, and there is still a strong community behind Fortran. The latest Fortran standard was released in 2018, bringing many new features and keeping Fortran a relevant, highly performant language for contemporary scientific computing challenges.

FORTRAN IV and FORTRAN 77 are the most common versions of FORTRAN. FORTRAN IV became a USASI standard in 1966 and FORTRAN 77 was approved by ANSI in 1978. In the early 1990s, a new ISO and ANSI standard for FORTRAN, called FORTRAN 90 was developed.

Fortran Programming Language:

Perhaps you have previously used other programming languages, such as Python, R, or MATLAB, which have developed with easy to understand syntax in mind, and with a programming style that favours more rapid development time at the expense of computational performance. Fortran will seem different to these languages in many ways, but the principles of programming remain broadly the same, and some syntax is shared or similar to elements of other programming languages.

3.3 Features:

Some of the key features of this language are as follows:

- 1) Simple language: Easy to learn and understand.
- 2) Machine independent: A program can be transported from one machine to another machine.
- 3) Expresses complex mathematical functions: It offers various natural ways to express complex mathematical functions.
- 4) Efficient execution: Only around 20% decrease in efficiency as compared to assembly or machine code.
- 5) Storage allocation: It allows programmers to control the allocation of storage.



6) Freedom in code layout: The programmers don't need to layout code in rigidly defined columns unlike assembly or machine language.

3.4 Basic Syntax of Fortran Program

A Fortran program is made of a collection of program units like a main program, modules, and external subprograms or procedures.

Each program contains one main program and may or may not contain other program units. The **syntax** of the main program is as follows –

```
program program_name
implicit none
! type declaration statements
! executable statements
end program program_name
```

Example

Let's write a program that adds two numbers and prints the result –

```
program addNumbers
! This simple program adds two numbers
implicit none
! Type declarations
real :: a, b, result
! Executable statements
a = 7.0
b = 5.0
result = a + b
print *, 'The total is ', result
```



end program add Numbers

When you compile and execute the above program, it produces the following result –

The total is 12.00000

Please note that –

- All Fortran programs start with the keyword program and end with the keyword end program, followed by the name of the program.
- The implicit none statement allows the compiler to check that all your variable types are declared properly. You must always use implicit none at the start of every program.
- Comments in Fortran are started with the exclamation mark (!), as all characters after this (except in a character string) are ignored by the compiler.
- The print * command displays data on the screen.
- Indentation of code lines is a good practice for keeping a program readable.
- Fortran allows both uppercase and lowercase letters. Fortran is case-insensitive, except for string literals.

Check your Progress A

1 Fortran stands for

2 Fortran is developed by

3 Fortran is used for

3.5 Basic Character Set

The basic character set of Fortran contains –

- 1) the letters A ... Z and a ... z
- 2) the digits 0 ... 9
- 3) the underscore () character
- 4) the special characters = : + blank - * / () [] , . \$ ' ! " % & ; < > ?



Tokens are made of characters in the basic character set. A token could be a keyword, an identifier, a constant, a string literal, or a symbol.

Program statements are made of tokens.

3.6 Identifier/ Variable Name

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

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

3.7 Keywords

Keywords are special words, reserved for the language. These reserved words cannot be used as identifiers or names.

The following table, lists the Fortran keywords –

The non-I/O keywords

Allocatable	Allocate	Assign	Assignment	block data
Call	Case	Character	Common	Complex
Contains	Continue	Cycle	Data	Deallocate
Default	Do	double precision	Else	else if
Elsewhere	end block data	end do	end function	end if
end interface	end module	end program	end select	end subroutine
end type	end where	Entry	equivalence	Exit



External	Function	go to	If	Implicit
In	Inout	Integer	Intent	Interface
Intrinsic	Kind	Len	Logical	Module
Namelist	Nullify	Only	Operator	Optional
Out	Parameter	Pause	Pointer	Private
Program	Public	Real	Recursive	Result
Return	Save	select case	Stop	Subroutine
Target	Then	Type	type()	use
Where	While			

The I/O related keywords

Backspace	Close	Endfile	Format	inquire
Open	Print	Read	Rewind	Write

3.8 Data Types

Fortran provides five intrinsic data types. The five intrinsic types are –

- 1) Integer type
- 2) Real type
- 3) Complex type
- 4) Logical type
- 5) Character type

3.8.1 Integer Type:

The integer types can hold only integer values. The following example extracts the largest value that can be held in a usual four byte integer –



```
program testingInt
implicit none
integer :: largeval
    print *, huge(largeval)
end program testingInt
```

When you compile and execute the above program it produces the following result –

```
2147483647
```

Note that the huge() function gives the largest number that can be held by the specific integer data type. You can also specify the number of bytes using the kind specifier. The following example demonstrates this –

```
program testingInt
implicit none
!two byte integer
integer(kind = 2) :: shortval
!four byte integer
integer(kind = 4) :: longval
!eight byte integer
integer(kind = 8) :: verylongval
!sixteen byte integer
integer(kind = 16) :: veryverylongval

!default integer
integer :: defval
```



```
print *, huge(shortval)
print *, huge(longval)
print *, huge(verylongval)
print *, huge(veryverylongval)
print *, huge(defval)
end program testingInt
```

When you compile and execute the above program, it produces the following result –

```
32767
2147483647
9223372036854775807
170141183460469231731687303715884105727
2147483647
```

3.8.2 Real Type:

It stores the floating point numbers, such as 2.0, 3.1415, -100.876, etc.

Traditionally there are two different real types, the default real type and double precision type.

However, Fortran 90/95 provides more control over the precision of real and integer data types through the kind specifier, which we will study in the chapter on Numbers.

The following example shows the use of real data type –

```
program division
implicit none

! Define real variables
real :: p, q, realRes
```



```
! Define integer variables
integer :: i, j, intRes

! Assigning values
p = 5.0
q = 6.0
i = 5
j = 6

! floating point division
realRes = p/q
intRes = i/j

print *, realRes
print *, intRes

end program division
```

When you compile and execute the above program it produces the following result –

```
0.8333333333
0
```

3.8.3 Complex Type:

This is used for storing complex numbers. A complex number has two parts, the real part and the imaginary part. Two consecutive numeric storage units store these two parts.

For example, the complex number (3.0, -5.0) is equal to $3.0 - 5.0i$

We will discuss Complex types in more detail, in the Numbers chapter.

3.8.4 Logical Type

There are only two logical values: `.true.` and `.false.`



Character Type

The character type stores characters and strings. The length of the string can be specified by len specifier. If no length is specified, it is 1.

For example,

```
character (len = 40) :: name
```

```
name = "Zara Ali"
```

The expression, name(1:4) would give the substring "Zara".

3.8.5 Implicit Typing

Older versions of Fortran allowed a feature called implicit typing, i.e., you do not have to declare the variables before use. If a variable is not declared, then the first letter of its name will determine its type.

Variable names starting with i, j, k, l, m, or n, are considered to be for integer variable and others are real variables. However, you must declare all the variables as it is good programming practice. For that you start your program with the statement –

```
implicit none
```

This statement turns off implicit typing.

Check your Progress B

- 1 is a name used to identify a variable, procedure, or any other user-defined item.
- 2 Older versions of Fortran allowed a feature called
- 3 number has two parts, the real part and the imaginary part.
- 4 are special words, reserved for the language.

3.9 Evolution of Fortran:

The Fortran language is a dynamic language that is constantly evolving to keep up with advances in programming practice and computing technology. A major new version appears about once per decade. The responsibility for developing new versions of the Fortran language lies with the International Organization for Standardization's (ISO) Fortran Working Group, WG5. That organization has



delegated authority to the J3 Committee of the Inter National Committee for Information Technology Standards (INCITS) to actually prepare new versions of the language. The preparation of each new version is an extended process involving first asking for suggestions for inclusion in the language, deciding which suggestions are feasible to implement, writing and circulating drafts to all interested parties throughout the world, and correcting the drafts and trying again until general agreement is reached. Eventually, a worldwide vote is held and the standard is adopted. The designers of new versions of the Fortran language must strike a delicate balance between backward compatibility with the existing base of Fortran programs and the introduction of desirable new features. Although modern structured programming features and approaches have been introduced into the language, many undesirable features from earlier versions of Fortran have been retained for backward compatibility.

The designers have developed a mechanism for identifying undesirable and obsolete features of the Fortran language that should no longer be used, and for eventually eliminating them from the language. Those parts of the language that have been superseded by new and better methods are declared to be obsolescent features.

Features that have been declared obsolescent should never be used in any new programs. As the use of these features declines in the existing Fortran code base, they will then be considered for deletion from the language. No feature will ever be deleted from a version of the language unless it was on the obsolescent list in at least one previous version, and unless the usage of the feature has dropped off to negligible levels. In this fashion, the language can evolve without threatening the existing Fortran code base.

The redundant, obsolescent, and deleted features of Fortran 2008 are described in We can get a feeling for just how much the Fortran language has evolved over the years by examining. These three figures show programs for calculating the solutions to the quadratic equation $ax^2 + bx + c = 0$ in the styles of the original FORTRAN I, of FORTRAN 77, and of Fortran 2008. It is obvious that the language has become more readable and structured over the years. Amazingly, though, Fortran 2008 compilers will still compile the FORTRAN I program with just a few minor changes!¹⁰.

Example:

A FORTRAN I program to solve for the roots of the quadratic equation $ax^2 + bx + c = 0$.



C SOLVE QUADRATIC EQUATION IN FORTRAN I

```
READ 100,A,B,C
100 FORMAT(3F12.4)
DISCR = B**2-4*A*C
IF (DISCR) 10,20,30
10 X1=(-B)/(2.*A)
X2=SQRTF(ABSF(DISCR))/(2.*A)
PRINT 110,X1,X2
110 FORMAT(5H X = ,F12.3,4H +i ,F12.3)
PRINT 120,X1,X2
120 FORMAT(5H X = ,F12.3,4H -i ,F12.3)
GOTO 40
20 X1=(-B)/(2.*A)
PRINT 130,X1
130 FORMAT(11H X1 = X2 = ,F12.3)
GOTO 40
30 X1=(-B)+SQRTF(ABSF(DISCR))/(2.*A)
X2=(-B)-SQRTF(ABSF(DISCR))/(2.*A)
PRINT 140,X1
140 FORMAT(6H X1 = ,F12.3)
PRINT 150,X2
150 FORMAT(6H X2 = ,F12.3)
40 CONTINUE
```



STOP 25252

3.10 Constants

The constants refer to the fixed values that the program cannot change during its execution. These fixed values are also called literals.

Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, a complex constant, or a string literal. There are only two logical constants : .true and .false.

The constants are treated just like regular variables, except that their values cannot be modified after their definition.

Constants can be classified into two types:

1. Integer Constants

2. Real Constants

3.10.1 Integer Constant

Integer constants are whole number without having fractional part are called as integer constant. The allowed digits in a Decimal constant are 0,1,2,3,4,5,6,7,8,9

Rule:

A integer constant must have at least on digit and must be written without decimal point. It may have either sign + or -.If either sign does not proceed the constant it is assumed to be constant.

- 1) Special symbols are not allowed.
- 2) Decimal numbers are not allowed.
- 3) Blank spaces are not allowed in between the digits.
- 4) Characters are not allowed.

Examples:

The following are the valid constant:

i. 12345



- ii. -1245
- iii. +1234

The following are the invalid constants:

- i. 11.0(Decimal number)
- ii. 12,34(comma not allowed)
- iii. \$1234(Special Symbol are not allowed)
- iv. y1234(first letter must be a digit)

The range of values of integer lies in between the $+(2^{31}-1)$ to -2^{31} that is $+217483647$ to -217483647 .

3.10.2 Real Constant

A real constant may be written in one or two forms called Fractional form or the exponent form.

Rule

A mantissa constant the fractional form at least one digit and must be written with a decimal point. It may have either the + or the sign preceding does precede then it is assumed to be positive.

Examples

The following are valid real constants:

- 1) 1.0
- 2) -0.5678
- 3) 5800000.0
- 4) -0.0000156

The following are invalid

- 1) 1 (Decimal Point missing)
- 2) -1/2 (Symbol/ illegal)
- 3) 5,867894 Comma not allowed



4) 25_685 Underscore not allowed)

In the examples done even though 58000000 and - 00000156 are valid real constants IL is more convenient to write them as 0.58×10^7 and $0.156 * 10^{-4}$ respectively.

Exponent:

The exponential notation for writing real constants provides this facility. In this notation these two numbers may be written

- 1) 0.58E7
- 2) -0.156E-4

In the above examples E7 and E4 are used to represent 10^7 and 10^4 respectively.

In this notation a real constant is represented in two parts a mantissa part (the part appearing before E) and an exponent part (the part following E) Thus 0.58 and -0.156 are the respective mantissas and 7 and -4 are real constants in the exponent form.

Rule

A real constant in the exponential form consists of a mantissa and an exponent. The mantissa must have at least one digit. It may have a sign. If a sign is omitted it is assumed to be positive. The mantissa is followed by the letter E or e and the exponent. The exponent must be an integer (without a decimal point) and must have at least one digit. A sign for the exponent is optional.

The actual number of digits in the mantissa and the exponent depends on the computer being used. The mantissa may have up to seven digits and the exponent may be between -38 and +38 in a 32-bit machine.

Examples

The following are valid real constants in the exponential form

1. (a) 152E08 b) 152.0E8 c) 152E +08 (d) 1520E7
2. -0.148E-5
3. 152 859E25
4. 0.01540E05



Observe that in (1) above four equivalent ways of writing the same constant are given.

The following are invalid:

1. 152, AE8 (letter A in mantissa incorrect)
2. 125 * E9 (* not allowed)
3. +145.8E (No digit specified for exponent)
4. -125,9E5.5 (exponent cannot be a fraction)
5. 0.158E +954 (exponent too large)
6. 125,458.25E -5 (comma not allowed in mantissa)
7. 0.158E (decimal point in exponent not allowed)

3.10.3 Keeping Constant Consistent:

It is important to always keep your physical constants consistent throughout a program. For example, do not use the value 3.14 for π at one point in a program, and 3.141593 at another point in the program. Also, you should always write your constants with at least as much precision as your computer will accept. If the real data type on your computer has seven significant digits of precision, then π should be written as 3.141593, not as 3.14!

The best way to achieve consistency and precision throughout a program is to assign a name to a constant, and then to use that name to refer to the constant throughout the program. If we assign the name PI to the constant 3.141593, then we can refer to PI by name throughout the program, and be certain that we are getting the same value everywhere. Furthermore, assigning meaningful names to constants improves the overall readability of our programs, because a programmer can tell at a glance just what the constant represents.

Named constants are created using the PARAMETER attribute of a type declaration statement. The form of a type declaration statement with a PARAMETER attribute is

type, PARAMETER :: name = value [, name2 = value2, ...] where type is the type of the constant (integer, real, logical, or character), and name is the name assigned to constant value. More than one



parameter may be declared on a single line if they are separated by commas. For example, the following statement assigns the name pi to the constant 3.141593.

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

If the named constant is of type character, then it is not necessary to declare the length of the character string. Since the named constant is being defined on the same line as its type declaration, the Fortran compiler can directly count the number of characters in the string. For example, the following statements declare a named constant error_message to be the 14-character string 'Unknown error!'.

```
CHARACTER, PARAMETER :: ERROR_MESSAGE = 'Unknown error!'
```

In languages such as C, C++, and Java, named constants are usually written in all capital letters. Many Fortran programmers are also familiar with these languages, and they have adopted the convention of writing named constants in capital letters in Fortran as well.

3.11 Named Constants & Literals

There are two types of constants –

- Literal constants
- Named constants

A literal constant have a value, but no name.

For example, following are the literal constants –

Type	Example
Integer constants	0 1 -1 300 123456789
Real constants	0.0 1.0 -1.0 123.456 7.1E+10 -52.715E-30
Complex constants	(0.0, 0.0) (-123.456E+30, 987.654E-29)
Logical constants	.true. .false.



Character constants

```
"PQR" "a" "123'abc$%#@!"
" a quote "" "
'PQR' 'a' '123"abc$%#@!'
' an apostrophe "'
```

When the above code is compiled and executed, it produces the following result –

Time = 5.00000000

Displacement = 127.374992

3.12 Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable should have a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. A name in Fortran must follow the following rules –

- 1) It cannot be longer than 31 characters.
- 2) It must be composed of alphanumeric characters (all the letters of the alphabet, and the digits 0 to 9) and underscores (_).
- 3) First character of a name must be a letter.
- 4) Names are case-insensitive.

Following are the variable types –

Sr.No	Type & Description
1	Integer It can hold only integer values.



2	Real It stores the floating point numbers.
3	Complex It is used for storing complex numbers.
4	Logical It stores logical Boolean values.
5	Character It stores characters or strings.

3.12.1 Integer Constant & Variable

The integer data type consists of integer constants and variables. This data type can only store integer values—it cannot represent numbers with fractional parts. An integer constant is any number that does not contain a decimal point. If a constant is positive, it may be written either with or without a + sign. No commas may be embedded within an integer constant. The following examples are valid integer constants:

0

-999

123456789

+17

The following examples are not valid integer constants:

1,000,000 (Embedded commas are illegal.)

-100. (If it has a decimal point, it is not an integer constant!)

An integer variable is a variable containing a value of the integer data type.

Constants and variables of the integer data type are usually stored in a single word on a computer. Since the length of a word varies from 32 bits to 64 bits on different computers, the largest integer that can be



stored in a computer also varies. The largest and smallest integers that can be stored in a particular computer can be determined from the word size by applying Equations (1-1) and (1-2).

Almost all Fortran compilers support integers with more than one length. For example, most PC compilers support 16-bit, 32-bit, and 64-bit integers. These different lengths of integers are known as different kinds of integers. Fortran has an explicit mechanism for choosing which kind of integer is used for a given value.

3.12.2 Real Constant & variable

The real data type consists of numbers stored in real or floating-point format. Unlike integers, the real data type can represent numbers with fractional components.

A real constant is a constant written with a decimal point. It may be written with or without an exponent. If the constant is positive, it may be written either with or without a + sign. No commas may be embedded within a real constant. Real constants may be written with or without an exponent. If used, the exponent consists of the letter E followed by a positive or negative integer, which corresponds to the power of 10 used when the number is written in scientific notation. If the exponent is positive, the + sign may be omitted. The mantissa of the number (the part of the number that precedes the exponent) should contain a decimal point. The following

examples are valid real constants:

10.

-999.9

+1.0E-3 (= 1.0×10^{-3} , or 0.001)

123.45E20 (= 123.45×10^{20} , or 1.2345×10^{22})

0.12E+1 (= 0.12×10^1

, or 1.2)

The following examples are not valid real constants:

1,000,000. (Embedded commas are illegal.)

111E3 (A decimal point is required in the mantissa.)



-12.0E1.5 (Decimal points are not allowed in exponents.)

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

A real value is stored in two parts: the mantissa and the exponent. The number of bits allocated to the mantissa determines the precision of the constant (that is, the number of significant digits to which the constant is known), while the number of bits allocated to the exponent determines the range of the constant (that is, the largest and the smallest values that can be represented).

Over the last 25 years, almost all computers have switched to using floating-point numbers that conform to IEEE Standard 754. Table 2-2 shows the precision and the range of typical real constants and variables on IEEE Standard 754 compliant computers.

All Fortran compilers support real numbers with more than one length. For example, PC compilers support both 32-bit real numbers and 64-bit real numbers. These different lengths of real numbers are known as different kinds. By selecting the proper kind, it is possible to increase the precision and range of a real constant or variable.

Fortran has an explicit mechanism for choosing which kind of real is used for a given value.

3.12.3 Character Constant & variable

The character data type consists of strings of alphanumeric characters. A character constant is a string of characters enclosed in single (') or double (") quotes. The minimum number of characters in a string is 0, while the maximum number of characters in a string varies from compiler to compiler.

The characters between the two single or double quotes are said to be in a character context. Any characters representable on a computer are legal in a character context, not just the 97 characters forming the Fortran character set.

The following are valid character constants:

'This is a test!'

'/b ' (a single blank)

'{^}' (These characters are legal in a character

context even though they are not a part of



the Fortran character set.)

"3.141593" (This is a character string, not a number.)

The following are not valid character constants:

This is a test! (No single or double quotes)

'This is a test!' (Mismatched quotes)

"Try this one.'" (Unbalanced single quotes)

If a character string must include an apostrophe, then that apostrophe may be represented by two consecutive single quotes. For example, the string "Man's best friend" would be written in a character constant as 'Man"s best friend'

Alternatively, the character string containing a single quote can be surrounded by double quotes. For example, the string "Man's best friend" could be written as

"Man's best friend"

Similarly, a character string containing double quotes can be surrounded by single quotes. The character string "Who cares?" could be written in a character

constant as

""Who cares?""

Character constants are most often used to print descriptive information using the

WRITE statement. WRITE (*,*) 'Result = ', k

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

3.12.4 Default and Explicit Variable Typing

When we look at a constant, it is easy to see whether it is of type integer, real, or character. If a number does not have a decimal point, it is of type integer; if it has a decimal point, it is of type real. If the constant is enclosed in single or double quotes, it is of type character. With variables, the situation is not so clear. How do we (or the compiler) know if the variable junk contains an integer, real, or character value?



There are two possible ways in which the type of a variable can be defined: default typing and explicit typing. If the type of a variable is not explicitly specified in the program, then default typing is used. By default:

Any variable names beginning with the letters i, j, k, l, m, or n are assumed to be of type INTEGER. Any variable names starting with another letter are assumed to be of type REAL.

Therefore, a variable called `incr` is assumed to be of type integer by default, while a variable called `big` is assumed to be of type real by default. This default typing convention goes all the way back to the original Fortran I in 1954. Note that no variable names are of type character by default, because this data type didn't exist in Fortran I!

The type of a variable may also be explicitly defined in the declaration section at the beginning of a program. The following Fortran statements can be used to specify the type of variables:

```
INTEGER :: var1 [, var2, var3, ...]
```

```
REAL :: var1 [, var2, var3, ...]
```

where the values inside the [] are optional. In this case, the values inside the brackets show that more than two variables may be declared on a single line if they are separated by commas.

These nonexecutable statements are called type declaration statements. They should be placed after the PROGRAM statement and before the first executable statement in the program, as shown in the example below.

```
PROGRAM example
```

```
INTEGER :: day, month, year
```

```
REAL :: second
```

```
(Executable statements follow here...)
```

There are no default names associated with the character data type, so all character variables must be explicitly typed using the CHARACTER type declaration statement.

This statement is a bit more complicated than the previous ones, since character variables may have different lengths. Its form is:



```
CHARACTER(len=<len>) :: var1 [, var2, var3, ...]
```

where <len> is the number of characters in the variables. The (len=<len>) portion

of the statement is optional. If only a number appears in the parentheses, then the character variables declared by the statement are of that length. If the parentheses are entirely absent, then the character variables declared by the statement have length 1. For

example, the type declaration statements

```
CHARACTER(len=10) :: first, last
```

```
CHARACTER :: initial
```

```
CHARACTER(15) :: id
```

define two 10-character variables called first and last, a 1-character variable called initial, and a 15-character variable called id.

Check your Progress C

1 is nothing but a name given to a storage area that our programs can manipulate.

2 Real constant may be written in one or two forms called

3..... refer to the fixed values that the program cannot change during its execution.

3.13 Declaration of Variables

Variables are declared at the beginning of a program (or subprogram) in a type declaration statement.

Syntax for variable declaration is as follows –

```
type-specifier :: variable_name
```

For example

```
integer :: total
```

```
real :: average
```

```
complex :: cx
```

```
logical :: done
```



```
character(len = 80) :: message ! a string of 80 characters
```

Later you can assign values to these variables, like,

```
total = 20000
```

```
average = 1666.67
```

```
done = .true.
```

```
message = "A big Hello from Tutorials Point"
```

```
cx = (3.0, 5.0) ! cx = 3.0 + 5.0i
```

You can also use the intrinsic function `cmplx`, to assign values to a complex variable –

```
cx = cmplx (1.0/2.0, -7.0) ! cx = 0.5 – 7.0i
```

```
cx = cmplx (x, y) ! cx = x + yi
```

Example

The following example demonstrates variable declaration, assignment and display on screen

```
program variableTesting
```

```
implicit none
```

```
! declaring variables
```

```
integer :: total
```

```
real :: average
```

```
complex :: cx
```

```
logical :: done
```

```
character(len=80) :: message ! a string of 80 characters
```

```
!assigning values
```

```
total = 20000
```



```
average = 1666.67  
done = .true.  
message = "Hello"  
cx = (3.0, 5.0) ! cx = 3.0 + 5.0i  
Print *, total  
Print *, average  
Print *, cx  
Print *, done  
Print *, message  
end program variableTesting
```

When the above code is compiled and executed, it produces the following result –

```
20000  
1666.67004  
(3.00000000, 5.00000000 )  
T  
Hello
```

3.14 Implicit Declaration of Variables

FORTRAN did not require variables to be explicitly declared. It was implicitly assumed that any variable name starting with I,J,K,L,M,N was a INTEGER and a variable name starting with any of the other letters in the alphabet REAL. Not requiring the declaration of variables before their use led to serious errors in FORTRAN programs which were difficult to detect. Thus FORTRAN designers wanted declaration of variables to be compulsory as in other languages such as C and Pascal. But this was not enforced as old FORTRAN 77 programs had to be accepted by FORTRAN 90 compilers. In other words, it was decided that FORTRAN 77 should be a subset of FORTRAN 90. Thus old syntax rules had to be honoured.



Thus if by mistake a Fortran 90 programmer forgot to declare a variable, the compiler will not defect and give it a default type based on the first letter of the name. To ensure that this mistake does not happen it is advised that all Fortran 90 programs must have the statement

IMPLICIT NONE

before declaring variables. In this book we will put IMPLICIT NONE at the beginning of every program before variables are declared

The statement

IMPLICIT NONE

informs the compiler that the variable name has a default type. Thus it is compulsory to dictate the type of each variable used in the program. If it is not done, the Fortran 90 compiler will signal a syntax error.

Check your Progress D

- 1 Fortran program consists of a mixture of statements.
- 2 consists of the nonexecutable statements at the beginning of the program.
- 3 First executable statement in this program is the statement.

3.15 SUMMARY:

FORTRAN is a first High Level Language. FORTRAN is compiled imperative programming language that is especially suited to numeric computation and scientific computing. Fortran encompasses a lineage of versions, each of which evolved to add extensions to the language while usually retaining compatibility with prior versions. Successive versions have added support for structured programming and processing of character-based data.

Fortran's design was the basis for many other programming languages. Amongst the better-known is Basic which is based on FORTRAN II with a number of syntax cleanups, notably better logical structures, and other changes to work more easily in an interactive environment.

3.16 KEYWORDS



Fortran: FORTRAN is the first high level programming language developed by John Backus in 1957. Originally developed for scientific calculations, it had very limited support for character strings and other structures needed for general purpose programming.

Identifiers: An identifier is a name used to identify a variable, procedure, or any other user-defined item.

Keywords: Keywords are special words, reserved for the language. These reserved words cannot be used as identifiers or names.

Data Types: Fortran provides five intrinsic data types, however, you can derive your own data types as well. The five intrinsic types are –(i) Integer (ii) Real (iii) Complex (iv) Logical (v) Character

Constant: An entity which does not change during the execution of the program.

Integer Constant: Integer constants are whole number they do not include fractional part.

Real Constant: Real constants may be written in one or two forms called Fractional form and Exponent form.

Variable: A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable should have a specific type, which determines the size and layout of the variable's memory.

Answer to check your Progress

Check your Progress A

- 1 Formula Translation
- 2 John Backus in 1957
- 3 scientific calculations

Check your Progress B

- 1 Identifiers
- 2 implicit typing
- 3 Complex



4 Keywords

Check your Progress C

1 Variable

2 Fractional form and Exponent form

3 Constants

Check your Progress D

1 non-executable and executable

2 Declaration Section

3 WRITE

3.17 SELF-ASSESSMENT QUESTIONS:

1. What do u mean by Fortran?
2. Explain a Syntax of a Fortran program and also describe the Data Types and Keywords used in Fortran
3. Explain the variable declaration?
4. Explain the concept of constants and variable?

3.18 SUGGESTED READINGS

1. Computer Programming in Fortran 90 and 95 by V Rajaraman
2. Introduction to Programming with Fortran by Chivers, Ian, Sleightholme, Jane
3. Fortran For Scientists and Engineers by Stephen J. Chapman, Mcgraw-Hill



Class : M.Sc. (Mathematics)	Course Code : MAL 516
Subject : Programming with FORTRAN (Theory)	

Chapter-4

INPUT/ OUTPUT STATEMENTS

STRUCTURE

- 4.0 Objective
- 4.1 List Directed Input Statement
- 4.2 Preparing Input data
- 4.3 List Directed Output Statement
- 4.4 Formatted Input/Output Statement
- 4.5 Format Specification
 - 4.5.1 Integer Output- I Descriptor
 - 4.5.2 Real Output- F Descriptor
 - 4.5.3 Real Output- E Descriptor
 - 4.5.4 Logical Output- L Descriptor
 - 4.5.5 Character Output- A Descriptor
 - 4.5.6 Changing Output- Slash Descriptor
 - 4.5.7 Formats are used During WRITE's
- 4.6 Summary
- 4.7 Keyword
- 4.8 Self Assessment Question



4.9 Suggested Readings

4.0 Objective:

After reading this chapter, you should be able to:

- 1) Understand the concept of input statement.
- 2) Understand the concept of output statement.
- 3) Understand the concept of format specification.
- 4) Understand how input/ output is formatted.

4.1 List Directed Input Statement:

List-directed input is carried out with the Fortran READ statements. The READ statement can read input values into a set of variables from the keyboard.

The READ statement has the following forms:

```
READ(*,*) var1, var2, ..., varn
```

```
READ *,var1,var2,...,varn
```

The first form starts with READ(*,*), followed by a list of variable names, separated by commas. The computer will read values from the keyboard successively and puts the value into the variables. The second form only has READ(*,*), which has a special meaning.

1. The following example reads in four values into variables Factor, N, Multiple and tolerance in this order.

```
INTEGER :: Factor, N
```

```
REAL    :: Multiple, tolerance
```

```
READ(*,*) Factor, N, Multiple, tolerance
```

2. The following example reads in a string into Title, followed by three real numbers into Height, Length and Area.

```
CHARACTER(LEN=10) :: Title
```



```
REAL          :: Height, Length, Area  
READ(*,*) Title, Height, Length, Area
```

4.2 Preparing Input Data:

Preparing input data is simple. Here are the rules:

1.If a READ statement needs some input values, start a new line that contains the input data. Make sure the type of the input value and the type of the corresponding variable are the same. The input data values must be separated by space or commas.

For the following READ

```
CHARACTER(LEN=5) :: Name  
REAL          :: height, length  
INTEGER       :: count, MaxLength  
READ(*,*) Name, height, count, length, MaxLength
```

The input data may look like the following:

```
"Smith" 100.0 25 123.579 10000
```

Note that all input data are on the same line and separated with spaces commas are not allowed. After reading in this line, the contents of the variables are

```
Name      "Smith"  
height    100.0  
count     25  
length    123.579  
MaxLength 100000
```

2. Input values can be on several lines. As long as the number of input values and the number of variables in the corresponding READ agree, the computer will search for the input values. Thus, the following input should produce the same result. Note that even blank lines are allowed in input.



```
"Smith" 100.0
```

```
25
```

```
123.579
```

```
10000
```

3. The execution of a READ statement always starts searching for input values with a new input line.

```
INTEGER :: I, J, K, L, M, N
```

```
READ(*,*) I, J
```

```
READ(*,*) K, L, M
```

```
READ(*,*) N
```

If the above READ statements are used to read the following input lines,

```
100 200
```

```
300 400 500
```

```
600
```

then I, J, K, L, M and N will receive 100, 200, 300, 400, 500 and 600, respectively.

4. Consequently, if the number of input values is larger than the number of variables in a READ statement, the extra values will be ignored. Consider the following:

```
INTEGER :: I, J, K, L, M, N
```

```
READ(*,*) I, J, K
```

```
READ(*,*) L, M, N
```

If the input lines are

```
100 200 300 400
```

```
500 600 700 800
```

```
900
```



Variables I, J and K receive 100, 200 and 300, respectively. Since the second READ starts with a new line, L, M and N receive 500, 600 and 700, respectively. 400 on the first input line is lost. The next READ will start reading with the third line, picking up 900. Hence, 800 is lost.

5. A limited type conversion is possible in a READ statement. If the input value is an integer and the corresponding variable is of REAL type, the input integer will be converted to a real number.

But, if the input value is a real number and the corresponding variable is of INTEGER type, an error will occur.

The length of the input string and the length of the corresponding CHARACTER variable do not have to be equal. If they are not equal, truncation or padding with spaces will occur as discussed in the PARAMETER attribute page.

6. Finally, a READ without a list of variables simply skips a line of input. Consider the following:

```
INTEGER :: P, Q, R, S
```

```
READ(*,*) P, Q
```

```
READ(*,*)
```

```
READ(*,*) R, S
```

If the input lines are

```
100 200 300
```

```
400 500 600
```

```
700 800 900
```

The first READ reads 100 and 200 into P and Q and 300 is lost. The second READ starts with a new input line, which is the second one. It does not read in anything. The third READ starts with the third line and reads 700 and 800 into R and S. As a result, the three input values (i.e., 400, 500 and 600) are all lost. The third value on the third line, 900, is also lost.

Check Your Progress A

1 Input is carried out with the Fortran statements.



2 The execution of a always starts searching for input values with a new input line.

3 Output is carried with the Fortran statement.

4.3 List Directed Output Statement

Listed-directed output is carried with the Fortran WRITE statement. The WRITE statement can display the results of a set of expressions and character strings. In general, WRITE displays the output on the screen. The WRITE statement has the following forms:

```
WRITE(*,*) exp1, exp2, ..., expn
```

```
WRITE *,exp1,exp2,...,expn
```

Or WRITE(*,*)

The first form starts with WRITE(*,*), followed by a list of arithmetic expressions or character strings, separated by commas. The computer will evaluate the arithmetic expressions and displays the results. Note that if a variable does not contain a value, its displayed result is unpredictable. The second form only has WRITE(*,*), which has a special meaning.

The following example displays the values of four variables on screen:

```
INTEGER :: Factor, N
```

```
REAL    :: Multiple, tolerance
```

```
WRITE(*,*) Factor, N, Multiple, tolerance
```

The following example displays the string content of Title, followed by the result of (Height + Length) * Area.

```
CHARACTER(LEN=10) :: Title
```

```
REAL              :: Height, Length, Area
```

```
WRITE(*,*) Title, (Height + Length) * Area
```

There are some useful rules:

1. Each WRITE starts with a new line.



2. Consequently, the second form in which the WRITE does not have a list of expressions just displays a blank line.

```

INTEGER          :: Target
REAL             :: Angle, Distance
CHARACTER(LEN=*), PARAMETER :: Time = "The time to hit target " &
                IS = " is "      &
UNIT = " sec."
Target = 10
Angle = 20.0
Distance = 1350.0
WRITE(*,*) 'Angle = ', Angle
WRITE(*,*) 'Distance = ', Distance
WRITE(*,*)
WRITE(*,*) Time, Target, IS, Angle * Distance, UNIT

```

This example may produce the following result:

Angle = 20.0

Distance = 1350.0

The time to hit target 10 is 27000sec.

The blank line is generated by the third WRITE.

The above example uses assumed length specifier (i.e., LEN=*) and continuation lines (i.e., symbol &).

3. If there are too many results that cannot be fit into a single line, the computer will display remaining results on the second, the third line and so on.

4.4 Formatted Input/Output Statement



Up to this point, we have not formatted our output or input specified how we want it to look.

```
READ *,var-name
```

```
PRINT *, "The output values are ", var1, var2, var3
```

The * indicates list-directed input and output, output printed according to FORTRAN's built-in rules.

Formatted output allows the programmer to control the appearance or format of the output.

Example-The user is asked to input the weight of the train in tons and the radius of the curve in meters. Each line of output contains the velocity (mph) and the force (Newtons) at that velocity.

```
PROGRAM force
```

```
    REAL weight, radius, force
```

```
    INTEGER mph
```

c Ask the user to enter the weight of the train (tons) and the

c radius of the curve (meters).

```
        PRINT *, 'Enter the weight of the train in tons.'
```

```
        READ *, weight
```

```
        PRINT *, 'Enter the radius of the curve in meters.'
```

```
        READ *, radius
```

c Converts the weight in tons to kilograms.

```
        mass = 1.016047e03 * weight
```

c Prints a header for the output.

```
        PRINT *, 'Velocity (mph)', ' ', 'Centrifugal Force (Newtons)'
```

c Computes the centrifugal force of a train on its tracks using

c the formula $force = (mass * velocity^2) / radius$. Each mph is

c converted to meters per second by the conversion equations:



c 1 mile = 1.609344e03 meters and 1 hour = 3600 seconds.

c The mph and forces are then printed.

```
DO 10 mph=50, 70
```

```
    veloc = (1.609344e03/3600.) * real(mph)
```

```
    force=(mass*veloc**2)/radius
```

```
    PRINT 200, mph, force
```

```
200    FORMAT (4x,I2, 15x, F10.1)
```

```
10    CONTINUE
```

```
    STOP
```

```
    END
```

The format of the output was specified in the statement

```
200    FORMAT (4x,I2, 15x, F10.1)
```

The specifier

4x prints 4 blanks.

I2 tells the computer to print the first variable of the PRINT list, mph, as a right-justified integer with two positions.

15x prints 15 blanks.

F10.1 prints the second variable, force, as a floating-point number with a total of 10 positions, 1 of which is a decimal position (to the right of the decimal point).

I2 and F10.1 are called format specifiers.

There is nothing to worry about the output format. The computer will use its own rules to display the results. In other words, integers and real numbers will be displayed as integers and real numbers. But, only the content of a string will be displayed. The computer will also guarantee that all significant digits will be shown so that one does not have to worry how many positions should be used for displaying a



number. The consequence is that displaying a good-looking table is a challenge. This will be discussed in FORMAT statement.

Check your Progress B

- 1 The descriptor used to describe the display format of integer data is the descriptor.
- 2 One format descriptor used to describe the display format of real data is the ... descriptor.
- 3 Character data is displayed using the format descriptor.

4.5 Format Specification

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

1. Format descriptors that describe the vertical position of a line of text.
2. Format descriptors that describe the horizontal position of data in a line.
3. Format descriptors that describe the output format of a particular value.
4. Format descriptors that control the repetition of portions of a format.

We will deal with some common examples of format descriptors in this chapter. Below Table contains a list of symbols used with format descriptors, together with their meanings.

4.5.1 Integer Output—The I Descriptor

The descriptor used to describe the display format of integer data is the I descriptor. It

has the general form

rIw or rIw.m

Symbol	Meaning
c	Column number
d	Number of digits to right of decimal place for real input or output
m	Minimum number of digits to be displayed
n	Number of spaces to skip



r	Repeat count-the number of times to use a descriptor or group of descriptors
w	Field width- the number of characters to use for the input or output.

where r, w, and m have the meanings give. This means that integers are printed out so that the last digit of the integer occupies the rightmost column of the field. If an integer is too large to fit into the field in which it is to be printed, then the field is filled with asterisks. For example,

the following statements:

```
INTEGER :: index = -12, junk = 4, number = -12345
```

```
WRITE (*,200) index, index+12, junk, number
```

```
WRITE (*,210) index, index+12, junk, number
```

```
WRITE (*,220) index, index+12, junk, number
```

```
200 FORMAT (' ', 2I5, I6, I10 )
```

```
210 FORMAT (' ', 2I5.0, I6, I10.8 )
```

```
220 FORMAT (' ', 2I5.3, I6, I5 )
```

will produce the output

```
-12 0 4 -12345
```

```
-12 4 -00012345
```

```
-012 000 4*****
```

```
----|----|----|----|----|----|
```

```
5 10 15 20 25 30
```

The special case of the zero length descriptor I0 causes the integer to be written out with a variable field width sufficient to hold the information contained in the integer. For example, the following statements:

```
INTEGER :: index = -12, junk = 4, number = -12345
```



```
WRITE (*,100) index, junk, number
```

```
100 FORMAT (I0,1X,I0,1X,I0)
```

will produce the output

```
-12 4 -12345
```

```
----|----|----|----|----|----|
```

```
5 10 15 20 25 30
```

This form of the format descriptor is especially useful for ensuring that the data will always be displayed, but it is not suitable for creating tables of data, because the columns of data will not be aligned properly.

4.5.2 Real Output—The F Descriptor

One format descriptor used to describe the display format of real data is the F descriptor. It has the form rFw.d

where r, w, and d have the meanings given in above Table. Real values are printed right justified within their fields. If necessary, the number will be rounded off before it is displayed. For example, suppose that the variable pi contains the value 3.141593. If this variable is displayed using the F7.3 format descriptor, the displayed value will be bb3.142. On the other hand, if the displayed number includes more significant digits than the internal representation of the number, extra zeros will be appended to the right of the decimal point. If the variable pi is displayed with an F10.8 format descriptor, the resulting value will be 3.14159300. If a real

number is too large to fit into the field in which it is to be printed, then the field is filled with asterisks.

For example, the following statements:

```
REAL :: a = -12.3, b = .123, c = 123.456
```

```
WRITE (*,200) a, b, c
```

```
WRITE (*,210) a, b, c
```

```
200 FORMAT (2F6.3, F8.3 )
```



210 FORMAT (3F10.2)

will produce the output

```
***** 0.123 123.456
```

```
-12.30 0.12 123.46
```

```
----|----|----|----|----|
```

```
5 10 15 20 25 30
```

4.5.3 Real Output—The E Descriptor

Real data can also be printed in exponential notation using the E descriptor. Scientific notation is a popular way for scientists and engineers to display very large or very small numbers. It consists of expressing a number as a normalized value between 1 and 10 multiplied by 10 raised to a power.

To understand the convenience of scientific notation, let's consider the following two examples from chemistry and physics. Avogadro's number is the number of atoms in a mole of a substance. It can be written out as 602,000,000,000,000,000,000 or it can be expressed in scientific notation as 6.02×10^{23} . On the other hand, the charge on an electron is 0.0000000000000000001602 coulombs. This number can be expressed in scientific notation as 1.602×10^{-19} . Scientific notation is clearly a much more convenient way to write these numbers!

The E format descriptor has the form

```
rEw.d
```

where r, w, and d have the meanings given in above Table. Unlike normal scientific notation, the real values displayed in exponential notation with the E descriptor are normalized to a range between 0.1 and 1.0. That is, they are displayed as a number between 0.1 and 1.0 multiplied by a power of 10. For example, the standard scientific notation for the number 4096.0 would be 4.096×10^3 , while the computer output

with the E descriptor would be 0.4096×10^4 . Since it is not easy to represent exponents on a line printer, the computer output would appear on the printer as

```
0.4096E+04.
```



If a real number cannot fit into the field in which it is to be printed, then the field is filled with asterisks. You should be especially careful with field sizes when working with the E format descriptor, since many items must be considered when sizing the output field. For example, suppose that we want to print out a variable in the E format with four significant digits of accuracy. Then a field width of 11 characters is required, as shown below: 1 for the sign of the mantissa, 2 for the zero and decimal point, 4 for the actual mantissa, 1 for the E, 1 for the sign of the exponent, and 2 for the exponent itself.

$\pm 0.d\text{ddd}E\pm ee$

In general, the width of an E format descriptor field must satisfy the expression

$$w \geq d + 7 \quad (5-1)$$

or the field may be filled with asterisks. The seven extra characters required are used as follows: 1 for the sign of the mantissa, 2 for the zero and decimal point, 1 for the E, 1 for the sign of the exponent, and 2 for the exponent itself.

For example, the following statements:

```
REAL :: a = 1.2346E6, b = 0.001, c = -77.7E10 , d = -77.7E10
```

```
WRITE (*,200) a, b, c, d
```

```
200 FORMAT (2E14.4, E13.6, E11.6 )
```

will produce the output

```
0.1235E+07 0.1000E-02-0.777000E+12*****
```

```
----|----|----|----|----|----|----|----|----|
```

```
5 10 15 20 25 30 35 40 45 50 55
```

4.5.4 Logical Output—The L Descriptor

The descriptor used to display logical data has the form

rLw



where r and w have the meanings given in above Table. The value of a logical variable can only be `.TRUE.` or `.FALSE.`. The output of logical variable is either a T or an F, right justified in the output field.

For example, the following statements:

```
LOGICAL :: output = .TRUE., debug = .FALSE.
```

```
WRITE (*,"(2L5)") output, debug
```

will produce the output

```
T F
```

```
----|----|----|
```

```
5 10 15
```

4.5.5 Character Output—The A Descriptor

Character data is displayed using the A format descriptor.

`rA` or `rAw`

where r and w have the meanings given in above Table . The `rA` descriptor displays character data in a field whose width is the same as the number of characters being displayed, while the `rAw` descriptor displays character data in a field of fixed width w.If the width w of the field is longer than the length of the character variable, the variable is printed out right justified in the field. If the width of the field is shorter than the length of the character variable, only the first w characters of the variable will be printed out in the field.

For example, the following statements:

```
CHARACTER(len=17) :: string = 'This is a string.'
```

```
WRITE (*,10) string
```

```
WRITE (*,11) string
```

```
WRITE (*,12) string
```

```
10 FORMAT (' ', A)
```



```
11 FORMAT (' ', A20)
```

```
12 FORMAT (' ', A6)
```

will produce the output

This is a string.

This is a string.

This i

```
----|----|----|----|----|
```

```
5 10 15 20 25
```

4.5.6 Changing Output Lines—The Slash (/) Descriptor

The slash (/) descriptor causes the current output buffer to be sent to the printer, and a new output buffer to be started. With slash descriptors, a single WRITE statement can display output values on more than one line. Several slashes can be used together to skip several lines. The slash is one of the special descriptors that does not have to be separated from other descriptors by commas. However, you may use commas if you wish. For example, suppose that we need to print out the results of an experiment in which we have measured the amplitude and phase of a signal at a certain time and depth. Assume that the integer variable index is 10 and the real variables time, depth, amplitude, and phase are 300., 330., 850.65, and 30., respectively. Then the statements

```
WRITE (*,100) index, time, depth, amplitude, phase
```

```
100 FORMAT (T20,'Results for Test Number ',I3,///, &
```

```
'Time = ',F7.0/, &
```

```
'Depth = ',F7.1,' meters',/, &
```

```
'Amplitude = ',F8.2/ &
```

```
'Phase = ',F7.1)
```

generate six separate output buffers. The first buffer puts a title on the output. The next two output buffers are empty, so two blank lines are printed. The final four output buffers contain the output for



one variable each, so the four values for time, depth, amplitude, and phase are printed on successive lines.

Notice the 1X descriptors after each slash. These descriptors place a blank in the character of each output buffer, so that each subsequent line starts in column 2.

4.5.7 Formats are Used during WRITES

Most Fortran compilers verify the syntax of FORMAT statements and character constants containing formats at compilation time, but do not otherwise process them. Character variables containing formats are not even checked at compilation time for valid syntax, since the format may be modified dynamically during program execution. In all cases, formats are saved unchanged as character strings within the compiled program.

When Results for Test Number 10

Time = 300.

Depth = 330.0 meters

Amplitude = 850.65

Phase = 30.2

the program is executed, the characters in a format

are used as a template to guide the operation of the formatted WRITE.

At execution time, the list of output variables associated with the WRITE statement is processed together with the format of the statement. The program begins at the left end of the variable list and the left end of the format, and scans from left to right, associating the first variable in the output list with the first format descriptor in the format, and so forth. The variables in the output list must be of the same type and in the same order as the format descriptors in the format, or a runtime error will occur.

Example:

A good way to illustrate the use of formatted WRITE statements is to generate and print out a table of data. It generates the square roots, squares, and cubes of all integers between 1 and 10, and presents the data in a table with appropriate headings.



A Fortran program to generate a table of square roots, squares, and cubes.

Solution:

```
PROGRAM table
```

```
!
```

```
! Purpose:
```

```
! To illustrate the use of formatted WRITE statements. This
```

```
! program generates a table containing the square roots, squares,
```

```
! and cubes of all integers between 1 and 10. The table includes
```

```
! a title and column headings.
```

```
!
```

```
! Record of revisions:
```

```
! Date Programmer Description of change
```

```
!
```

```
IMPLICIT NONE
```

```
INTEGER :: cube ! The cube of i
```

```
INTEGER :: i ! Index variable
```

```
INTEGER :: square ! The square of i
```

```
REAL :: square_root ! The square root of i
```

```
! Print the title of the table on a new page.
```

```
WRITE (*,100)
```

```
100 FORMAT (T3, 'Table of Square Roots, Squares, and Cubes'/)
```

```
! Print the column headings after skipping one line.
```

```
WRITE (*,110)
```



(concluded)

```
110 FORMAT (T4,'Number',T13,'Square Root',T29,'Square',T39,'Cube')
```

```
WRITE (*,120)
```

```
120 FORMAT (T4,'=====',T13,'=====',T29,'=====',T39,'=====')
```

! Generate the required values, and print them out.

```
DO i = 1, 10
```

```
square_root = SQRT ( REAL(i) )
```

```
square = i**2
```

```
cube = i**3
```

```
WRITE (*,130) i, square_root, square, cube
```

```
130 FORMAT (1X, T4, I4, T13, F10.6, T27, I6, T37, I6)
```

```
END DO
```

```
END PROGRAM table
```

This program uses the tab format descriptor to set up neat columns of data for the table. When this program is compiled and executed using the Intel Fortran compiler, the results are

Table of Square Roots, Squares, and Cubes

Number	Square Root	Square	Cube
1	1.000000	1	1
2	1.414214	4	8
3	1.732051	9	27
4	2.000000	16	64
5	2.236068	25	125



6	2.449490	36	216
7	2.645751	49	343
8	2.828427	64	512
9	3.000000	81	729
10	3.162278	100	1000

Example:

A capacitor is an electrical device that stores electric charge. It essentially consists of two flat plates with an insulating material (the dielectric) between them. The capacitance of a capacitor is defined as

$$C = \frac{Q}{V} \quad (5-2)$$

$$V = \frac{Q}{C} \quad (5-2)$$

where Q is the amount of charge stored in a capacitor in units of coulombs and V is the voltage between the two plates of the capacitor in volts. The units of capacitance are farads (F), with 1 farad = 1 coulomb per volt. When a charge is present on the plates of the capacitor, there is an electric field between the two plates. The energy stored in this electric field is given by the equation

$$E = \frac{1}{2} CV^2 \quad (5-3)$$

$$E = \frac{1}{2} QV$$

$$E = \frac{1}{2} \frac{Q^2}{C} \quad (5-3)$$

where E is the energy in joules. Write a program that will perform one of the following calculations:

1. For a known capacitance and voltage, calculate the charge on the plates, the number of electrons on the plates, and the energy stored in the electric field.
2. For a known charge and voltage, calculate the capacitance of the capacitor, the number of electrons on the plates, and the energy stored in the electric field.



Solution

This program must be able to ask the user which calculation he or she wishes to perform, read in the appropriate values for that calculation, and write out the results in a reasonable format. Note that this problem will require us to work with very small and very large numbers, so we will have to pay special attention to the FORMAT statements in the program.

For example, capacitors are typically rated in microfarads (μF or 10^{-6} F)

(pF or 10^{-12} F), and there are 6.241461×10^{18} electrons per coulomb of charge.

1. Define the problem.

The problem may be succinctly stated as follows:

(a) For a known capacitance and voltage, calculate the charge on a capacitor, the number of electrons stored, and the energy stored in its electric field.

(b) For a known charge and voltage, calculate the capacitance of the capacitor, the number of electrons stored, and the energy stored in its electric field.

2. Define the inputs and outputs.

There are two possible sets of input values to this program:

(a) Capacitance in farads and voltage in volts.

(b) Charge in coulombs and voltage in volts.

The outputs from the program in either mode will be the capacitance of the capacitor, the voltage across the capacitor, the charge on the plates of the capacitor, and the number of electrons on the plates of the capacitor. The output must be printed out in a reasonable and understandable format.

3. Algorithm Development.

This program can be broken down into four major steps:

- ✓ Decide which calculation is required
- ✓ Get the input data for that calculation
- ✓ Calculate the unknown quantities



Write out the capacitance, voltage, charge and number of electrons.

The first major step of the program is to decide which calculation is required. There are two types of calculations: Type 1 requires capacitance and voltage, while Type 2 requires charge and voltage. We must prompt the user for the type of input data, read his or her answer, and then read in the appropriate data. The pseudocode for these steps is:

Prompt user for the type of calculation "type"

WHILE

Read type

IF type == 1 or type == 2 EXIT

Tell user of invalid value

End of WHILE

IF type == 1 THEN

Prompt the user for the capacitance c in farads

Read capacitance c

Prompt the user for the voltage v in volts

Read voltage v

ELSE IF type == 2 THEN

Prompt the user for the charge "charge" in coulombs

Read "charge"

Prompt the user for the voltage v in volts

Read voltage v

END IF

Next, we must calculate unknown values. For Type 1 calculations, the unknown values are charge, the number of electrons, and the energy in the electric field, while for Type 2 calculations, the unknown



values are capacitance, the number of electrons, and the energy in the electric field. The pseudocode for this step is shown below:

```
IF type == 1 THEN
```

```
  charge ← c * v
```

```
ELSE
```

```
  c ← charge / v
```

```
END IF
```

```
electrons ← charge * electrons_per_coulomb
```

```
energy ← 0.5 * c * v**2
```

where `electrons_per_coulomb` is the number of electrons per coulomb of charge (6.241461×10^{18}). Finally, we must write out the results in a useful format.

```
WRITE v, c, charge, electrons, energy
```

4. Turn the program completion.

The final Fortran program is:

Program to perform capacitor calculations.

```
PROGRAM capacitor
```

```
!
```

```
! Purpose:
```

```
! To calculate the behaviour of a capacitor as follows:
```

```
! 1. If capacitance and voltage are known, calculate
```

```
! charge, number of electrons, and energy stored.
```

```
(continued)
```

```
! 2. If charge and voltage are known, calculate capa-
```



```
! citance, number of electrons, and energy stored.
!
!
IMPLICIT NONE

! Data dictionary: declare constants
REAL, PARAMETER :: ELECTRONS_PER_COULOMB = 6.241461E18

! Data dictionary: declare variable types, definitions, & units
REAL :: c ! Capacitance of the capacitor (farads).
REAL :: charge ! Charge on the capacitor (coulombs).
REAL :: electrons ! Number of electrons on the plates of the capacitor
REAL :: energy ! Energy stored in the electric field (joules)
INTEGER :: type ! Type of input data available for the calculation:
! 1: C and V
! 2: CHARGE and V
REAL :: v ! Voltage on the capacitor (volts).

! Prompt user for the type of input data available.
WRITE (*, 100)
100 FORMAT (' This program calculates information about a ' &
'capacitor.',/, ' Please specify the type of information',&
' available from the following list:',/,&
' 1 -- capacitance and voltage ',/,&
' 2 -- charge and voltage ',//,&
' Select options 1 or 2: ')
```



! Get response and validate it.

DO

READ (*,*) type

IF ((type == 1) .OR. (type == 2)) EXIT

WRITE (*,110) type

110 FORMAT (' Invalid response: ', I6, '. Please enter 1 or 2:')

END DO

! Get additional data based upon the type of calculation.

input: IF (type == 1) THEN

! Get capacitance.

WRITE (*,'Enter capacitance in farads: ')

READ (*,*) c

! Get voltage.

WRITE (*,'Enter voltage in volts: ')

READ (*,*) v

ELSE

(concluded)

! Get charge.

WRITE (*,'Enter charge in coulombs: ')

READ (*,*) charge

! Get voltage.

WRITE (*,'Enter voltage in volts: ')

READ (*,*) v



```
END IF input

! Calculate the unknown quantities.

calculate: IF ( type == 1 ) THEN

  charge = c * v ! Charge

ELSE

  c = charge / v ! Capacitance

END IF calculate

electrons = charge * ELECTRONS_PER_COULOMB ! Electrons

energy = 0.5 * c * v**2 ! Energy

! Write out answers.

WRITE (*,120) v, c, charge, electrons, energy

120 FORMAT ('For this capacitor: ',/, &
' Voltage = ', F10.2, ' V',/, &
' Capacitance = ', ES10.3, ' F',/, &
' Total charge = ', ES10.3, ' C',/, &
' Number of electrons = ', ES10.3,/, &
' Total energy = ', F10.4, ' joules' )

END PROGRAM capacitor
```

5. Test & Debug the program.

To test this program, we will calculate the answers by hand for a simple data set, and then compare the answers to the results of the program. If we use a voltage of 100 V and a capacitance of 100 μF , the resulting charge on the plates of the capacitor is 0.01 C, there are 6.241×10^{16} electrons on the capacitor, and the energy stored is 0.5 joules.

Running these values through the program using both options 1 and 2 yields the following results:



```
C:\book\fortran>capacitor
```

This program calculates information about a capacitor.

Please specify the type of information available from the following list:

1 -- capacitance and voltage

2 -- charge and voltage

Select options 1 or 2:

1

Enter capacitance in farads:

100.e-6

Enter voltage in volts:

100.

(concluded)

For this capacitor:

Voltage = 100.00 V

Capacitance = 1.000E-04 F

Total charge = 1.000E-02 C

Number of electrons = 6.241E+16

Total energy = .5000 joules

```
C:\book\fortran>capacitor
```

This program calculates information about a capacitor.

Please specify the type of information available from the following list:

1 -- capacitance and voltage

2 -- charge and voltage



Select options 1 or 2:

2

Enter charge in coulombs:

0.01

Enter voltage in volts:

100.

For this capacitor:

Voltage = 100.00 V

Capacitance = 1.000E-04 F

Total charge = 1.000E-02 C

Number of electrons = 6.241E+16

Total energy = .5000 joules

6. FORtran Documenter

A great program for documenting fortran source code is FORD.

Documenting Fortran using Doxygen

Documentation can be created right from source code using Doxygen.

The commandline program doxygen creates the documentation using configuration files. The gui program doxygen-wizard helps creating those files.

The source code needs to be documented using special comment syntax: `!>`, and `!!`.

One should always set `OPTIMIZE_FOR_FORTRAN = YES` within the configuration file.

Doxygen commands are usually ended by an empty comment line or a new doxygen command.

Note that support for Fortran is rather bad in doxygen. Even simple constructs such as `public/private` statements inside of types are not supported.

LaTeX



One can also include LaTeX code within the documentation. Doxygen's website gives detailed information.

4.6 Summary:

List-directed input is carried out with the Fortran READ statements. The READ statement can read input values into a set of variables from the keyboard. Listed-directed output is carried with the Fortran WRITE statement. The WRITE statement can display the results of a set of expressions and character strings. In general, WRITE displays the output on the screen. This is useful when data are not separated by delimiters (spaces or commas) so that the standard unformatted READ command is inapplicable. A Fortran format specification is a list of format elements describing the variable format (real number in either decimal or exponential form), the width (number of characters) of each variable, and (optionally) the number of decimal places.

4.7 Keyword:

Directed Input: List-directed input is carried out with the Fortran READ statements. The READ statement can read input values into a set of variables from the keyboard.

Directed Output: Listed-directed output is carried with the Fortran WRITE statement. The WRITE statement can display the results of a set of expressions and character strings. In general, WRITE displays the output on the screen.

Format Specification: Listed-directed output is carried with the Fortran WRITE statement. The WRITE statement can display the results of a set of expressions and character strings. In general, WRITE displays the output on the screen.

Answer to check your progress

Check your Progress A

1 READ

2 READ

3 WRITE

Check your Progress B



1 I

2 F

3 A

4.8 Self Assessed Question:

- 1) What is directed input?
- 2) What is directed output?
- 3) What is format specification?

4.9 SUGGESTED READINGS

1. Computer Programming in Fortran 90 and 95 by V Rajaraman
2. Introduction to Programming with Fortran by Chivers, Ian, Sleightholme, Jane
3. Fortran For Scientists and Engineers by Stephen J. Chapman, Mcgraw-Hill



Class : M.Sc. (Mathematics)

Course Code : MAL 516

Subject : Programming with FORTRAN (Theory)

Chapter-5

CONTROL STRUCTURE

STRUCTURE

- 5.0 Objective
- 5.1 Structure of Fortran Program
 - 5.1.1 Declaration Section
 - 5.1.2 Execution Section
 - 5.1.3 Termination Section
 - 5.1.4 Program Style
 - 5.1.5 Compiling, Linking & Execution
- 5.2 Control Flow and Types
- 5.3 If Statements
- 5.4 If else statements
- 5.5 Nested If statements
- 5.6 Select Case statement
- 5.7 Looping
- 5.8 Do Loop
- 5.9 Do While Loop
- 5.10 Repeat Until Loop



- 5.11 GOTO
- 5.12 Summary
- 5.13 Keyword
- 5.14 Self Assessment Question
- 5.15 Suggested Readings

5.0 Objective:

After reading this chapter, you should be able to:

- 1) Understand flow of control statements.
- 2) Usage of switch statements.
- 3) Understand the concept of if, if else and nested if statements.
- 4) Understand the looping construct.

5.1 Structure of a Fortran Program

Each Fortran program consists of a mixture of executable and nonexecutable statements, which must occur in a specific order.

Example

A simple Fortran program.

```
PROGRAM my_first_program
```

```
PROGRAM my_first_program
```

```
! Purpose:
```

```
! To illustrate some of the basic features of a Fortran program.
```

```
!
```

```
! Declare the variables used in this program.
```



```
INTEGER :: i, j, k ! All variables are integers
! Get two values to store in variables i and j
WRITE (*,*) 'Enter the numbers to multiply: '
READ (*,*) i, j
(concluded)
! Multiply the numbers together
k = i * j
! Write out the result.
WRITE (*,*) 'Result = ', k
! Finish up.
STOP
END PROGRAM my_first_program
```

This Fortran program, like all Fortran program units, is divided into three sections:

1. The declaration section. This section consists of a group of nonexecutable statements at the beginning of the program that define the name of the program and the number and types of variables referenced in the program.
2. The execution section. This section consists of one or more statements describing the actions to be performed by the program.
3. The termination section. This section consists of a statement or statements stopping the execution of the program and telling the compiler that the program is complete.

Note that comments may be inserted freely anywhere within, before, or after the program.

5.1.1 Declaration Section



The declaration section consists of the nonexecutable statements at the beginning of the program that define the name of the program and the number and types of variables referenced in the program.

The first statement in this section is the PROGRAM statement. It is a non executable statement that specifies the name of the program to the Fortran compiler. Fortran program names may be up to 63 characters long and contain any combination of alphabetic characters, digits, and the underscore (_) character. However, the first character in a program name must always be alphabetic. If present, the PROGRAM statement must be the first line of the program. In this example, the program has been named my_first_program. The next several lines in the program are comments that describe the purpose of the program. Next comes the INTEGER type declaration statement. This non-executable statement will be described later in this chapter. Here, it declares that three integer variables called i, j, and k will be used in this program.

5.1.2 Execution Section

The execution section consists of one or more executable statements describing the actions to be performed by the program. The first executable statement in this program is the WRITE statement, which writes out a message prompting the user to enter the two numbers to be multiplied together. The next executable statement is a READ statement, which reads in the two integers supplied by the user. The third executable statement instructs the computer to multiply the two numbers i and j together, and to store the result in variable k. The final WRITE statement prints out the result for the user to see. Comments may be embedded anywhere throughout the execution section.

5.1.3 Termination Section

The termination section consists of the STOP and END PROGRAM statements. The STOP statement is a statement that tells the computer to stop running the program. The END PROGRAM statement is a statement that tells the compiler that there are no more statements to be compiled in the program.

The STOP statement takes one of the following forms:

```
STOP
```

```
STOP 3
```

```
STOP 'Error stop'
```



If the STOP statement is used by itself, execution will stop. If the STOP statement is used with a number, that number will be printed out when the program stops, and will normally be returned to the operating system as an error code. If the STOP statement is used with a character string, that string will be printed out when the program stops.

When the STOP statement immediately precedes the END PROGRAM statement as in this example, it is optional. The compiler will automatically generate a STOP command when the END PROGRAM statement is reached. The STOP statement is therefore rarely used.

There is an alternate version of the STOP statement called ERROR STOP. This version stops the program, but it also notifies the operating system that the program failed

to execute properly. An example might be:

```
ERROR STOP 'Cannot access database'
```

This version of the STOP statement was added in Fortran 2008, and it might be useful if you need to inform an operating system script that a program failed abnormally.

5.1.4 Program Style

This example program follows a commonly used Fortran convention of capitalizing keywords such as PROGRAM, READ, and WRITE, while using lowercase for the program variables. Names are written with underscores between the words, as in my_first_ program above. It also uses capital letters for named constants such as PI (π). This is not a Fortran requirement; the program would have worked just as well if all capital letters or all lowercase letters were used. Since uppercase and lowercase letters are equivalent in Fortran, the program functions identically in either case. Throughout this book, we will follow this convention of capitalizing Fortran keywords and constants, and using lowercase for variables, procedure names, etc. Some programmers use other styles to write Fortran programs. For example, Java programmers who also work with Fortran might adopt a Java-like convention in which keywords and names are in lowercase, with capital letters at the beginning of each word (sometimes called “camel case”). Such a programmer might give this program the name myFirstProgram. This is an equally valid way to write a Fortran program. It is not necessary for you to follow any specific convention to write a Fortran program, but you should always be consistent in your programming style.



Establish a standard practice, or adopt the standard practice of the organization in which you work, and then follow it consistently in all of your programs.

5.1.5 Compiling, Linking & Loading

Before the sample program can be run, it must be compiled into object code with a Fortran compiler, and then linked with a computer's system libraries to produce an executable program. These two steps are usually done together in response to a single programmer command. The details of compiling and linking are different for every compiler and operating system. You should ask your instructor or consult the appropriate manuals to determine the proper procedure for your system.



Figure 5.1 describe how fortran code is to be executed

Fortran programs can be compiled, linked, and executed in one of two possible modes: batch and interactive. In batch mode, a program is executed without an input from or interaction with a user. This is the way most Fortran programs worked in the early days. A program would be submitted as a deck of punched cards or in a file, and it would be compiled, linked, and executed without any user interaction. All input data for the program had to be placed on cards or put in files before the job was started, and all output went to output files or to a line printer. By contrast, a program that is run in interactive mode is compiled, linked, and executed while a user is waiting at an input device such as the computer keyboard or a terminal. Since the program executes with the human present, it can ask for input data from the user as it is executing, and it can display intermediate and final results as soon as they are computed. Today, most Fortran programs are executed in interactive mode. However, some very large Fortran programs that execute for days at a time are still run in batch mode.

5.2 CONTROL FLOW AND TYPES

Types of Control Statements:

There are three types of Control Statements. They are

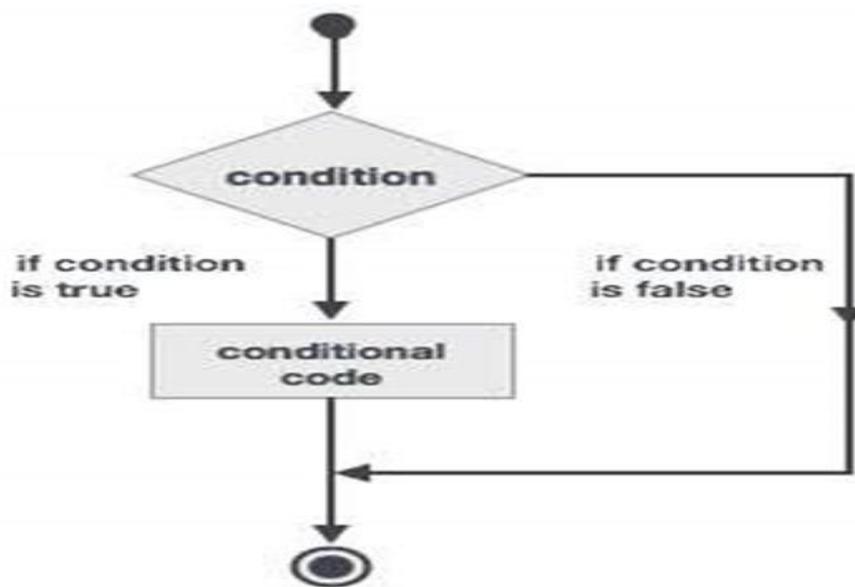


- 1) Branching
- 2) Looping
- 3) Jumping

Decision Control Statements

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed, if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages –



Flow chart for decision control statement

- 1) If statement
- 2) The If else statement
- 3) Nested if statement
- 4) Switch statement



5.3 IF STATEMENT

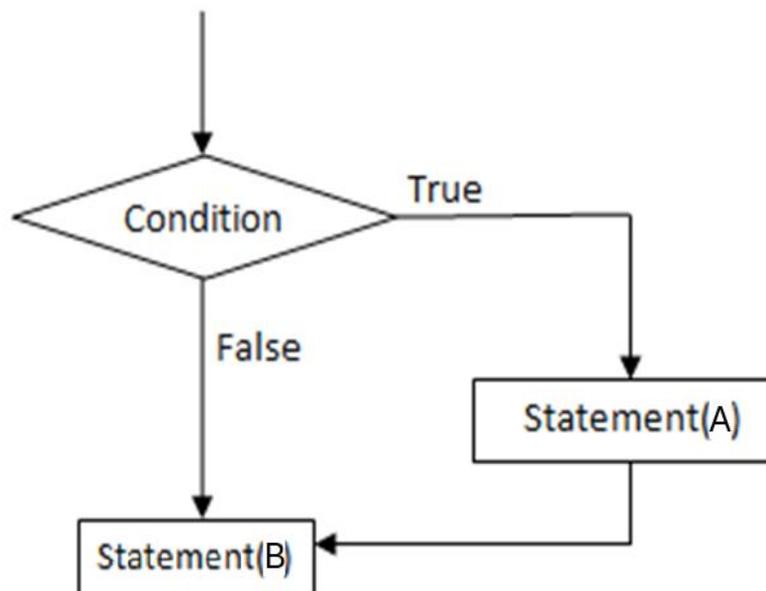
This is the simplest form of the control statement. It is very frequently used in allowing the flow of program execution and decision making. An if... then... end if statement consists of a logical expression followed by one or more statements.

Syntax:

```
IF( logical-expression ) statementA
```

```
statementB
```

The if statement evaluates the test expression inside the parenthesis. If the test statement is evaluated to true (non-zero), statements inside the body of if is executed. If the test expression is evaluated to false (0), statements inside the body of if is skipped from execution. The test expressions are written with the help of operators, which have already been described earlier in this book.



Flow chat for IF statement

5.4 If-else Statement:



An if... then statement can be followed by an optional else statement, which executes when the logical expression is false.

Syntax:

```
If(test expression)
```

```
{
```

```
Block A;
```

```
}
```

```
else
```

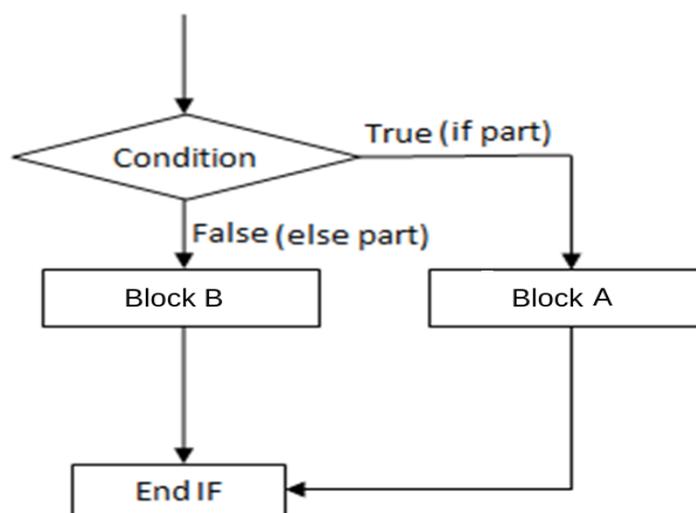
```
{
```

```
Block B;
```

```
}
```

```
End if
```

If the test expression is true, codes inside the body of the if statement is executed and codes inside the body of else statement is skipped. If the test expression is false, codes inside the body of the else statement is executed and codes inside the body of if statement is skipped. The sequence of execution for the flow of control for if-else is shown in the figure.





Flow chart for If-Else Statement

Example:

```
INTEGER :: x
if (x .ge. 0) then
    y = sqrt(x)
    print *, y, " squared = ", x
else
    print *, "x has no square root"
end if
```

5.5 ELSE-IF LADER

An if statement construct can have one or more optional else-if constructs. When the if condition fails, the immediately followed else-if is executed. When the else-if also fails, its successor else-if statement (if any) is executed, and so on.

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

Syntax:

```
IF ( expression 1 ) THEN

    IF( expression 2 ) THEN

        ELSE

    ENDIF

ELSE

    IF( expression 3 ) THEN
```



```
else
```

```
ENDIF
```

```
ENDIF
```

(A further IF construct may appear in the then-block , if block, the else block or the else-if block. This is now a nested IF structure.)

Statements in either the then-block , the else-block or the elseif-block may be labelled but jumps to such labelled statements are permitted only from within the block containing them. Entry into a block-IF construct is allowed only via the initial IF statement. Transfer out of either the then-block , the else-block or the elseif-block is permitted but only to a statement entirely outside of the whole block defined by the IF...END IF statements. A transfer within the same block-IF between any of the blocks is not permitted.

Certain types of statement, eg, END SUBROUTINE, END FUNCTION or END PROGRAM, statement are not allowed in the then-block , the else-block or the elseif-block.

Check Your Progress C

- 1 structures require that the programmer specify one or more conditions to be evaluated or tested by the program
- 2 statement evaluates the test expression inside the parenthesis.
- 3 statement is a multiway branch statement.

5.6 Select Case Statement:

Select case statements are used as substitute in case of long if statements that compare a variable to several integer values.

Select Case statement is a multiway branch statement. It provides an easy way of execution to different parts of code based on the value of the expression. It is a control statement that allows a value to change control of execution.

Syntax:



SELECT CASE (exp)

CASE list-1

statements-1

CASE list-2

statements-2

CASE list-3

statements-3

.....

CASE list-n

statements-n

CASE DEFAULT

statements-DEFAULT

END SELECT

where statements-1, statements-2, statements-3, ..., statements-n and statements-DEFAULT are sequences of executable statements, including the SELECT CASE statement itself, and selector is an expression whose result is of type INTEGER, CHARACTER or LOGICAL (i.e., no REAL type can be used for the selector). The label lists label-list-1, label-list-2, label-list-3, ..., and label-list-n are called case labels.

A label-list is a list of labels, separated by commas. Each label must be one of the following forms. In fact, three of these four are identical to an extent specifier for substrings:

value

value-1 : value-2

value-1 :

: value-2



where value, value-1 and value-2 are constants or alias defined by PARAMETER. The type of these constants must be identical to that of the selector.

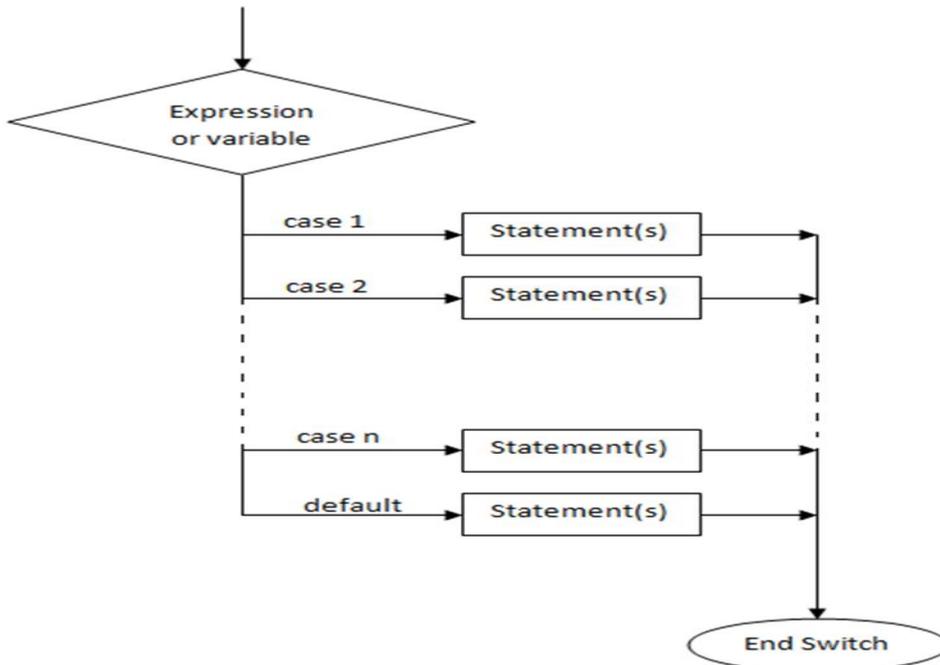
- The first form has only one value
- The second form means all values in the range of value-1 and value-2. In this form, value-1 must be less than value-2.
- The third form means all values that are greater than or equal to value-1
- The fourth form means all values that are less than or equal to value-2

The rule of executing the SELECT CASE statement goes as follows:

- The selector expression is evaluated
- If the result is in label-list-i, then the sequence of statements in statements-i are executed, followed by the statement following END SELECT
- If the result is not in any one of the label lists, there are two possibilities:
 - if CASE DEFAULT is there, then the sequence of statements in statements-DEFAULT are executed, followed by the statement following END SELECT
 - if there is no CASE DEFAULT is not there, the statement following END SELECT is executed.

There are some notes for writing good Fortran programs:

- The constants listed in label lists must be unique
- CASE DEFAULT is optional. But with a CASE DEFAULT, you are guaranteed that whatever the selector value, one of the labels will be used. I strongly suggest to add a CASE DEFAULT in every SELECT CASE statement.
- The place for CASE DEFAULT can be anywhere within a SELECT CASE statement; however, put it at the end would be more natural.



Flowchart for Select case Statement

Example:

The following uses Class to execute a selection. If Class is 1, Freshman is displayed; if Class is 2, Sophomore is displayed; if Class is 3, Junior is displayed; if Class is 4, Senior is displayed; and if Class is none of the above, Hm, I don't know is displayed. After displaying one of the above, Done is displayed.

```
INTEGER :: Class
```

```
SELECT CASE (Class)
```

```
  CASE 1
```

```
  WRITE(*,*) 'Freshman'
```

```
  CASE 2
```

```
  WRITE(*,*) 'Sophomore'
```

```
  CASE 3
```

```
  WRITE(*,*) 'Junior'
```



CASE 4

```
WRITE(*,*) 'Senior'
```

```
CASE DEFAULT
```

```
WRITE(*,*) "Hmmm, I don't know"
```

```
END SELECT
```

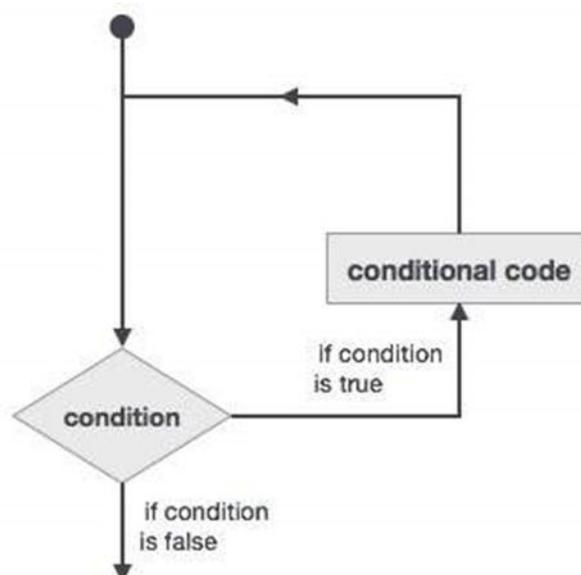
```
WRITE(*,*) 'Done'
```

5.7 Loops:

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially : The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages –





Fortran provides the following types of loop constructs to handle looping requirements. Click the following links to check their detail.

5.8 Do-loop

The loop can be named and the `exec-stmts` could contain EXIT or CYCLE statements, however, a WHILE clause cannot be used but this can be simulated with an EXIT statement if desired.

Loops can be written which cycle a fixed number of times.

Syntax:

```
DO control-var = initial, final [, step] statements
```

```
END DO
```

The number of iterations, which is evaluated before execution of the loop begins, is calculated as

$$\text{MAX}(\text{INT}((\text{final value} - \text{initial value} + \text{increment}) / \text{increment}), 0)$$

in other words the loop runs from `final value` to `initial value` in steps of `increment`. If this gives a zero or negative count then the loop is not executed.

If `increment` is absent it is assumed to be 1.

The iteration count is worked out as follows (adapted from the standard, [1]):

1. `final value` is calculated,
2. `initial value` is calculated,
3. `increment`, if present, is calculated,
4. the DO variable is assigned the value of `final value`,
5. the iteration count is established (using the formula given above).

The execution cycle is performed as follows (adapted from the standard):

1. the iteration count is tested and if it is zero then the loop terminates.
2. if it is non zero the loop is executed.



3. (conceptually) at the END DO the iteration count is decreased by one and the DO variable is incremented by `expr3` .

4. control passes to the top of the loop again and the cycle begins again.

For example,

```
DO i = 1, 100, 1
```

```
...
```

```
END DO
```

is a DO loop that will execute 10 times, it is exactly equivalent to

```
DO i = 1, 100
```

```
...
```

```
END DO
```

More complex examples may involve expressions and loops running from high to low:

```
DO i1 = 24, k*j, -1
```

```
DO i2 = k, k*j, j/k
```

```
...
```

```
END DO
```

```
END DO
```

An indexed loop could be achieved using an induction variable and EXIT statement, however, the indexed DO loop is better suited as there is less scope for error.

The DO variable cannot be assigned to within the loop.

Example:

```
INTEGER :: N, k
```

```
READ(*,*) N
```

```
WRITE(*,*) "Odd number between 1 and ", N
```



```
DO k = 1, N, 2  
WRITE(*,*) k  
END DO
```

5.9 Block Do Statement Loop

Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

Syntax:

```
label IF (logical-expression) THEN  
    statement block  
    GO TO label  
END IF
```

The loop only executes if the logical expression evaluates to `.TRUE.`. Clearly, here, the values of `a` or `b` must be modified within the loop otherwise it will never terminate.

The above loop is functionally equivalent to,

```
DO;  
....  
IF (a .NE. b) EXIT  
END DO
```

The loop only executes if the logical expression evaluates to `.TRUE.`. Clearly, here, the values of `a` or `b` must be modified within the loop otherwise it will never terminate.

`EXIT` and `CYCLE` can still be used in a `DO WHILE` loop, just as there could be multiple `EXIT` and `CYCLE` statements in a regular loop.

Example:



```
10 IF (Z .GE. 0D0) THEN
    Z = Z - SQRT(Z)
    GO TO 10
END IF
```

The statement labelled 10 is a block IF. If the value stored in the variable Z is non-negative, then the statement block in the block IF is executed. In this case, the value of $Z - \text{SQRT}(Z)$ is calculated and assigned to the variable Z. Then the unconditional GO TO statement is encountered and control is passed out of the body of block IF and back to the statement labelled with the number 10 which is the beginning of the block IF. This loop continues until the logical expression $Z .GE. 0D0$ is .FALSE. at which time the loop is finished and control passes on to the next executable statement past the END IF.

5.10 Repeat until Loop:

Another type of loop is the repeat-until loop. This loop iterates one or more times. Unlike the do-while loop, the repeat-until loop tests at the bottom of the loop so it always executes at least once. Again, FORTRAN does not have a formal repeat-until loop but it is easy to construct one using IF and GO TO statements.

Syntax:

```
label CONTINUE
    statement block
    IF (logical-expression) GO TO label
```

Example:

```
10 CONTINUE
    WRITE(*,*)'Enter a value 1-12 for the month'
    READ(*,*)MONTH
    IF (MONTH .LT. 1 .OR. MONTH .GT. 12) GO TO 10
```



The statement labelled 10 is a CONTINUE statement which is often used at the beginning or end of a loop structure. This program fragment executes the CONTINUE statement (which does nothing) and then prints out the statement Enter a value 1-12 for the month. The program then reads in a value from the standard input device and stores it in the variable MONTH. At this point MONTH is tested to see if it is between the values of 1 and 12 inclusive. If it is, then control passes to the next executable statement, but if it isn't, then the GO TO 10 statement is executed and the program returns to the CONTINUE statement. The program then goes through the WRITE and READ statements again and tests the new value of MONTH. The program will not break out of this loop until MONTH has a legal value.

5.11 GOTO Statement:

The assigned GOTO statement causes a transfer of control to the branch target statement indicated by a variable that was assigned a statement label in an ASSIGN statement. If the parenthesized list of labels is present, the variable must be one of the labels in the list.

Syntax

GOTO label

Where:

label is given to that statement where it needs to be transferred.

labels is a comma-separated list of statement labels.

Remarks

At the time of execution of the GOTO statement, assign-variable must be defined with the value of a label of a branch target statement in the same scoping unit.

The assigned GOTO statement is a construct created in the early days of Fortran, and was suitable for the tiny programs which the machines of that era were able to execute. As hardware got better and programs grew larger, the assigned GOTO statement was identified as a major contributor to a logic snarled condition known as "spaghetti code", which made a program difficult to read and debug. The assigned GOTO statement may be replaced by the CASE Construct or the IF Construct. Although the



assigned GOTO statement is obsolescent and should never be used when writing new code, it is fully supported.

Example

```
    assign 10 to i
    goto 20
20 assign 30 to i
    goto i
10 write(*,*) " assigned goto construct"
    assign 20 to i
    goto i, (10,20,30)
30 continue
```

Check your Progress B

- 1 are used to execute a block of code several number of times.
- 2repeats a statement or group of statements while a given condition is true.
- 3 statement causes a transfer of control to the branch target statement.

Examples:

The marks distribution of a course is: Attendance - 10%, Class Tests - 20%, Midterm Exam - 20% and Final Exam - 50%. The grades given are: A for 90%, B for 80% and 90%, C for 70% and 80%, D for 60% and 70%, F for 60%. If the teacher takes 40 classes and if Class Tests, Midterm and Final Exam have full marks of 100 each, write a program to calculate a student's grade.

Solution:

```
REAL MT
PRINT*, 'ENTER
AT,CT,MT,FE'
```



```
READ*,AT,CT,MT,FE
TOT=AT*10/40+CT*20/100+MT*20/100+FE*50/100
PRINT*,'TOTAL IS',TOT
IF(TOT>=90) THEN
PRINT*,'GRADE IS A'
ELSE
IF(TOT>=80)THEN
'GRADE IS B'
ELSE
IF(TOT>=70)THEN
PRINT*,'GRADE IS C'
ELSE
IF(TOT>=60)THEN
PRINT*,'GRADE IS D'
ELSE
PRINT*,'GRADE IS F'
ENDIF
ENDIF
ENDIF
ENDIF
END
```

Example

Program for calculating the cylinder area?



Solution:

```
program cylinder
```

```
! Calculate the surface area of a cylinder.
```

```
!
```

```
! Declare variables and constants.
```

```
! constants=pi
```

```
! variables=radius squared and height
```

```
implicit none ! Require all variables to be explicitly declared
```

```
integer :: ierr
```

```
character(1) :: yn
```

```
real :: radius, height, area
```

```
real, parameter :: pi = 3.141592653589793
```

```
interactive_loop: do
```

```
! Prompt the user for radius and height
```

```
! and read them.
```

```
write (*,*) 'Enter radius and height.'
```

```
read (*,*,iostat=ierr) radius,height
```



! If radius and height could not be read from input,
! then cycle through the loop.

```
if (ierr /= 0) then
  write(*,*) 'Error, invalid input.'
  cycle interactive_loop
end if
```

! Compute area. The ** means "raise to a power."

```
area = 2*pi * (radius**2 + radius*height)
```

! Write the input variables (radius, height)
! and output (area) to the screen.

```
write (*, '(1x,a7,f6.2,5x,a7,f6.2,5x,a5,f6.2)') &
  'radius=',radius,'height=',height,'area=',area
```

```
yn = ''
```

```
yn_loop: do
```

```
  write(*,*) 'Perform another calculation? y[n]'
```

```
  read(*, '(a1)') yn
```

```
  if (yn=='y' .or. yn=='Y') exit yn_loop
```



```
if (yn=='n' .or. yn=='N' .or. yn==' ') exit interactive_loop  
end do yn_loop
```

```
end do interactive_loop
```

```
end program cylinder
```

Example:

Program to get the summations?

Solution:

```
! sum.f90
```

```
! Performs summations using in a loop using EXIT statement
```

```
! Saves input information and the summation in a data file
```

```
program summation
```

```
implicit none
```

```
integer :: sum, a
```

```
print*, "This program performs summations. Enter 0 to stop."
```

```
open(unit=10, file="SumData.DAT")
```

```
sum = 0
```

```
do
```



```
print*, "Add:"  
read*, a  
if (a == 0) then  
  exit  
else  
  sum = sum + a  
end if  
write(10,*) a  
end do  
  
print*, "Summation =", sum  
write(10,*) "Summation =", sum  
close(10)  
  
end
```

When executed, the console would display the following:

This program performs summations. Enter 0 to stop.

Add:

1

Add:

2

Add:

3



Add:

0

Summation = 6

And the file SumData.DAT would contain:

1

2

3

Summation = 6

5.12 Summary:

Decision control statements are used to make a decision by checking the condition, whether true or false. The flow of control for a program based on the value of test expression inside the if statement. If, if else, nested if and goto statements are decision making statements are available. In this we also use the branching concept. Branching statement is used to repeat the loop again and again till the condition is satisfied.

5.13 Keywords:

If: It is a decision control statement. It is always executed whenever there is a non zero value for a test expression.

If-else: If else is a decision control statement where one of the part get executed either from the if block or from the else block.

Select Case: It is a type of control statement, which allows to make a decision from multiple choices.

Do Loop: This construct enables a statement, or a series of statements, to be carried out iteratively, while a given condition is true.

Do-While Loop: Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.



Answer to check your Progress

Check your Progress C

1 Decision control

2 if

3 Select case

5.14 Self Assessment Questions:

- 1) What is a control statement?
- 2) Describe decision control statements.
- 3) Explain the concept of nested if in detail. What is the purpose of Select case?
- 4) What is loop? Explain in detail?

5.15 SUGGESTED READINGS

1. Computer Programming in Fortran 90 and 95 by V Rajaraman
2. Introduction to Programming with Fortran by Chivers, Ian, Sleightholme, Jane
3. Fortran For Scientists and Engineers by Stephen J. Chapman, Mcgraw-Hill



Class : M.Sc. (Mathematics)

Course Code : MAL 516

Subject : Programming with FORTRAN (Theory)

Lesson-6

OPERATORS & EXPRESSION

STRUCTURE

6.0 Objective

6.1 Operators

6.1.1 Arithmetic Operators.....

6.1.2 Character Operator

6.1.3 Relational Operators.....

6.1.4 Logical Operators.....

6.2 Operators Precedence

6.3 Assignment Statement

6.3.1 Arithmetic Assignment.....

6.3.2 Logical Assignment.....

6.3.3 Character Assignment.....

6.3.4 Record Assignment.....

6.4 Summary

6.5 Keyword

6.6 Self Assessment Question

6.7 Suggested Readings



6.0 Objective:

After reading this chapter, you should be able to:

- 1) Understand the concept of arithmetic operator.
- 2) Understand the concept relational operators.
- 3) Understand the concept of logical operators.
- 4) Understand the concept of assignment statement.

6.1 Operators:

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. Fortran provides the following types of operators –

- 1) Arithmetic Operators
- 2) Relational Operators
- 3) Logical Operators

Let us look at all these types of operators one by one.

6.1.1 Arithmetic Operators

Following table shows all the arithmetic operators supported by Fortran. Assume variable A holds 5 and variable B holds 3 then –

Show Examples

Sno.	Operator	Description	Example
1.	+	Addition Operator, adds two operands.	A + B will give 8
2.	-	Subtraction Operator, subtracts second operand from the first.	A - B will give 2
3.	*	Multiplication Operator, multiplies both operands.	A * B will give 15



4.	/	Division Operator, divides numerator by de-numerator.	A / B will give 1
5.	**	Exponentiation Operator, raises one operand to the power of the other.	A ** B will give 125
6	(Parenthesis	

Example:

Consider an expression: $2x^3+4x^2+2x+10$

A correct syntactic translation is $2 * x **3 +4 * x **2 +2 * x +10$

It is an inefficient way of writing a expression for computer evaluation as it involves a total of 3 additional operators and 7 multiplication operators. Another technique of writing a expression require far lesser number of arithmetic operations and is given below:

$$\begin{aligned}
 2x^3+4x^2+2x+10 &= 10+2x+4x^2+2x^3 \\
 &= 10+2x(1+2x+x^2) \\
 &= 10+2x(1+x(2+x))
 \end{aligned}$$

Translated into FORTRAN is

$$10+2 * x * (1+x * (2+x))$$

The above expression requires only 3 addition and 3 multiplication. However this expression is efficient and less readable. Errors could be made in parenthesizing. Whenever there are large number of parentheses in an expression one check is to separately count the number of left and right parentheses. These count should be equal.

6.1.2Character Operator:

A character expression yields a character string value on evaluation. The simplest form of a character expression can be one of these types of characters:

- constant



- variable reference
- array element reference
- substring reference
- function reference

Construct complicated character expressions from one or more operands together with the concatenate operator and parentheses.

Concatenate Operator

The concatenate operator (`//`) is the only character operator defined in Fortran. A character expression formed from the concatenation of two character operands `x1` and `x2` is specified as

`x1 // x2`

The result of this operation is a character string with a value of `x1` extended on the right with the value of `x2`. The length of the result is the sum of the lengths of the character operands. For example,

`'HEL' // 'LO2'`

The result of the above expression is the string `HELLO2` of length six.

Character Operands:

A character operand must identify a value of type character and must be a character expression. The basic component in a character expression is the character primary. The forms of a character primary are

- character constant
- symbolic name of a character constant
- character variable reference
- character array element reference
- character substring reference
- character function reference



- character expression enclosed in parentheses

A character expression consists of one or more character primaries separated by the concatenation operator. Its forms are

- character primary
- character expression // character primary

In a character expression containing two or more concatenation operators, the primaries are combined from left to right. Thus, the character expression

'A' // 'BCD' // 'EF'

is interpreted the same as

('A' // 'BCD') // 'EF'

The value of the above character expression is ABCDEF.

Except in a character assignment statement, concatenation of an operand with an asterisk (*) as its length specification is not allowed unless the operand is the symbolic name of a constant.

6.1.3 Relational Operators:

Following table shows all the relational operators supported by Fortran. Assume variable A holds 10 and variable B holds 20, then –

Show Examples

Sno.	Operator	Equivalent	Description	Example
1.	==	.eq.	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
2.	/=	.ne.	Checks if the values of two operands are equal or not, if	(A != B) is true.



			values are not equal then condition becomes true.	
3.	>	.gt.	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
4.	<	.lt.	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
5.	>=	.ge.	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
6.	<=	.le.	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

A relational expression yields a logical value of either `.TRUE.` or `.FALSE.` on evaluation and comparison of two arithmetic expressions or two character expressions. A relational expression can appear only within a logical expression.

Table below lists the Fortran relational operators. Arithmetic and character operators are evaluated before relational operators.

Relational Operands:

The operands of a relational operator can be arithmetic or character expressions. The relational expression requires exactly two operands and is written in the following form:

`e1 relop e2`



where

e1 and e2 are arithmetic or character expressions.

relop is the relational operator.

Note: Both e1 and e2 must be the same type of expression, either arithmetic or character.

The result of a relational expression is of type logical, with a value of `.TRUE.` or `.FALSE.`. The manner in which the expression is evaluated depends on the data type of the operands.

Arithmetic Relational Expression:

In an arithmetic relational expression, e1 and e2 must each be an integer, real, double precision, complex, or double complex expression. relop must be a relational operator.

The following are examples of arithmetic relational expressions:

`(a + b) .EQ. (c + 1)`

`HOURS .LE. 40`

You can use complex type operands only when specifying either the `.EQ.` or `.NE.` relational operator.

An arithmetic relational expression has the logical value `.TRUE.` only if the values of the operands satisfy the relation specified by the operator.

Otherwise, the value is `.FALSE.`

If the two arithmetic expressions e1 and e2 differ in type, the expression is evaluated as follows:

`((e1) - (e2)) relop 0`

where the value 0 (zero) is of the same type as the expression `((e1) - (e2))` and the type conversion rules apply to the expression. Do not compare a double precision value with a complex value.

Character Relational Expression:

In a character relational expression, e1 and e2 are character expressions and relop is a relational operator.

The following is an example of a character relational expression:



NAME .EQ. 'HOMER'

A character relational expression has the logical value `.TRUE.` only if the values of the operands satisfy the relation specified by the operator. Otherwise, the value is `.FALSE.`.

The result of a character relational expression depends on the collating sequence as follows:

- If `e1` and `e2` are single characters, their relationship in the collating sequence determines the value of the operator. `e1` is less than or greater than `e2` if `e1` is before or after `e2`, respectively, in the collating sequence.
- If either `e1` or `e2` are character strings with lengths greater than 1, corresponding individual characters are compared from left to right until a relationship other than `.EQ.` can be determined.
- If the operands are of unequal length, the shorter operand is extended on the right with blanks to the length of the longer operand for the comparison.
- If no other relationship can be determined after the strings are exhausted, the strings are equal.

The collating sequence depends partially on the processor; however, equality tests `.EQ.` and `.NE.` do not depend on the processor collating sequence and can be used on any processor.

6.1.4 Logical Operators:

Logical operators in Fortran work only on logical values `.true.` and `.false.`

The following table shows all the logical operators supported by Fortran. Assume variable `A` holds `.true.` and variable `B` holds `.false.`, then –

Show Examples

Sno.	Operator	Description	Example
1.	<code>.and.</code>	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	<code>(A .and. B)</code> is false.
2.	<code>.or.</code>	Called Logical OR Operator. If any of the two operands is non-	<code>(A .or. B)</code> is true.



		zero, then condition becomes true.	
3.	.not.	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A .and. B) is true.
4.	.eqv.	Called Logical EQUIVALENT Operator. Used to check equivalence of two logical values.	(A .eqv. B) is false.
5.	.neqv	Called Logical NON-EQUIVALENT Operator. Used to check non-equivalence of two logical values.	(A .neqv. B) is true.

A logical expression specifies a logical computation that yields a logical value. The simplest form of a logical expression is one of the following:

- logical constant
- logical variable reference
- logical array element reference
- logical function reference
- relational expression

Construct complicated logical expressions from one or more logical operands together with logical operators and parentheses.

Table below defines the Fortran logical operators.

Table: Logical Operators



Logical Operator	Meaning
.NOT.	Logical negation
.AND.	Logical conjunct
.OR.	Logical disjunct
.EQV.	Logical equivalence
.NEQV.	Logical exclusive or

All logical operators require at least two operands, except the logical negation operator .NOT. , which requires only one.

A logical expression containing two or more logical operators is evaluated based on a precedence relation between the logical operators. This precedence, from highest to lowest, is

- .NOT.
- .AND.
- .OR.
- .EQV. and .NEQV.

For example, in the following expression

`W .NEQV. X .OR. Y .AND. Z`

the operators are executed in the following sequence:

1. `Y .AND. Z` is evaluated first (A represents the result).
2. `X .OR. A` is evaluated second (B represents the result).
3. `W .NEQV. B` is evaluated to produce the final result.

You can use parentheses to override the precedence of the operators.

Logical Operands

Logical operands specify values with a logical data type. The forms of a logical operands are



- logical primary
- logical factor
- logical term
- logical disjunct
- logical expression

Logical Primary:

The logical primary is the basic component of a logical expression. The forms of a logical primary are

- logical constant
- symbolic name of a logical constant
- integer or logical variable reference
- logical array element reference
- integer or logical function reference
- relational expression
- integer or logical expression in parentheses

When an integer appears as an operand to a logical operator, the other operand is promoted to type integer if necessary and the operation is performed on a bit-by-bit basis producing an integer result. Whenever an arithmetic datum appears in a logical expression, the result of that expression will be of type integer because of type promotion rules. If necessary, the result can be converted back to LOGICAL.

Do not specify two logical operators consecutively and do not use implied logical operators.

Logical Factor:

The logical factor uses the logical negation operator `.NOT.` to reverse the logical value to which it is applied. For example, applying `.NOT.` to a false relational expression makes the expression true. Therefore, if UP is true,

`.NOT. UP` is false. The logical factor has the following forms:



- logical primary
- .NOT. logical primary

Logical Term:

The logical term uses the logical conjunct operator .AND. to combine logical factors. It takes the forms

- Logical factor
- Logical term .AND. logical factor

In evaluating a logical term with two or more .AND. operators, the logical factors are combined from left to right. For example, X .AND. Y .AND. Z has the same interpretation as (X .AND. Y) .AND. Z.

Logical Disjunct:

The logical disjunct is a sequence of logical terms separated by the .OR. operator and has the following two forms:

- Logical term
- Logical disjunct .OR. logical term

In an expression containing two or more .OR. operators, the logical terms are combined from left to right in succession. For example, the expression X.OR. Y .OR. Z has the same interpretation as (X .OR. Y) .OR. Z.

Logical Expression:

At the highest level of complexity is the logical expression. A logical expression is a sequence of logical disjuncts separated by the .EQV., .NEQV., or .XOR. operators. Its forms are

- logical disjunct
- logical expression .EQV. logical disjunct
- logical expression .NEQV. logical disjunct

The logical disjuncts are combined from left to right when a logical expression contains two or more .EQV., .NEQV. operators.



A logical constant expression is a logical expression in which each primary is a logical constant, the symbolic name of a logical constant, a relational expression in which each primary is a constant, or a logical constant expression enclosed in parentheses. A logical constant expression can contain arithmetic and character constant expressions but not variables, array elements, or function references.

Interpretation of Logical Expressions:

In general, logical expressions containing two or more logical operators are executed according to the hierarchy of operators described previously, unless the order has been overridden by the use of parentheses. Table below defines the form and interpretation of the logical expressions.

Table: Truth Table

IFA=	B=	THEN .NOT.B	A.AND.B	A.OR.B	A.EQV.B	A.XOR.B A.NEQV.B
F	F	T	F	F	T	F
F	T	F	F	T	F	T
T	F	–	F	T	F	T
T	T	–	T	T	T	F

6.2 Operators & Precedence:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example, $x = 7 + 3 * 2$; here, x is assigned 13, not 20 because operator $*$ has higher precedence than $+$, so it first gets multiplied with $3*2$ and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Show Examples



Sno.	Category	Operator	Associativity
1.	Logical NOT and negative sign	.not. (-)	Left to right
2.	Exponentiation	**	Left to right
3.	Multiplicative	* /	Left to right
4.	Additive	+ -	Left to right
5.	Relational	< <= > >=	Left to right
6.	Equality	== /=	Left to right
7.	Logical AND	.and.	Left to right
8.	Logical OR	.or.	Left to right
9.	Assignment	=	Right to left

Check Your Progress A

- 1 must identify a value of type character and must be a character expression.
- 2 in Fortran work only on values .true. and .false.
- 3 The logical term uses the logical to combine logical factors

6.3 Assignment Statement:

The assignment statement assigns a value to a variable, substring, array element, record, or record field.

v = e

Parameter	Description
V	Variable, substring, array element, record, or record field
E	Expression giving the value to be assigned



6.3.1 Arithmetic Assignment:

v is of numeric type and is the name of a variable, array element, or record field.

e is an arithmetic expression, a character constant, or a logical expression. Assigning logical to numerics is nonstandard, and may not be portable; the resultant data type is, of course, the data type of v .

Execution of an arithmetic assignment statement causes the evaluation of the expression e , and conversion to the type of v (if types differ), and assignment of v with the resulting value typed according to the following table.

Type of v	Type of e
INTEGER*2, INTEGER*4, or INTEGER*8	INT(e)
REAL	REAL(e)
REAL*8	REAL*8
REAL*16 (SPARC only)	QREAL(e) (SPARC only)
DOUBLE PRECISION	DBLE(e)
COMPLEX*8	CMPLX(e)
COMPLEX*16	DCMPLX(e)
COMPLEX*32 (SPARC only)	QCMPLX(e) (SPARC only)

Example:

```
REAL A,B
```

```
DOUBLE PRECISION V
```

```
V=A*B
```

The above code is compiled exactly as if it were the following:

```
REAL A,B
```



DOUBLE PRECISION V

```
V=DBLE(A*B)
```

6.3.2 Logical Assignment:

v is the name of a variable, array element, or record field of type logical.

e is a logical expression, or an integer between -128 and 127, or a single character constant.

Execution of a logical assignment statement causes evaluation of the logical expression e and assignment of the resulting value to v. If e is a logical expression (rather than an integer between -128 and 127, or a single character constant), then e must have a value of either true or false.

Logical expressions of any size can be assigned to logical variables of any size. The section on the LOGICAL statement provides more details on the size of logical variables.

Example:

```
LOGICAL B1*1, B2*1
```

```
LOGICAL L3, L4
```

```
L4 = .TRUE.
```

```
B1 = L4
```

```
B2 = B1
```

6.3.3 Character Assignment:

The constant can be a Hollerith constant or a string of characters delimited by apostrophes (') or quotes ("). The character string cannot include the control characters Control-A, Control-B, or Control-C; that is, you cannot hold down the Control key and press the A, B, or C keys. If you need those control characters, use the char() function.

If you use quotes to delimit a character constant, then you cannot compile with the -xl option, because, in that case, a quote introduces an octal constant. The characters are transferred to the variables without any conversion of data, and may not be portable.



Character expressions which include the // operator can be assigned only to items of type CHARACTER. Here, the v is the name of a variable, substring, array element, or record field of type CHARACTER; e is a character expression.

Execution of a character assignment statement causes evaluation of the character expression and assignment of the resulting value to v. If the length of e is more than that of v, characters on the right are truncated. If the length of e is less than that of v, blank characters are padded on the right.

Example:

```
CHARACTER BELL*1, C2*2, C3*3, C5*5, C6*6
```

```
REAL Z
```

```
C2 = 'z'
```

```
C3 = 'uvwxyz'
```

```
C5 = 'vwxyz'
```

```
C5(1:2) = 'AB'
```

```
C6 = C5 // C2
```

```
BELL = CHAR(7) Control Character (^G)
```

6.3.4 Record Assignment:

v and e are each a record or record field.

The e and v must have the same structure. They have the same structure if any of the following occur:

- Both e and v are fields with the same elementary data type.
- Both e and v are records with the same number of fields such that corresponding fields are the same elementary data type.
- Both e and v are records with the same number of fields such that corresponding fields are substructures with the same structure as defined in 2, above.

The sections on the RECORD and STRUCTURE statements have more details on the structure of records.

**Check your Progress B**

- 1 assigns a value to a variable, substring, array element, record, or record field.
- 2 containing two or more logical operators are executed according to the hierarchy of operators
- 3 Higher level of complexity is the

Example:

```
STRUCTURE /PRODUCT/
```

```
    INTEGER*4 ID
```

```
    CHARACTER*16 NAME
```

```
    CHARACTER*8 MODEL
```

```
    REAL*4 COST
```

```
    REAL*4 PRICE
```

```
END STRUCTURE
```

```
RECORD /PRODUCT/ CURRENT, PRIOR, NEXT, LINE(10)
```

```
CURRENT = NEXT      Record to record
```

```
LINE(1) = CURRENT   Record to array element
```

```
WRITE ( 9 ) CURRENT Write whole record
```

```
NEXT.ID = 82        Assign a value to a field
```

Example:

Program to add two numbers?

Solution:

Implicit none

real, parameter::a=20



```
real, parameter:: b=30
integer :: c
c= a+ b
write(*, *) This result is:', c
end program addition
```

This program will give the result

C=50

Example:

Program to subtract two numbers?

Solution:

Implicit none

```
real, parameter::a=50
real, parameter:: b=30
integer :: c
c= a- b
write(*, *) This result is:', c
end program addition
```

This program will give the result

C=20

Example:

Program to multiply two numbers?

Solution:

Implicit none



```
real, parameter::a=20
real, parameter:: b=30
integer :: c
c= a* b
write(*, *) This result is:', c
end program addition
```

This program will give the result

C=600

Example:

Program to divide two numbers?

Solution:

Implicit none

```
real, parameter::a=6
real, parameter:: b=3
integer :: c
c= a/ b
write(*, *) This result is:', c
end program addition
```

This program will give the result

C=2

Example:

Program to check the two numbers are equal or not.

Solution:



Implicit none

```
real, parameter::a=6
```

```
real, parameter:: b=3
```

```
if(a==b) then
```

```
write(*,*)"Numbers are equal. "
```

```
else
```

```
write(*,*)" Numbers are not equal."
```

```
end if
```

This program will give the result

Numbers are not equal.

Example:

Program to find the largest numbers in between the two.

Solution:

Implicit none

```
real, parameter::a=6
```

```
real, parameter:: b=3
```

```
if(a .gt. b) then
```

```
write(*,*)" a is greater."
```

```
else
```

```
write(*,*)"b is greater"
```

```
end if
```

This program will give the result

a is greater.

**Example:**

Program to find the smallest numbers in between the two.

Solution:

Implicit none

real, parameter::a=6

real, parameter:: b=3

if(a .lt. b) then

write(*,*)" a is smaller."

else

write(*,*)"b is smaller"

end if

This program will give the result

b is smaller.

Example:

Some combinations of both integer and logical type:

```
COMPLEX C1 / ( 1.0, 2.0 ) /
```

```
INTEGER*2 I1, I2, I3
```

```
LOGICAL L1, L2, L3, L4, L5
```

```
REAL R1 / 1.0 /
```

```
DATA I1 / 8 /, I2 / 'W' /, I3 / 0 /
```

```
DATA L1/.TRUE./, L2/.TRUE./, L3/.TRUE./, L4/.TRUE./,
```

```
& L5/.TRUE./
```

```
L1 = L1 + 1
```



I2 = .NOT. I2

L2 = I1 .AND. I3

L3 = I1 .OR. I2

L4 = L4 + C1

L5 = L5 + R1

6.4 Summary:

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. Fortran provides the following types of operators- Arithmetic, Logical, Relational. The assignment statement assigns a value to a variable, substring, array element, record, or record field. There are different type of assignment statements are: Arithmetic assignment, logical assignment, relational assignment.

6.5 Keyword:

Arithmetic Operator: It takes numerical value as their operands and return a single numerical value.

Logical Operator: It is used to perform logical operation on the expression.

Relational Operator: It is used to compare values between two expression.

Assignment Statement: Assign the value to a variable we use assignment statement.

Answer to check your progress

Check your Progress A

1 Character Operand

2 Logical operators

3 conjunct operator .AND.

Check your Progress B

1 Assignment statement

2 Logical Expression



3 logical expression

6.6 Self Assessment Question:

- 1) Describe the assignment statement?
- 2) Explain the concept of arithmetic operators?
- 3) Explain the concept of logical operators?
- 4) Explain the concept of relational operators?

6.7 SUGGESTED READINGS

1. Computer Programming in Fortran 90 and 95 by V Rajaraman
2. Introduction to Programming with Fortran by Chivers, Ian, Sleightholme, Jane
3. Fortran For Scientists and Engineers by Stephen J. Chapman, Mcgraw-Hill



Class : M.Sc. (Mathematics)	Course Code : MAL 516
Subject : Programming with FORTRAN (Theory)	

CHAPTER-7

ARRAYS & STRING

STRUCTURE

- 7.0 Objectives
- 7.1 Introduction
- 7.2 Declaring Arrays of Fixed Size
- 7.3 Assigning values
- 7.4 Procedure
 - 7.4.1 Passing arrays to Procedure
- 7.5 Array Sections
- 7.6 Variable size Arrays
 - 7.6.1 Use of Multidimensional array
 - 7.6.2 Where Statement
- 7.7 Strings
- 7.8 String Declaration
- 7.9 String Operation
 - 7.9.1 String Concatenation
 - 7.9.2 Extracting Substring
 - 7.9.3 Trimming String



7.9.4 Left/ Right Adjustment of String

7.9.5 Searching of Substring

7.10 Summary

7.11 Keywords

7.12 Self-Assessment Questions

7.13 Suggested Readings

7.0 OBJECTIVES:

After reading this lesson, you should be able to:

- 1) Define the arrays and how they are initialized.
- 2) Explain the concept of fixed size and variable size arrays.
- 3) Define string.
- 4) Understand the concept of string operations?

7.1 Introduction

Arrays can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Numbers(1) Numbers(2) Numbers(3) Numbers(4) ...

Arrays can be one- dimensional (like vectors), two-dimensional (like matrices) and Fortran allows you to create up to 7-dimensional arrays.

7.2 Declaring Array of Fixed Size



Arrays are declared with the dimension attribute.

For example, to declare a one-dimensional array named `number`, of real numbers containing 5 elements, you write,

```
Data type, dimension(lower-limit : upper-limit) :: arrayname
```

```
real, dimension(5) :: numbers
```

The individual elements of arrays are referenced by specifying their subscripts. The first element of an array has a subscript of one. The array `numbers` contains five real variables – `numbers(1)`, `numbers(2)`, `numbers(3)`, `numbers(4)`, and `numbers(5)`.

To create a 5 x 5 two-dimensional array of integers named `matrix`, you write –

```
integer, dimension (5,5) :: matrix
```

You can also declare an array with some explicit lower bound, for example –

```
real, dimension(2:6) :: numbers
```

```
integer, dimension (-3:2,0:4) :: matrix
```

7.3 Assigning values along with Declaration

You can either assign values to individual members, like,

```
numbers(1) = 2.0
```

or, you can use a loop,

```
do i =1,5
```

```
    numbers(i) = i * 2.0
```

```
end do
```

1-D array

```
arrayname=( / val1,val2,.....,valn /)
```

One-dimensional array elements can be directly assigned values using a short hand symbol, called array constructor, like,



```
numbers = (/1.5, 3.2,4.5,0.9,7.2 /)
```

Note: Spaces are not allowed between the brackets ‘(‘and the back slash ‘/’

Example

The following example demonstrates the concepts discussed above.

```
program arrayProg
```

```
real :: numbers(5) !one dimensional integer array
```

```
integer :: matrix(3,3), i , j !two dimensional real array
```

```
!assigning some values to the array numbers
```

```
do i=1,5
```

```
    numbers(i) = i * 2.0
```

```
end do
```

```
!display the values
```

```
do i = 1, 5
```

```
    Print *, numbers(i)
```

```
end do
```

```
!assigning some values to the array matrix
```

```
do i=1,3
```

```
    do j = 1, 3
```

```
matrix(i, j) = i+j
```

```
    end do
```



```
end do

!display the values
do i=1,3
  do j = 1, 3
    Print *, matrix(i,j)
  end do
end do

!short hand assignment
numbers = (/1.5, 3.2,4.5,0.9,7.2 /)

!display the values
do i = 1, 5
  Print *, numbers(i)
end do

end program arrayProg

When the above code is compiled and executed, it produces the following result –

2.00000000
4.00000000
6.00000000
8.00000000
10.00000000
```



2

3

4

3

4

5

4

5

6

1.50000000

3.20000005

4.50000000

0.899999976

7.19999981

Some Array Related Terms

The following table gives some array related terms –

Term	Meaning
Rank	It is the number of dimensions an array has. For example, for the array named matrix, rank is 2, and for the array named numbers, rank is 1.
Extent	It is the number of elements along a dimension. For example, the



	array numbers has extent 5 and the array named matrix has extent 3 in both dimensions.
Shape	The shape of an array is a one-dimensional integer array, containing the number of elements (the extent) in each dimension. For example, for the array matrix, shape is (3, 3) and the array numbers it is (5).
Size	It is the number of elements an array contains. For the array matrix, it is 9, and for the array numbers, it is 5.

7.4 Procedure

A procedure is a group of statements that perform a well-defined task and can be invoked from your program. Information (or data) is passed to the calling program, to the procedure as arguments.

7.4.1 Passing Arrays to Procedures

You can pass an array to a procedure as an argument. The following example demonstrates the concept –

```

program arrayToProcedure
implicit none

integer, dimension (5) :: myArray
integer :: i

call fillArray (myArray)
call printArray(myArray)

end program arrayToProcedure

```



```
subroutine fillArray (a)
implicit none
  integer, dimension (5), intent (out) :: a
  ! local variables
integer :: i
  do i = 1, 5
    a(i) = i
  end do
end subroutine fillArray
subroutine printArray(a)
  integer, dimension (5) :: a
integer::i
  do i = 1, 5
    Print *, a(i)
  end do
end subroutine printArray
```

When the above code is compiled and executed, it produces the following result –

```
1
2
3
4
5
```



In the above example, the subroutine fillArray and printArray can only be called with arrays with dimension 5. However, to write subroutines that can be used for arrays of any size, you can rewrite it using the following technique –

```
program arrayToProcedure
```

```
implicit none
```

```
integer, dimension (10) :: myArray
```

```
integer :: i
```

```
interface
```

```
subroutine fillArray (a)
```

```
integer, dimension(:), intent (out) :: a
```

```
integer :: i
```

```
end subroutine fillArray
```

```
subroutine printArray (a)
```

```
integer, dimension(:) :: a
```

```
integer :: i
```

```
end subroutine printArray
```

```
end interface
```

```
call fillArray (myArray)
```

```
call printArray(myArray)
```



```
end program arrayToProcedure
```

```
subroutine fillArray (a)
```

```
implicit none
```

```
integer,dimension (:), intent (out) :: a
```

```
! local variables
```

```
integer :: i, arraySize
```

```
arraySize = size(a)
```

```
do i = 1, arraySize
```

```
    a(i) = i
```

```
end do
```

```
end subroutine fillArray
```

```
subroutine printArray(a)
```

```
implicit none
```

```
integer,dimension (:) :: a
```

```
integer::i, arraySize
```

```
arraySize = size(a)
```

```
do i = 1, arraySize
```

```
    Print *, a(i)
```



```
end do
```

```
end subroutine printArray
```

Please note that the program is using the size function to get the size of the array.

When the above code is compiled and executed, it produces the following result –

```
1
2
3
4
5
6
7
8
9
10
```

Check your Progress A

1 can store a fixed-size sequential collection of elements of the same type.

2 Arrays are declared with the attribute.

3 To access an array section, you need to provide the of the section

7.5 Array Sections:

So far we have referred to the whole array, Fortran provides an easy way to refer several elements, or a section of an array, using a single statement.

To access an array section, you need to provide the lower and the upper bound of the section, as well as a stride (increment), for all the dimensions. This notation is called a subscript triplet:



array ([lower]:[upper][:stride], ...)

When no lower and upper bounds are mentioned, it defaults to the extents you declared, and stride value defaults to 1.

The following example demonstrates the concept –

program arraySubsection

```
real, dimension(10) :: a, b
```

```
integer::i, asize, bsize
```

```
a(1:7) = 5.0 ! a(1) to a(7) assigned 5.0
```

```
a(8:) = 0.0 ! rest are 0.0
```

```
b(2:10:2) = 3.9
```

```
b(1:9:2) = 2.5
```

```
!display
```

```
asize = size(a)
```

```
bsize = size(b)
```

```
do i = 1, asize
```

```
Print *, a(i)
```

```
end do
```

```
do i = 1, bsize
```



```
Print *, b(i)
end do

end program arraySubsection
```

When the above code is compiled and executed, it produces the following result –

```
5.00000000
5.00000000
5.00000000
5.00000000
5.00000000
5.00000000
5.00000000
0.00000000E+00
0.00000000E+00
0.00000000E+00
2.50000000
3.90000010
2.50000000
3.90000010
2.50000000
3.90000010
2.50000000
3.90000010
```



2.50000000

3.90000010

7.6 Variable Size Arrays:

A variable size array is an array, the size of which is not known at compile time, but will be known at execution time.

Variable size arrays are declared with the attribute `allocatable`.

For example:

```
real, dimension (:,:), allocatable :: darray
```

The rank of the array, i.e., the dimensions has to be mentioned however, to allocate memory to such an array, you use the `allocate` function.

```
allocate ( darray(s1,s2) )
```

After the array is used, in the program, the memory created should be freed using the `deallocate` function

```
deallocate (darray)
```

Example

The following example demonstrates the concepts discussed above.

```
program dynamic_array
```

```
implicit none
```

```
!rank is 2, but size not known
```

```
    real, dimension (:,:), allocatable :: darray
```

```
integer :: s1, s2
```

```
integer :: i, j
```

```
    print*, "Enter the size of the array:"
```

```
    read*, s1, s2
```



```
! allocate memory
allocate ( darray(s1,s2) )
do i = 1, s1
  do j = 1, s2
darray(i,j) = i*j
    print*, "darray(",i,",",j,") = ", darray(i,j)
  end do
end do
deallocate (darray)
end program dynamic_array
```

When the above code is compiled and executed, it produces the following result –

Enter the size of the array: 3,4

darray(1 , 1) = 1.00000000

darray(1 , 2) = 2.00000000

darray(1 , 3) = 3.00000000

darray(1 , 4) = 4.00000000

darray(2 , 1) = 2.00000000

darray(2 , 2) = 4.00000000

darray(2 , 3) = 6.00000000

darray(2 , 4) = 8.00000000

darray(3 , 1) = 3.00000000

darray(3 , 2) = 6.00000000

darray(3 , 3) = 9.00000000



```
darray( 3 , 4 ) = 12.0000000
```

7.6.1 Use of Multidimensional array:

FORTRAN 90 arrays may have up to seven dimensions. The elements in each dimension are of the same type so it is not possible to have an array with INTEGER values in the first dimension, REAL values in the second dimension, CHARACTER values in the third dimension, etc.

Multidimensional arrays are declared in the same way as one-dimensional arrays. The array bounds or lengths of each dimension are separated by commas.

```
INTEGER BATTLE(31,12,1939:1945)
```

BATTLE is a three-dimensional array. The first dimension has length 31, the second has length 12 and the third length 7. The subscript for the first dimension range from 1 to 31, the subscripts for the second dimension range from 1 to 12 and the subscript for the third dimension range from 1939 to 1945.

The syntax of data statement is –

```
data variable / list / ...
```

Example

The following example demonstrates the concept –

```
program dataStatement
```

```
implicit none
```

```
integer :: a(5), b(3,3), c(10),i, j
```

```
data a /7,8,9,10,11/
```

```
data b(1,:) /1,1,1/
```

```
data b(2,:) /2,2,2/
```

```
data b(3,:) /3,3,3/
```

```
data (c(i),i = 1,10,2) /4,5,6,7,8/
```

```
data (c(i),i = 2,10,2) /5*2/
```



```
Print *, 'The A array:'  
  
do j = 1, 5  
    print*, a(j)  
end do  
  
Print *, 'The B array:'  
  
do i = lbound(b,1), ubound(b,1)  
write(*,*) (b(i,j), j = lbound(b,2), ubound(b,2))  
end do  
  
Print *, 'The C array:'  
  
do j = 1, 10  
    print*, c(j)  
end do  
  
end program dataStatement
```

When the above code is compiled and executed, it produces the following result –

The A array:

7
8
9
10
11

The B array:

1 1 1
2 2 2



3 3 3

The C array:

4

2

5

2

6

2

7

2

8

2

7.6.2 Use of Where Statement:

The where statement allows you to use some elements of an array in an expression, depending on the outcome of some logical condition. It allows the execution of the expression, on an element, if the given condition is true.

Example

The following example demonstrates the concept –

program whereStatement

implicit none

integer :: a(3,5), i , j

do i = 1,3

do j = 1, 5

a(i,j) = j-i



```
    end do

end do

Print *, 'The A array:'

do i = lbound(a,1), ubound(a,1)
write(*,*) (a(i,j), j = lbound(a,2), ubound(a,2))
end do

where( a<0 )
    a = 1
elsewhere
    a = 5
end where

Print *, 'The A array:'

do i = lbound(a,1), ubound(a,1)
write(*,*) (a(i,j), j = lbound(a,2), ubound(a,2))
end do

end program whereStatement
```

When the above code is compiled and executed, it produces the following result –

The A array:

0	1	2	3	4
-1	0	1	2	3
-2	-1	0	1	2

The A array:

5	5	5	5	5
---	---	---	---	---



1 5 5 5 5

1 1 5 5 5

7.7 String:

A string is a sequence of characters.

A character string may be only one character in length, or it could even be of many characters in length. In Fortran, character constants are given between a pair of double or single quotes.

The intrinsic data type character stores characters and strings. The length of the string can be specified by len specifier. If no length is specified, it is 1. You can refer individual characters within a string referring by position; the left most character is at position 1.

7.8 String Declaration:

Declaring a string is same as other variables –

Data-type :: variable_name

For example,

Character(n) :: val1,val2,.....,valn

Character(len=n) :: val1,val2,.....,valn

Character(len = 20) :: firstname, surname

you can assign a value like,

character (len = 40) :: name

name = “Zara Ali”

The following example demonstrates declaration and use of character data type –

program hello

implicit none

character(len = 15) :: surname, firstname



```
character(len = 6) :: title
character(len = 25)::greetings

title = 'Mr.'
firstname = 'Rowan'
surname = 'Atkinson'
greetings = 'A big hello from Mr. Beans'

print *, 'Here is', title, firstname, surname
print *, greetings

end program hello
```

When you compile and execute the above program it produces the following result –

```
Here isMr. Rowan Atkinson
```

```
A big hello from Mr. Bean
```

Check Your Progress B

- 1..... array size is not known at compile time.
- 2 The Fortran language can treat characters as
- 3 Joining of two strings one after the another is called

7.9 String Operations:

There are various operations applied on string are:

- 1) String Concatenation
- 2) Extracting Substring



- 3) Trimming Strings
- 4) Left/ Right Adjustments of String
- 5) Searching of Substring

7.9.1 String Concatenation:

The concatenation operator //, concatenates strings.

The following example demonstrates this –

```
program hello
implicit none
character(len = 15) :: surname, firstname
character(len = 6) :: title
character(len = 40):: name
character(len = 25)::greetings

  title = 'Mr.'
  firstname = 'Rowan'
  surname = 'Atkinson'

  name = title//firstname//surname
  greetings = 'A big hello from Mr. Beans'

  print *, 'Here is', name
  print *, greetings

end program hello
```



When you compile and execute the above program it produces the following result –

Here is Mr. Rowan Atkinson

A big hello from Mr. Bean

7.9.2 Extracting Substrings:

In Fortran, you can extract a substring from a string by indexing the string, giving the start and the end index of the substring in a pair of brackets. This is called extent specifier.

The following example shows how to extract the substring ‘world’ from the string ‘hello world’ –

```
program subString
character(len = 11)::hello
  hello = "Hello World"
  print*, hello(7:11)
end program subString
```

When you compile and execute the above program it produces the following result –

World

Example

The following example uses the `date_and_time` function to give the date and time string. We use extent specifiers to extract the year, date, month, hour, minutes and second information separately.

```
program datetime
implicit none
character(len = 8) :: dateinfo ! ccyymmdd
character(len = 4) :: year, month*2, day*2
character(len = 10) :: timeinfo ! hhmmss.sss
character(len = 2) :: hour, minute, second*6
call date_and_time(dateinfo, timeinfo)
```



```
! let's break dateinfo into year, month and day.

! dateinfo has a form of ccyyymmdd, where cc = century, yy = year

! mm = month and dd = day
year =dateinfo(1:4)

month = dateinfo(5:6)

day = dateinfo(7:8)

print*, 'Date String:', dateinfo

print*, 'Year:', year

print *, 'Month:', month

print *, 'Day:', day

! let's break timeinfo into hour, minute and second.

! timeinfo has a form of hhmmss.sss, where h = hour, m = minute

! and s = second

hour = timeinfo(1:2)

minute = timeinfo(3:4)

second = timeinfo(5:10)

print*, 'Time String:', timeinfo

print*, 'Hour:', hour

print*, 'Minute:', minute

print*, 'Second:', second

end program datetime
```

When you compile and execute the above program, it gives the detailed date and time information –

```
Date String: 20140803
```



Year: 2014

Month: 08

Day: 03

Time String: 075835.466

Hour: 07

Minute: 58

Second: 35.466

7.9.3 Trimming Strings

The trim function takes a string, and returns the input string after removing all trailing blanks.

Example

```
program trimString
```

```
implicit none
```

```
character (len = *n), parameter :: fname="Susanne", sname="Rizwan"
```

```
character (len = 20) :: fullname
```

```
fullname = fname// " " //sname !concatenating the strings
```

```
print*,fullname," , the beautiful dancer from the east!"
```

```
print*,trim(fullname)," , the beautiful dancer from the east!"
```

```
end program trimString
```

When you compile and execute the above program it produces the following result –

Susanne Rizwan , the beautiful dancer from the east!



Susanne Rizwan, the beautiful dancer from the east!

7.9.4 Left and Right Adjustment of Strings:

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

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

Example

```
program hello
```

```
implicit none
```

```
character(len = 15) :: surname, firstname
```

```
character(len = 6) :: title
```

```
character(len = 40):: name
```

```
character(len = 25):: greetings
```

```
title = 'Mr. '
```

```
firstname = 'Rowan'
```

```
surname = 'Atkinson'
```

```
greetings = 'A big hello from Mr. Beans'
```

```
name = adjustl(title)//adjustl(firstname)//adjustl(surname)
```

```
print *, 'Here is', name
```

```
print *, greetings
```



```
name = adjustr(title)//adjustr(firstname)//adjustr(surname)

print *, 'Here is', name

print *, greetings

name = trim(title)//trim(firstname)//trim(surname)

print *, 'Here is', name

print *, greetings

end program hello
```

When you compile and execute the above program it produces the following result –

Here is Mr. Rowan Atkinson

A big hello from Mr. Bean

Here is Mr. Rowan Atkinson

A big hello from Mr. Bean

Here is Mr.RowanAtkinson

A big hello from Mr. Bean

7.9.5 Searching for a Substring in a String:

The index function takes two strings and checks if the second string is a substring of the first string. If the second argument is a substring of the first argument, then it returns an integer which is the starting index of the second string in the first string, else it returns zero.

Example

```
program hello

implicit none

character(len=30) :: myString

character(len=10) :: testString
```



```
myString = 'This is a test'
testString = 'test'
if(index(myString, testString) == 0)then
    print *, 'test is not found'
else
    print *, 'test is found at index: ', index(myString, testString)
end if
end program hello
```

When you compile and execute the above program it produces the following result –

```
test is found at index: 11
```

7.10 Summary:

Arrays are a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type. Compiler does not perform any bound checking on an array. The array variable acts as a pointer to the zeroth element of the array. Array elements are stored in contiguous memory locations and they can be accessed using pointers. String is a collection of characters where the last character is null. As it is an array, all the characters are stored in contiguous memory locations. There are various operations that can be performed on the string.

7.11 KEYWORDS

Array: An array is an ordered and finite set of homogenous elements.

Dynamic Array: A dynamic array is an array, the size of which is not known at compile time, but will be known at execution time.

String: A string is a group of characters of any length.

String Operations: There are various operations applied on string like concatenation, trimming, substring.

**Answer to check your Progress****Check Your Progress A**

- 1 Arrays
- 2 dimension
- 3 lower and upper bound

Check Your Progress B

- 1 Dynamic
- 2 single character or contiguous strings.
- 3 String concatenation

7.12 Self Assessment Questions:

- 1) Why an array is called a derived data type?
- 2) Can an array is assigned to another of same type and size?
- 3) What would be contents of the following array after initialization?
int A[5]= {4,5,6};
- 4) Write a program to copy the contents of an array into the another in a reverse array?
- 5) What is String?
- 6) Describe the operations of string?

7.13 SUGGESTED READINGS

1. Computer Programming in Fortran 90 and 95 by V Rajaraman
2. Introduction to Programming with Fortran by Chivers, Ian, Sleightholme, Jane
3. Fortran For Scientists and Engineers by Stephen J. Chapman, Mcgraw-Hill



Class : M.Sc. (Mathematics)	Course Code : MAL 516
Subject : Programming with FORTRAN (Theory)	

CHAPTER-8

FUNCTIONS AND SUBROUTINES

STRUCTURE

- 8.0 Objectives
- 8.1 Introduction
- 8.2 Subprogram
- 8.3 Advantages of Subprogram
- 8.4 Function
- 8.5 Subroutine
- 8.6 Specifying Arguments
- 8.7 Recursive Procedure
- 8.8 Internal Procedure
- 8.9 Difference between function subprogram and Subroutine subprogram
- 8.10 Intrinsic Function
- 8.11 Summary
- 8.12 Keywords
- 8.13 Self-Assessment Questions
- 8.14 Suggested Readings



8.0 OBJECTIVES

After reading this lesson, you should be able to:

- 1) Define the functions and how they are initialized.
- 2) Explain the concept of encryption and decryption.
- 3) Define multidimensional arrays.
- 4) How a string can be stored in array?

8.1 Introduction:

In fortran language procedure are program segments which have an independent existence. In other words procedure may be developed separately and tested. They can be than used by other programs. There are two types of procedure in FORAN 90 known as function and subroutine. There are some inbuilt function known as intrinsic function, such as **SIN** and **EXP** Programs for evaluating such such functions have already been written, tested and incorporated in the Fortran Processor by a professional programmer. when we write **SIN(y)** we invoke such an independent fuction. Besides such predefined function we may write to program functions which we may need frequently but which are notlibrary. In such a case it is a good idea to write a good efficient program for this, put it in our own “private library” and use it just like a predefined functions.

Another motivation for providing subroutines and functions in fortran is to enable a fortran user to use programs developed by others, when it satisfies his requirements. There are several of efficient , well tested program for common problems in matrix manipulation, algebraic equation solution, statistical computation etc, have been developed and tested by companies.

In this chapter we will discuss the method of defining and using function and subroutines. Functions and subroutines may be classified as:

Predefined (built-in) or Intrinsic function

External functions or Function subprograms

Generic function

Subroutine subprogram



Intrinsic function such as SQRT, SIN, COS, MOD etc

External functions or Function subprograms is defined by a group of statements. It is implicitly referenced by the appearance of its name in the program.

A generic function is function which returns a value of same type as that's of its argument.

A subroutine subprogram is also defined by an independent group of statements. It is explicitly referenced by using a statement known as CALL in fortran.

8.2 Subprogram

When a programs 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 subprogram 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.

There are two types of subprograms –

- Functions
- Subroutines

8.3 Advantages of Subprogram

There are several advantages to using subprograms:

1. They help keep the code simple, and, thus, more readable.
2. They allow the programmer to use the same code as many times as needed throughout the program.
3. They allow the programmer to define needed functions.
4. They can be used in other programs.

8.4 Function



A function is a procedure that returns a single quantity. A function should not modify its arguments.

The returned quantity is known as **function value**, and it is denoted by the function name.

Syntax

Syntax for a function is as follows –

```
function name(arg1, arg2, ....)
```

```
    [declarations, including those for the arguments]
```

```
    [executable statements]
```

```
end function [name]
```

The following example demonstrates a function named `area_of_circle`. It calculates the area of a circle with radius `r`.

```
program calling_func

    real :: a
    a = area_of_circle(2.0)

    Print *, "The area of a circle with radius 2.0 is"

    Print *, a

end program calling_func

! this function computes the area of a circle with radius r
function area_of_circle (r)
```



```
! function result
implicit none

! dummy arguments
real :: area_of_circle

! local variables
real :: r
real :: pi

pi = 4 * atan (1.0)
area_of_circle = pi * r**2

end function area_of_circle
```

When you compile and execute the above program, it produces the following result –

The area of a circle with radius 2.0 is

12.5663710

Please note that –

- You must specify **implicit none** in both the main program as well as the procedure.
- The argument r in the called function is called **dummy argument**.

The result Option



If you want the returned value to be stored in some other name than the function name, you can use the **result** option.

You can specify the return variable name as –

```
function name(arg1, arg2, ....) result (return_var_name)
```

```
  [declarations, including those for the arguments]
```

```
  [executable statements]
```

```
end function [name]
```

8.5 Subroutine

subroutine does not return a value, however it can modify its arguments.

Syntax

```
subroutine name(arg1, arg2, ....)
```

```
{declarations, including those for the arguments}
```

```
  data-type, intent(in):: var1, var2, ..., varn
```

```
  data-type, intent(out):: var1, var2, ..., varn
```

```
end subroutine [name]
```

How to Calling a Subroutine

You need to invoke a subroutine using the call statement.

The following example demonstrates the definition and use of a subroutine swap, that changes the values of its arguments.

```
program calling_func
```

```
  implicit none
```

```
  real :: a, b
```

```
    a = 5.0
```



```
b = 7.0  
  
Print *, "Before calling swap"  
  
Print *, "a = ", a  
Print *, "b = ", b  
  
call swap(a, b)  
  
Print *, "After calling swap"  
  
Print *, "a = ", a  
Print *, "b = ", b  
  
end program calling_func  
  
subroutine swap(x, y)  
  
implicit none  
  
real :: x, y, temp  
  
    temp = x  
  
    x = y  
  
    y = temp  
  
end subroutine swap
```

When you compile and execute the above program, it produces the following result –

Before calling swap

a =5.00000000

b = 7.00000000

After calling swap

a = 7.00000000

b = 5.00000000



8.6 Specifying the Intent of the Arguments:

The intent attribute allows you to specify the intention with which arguments are used in the procedure.

The following table provides the values of the intent attribute –

Value	Used as	Explanation
In	intent(in)	Used as input values, not changed in the function
Out	intent(out)	Used as output value, they are overwritten
inout	intent(inout)	Arguments are both used and overwritten

The following example demonstrates the concept –

```
program calling_func
```

```
implicit none
```

```
real :: x, y, z, disc
```

```
  x = 2.0
```

```
  y = 6.0
```

```
  z = 3.0
```

```
  call intent_example(x, y, z, disc)
```

```
  Print *, "The value of the discriminant is"
```

```
  Print *, disc
```

```
end program calling_func
```

```
subroutine intent_example (a, b, c, d)
```

```
implicit none
```

```
  ! dummy arguments
```



```
real, intent (in) :: a
real, intent (in) :: b
real, intent (in) :: c
real, intent (out) :: d
d = b * b - 4.0 * a * c
end subroutine intent_example
```

When you compile and execute the above program, it produces the following result –

The value of the discriminant is

12.0000000

Check Your Progress A

- 1 is a procedure that returns a single quantity.
- 2 Function should not modify its
- 3 does not return a value.

8.7 Recursive Procedures:

Recursion occurs when a programming languages allows you to call a function inside the same function. It is called recursive call of the function.

When a procedure calls itself, directly or indirectly, is called a recursive procedure. You should declare this type of procedures by preceding the word recursive before its declaration.

When a function is used recursively, the result option has to be used.

Following is an example, which calculates factorial for a given number using a recursive procedure –

program calling_func

implicit none



```
integer :: i, f
```

```
i = 15
```

```
Print *, "The value of factorial 15 is"
```

```
f = myfactorial(15)
```

```
Print *, f
```

```
end program calling_func
```

```
! computes the factorial of n (n!)
```

```
recursive function myfactorial (n) result (fac)
```

```
! function result
```

```
implicit none
```

```
! dummy arguments
```

```
integer :: fac
```

```
integer, intent (in) :: n
```

```
select case (n)
```

```
case (0:1)
```

```
fac = 1
```

```
case default
```

```
fac = n * myfactorial (n-1)
```



```
end select
```

```
end function myfactorial
```

8.8 Internal Procedures:

When a procedure is contained within a program, it is called the internal procedure of the program. The syntax for containing an internal procedure is as follows –

```
program program_name  
  implicit none  
  ! type declaration statements  
  ! executable statements  
  ...  
  contains  
  ! internal procedures  
  ...  
end program program_name
```

The following example demonstrates the concept –

```
program mainprog  
  implicit none  
  
  real :: a, b  
  
  a = 2.0  
  
  b = 3.0
```



```
Print *, "Before calling swap"
```

```
Print *, "a = ", a
```

```
Print *, "b = ", b
```

```
call swap(a, b)
```

```
Print *, "After calling swap"
```

```
Print *, "a = ", a
```

```
Print *, "b = ", b
```

```
contains
```

```
subroutine swap(x, y)
```

```
real :: x, y, temp
```

```
temp = x
```

```
x = y
```

```
y = temp
```

```
end subroutine swap
```

```
end program mainprog
```

When you compile and execute the above program, it produces the following result –

Before calling swap

a = 2.00000000



b = 3.00000000

After calling swap

a = 3.00000000

b = 2.00000000

Check your Progress B

- 1 occurs when a programming languages allows you to call a function inside the same function.
- 2 Procedure is contained within a program, it is called the
- 3 are some common and important functions that are provided as a part of the Fortran language

8.9 Difference between Function and Subroutine

S.No.	Function Subprogram	Subroutine Subprogram
1	It can return only one value to the calling program.	It can return more than one value to the calling program.
2	It returns value through its name.	It returns value through its arguments.
3	There must be at least one argument in the dummy list.	There is no restriction i.e it can be without any arguments.
4	It is called by referencing by its name.	It is called by CALL statement.
5	It has a type associated with its name which identifies types value returned by it.	It has no type associated with its name and value returned can be of different type arguments returning values to the calling program.

8.10 Intrinsic Function:

Intrinsic functions are some common and important functions that are provided as a part of the Fortran compiler supplied by main factors of compiler. We have already discussed some of these functions in the Arrays, Characters and String chapters.



Intrinsic functions can be categorised as –

- 1) Numeric Functions
- 2) Mathematical Functions
- 3) Numeric Inquiry Functions
- 4) Floating-Point Manipulation Functions
- 5) Bit Manipulation Functions
- 6) Character Functions

In the following section we provide brief descriptions of all these functions from other categories.

In the function name column,

- 1) A represents any type of numeric variable
- 2) R represents a real or integer variable
- 3) X and Y represent real variables
- 4) Z represents complex variable
- 5) W represents real or complex variable

8.10.1 Numeric Functions:

Sr.No	Function & Description
	The expression in [] is optional.
1	<p>ABS (A)</p> <p>It returns the absolute value of A</p>
2	<p>AIMAG (Z)</p> <p>It returns the imaginary part of a complex number Z</p>



3	AINT (A [, KIND]) It truncates fractional part of A towards zero, returning a real, whole number.
4	ANINT (A [, KIND]) It returns a real value, the nearest integer or whole number.
5	CEILING (A [, KIND]) It returns the least integer greater than or equal to number A.
6	CMPLX (X [, Y, KIND]) It converts the real variables X and Y to a complex number $X+iY$; if Y is absent, 0 is used.
7	CONJG (Z) It returns the complex conjugate of any complex number Z.
8	DBLE (A) It converts A to a double precision real number.
9	DIM (X, Y) It returns the positive difference of X and Y.
10	DPROD (X, Y) It returns the double precision real product of X and Y.
11	FLOOR (A [, KIND]) It provides the greatest integer less than or equal to number A.



12	INT (A [, KIND]) It converts a number (real or integer) to integer, truncating the real part towards zero.
13	MAX (A1, A2 [, A3,...]) It returns the maximum value from the arguments, all being of same type.
14	MIN (A1, A2 [, A3,...]) It returns the minimum value from the arguments, all being of same type.
15	MOD (A, P) It returns the remainder of A on division by P, both arguments being of the same type $(A - \text{INT}(A/P) * P)$
16	MODULO (A, P) It returns A modulo P: $(A - \text{FLOOR}(A/P) * P)$
17	NINT (A [, KIND]) It returns the nearest integer of number A
18	REAL (A [, KIND]) It Converts to real type
19	SIGN (A, B) It returns the absolute value of A multiplied by the sign of P. Basically it transfers the of sign of B to A.



Example

```
program numericFunctions
implicit none

! define constants

! define variables
real :: a, b
complex :: z

! values for a, b
a = 15.2345
b = -20.7689

write(*,*) 'abs(a): ',abs(a),' abs(b): ',abs(b)
write(*,*) 'aint(a): ',aint(a),' aint(b): ',aint(b)
write(*,*) 'ceiling(a): ',ceiling(a),' ceiling(b): ',ceiling(b)
write(*,*) 'floor(a): ',floor(a),' floor(b): ',floor(b)

z = cmplx(a, b)
write(*,*) 'z: ',z

end program numericFunctions
```

When you compile and execute the above program, it produces the following result –



abs(a): 15.2344999 abs(b): 20.7688999

aint(a): 15.0000000 aint(b): -20.0000000

ceiling(a): 16 ceiling(b): -20

floor(a): 15 floor(b): -21

z: (15.2344999, -20.7688999)

8.10.2 Mathematical Functions:

Sr.No	Function & Description
1	ACOS (X) It returns the inverse cosine in the range $(0, \pi)$, in radians.
2	ASIN (X) It returns the inverse sine in the range $(-\pi/2, \pi/2)$, in radians.
3	ATAN (X) It returns the inverse tangent in the range $(-\pi/2, \pi/2)$, in radians.
4	ATAN2 (Y, X) It returns the inverse tangent in the range $(-\pi, \pi)$, in radians.
5	COS (X) It returns the cosine of argument in radians.



6	COSH (X) It returns the hyperbolic cosine of argument in radians.
7	EXP (X) It returns the exponential value of X.
8	LOG (X) It returns the natural logarithmic value of X.
9	LOG10 (X) It returns the common logarithmic (base 10) value of X.
10	SIN (X) It returns the sine of argument in radians.
11	SINH (X) It returns the hyperbolic sine of argument in radians.
12	SQRT (X) It returns square root of X.
13	TAN (X) It returns the tangent of argument in radians.



14	<p>TANH (X)</p> <p>It returns the hyperbolic tangent of argument in radians.</p>
----	---

Example

The following program computes the horizontal and vertical position x and y respectively of a projectile after a time, t –

Where, $x = u t \cos a$ and $y = u t \sin a - g t^2 / 2$

```
program projectileMotion
```

```
implicit none
```

```
! define constants
```

```
real, parameter :: g = 9.8
```

```
real, parameter :: pi = 3.1415927
```

```
!define variables
```

```
real :: a, t, u, x, y
```

```
!values for a, t, and u
```

```
a = 45.0
```

```
t = 20.0
```

```
u = 10.0
```

```
! convert angle to radians
```

```
a = a * pi / 180.0
```

```
x = u * cos(a) * t
```

```
y = u * sin(a) * t - 0.5 * g * t * t
```



```
write(*,*) 'x: ',x,' y: ',y
```

```
end program projectileMotion
```

When you compile and execute the above program, it produces the following result –

```
x: 141.421356 y: -1818.57861
```

8.10.3 Numeric Inquiry Functions:

These functions work with a certain model of integer and floating-point arithmetic. The functions return properties of numbers of the same kind as the variable X, which can be real and in some cases integer.

Sr.No	Function & Description
1	DIGITS (X) It returns the number of significant digits of the model.
2	EPSILON (X) It returns the number that is almost negligible compared to one. In other words, it returns the smallest value such that $\text{REAL}(1.0, \text{KIND}(X)) + \text{EPSILON}(X)$ is not equal to $\text{REAL}(1.0, \text{KIND}(X))$.
3	HUGE (X) It returns the largest number of the model
4	MAXEXPONENT (X) It returns the maximum exponent of the model
5	MINEXPONENT (X) It returns the minimum exponent of the model
6	PRECISION (X) It returns the decimal precision
7	RADIX (X)



	It returns the base of the model
8	RANGE (X) It returns the decimal exponent range
9	TINY (X) It returns the smallest positive number of the model

8.10.4 Floating-Point Manipulation Functions:

Sr.No	Function & Description
1	EXPONENT (X) It returns the exponent part of a model number
2	FRACTION (X) It returns the fractional part of a number
3	NEAREST (X, S) It returns the nearest different processor number in given direction
4	RRSPACING (X) It returns the reciprocal of the relative spacing of model numbers near given number
5	SCALE (X, I) It multiplies a real by its base to an integer power
6	SET_EXPONENT (X, I) it returns the exponent part of a number
7	SPACING (X)



It returns the absolute spacing of model numbers near given number

8.10.5 Bit Manipulation Functions:

Sr.No	Function & Description
1	BIT_SIZE (I) It returns the number of bits of the model
2	BTEST (I, POS) Bit testing
3	IAND (I, J) Logical AND
4	IBCLR (I, POS) Clear bit
5	IBITS (I, POS, LEN) Bit extraction
6	IBSET (I, POS) Set bit
7	IEOR (I, J) Exclusive OR
8	IOR (I, J) Inclusive OR
9	ISHFT (I, SHIFT) Logical shift



10	ISHFTC (I, SHIFT [, SIZE]) Circular shift
11	NOT (I) Logical complement

8.10.6 Character Functions:

Sr.No	Function & Description
1	ACHAR (I) It returns the I character in the ASCII collating sequence.
2	ADJUSTL (STRING) It adjusts string left by removing any leading blanks and inserting trailing blanks
3	ADJUSTR (STRING) It adjusts string right by removing trailing blanks and inserting leading blanks.
4	CHAR (I [, KIND]) It returns the I character in the machine specific collating sequence
5	IACHAR (C) It returns the position of the character in the ASCII collating sequence.
6	ICHAR (C) It returns the position of the character in the machine (processor) specific collating sequence.
7	INDEX (STRING, SUBSTRING [, BACK]) It returns the leftmost (rightmost if BACK is .TRUE.) starting position of



	SUBSTRING within STRING.
8	LEN (STRING) It returns the length of a string.
9	LEN_TRIM (STRING) It returns the length of a string without trailing blank characters.
10	LGE (STRING_A, STRING_B) Lexically greater than or equal
11	LGT (STRING_A, STRING_B) Lexically greater than
12	LLE (STRING_A, STRING_B) Lexically less than or equal
13	LLT (STRING_A, STRING_B) Lexically less than
14	REPEAT (STRING, NCOPIES) Repeated concatenation
15	SCAN (STRING, SET [, BACK]) It returns the index of the leftmost (rightmost if BACK is .TRUE.) character of STRING that belong to SET, or 0 if none belong.
16	TRIM (STRING) Removes trailing blank characters
17	VERIFY (STRING, SET [, BACK]) Verifies the set of characters in a string



8.11 Summary

A function is a block of statements that performs a specific task. Functions are used to improve the readability and reusability of the code, same function can be used in any program rather than writing the same code from scratch, debugging of the code would be easier if you use functions, as errors are easy to be traced and reduces the size of the code, duplicate set of statements are replaced by function calls.

8.12 KEYWORDS

Function: A function is a subprogram that can be defined by the user in his/her program.

Scope: The range of code in a program over which a variable has a meaning is called as scope of the variable.

Subroutine: Subroutine does not return a value however it can modify its argument.

Recursive Procedure: Recursion occur when a programming language allow you to call a function inside the same function is called Recursion.

Internal Procedure: A procedure is contained within a program is called internal procedure.

Answer to check your progress

Check Your Progress A

- 1 Function
- 2 argument
- 3 Subroutine

Check Your Progress B

- 1 Recursion
- 2 Internal Procedure
- 3 Intrinsic Function

8.13 SELF ASSESSENT QUESTION:

- 1) Explain in brief function.



- 2) What is meant by recursion?
- 3) Explain in brief the intrinsic functions.
- 4) What is meant by subroutine?

8.14 SUGGESTED READINGS

1. Computer Programming in Fortran 90 and 95 by V Rajaraman
2. Introduction to Programming with Fortran by Chivers, Ian, Sleightholme, Jane
3. Fortran For Scientists and Engineers by Stephen J. Chapman, McGraw-Hill



Class : M.Sc. (Mathematics)

Course Code : MAL 516

Subject : Programming with FORTRAN (Theory)

CHAPTER-9

DERIVED TYPE AND POINTERS

STRUCTURE

- 9.0 Objectives
- 9.1 Introduction
- 9.2 Definition of Derived Data Type
- 9.3 Pointers & Targets
 - 9.3.1 Pointer Assignment Statement
 - 9.3.2 Pointer Association Status
- 9.4 Pointers in Assignment Statement
- 9.5 Pointers with Arrays
- 9.6 Dynamic Memory Allocation with Pointers
- 9.7 Pointers as Component of Derived Data Types
- 9.8 Arrays of Pointers
- 9.9 Pointers in Procedures
 - 9.9.1 Using ITENT Attribute
 - 9.9.2 Pointer Valued Function
- 9.10 Procedure Pointers
- 9.11 Summary



9.12 Keywords

9.13 Self-Assessment Questions

9.14 Suggested Readings

9.0 Objective:

After reading this lesson, you should be able to:

1. Understand how to declare a pointer variable.
2. Understand how to allocate memory to the pointers.
3. Understand dynamic memory allocation of pointers.
4. Understand the concept of derived data type using pointers.

9.1 Introduction

In earlier chapters, we have created and used variables of the five intrinsic Fortran data types and of derived data types. These variables all had two characteristics in common: They all stored some form of data, and they were almost all static, meaning that the number and types of variables in a program were declared before program execution, and remained local the same throughout program execution. Fortran includes another type of variable that contains no data at all. Instead, it contains the address in memory of another variable where the data is actually stored. Because this type of variable points to another variable, it is called a pointer. The difference between a pointer and an ordinary variable is illustrated.



Figure 9.1 show pointer variable and data variable

The difference between a pointer and an ordinary variable: (a) A pointer stores the address of an ordinary variable in its memory location. (b) An ordinary variable stores a data value. Both pointers and ordinary variables have names, but pointers store the addresses of ordinary variables, while



ordinary variables store data values. Pointers are primarily used in situations where variables and arrays must be created and destroyed dynamically during the execution of a program, and where it is not known before the program executes just how many of any given type of variable will be needed during a run. For example, suppose that a mailing list program must read in an unknown number of names and addresses, sort them into a user-specified order, and then print mailing labels in that order. The names and addresses will be stored in variables of a derived data type. If this program is implemented with static arrays, then the arrays must be as large as the largest possible mailing list ever to be processed. Most of the time the mailing lists will be much smaller, and this will produce a terrible waste of computer memory. If the program is implemented with allocatable arrays, then we can allocate just the required amount of memory, but we must still know in advance how many addresses there will be before the first one is read. By contrast, we will now learn how to dynamically allocate a variable for each address as it is read in, and how to use pointers to manipulate those addresses in any desired fashion. This flexibility will produce a much more efficient program.

We will first learn the basics of creating and using pointers, and then see several examples of how they can be used to write flexible and powerful programs.

9.2 DEFINITION OF DERIVED DATA TYPE

Fortran allows you to define derived data types. A derived data type is also called a structure, and it can consist of data objects of different types.

Derived data types are used to represent a record. E.g. you want to keep track of your books in a library, you might want to track the following attributes about each book –

- Title
- Author
- Subject
- Book ID

Derived data type

To define a derived data **type**, the **type** and **end type** statements are used. . The **type** statement defines a new data type, with more than one member for your program. The format of the **type** statement is this –



```
type type-name
```

```
    declaration
```

```
end type
```

Here is the way to declare a book structure

```
type Books
```

```
    character(len = 50) :: title
```

```
    character(len = 50) :: author
```

```
    character(len = 150) :: subject
```

```
    integer :: book_id
```

```
end type Books
```

Derived data types & Elements

Elements of derived types are accessed with the "%" operator.

A structure of type Books can be created in a type declaration statement like –

```
type(Books) :: book1
```

The components of the structure can be accessed using the component selector character (%) –

```
book1%title = "C Programming"
```

```
book1%author = "Nuha Ali"
```

```
book1%subject = "C Programming Tutorial"
```

```
book1%book_id = 6495407
```

Note that there are no spaces before and after the % symbol.

Derived data type & Array

We can also create arrays of a derived type –

```
type(Books), dimension(2) :: list
```



Individual elements of the array could be accessed as –

```
list(1)%title = "C Programming"
```

```
list(1)%author = "Nuha Ali"
```

```
list(1)%subject = "C Programming Tutorial"
```

```
list(1)%book_id = 6495407
```

9.3 POINTERS AND TARGETS

A Fortran variable is declared to be a pointer by either including the `POINTER` attribute in its type definition statement (the preferred choice) or by listing it in a separate `POINTER` statement. For example, each of the following statements declares a pointer `p1` that must point to a real variable.

```
REAL, POINTER :: p1
```

or

```
REAL :: p1
```

```
POINTER :: p1
```

Note that the type of a pointer must be declared, even though the pointer does not contain any data of that type. Instead, it contains the address of a variable of the declared type. A pointer is only allowed to point to variables of its declared type. Any attempt to point to a variable of a different type will produce a compilation error.

Pointers to variables of derived data types may also be declared. For example,

```
TYPE (vector), POINTER :: vector_pointer
```

declares a pointer to a variable of derived data type `vector`. Pointers may also point to an array. A pointer to an array is declared with a deferred-shape array specification, meaning that the rank of the array is specified, but the actual extent of the array in each dimension is indicated by colons. Two pointers to arrays are:

```
INTEGER, DIMENSION(:), POINTER :: ptr1
```

```
REAL, DIMENSION(:,:), POINTER :: ptr2
```



The first pointer can point to any 1D integer array, while the second pointer can point to any 2D real array. A pointer can point to any variable or array of the pointer's type as long as the variable or array has been declared to be a target. A target is a data object whose address has been made available for use with pointers. A Fortran variable or array is declared to be a target by either including the TARGET attribute in its type definition statement (the preferred choice) or by listing it in a separate TARGET statement. For example, each of the following sets of statements declares two targets to which pointers may point.

```
REAL, TARGET :: a1 = 7
```

```
INTEGER, DIMENSION(10), TARGET :: int_array
```

or

```
REAL :: a1 = 7
```

```
INTEGER, DIMENSION(10) :: int_array
```

```
TARGET :: a1, int_array
```

They declare a real scalar value `a1` and a rank 1 integer array `int_array`. Variable `a1` may be pointed to by any real scalar pointer (such as the pointer `p1` declared above), and `int_array` may be pointed to by any integer rank 1 pointer (such as pointer `ptr1` above).

9.3.1 Pointer Assignment Statements

A pointer can be associated with a given target by means of a pointer assignment statement. A pointer assignment statement takes the form

```
pointer => target
```

where `pointer` is the name of a pointer, and `target` is the name of a variable or array of the same type as the pointer. The pointer assignment operator consists of an equal sign followed by a greater than sign with no space in between.

When this statement is executed, the memory address of the target is stored in the pointer. After the pointer assignment statement, any reference to the pointer will actually be a reference to the data stored in the target.



If a pointer is already associated with a target, and another pointer assignment statement is executed using the same pointer, then the association with the first target is lost and the pointer now points to the second target. Any reference to the pointer after the second pointer assignment statement will actually be a reference to the data stored in the second target.

For example, the program defines a real pointer `p` and two target variables `t1` and `t2`. The pointer is first associated with variable `t1` by a pointer assignment statement, and `p` is written out by a `WRITE` statement. Then the pointer is associated with variable `t2` by another pointer assignment statement, and `p` is written out by a second `WRITE` statement.

Example:

Program to illustrate pointer assignment statements.

```
PROGRAM test_ptr
IMPLICIT NONE
REAL, POINTER :: p
REAL, TARGET :: t1 = 10., t2 = -17.

p => t1
WRITE (*,*) 'p, t1, t2 = ', p, t1, t2

p => t2
WRITE (*,*) 'p, t1, t2 = ', p, t1, t2
END PROGRAM test_ptr
```

When this program is executed, the results are:

```
C:\book\fortran>test_ptr
p, t1, t2 = 10.000000 10.000000 -17.000000
p, t1, t2 = -17.000000 10.000000 -17.000000
```

It is important to note that `p` never contains either 10. or `-17`. Instead, it contains the addresses of the variables in which those values were stored, and the Fortran compiler treats a reference to the pointer as



a reference to those addresses. Also, note that value could be accessed either through a pointer to a variable or through the variable’s name, and the two forms of access can be mixed even within a single statement. It is also possible to assign the value of one pointer to another pointer in a pointer assignment statement.

```
pointer1 => pointer2
```

After such a statement, both pointers point directly and independently to the same target. If either pointer is changed in a later assignment, the other one will be unaffected and will continue to point to the original target. If pointer2 is disassociated (does not point to a target) at the time the statement is executed, then pointer1 also becomes disassociated. For example, the program defines two real pointers p1 and p2, and two target variables t1 and t2. The pointer p1 is first associated with variable t1 by a pointer assignment statement, and then pointer p2 is assigned the value of pointer p1 by another pointer assignment statement. After these statements, both pointers p1 and p2 are independently associated with variable t1. When pointer p1 is later associated with variable t2, pointer p2 remains associated with t1.

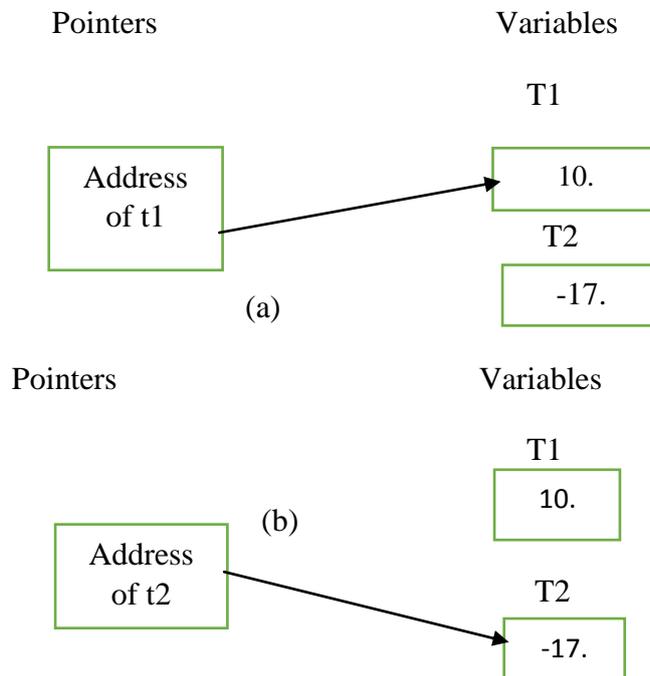


Figure 9.2 variable points to its address

The relationship between the pointer and the variables in program test_ptr. (a) The situation



after the first executable statement: p contains the address of variable t1, and a reference to p is the same as a reference to t1. (b) The situation after the third executable statement: p contains the address of variable t2, and a reference to p is the same as a reference to t2.

9.3.2 Pointer Association Status

The association status of a pointer indicates whether or not the pointer currently points to a valid target. There are three possible statuses: undefined, associated, and disassociated. When a pointer is first declared in a type declaration statement, its pointer association status is undefined. Once a pointer has been associated with a target by a pointer assignment statement, its association status becomes associated. If a pointer is later disassociated from its target and is not associated with any new target, then its association status becomes disassociated. How can a pointer be disassociated from its target? It can be disassociated from one target and simultaneously associated with another target by executing a pointer assignment statement. In addition, a pointer can be disassociated from all targets by executing a NULLIFY statement. A NULLIFY statement has the form

```
NULLIFY ( ptr1 [,ptr2, ...] )
```

where ptr1, ptr2, etc., are pointers. After the statement is executed, the pointers listed in the statement are disassociated from all targets. A pointer can only be used to reference a target when it is associated with that target. Any attempt to use a pointer when it is not associated with a target will result in an error, and the program containing the error will abort. Therefore, we must be able to tell whether or not a particular pointer is associated with a particular target, or with any target at all. This can be done using the logical intrinsic function ASSOCIATED. The function comes in two forms, one containing a pointer as its only argument and one containing both a pointer and a target. The first form is

```
status = ASSOCIATED ( pointer )
```

This function returns a true value if the pointer is associated with any target, and a false value if it is not associated with any target. The second form is

```
status = ASSOCIATED ( pointer, target )
```



This function returns a true value if the pointer is associated with the particular target included in the function, and a false value otherwise. A pointer's association status can only be undefined from the time that it is

declared until it is first used. Thereafter, the pointer's status will always be either associated or disassociated. Because the undefined status is ambiguous, it is recommended that every pointer's status be clarified as soon as it is created by either assigning it to a target or nullifying it. For example, pointers could be declared and nullified in a program as follows:

```
REAL, POINTER :: p1, p2
```

```
INTEGER, POINTER :: i1
```

```
...
```

```
(additional specification statements)
```

```
...
```

```
NULLIFY (p1, p2, i1)
```

Fortran also provides an intrinsic function `NULL()` that can be used to nullify a pointer at the time it is declared (or at any time during the execution of a program). Thus, pointers can be declared and nullified as follows:

```
REAL, POINTER :: p1 => NULL(), p2 => NULL()
```

```
INTEGER, POINTER :: i1 => NULL()
```

```
...
```

```
(additional specification statements)
```

9.4 POINTERS IN ASSIGNMENT STATEMENTS

Whenever a pointer appears in a Fortran expression where a value is expected, the value of the target pointed to is used instead of the pointer itself. This process is known as dereferencing the pointer. We have already seen an example of dereferencing in the previous section: Whenever a pointer appeared in



a WRITE statement, the value of the target pointed to was printed out instead. As another example, consider two pointers

p1 and p2 that are associated with variables a and b, respectively. In the ordinary assignment statement

$$p2 = p1$$

both p1 and p2 appear in places where variables are expected, so they are dereferenced, and this statement is exactly identical to the statement

$$b = a$$

By contrast, in the pointer assignment statement

$$p2 \Rightarrow p1$$

p2 appears in a place where a pointer is expected, while p1 appears in a place where a target (an ordinary variable) is expected. As a result, p1 is dereferenced, while p2 refers to the pointer itself. The result is that the target pointed to by p1 is assigned to the pointer p2.

The program provides another example of using pointers in place of variables:

9.5 POINTERS WITH ARRAYS

A pointer can point to an array as well as a scalar. A pointer to an array must declare the type and the rank of the array that it will point to, but does not declare the extent in each dimension. Thus, the following statements are legal:

```
REAL, DIMENSION(100,1000), TARGET :: mydata
```

```
REAL, DIMENSION(:,:), POINTER :: pointer
```

```
pointer => array
```

A pointer can point not only to an array but also to a subset of an array (an array section). Any array section that can be defined by a subscript triplet can be used as the target of a pointer. For example, the program declares a 16-element integer array info, and fills the array with the values 1 through 16. This array serves as the target for a series of pointers. The first pointer ptr1 points to the entire array,



while the second one points to the array section defined by the subscript triplet `ptr1(2::2)`. This will consist of the even subscripts 2, 4, 6, 8, 10, 12, 14, and 16 from the original array. The third pointer also uses the subscript triplet `2::2`, and it points the even elements from the list pointed to by second pointer. This will consist of the subscripts 4, 8, 12, and 16 from the original array. This process of selection continues with the remaining pointers.

Example

Program to illustrate invalid pointer assignments to array sections defined with vector subscripts.

```
PROGRAM bad
IMPLICIT NONE
INTEGER :: i
INTEGER, DIMENSION(3) :: subs = [ 1, 8, 11 ]
INTEGER, DIMENSION(16), TARGET :: info = [ (i, i=1,16) ]
INTEGER, DIMENSION(:), POINTER :: ptr1
ptr1 => info(subs)
WRITE (*,'(A,16I3)') ' ptr1 = ', ptr1
END PROGRAM bad
```

Check your Progress A

- 1 contains more information about a particular object, like type, rank, extents, and memory address.
- 2 Pointer is a that has more functionalities than just storing the memory address.
- 3 statement allows you to allocate space for a pointer object.

9.6 DYNAMIC MEMORY ALLOCATION WITH POINTERS



One of the most powerful features of pointers is that they can be used to dynamically create variables or arrays whenever required, and then to release the space used by the dynamic variables or arrays once they are no longer needed. The procedure for doing this is similar to that used to create allocatable arrays. Memory is allocated using an `ALLOCATE` statement, and it is deallocated using a `DEALLOCATE` statement. The `ALLOCATE` statement has the same form as the `ALLOCATE` statement for an allocatable array. The statement takes the form

```
ALLOCATE (pointer(size),[ ...,] STAT=status)
```

where `pointer` is the name of a pointer to the variable or array being created, `size` is the dimension specification if the object being created is an array, and `status` is the result of the operation. If the allocation is successful, then the status will be 0. If it fails, a processor-dependent positive integer will be returned in the status variable. The `STAT=` clause is optional but should always be used, since a failed allocation statement without a `STAT=` clause will cause a program to abort. This statement creates an unnamed data object of the specified size and the pointer's type, and sets the pointer to point to the object. Because the new data object is unnamed, it can only be accessed by using the pointer. After the statement is executed, the association status of the pointer will become associated. If the pointer was associated with another data object before the `ALLOCATE` statement is executed, then that association is lost. The data object created by using the pointer `ALLOCATE` statement is unnamed, and so can only be accessed by the pointer. If all pointers to that memory are either nullified or reassigned with other targets, then the data object will no longer be accessible by the program. The object will still be present in memory, but it will no longer be possible to use it. Thus, careless programming with pointers can result in memory being filled with unusable space. This unusable memory is commonly referred to as a "memory leak". One symptom of this problem is that a program seems to grow larger and larger as it continues to execute, until it either fills the entire computer or uses all available memory. An example of a program with a memory leak. In this program, 10-element arrays are allocated using both `ptr1` and `ptr2`. The two arrays are initialized to different values, and those values are printed out. Then `ptr2` is assigned to point to the same memory as `ptr1` in a pointer assignment statement. After that statement, the memory that was assigned to `ptr2` is no longer accessible to the program. That memory has been "lost", and will not be recovered until the program stops executing.

Example



Program to illustrate “memory leaks” in a program.

```
PROGRAM mem_leak

IMPLICIT NONE

INTEGER :: i, istat

INTEGER, DIMENSION(:), POINTER :: ptr1, ptr2

(concluded)

! Check associated status of ptrs.

WRITE (*,'(A,2L5)') ' Are ptr1, ptr2 associated? ', &
ASSOCIATED(ptr1), ASSOCIATED(ptr2)

! Allocate and initialize memory

ALLOCATE (ptr1(1:10), STAT=istat)

ALLOCATE (ptr2(1:10), STAT=istat)

ptr1 = [ (i, i = 1,10 ) ]

ptr2 = [ (i, i = 11,20 ) ]

! Check associated status of ptrs.

WRITE (*,'(A,2L5)') ' Are ptr1, ptr2 associated? ', &
ASSOCIATED(ptr1), ASSOCIATED(ptr2)

WRITE (*,'(A,10I3)') ' ptr1 = ', ptr1 ! Write out data

WRITE (*,'(A,10I3)') ' ptr2 = ', ptr2

ptr2 => ptr1 ! Reassign ptr2

WRITE (*,'(A,10I3)') ' ptr1 = ', ptr1 ! Write out data

WRITE (*,'(A,10I3)') ' ptr2 = ', ptr2

NULLIFY(ptr1) ! Nullify pointer
```



```
DEALLOCATE(ptr2, STAT=istat) ! Deallocate memory  
END PROGRAM mem_leak
```

When program mem_leak executes, the results are:

```
C:\book\fortran>mem_leak
```

```
Are ptr1, ptr2 associated? F F
```

```
Are ptr1, ptr2 associated? T T
```

```
ptr1 = 1 2 3 4 5 6 7 8 9 10
```

```
ptr2 = 11 12 13 14 15 16 17 18 19 20
```

```
ptr1 = 1 2 3 4 5 6 7 8 9 10
```

```
ptr2 = 1 2 3 4 5 6 7 8 9 10
```

Memory that has been allocated with an `ALLOCATE` statement should be deallocated with a `DEALLOCATE` statement when the program is finished using it. If it is not deallocated, then that memory will be unavailable for any other use until the program finishes executing. When memory is deallocated in a pointer `DEALLOCATE` statement, the pointer to that memory is nullified at the same time. Thus, the statement

```
DEALLOCATE(ptr2, STAT=istat)
```

both deallocates the memory pointed to and nullifies the pointer ptr2. The pointer `DEALLOCATE` statement can only deallocate memory that was created by an `ALLOCATE` statement. It is important to remember this fact. If the pointer in the statement happens to point to a target that was not created with an `ALLOCATE` statement, then the `DEALLOCATE` statement will fail and the program will abort unless the `STAT=` clause was specified. The association between such pointers and their targets can be broken by the use of the `NULLIFY` statement.

A potentially serious problem can occur when deallocating memory. Suppose that two pointers ptr1 and ptr2 both point to the same allocated array. If pointer ptr1 is used in a `DEALLOCATE` statement to deallocate the array, then that pointer is nullified. However, ptr2 will not be nullified. It will continue to point to the memory location where the array used to be, even if that memory location is reused for



some other purpose by the program. If that pointer is used to either read data from or write data to the memory location, it will be either reading unpredictable values or overwriting memory

used for some other purpose. In either case, using that pointer is a recipe for disaster! If a piece of allocated memory is deallocated, then all of the pointers to that memory should be nullified or reassigned. One of them will be automatically nullified by the DEALLOCATE statement, and any others should be nullified in NULLIFY statement(s).

9.7 POINTERS AS COMPONENTS OF DERIVED DATA TYPES

Pointers may appear as components of derived data types. Pointers in derived data types may even point to the derived data type being defined. This feature is very useful, since it permits us to construct various types of dynamic data structures linked together by successive pointers during the execution of a program. The simplest such structure is a linked list, which is a list of values linked together in a linear fashion by

pointers. For example, the following derived data type contains a real number and a pointer to another variable of the same type:

```
TYPE :: real_value
```

```
REAL :: value
```

```
TYPE (real_value), POINTER :: p
```

```
END TYPE
```

A linked list is a series of variables of a derived data type, with the pointer from each variable pointing to the next variable in the list. The pointer in the last variable is nullified, since there is no variable after it in the list. Two pointers (say, head and tail) are also defined to point to the first and last variables in the list. Recall that a static array must be declared with a fixed size when a program is compiled. As a result, we must size each such array to be large enough to handle the largest problem that a program will ever be required to solve. This large memory requirement can result in a program being too large to run on some computers, and also results in a waste of memory most of the time that the program is executed. Even allocatable arrays don't completely solve the problem. Allocatable arrays prevent memory waste by allowing us to allocate only the amount of memory needed for a specific problem, but



we must know before we allocate the memory just how many values will be present during a particular run. In contrast, linked lists permit us to add elements one at a time, and we do not have to know in advance how many elements will ultimately be in the list.

When a program containing a linked list first starts to execute, there are no values in the list. In that case, the head and tail pointers have nothing to point to, so they are both nullified. When the first value is read, a variable of the derived data type is created, and the value is stored in that variable.

9.8 ARRAYS OF POINTERS

It is not possible to declare an array of pointers in Fortran. In a pointer declaration, the DIMENSION attribute refers to the dimension of the pointer's target, not to the dimension of the pointer itself. The dimension must be declared with a deferred-shape specification, and the actual size will be the size of the target with which the pointer is associated. In the example shown below, the subscript on the pointer refers the corresponding position in the target array, so the value of ptr(4) is 6.

```
REAL, DIMENSION(:), POINTER :: ptr
REAL, DIMENSION(5), TARGET :: tgt = { -2, 5., 0., 6., 1 }

ptr =>tgt

WRITE (*,*) ptr(4)
```

There are many applications in which arrays of pointers are useful. Fortunately, we can create an array of pointers for those applications by using derived data types. It is illegal to have an array of pointers in Fortran, but it is perfectly legal to have an array of any derived data type. Therefore, we can declare a derived data type containing only a pointer, and then create an array of that data type! For example, the program in declares an array of a derived data type containing real pointers, each of which points to a real array.

EXAMPLE

Program illustrating how to create an array of pointers using a derived data type.

```
PROGRAM ptr_array

IMPLICIT NONE
```



```
TYPE :: ptr
  REAL, DIMENSION(:), POINTER :: p
END TYPE

TYPE (ptr), DIMENSION(3) :: p1
REAL, DIMENSION(4), TARGET :: a = [ 1., 2., 3., 4. ]
REAL, DIMENSION(4), TARGET :: b = [ 5., 6., 7., 8. ]
REAL, DIMENSION(4), TARGET :: c = [ 9., 10., 11., 12. ]

p1(1)%p => a
p1(2)%p => b
p1(3)%p => c

WRITE (*,*) p1(3)%p
WRITE (*,*) p1(2)%p(3)

END PROGRAM ptr_array
```

With the declarations in program `ptr_array`, the expression `p1(3)%p` refers to the third array (array `c`), so the first `WRITE` statement should print out 9., 10., 11., and 12. The expression `p1(2)%p(3)` refers to the third value of the second array (array `b`), so

the second `WRITE` statement prints out the value 7. When this program is compiled and executed with the Compaq Visual Fortran compiler, the results are:

```
C:\book\fortran>ptr_array
9.000000 10.000000 11.000000 12.000000
7.000000
(concluded)
ptr1 = 0
ptr1(3) = 17
```



```
ptr2 => ptr1
DEALLOCATE (ptr1)
WRITE (*,*) ptr2
12. TYPE mytype
    INTEGER, DIMENSION(:), POINTER :: array
END TYPE
TYPE (mytype), DIMENSION(10) :: p
INTEGER :: i, istat
DO i = 1, 10
    ALLOCATE (p(i).array(10), STAT=istat)
    DO j = 1, 10
        p(i)%array(j) = 10*(i-1) + j
    END DO
END DO
WRITE (*,'(10I4)') p(4).array
WRITE (*,'(10I4)') p(7).array(1)
```

9.9 USING POINTERS IN PROCEDURES

Pointers may be used as dummy arguments in procedures and may be passed as actual arguments to procedures. In addition, a function result can be a pointer. The following restrictions apply if pointers are used in procedures:

1. If a procedure has dummy arguments with either the `POINTER` or `TARGET` attributes, then the procedure must have an explicit interface.
2. If a dummy argument is a pointer, then the actual argument passed to the procedure must be a pointer of the same type, kind, and rank.



3. A pointer dummy argument cannot appear in an ELEMENTAL procedure.

It is important to be careful when passing pointers to procedures. As programs get larger and more flexible, we will often get to a situation where pointers are allocated in one procedure, used in others, and finally deallocated and nullified in yet another. In such a complex program, it is very easy to make errors such as attempting to work with disassociated pointers, or allocating new arrays with pointers that are already in use. It is very important that the status results be checked for all ALLOCATE and DEALLOCATE statements, and that the status of pointers be checked using the ASSOCIATED function. When a pointer is used to pass data to a procedure, we automatically know the type of the data associated with the pointer from the type of the pointer itself. If the pointer points to an array, we will know the rank of the array, but not its extent or size. If we need to know the extent or size of the array, then we can use the intrinsic functions LBOUND and UBOUND to determine the bounds of each dimension of the array.

```
ptr1 = 0
```

```
ptr1(3) = 17
```

```
ptr2 => ptr1
```

```
DEALLOCATE (ptr1)
```

```
WRITE (*,*) ptr2
```

12. TYPE mytype

```
INTEGER, DIMENSION(:), POINTER :: array
```

```
END TYPE
```

```
TYPE (mytype), DIMENSION(10) :: p
```

```
INTEGER :: i, istat
```

```
DO i = 1, 10
```

```
ALLOCATE (p(i).array(10), STAT=istat)
```

```
DO j = 1, 10
```



```
p(i)%array(j) = 10*(i-1) + j
```

```
END DO
```

```
END DO
```

```
WRITE (*,'(10I4)') p(4).array
```

```
WRITE (*,'(10I4)') p(7).array(1)
```

9.9.1 Using the INTENT Attribute with Pointers

If the INTENT attribute appears on a pointer dummy argument, it refers to the pointer and not to its target. Thus, if a subroutine has the following declaration

```
SUBROUTINE test(xval)
```

```
REAL, POINTER, DIMENSION(:), INTENT(IN) :: xval
```

```
...
```

then the pointer xval cannot be allocated, deallocated, or reassigned within the subroutine. However, the contents of the pointer's target can be changed. Therefore, the statement

```
xval(90:100) = -2.
```

would be legal within this subroutine if the target of the pointer has at least 100 elements.

9.9.2 Pointer-Valued Functions

It is also possible for a function to return a pointer value. If a function is to return a pointer, then the RESULT clause must be used in the function definition, and the RESULT variable must be declared to be a pointer.

Example

A pointer-valued function.

```
FUNCTION every_fifth (ptr_array) RESULT (ptr_fifth)
```

```
!
```



```

! Purpose:
! To produce a pointer to every fifth element in an
! input rank one array.
!
!
IMPLICIT NONE
! Data dictionary: declare calling parameter types & definitions
INTEGER, DIMENSION(:), POINTER :: ptr_array
INTEGER, DIMENSION(:), POINTER :: ptr_fifth
! Data dictionary: declare local variable types & definitions
INTEGER :: low ! Array lower bound
INTEGER :: high ! Array upper bound
low = LBOUND(ptr_array,1)
high = UBOUND(ptr_array,1)
ptr_fifth =>ptr_array(low:high:5)
END FUNCTION every_fifth

```

A pointer-valued function must always have an explicit interface in any procedure that uses it. The explicit interface may be specified by an interface or by placing the function in a module and then using the module in the procedure. Once the function is defined, it can be used any place that a pointer expression can be used. For example, it can be used on the right-hand side of a pointer assignment statement as follows:

```
ptr_2 =>every_fifth( ptr_1 )
```

The function can also be used in a location where an integer array is expected. In that case, the pointer returned by the function will automatically be dereferenced, and the values pointed to will be used. Thus, the following statement is legal, and will print out



the values pointed to by the pointer returned from the function.

```
WRITE (*,*) every_fifth( ptr_1 )
```

As with any function, a pointer-valued function cannot be used on the left-hand side of an assignment statement.

Check Your Progress B:

Fill in the blanks:

1. Pointer can point to an array as well as a
2. The most powerful feature of pointer is to
3. User can define own type called

9.10 PROCEDURE POINTERS

It is also possible for a Fortran pointer to refer to a procedure instead of a variable or array. A procedure pointer is declared by the statement:

```
PROCEDURE (proc), POINTER :: p => NULL()
```

This statement declares a pointer to a procedure that has the same calling sequence as procedure proc, which must have an explicit interface. Once a procedure pointer is declared, a procedure can be assigned to it in the same

fashion as for variables or arrays. For example, suppose that subroutine sub1 has an explicit interface. Then a pointer to sub1 could be declared as

```
PROCEDURE (sub1), POINTER :: p => NULL()
```

and the following assignment would be legal

```
p => sub1
```

After such an assignment, the following two subroutine calls are identical, producing exactly the same results.

```
CALL sub1(a, b, c)
```



```
CALL p(a, b, c)
```

Note that this pointer will work for any subroutine that has the same interface as sub1. For example, suppose that subroutines sub1 and sub2 both have the same interface (number, sequence, type, and intent of calling parameters). Then the first call to p below would call sub1 and the second one would call sub2.

```
p => sub1
```

```
CALL p(a, b, c)
```

```
p => sub2
```

```
CALL p(a, b, c)
```

This program declares three functions with the same signature in a module so that they have an explicit interface. The main program declares a procedure pointer of type func1, and it is useable with any function having the same signature as func1. The program assigns a function to the pointer based on user selection, and then evaluates the function using the pointer.

EXAMPLE

A program to store a database of names and phone numbers in a binary tree structure, and to retrieve a selected item from that tree.

```
MODULE test_functions
```

```
!
```

```
! Purpose:
```

```
! Module containing test functions. The module creates
```

```
! an explicit interface for the functions.
```

```
(continued)
```

```
!
```

```
!
```

```
IMPLICIT NONE
```



CONTAINS

! All of the following functions have the same signature,
! and they have an explicit interface because they are
! contained in a module.

```
REAL FUNCTION func1(x)
```

```
IMPLICIT NONE
```

```
REAL,INTENT(IN) :: x
```

```
func1 = x**2 - 2*x + 4
```

```
END FUNCTION func1
```

```
REAL FUNCTION func2(x)
```

```
IMPLICIT NONE
```

```
REAL,INTENT(IN) :: x
```

```
func2 = exp(-x/5) * sin(2*x)
```

```
END FUNCTION func2
```

```
REAL FUNCTION func3(x)
```

```
IMPLICIT NONE
```

```
REAL,INTENT(IN) :: x
```

```
func3 = cos(x)
```

```
END FUNCTION func3
```

```
END MODULE test_functions
```

```
PROGRAM test_function_pointers
```

```
!
```

```
! Purpose:
```



```
! To test Fortran procedure pointers. The function
! point will work with any procedure with an explicit
! interface that has same signature as "func1".
!
!
USE test_functions
IMPLICIT NONE
! Declare variables
INTEGER :: index ! Selection index
PROCEDURE(func1), POINTER :: p ! Function pointer
REAL :: x ! Calling argument
(concluded)
! Get the name of the file containing the input data.
WRITE (*,*) 'Select a function to associate with the pointer:'
WRITE (*,*) ' 1: func1'
WRITE (*,*) ' 2: func2'
WRITE (*,*) ' 3: func3'
READ (*,*) index
! Is it valid?
IF ( (index < 1) .OR. (index > 3) ) THEN
  WRITE (*,*) 'Invalid selection made!'
  ERROR STOP 'Bad index'
ELSE
```



```
! Associate the pointer

SELECT CASE (index)

CASE (1)
WRITE (*,*) 'func1 selected...'

p => func1

CASE (2)
WRITE (*,*) 'func2 selected...'

p => func2

CASE (3)
WRITE (*,*) 'func3 selected...'

p => func3

END SELECT

! Execute the function

WRITE (*,'(A)',ADVANCE='NO') 'Enter x: '

READ (*,*) x

WRITE (*,'(A,F13.6)') 'f(x) = ', p(x)

END IF

END PROGRAM test_function_pointers
```

When this program is executed, the results are:

```
C:\book\fortran>test_function_pointers
```

Select a function to associate with the pointer:

1: func1

2: func2



3: func3

3

func3 selected...

Enter x: 3.14159

f(x) = -1.000000

Since $\cos(\pi) = -1$, this is the correct answer.

Procedure pointers are very useful in Fortran programs, because a user can associate a specific procedure with a defined data type. For example, the following type declaration includes a pointer to a procedure that can invert the matrix declared in the derived data type.

```
TYPE matrix(m,n)
```

```
  INTEGER, LEN :: m,n
```

```
  REAL :: element(m,n)
```

```
  PROCEDURE (lu), POINTER :: invert
```

```
END TYPE
```

```
:
```

```
TYPE(m=10,n=10) :: a
```

```
:
```

```
CALL a%invert(...)
```

Note that this is different from binding the procedure to the data type in that binding is permanent, while the procedure pointed to by the function pointer can change during the course of program execution.

9.11 SUMMARY:

In Fortran, a pointer is a data object that has more functionalities than just storing the memory address. It contains more information about a particular object, like type, rank, extents, and memory address. A



pointer to an array must declare the type and the rank of the array that it will point to, but does not declare the extent in each dimension. Users can define their own data type called derived type.

9.12 KEYWORD:

Pointers: Pointer is a data object that has more functionalities than just storing the memory address.

Derived type: Users can define their own data type.

Pointer with Array: A pointer to an array must declare the type and the rank of the array that it will point to, but does not declare the extent in each dimension.

Answer to check your Progress

Check Your Progress A

- 1 Pointer
- 2 data object
- 3 Allocate

Check Your Progress B

- 1 scalar
- 2 create dynamic variables
- 3 derived type

9.13 SELF ASSESSMENT QUESTION:

- 1)What is Pointer?
- 2)Describe how to define and allocate space to the Pointer?
- 3) Describe how to dynamically allocate memory to pointers variables?
- 4) Explain derived data types in brief?

9.14 SUGGESTED READINGS

1. Computer Programming in Fortran 90 and 95 by V Rajaraman



2. Introduction to Programming with Fortran by Chivers, Ian, Sleightholme, Jane
3. Fortran For Scientists and Engineers by Stephen J. Chapman, McGraw-Hill



Class : M.Sc. (Mathematics)

Course Code : MAL 516

Subject : Programming with FORTRAN (Theory)

CHAPTER-10

FILE PROCESSING

STRUCTURE

- 10.0 Objectives
- 10.1 File Introduction
- 10.2 Opening and closing File
 - 10.2.1 Open Statement
 - 10.2.2 Close Statement
 - 10.2.3 Read & Write to Disk File
 - 10.2.4 IOSTAT= and IOMSG= Clauses in the READ Statement
 - 10.2.5 File Positioning
- 10.3 Reading from and Writing into the File
- 10.4 Summary
- 10.5 Keywords
- 10.6 Self-Assessment Questions
- 10.7 Suggested Readings

10.0 Objective:

After reading this lesson, you should be able to:



- 1) Understand how to open and close the file.
- 2) Understand how to read and write into the file.
- 3) Understand how to declare a pointer variable.
- 4) Understand how to allocate memory to the pointers.

10.1 File Introduction:

Fortran allows you to read data from, and write data into files. The programs that we have written up to now have involved relatively small amounts of input and output data. We have typed in the input data from the keyboard each time that a program has been run, and the output data has gone directly to a terminal or printer. This is acceptable for small data sets, but it rapidly becomes prohibitive when working with large volumes of data. Imagine having to type in 100,000 input values each time a program is run! Such a process would be both time consuming and prone to typing errors. We need a convenient way to read in and write out large data sets, and to be able to use them repeatedly without retyping. Fortunately, computers have a standard structure for holding data that we will be able to use in our programs. This structure is called a file. A file consists of many lines of data that are related to each other, and that can be accessed as a unit. Each line of information in a file is called a record. Fortran can read information from a file or write information to a file one record at a time. The files on a computer can be stored on various types of devices, which are collectively known as secondary memory. (The computer's RAM is its primary memory.) Secondary memory is slower than the computer's main memory, but it still allows relatively quick access to the data. Common secondary storage devices include hard disk drives, USB memory sticks, and CDs or DVDs. In the early days of computers, magnetic tapes were the most common type of secondary storage device. Computer magnetic tapes store data in a manner similar to the audio cassette tapes that were used to play music. Like them, computer magnetic tapes must be read (or "played") in order from the beginning of the tape to the end of it. When we read data in consecutive order one record after another in this manner, we are using sequential access. Other devices such as hard disks have the ability to jump from one record to another anywhere within a file. When we jump freely from one record to another following no specific order, we are using direct access. For historical reasons, sequential access is the default access technique in Fortran, even if we are working with devices capable of direct access. To use files within a Fortran



program, we will need some way to select the desired file and to read from or write to it. Fortunately, Fortran has a wonderfully flexible method to read a file and write results into files, whether they are on disk, magnetic tape, or some other device attached to the computer. This mechanism is known as the input/output unit (i/o unit, sometimes called a “logical unit”, or simply a “unit”). The i/o unit corresponds to the first asterisk in the READ (*,*) and WRITE (*,*) statements. If that asterisk is replaced by an i/o unit number, then the corresponding read or write will be to the device assigned to that unit instead of to the standard input or output device. The statements to read or write any file or device attached to the computer are exactly the same except for the i/o unit number in the first position, so we already know most of what we need to know to use file i/o. An i/o unit number must be of type INTEGER. Several Fortran statements may be used to control disk file input and output.

Fortran file control statements

I/O statement Function

OPEN Associate a specific disk file with a specific i/o unit number.

CLOSE End the association of a specific disk file with a specific i/o unit number.

READ Read data from a specified i/o unit number.

WRITE Write data to a specified i/o unit number.

REWIND Move to the beginning of a file.

BACKSPACE Move back one record in a file.

I/O unit numbers are assigned to disk files or devices using the OPEN statement, and detached from them using the CLOSE statement. Once a file is attached to an i/o unit using the OPEN statement, we can read and write in exactly the same manner that we have already learned. When we are through with the file, the CLOSE statement closes the file and releases the i/o unit to be assigned to some other file. The REWIND and BACKSPACE statements may be used to change the current reading or writing position in a file while it is open. Certain unit numbers are pre-defined to be connected to certain input or output devices, so that we don't need an OPEN statement to use these devices. These predefined units vary from processor to processor.⁵ Typically, i/o unit 5 is pre-defined to be the standard input device for your program (i.e., the keyboard if you are running at a terminal, or the input batch file if you



are running in batch mode). Similarly, i/o unit 6 is usually pre-defined to be the standard output device for your program (the screen if you are running at a terminal, or the line printer if you are running in batch mode). These assignments date back to the early days of Fortran on IBM computers, so they have been copied by most other vendors in their Fortran compilers. Another common association is i/o unit 0 for the standard error device for your program. This assignment goes back to the C language and Unix-based computers. However, you cannot count on any of these associations always being true for every processor. If you need to read from and write to the standard devices, always use the asterisk instead of the standard unit number for that device. The asterisk is guaranteed to work correctly on any computer system

10.2 Opening and Closing Files

Before using a file you must open the file. The open command is used to open files for reading or writing. The simplest form of the command is –

```
open (unit = number, file = "name").
```

However, the open statement may have a general form –

```
open (list-of-specifiers)
```

10.2.1 Open Statement:

The OPEN statement associates a file with a given i/o unit number. Its form is

```
OPEN (specified_list)
```

where specified_list contains a series of clauses specifying the i/o unit number, the file name, and information about how to access the file. The clauses in the list are separated by commas. The full list of possible clauses in the OPEN statement will be postponed until. For now, we will introduce only the six most important items from the list. They are

S.no	Specifier	Description	Syntax
1.	UNIT	UNIT clause indicating the i/o unit number to associate with this file	UNIT=int-expr where int_expr can be a nonnegative integer value.



2.	FILE	FILE clause specifying the name of the file to be opened.	FILE= char_expr where char_expr is a character value containing name of the file to be opened.
3.	STATUS	STATUS clause specifying the status of the file to be opened.	STATUS= char_expr where char_expr is one of the following: 'OLD', 'NEW', 'REPLACE', 'SCRATCH', or 'UNKNOWN'.
4.	ACTION	ACTION clause specifying whether a file is to be opened for reading only, for writing only, or for both reading and writing.	ACTION= char_expr where char_expr is one of the following: 'READ', 'WRITE', or 'READWRITE'. If no action is specified, the file is opened for both reading and writing.
5.	IOSTAT	IOSTAT clause specifying the name of an integer variable in which the status of the open operation can be returned.	IOSTAT= int_var where int_var is an integer variable. If the OPEN statement is successful, a 0 will be returned in the integer variable. If it is not successful, a positive number corresponding to a system error message will be returned in the variable. The system error messages vary from processor to processor, but a zero always means success.
6.	IOMSG	IOMSG clause specifying the name of a character variable that will contain a message if an error occurs.	IOMSG= char_var where char_var is a character variable. If the OPEN statement is successful, the contents of the character variable will be unchanged. If it is not successful, a



			descriptive error message will be returned in this string correct OPEN statements are shown below.
--	--	--	--

1. Opening a File for Input

The statement below opens a file named EXAMPLE.DAT and attaches it to i/o unit 8.

```
INTEGER :: ierror
```

```
OPEN (UNIT=8, FILE='EXAMPLE.DAT', STATUS='OLD', ACTION='READ', &
      IOSTAT=ierror, IOMSG=err_string)
```

The STATUS='OLD' clause specifies that the file already exists; if it does not exist, then the OPEN statement will return an error code in variable ierror, and an error message in character string err_string. This is the proper form of the OPEN statement for an input file. If we are opening a file to read input data from, then the file had better be present with data in it! If it is not there, something is obviously wrong. By checking the returned value in error, we can tell that there is a problem and take appropriate action.

The ACTION='READ' clause specifies that the file should be read-only. If an attempt is made to write to the file, an error will occur. This behaviour is appropriate for an input file.

2. Opening a File for Output

```
INTEGER :: unit, ierror
```

```
CHARACTER(len=6) :: filename
```

```
unit = 25
```

```
filename = 'OUTDAT'
```

```
OPEN (UNIT=unit, FILE=filename, STATUS='NEW', ACTION='WRITE', &
      IOSTAT=ierror, IOMSG=err_string)
```



or

```
OPEN (UNIT=unit, FILE=filename, STATUS='REPLACE', ACTION='WRITE', &  
      IOSTAT=ierror, IOMSG=err_string)
```

The STATUS='NEW' clause specifies that the file is a new file; if it already exists, then the OPEN statement will return an error code in variable ierror. This is the proper form of the OPEN statement for an output file if we want to make sure that we don't overwrite the data in a file that already exists.

The STATUS='REPLACE' clause specifies that a new file should be opened for output whether a file by the same name exists or not. If the file already exists, the program will delete it, create a new file, and open it for output. The old contents of the file will be lost. If it does not exist, the program will create a new file by that name and open it. This is the proper form of the OPEN statement for an output file if we want to open the file whether or not a previous file exists with the same name.

The ACTION='WRITE' clause specifies that the file should be write-only. If an attempt is made to read from the file, an error will occur. This behavior is appropriate for an output file.

3. Opening a Scratch File

```
OPEN (UNIT=12, STATUS='SCRATCH', IOSTAT=ierror)
```

A scratch file is a temporary file that is created by the program, and that will be deleted automatically when the file is closed or when the program terminates. This type of file may be used for saving intermediate results while a program is running, but it may not be used to save anything that we want to keep after the program finishes. Notice that no file name is specified in the OPEN statement. In fact, it is an error to specify a file name with a scratch file. Since no ACTION= clause is included, the file has been opened for both reading and writing.

9.2.2 CLOSE Statement

The CLOSE statement closes a file and releases the i/o unit number associated with it.

Its form is

```
CLOSE (close_list)
```



where `close_list` must contain a clause specifying the i/o number, and may specify other options that will be discussed with the advanced i/o material. If no `CLOSE` statement is included in the program for a given file, that file will be closed automatically when the program terminates. After a nonscratch file is closed, it may be reopened at any time using a new `OPEN` statement. When it is reopened, it may be associated with the same i/o unit or with a different i/o unit. After the file is closed, the i/o unit that was associated with it is free

to be reassigned to any other file in a new `OPEN` statement.

The following table describes the most commonly used specifiers –

Sr.No	Specifier & Description
1	<p>[UNIT=] u</p> <p>The unit number <code>u</code> could be any number in the range 9-99999 and it indicates the file, you may choose any number but every open file in the program must have a unique number</p>
2	<p>IOSTAT= ios</p> <p>It is the I/O status identifier and should be an integer variable. If the open statement is successful then the <code>ios</code> value returned is zero else a non-zero value.</p>
3	<p>ERR = err</p> <p>It is a label to which the control jumps in case of any error.</p>
4	<p>FILE = fname</p> <p>File name, a character string.</p>
5	<p>STATUS = sta</p> <p>It shows the prior status of the file. A character string and can have one of the three</p>



	values NEW, OLD or SCRATCH. A scratch file is created and deleted when closed or the program ends.
6	ACCESS = acc It is the file access mode. Can have either of the two values, SEQUENTIAL or DIRECT. The default is SEQUENTIAL.
7	FORM = frm It gives the formatting status of the file. Can have either of the two values FORMATTED or UNFORMATTED. The default is UNFORMATTED
8	RECL = rl It specifies the length of each record in a direct access file.

After the file has been opened, it is accessed by read and write statements. Once done, it should be closed using the close statement.

The close statement has the following syntax –

```
close ([UNIT = ]u[,IOSTAT = ios, ERR = err, STATUS = sta])
```

Please note that the parameters in brackets are optional.

Example

This example demonstrates opening a new file for writing some data into the file.

```
program outputdata
```

```
implicit none
```

```
real, dimension(100) :: x, y
```

```
real, dimension(100) :: p, q
```



```
integer :: i

! data
do i=1,100
  x(i) = i * 0.1
  y(i) = sin(x(i)) * (1-cos(x(i)/3.0))
end do

! output data into a file
open(1, file = 'data1.dat', status = 'new')
do i=1,100
write(1,*) x(i), y(i)
end do

close(1)
end program outputdata
```

When the above code is compiled and executed, it creates the file data1.dat and writes the x and y array values into it. And then closes the file.

10.2.3 READs and WRITEs to Disk Files

Once a file has been connected to an i/o unit via the OPEN statement, it is possible to read from or write to the file using the same READ and WRITE statements that we have been using. For example, the statements

```
OPEN (UNIT=8, FILE='INPUT.DAT',STATUS='OLD',I0STAT=ierror)
READ (8,*) x, y, z
```



will read the values of variables x, y, and z in free format from the file INPUT.DAT, and the statements

```
OPEN (UNIT=9, FILE='OUTPUT.DAT',STATUS='REPLACE',IOSTAT=ierror)
```

```
WRITE (9,100) x, y, z
```

```
100 FORMAT (' X = ', F10.2, ' Y = ', F10.2, ' Z = ', F10.2 )
```

will write the values of variables x, y, and z to the file OUTPUT.DAT in the specified format.

10.2.4 The IOSTAT= and IOMSG= Clauses in the READ Statement

The IOSTAT= and IOMSG= clauses are important additional features that may be added to the READ statement when working with disk files. The form of the IOSTAT=clause is

```
IOSTAT= int_var
```

where int_var is an integer variable. If the READ statement is successful, a 0 will be returned in the integer variable. If it is not successful due to a file or format error, a positive number corresponding to a system error message will be returned in the variable. If it is not successful because the end of the input data file has been reached, a negative number will be returned in the variable.

If an IOMSG= clause is included in a READ statement and the returned i/o status is nonzero, then the character string returned by the IOMSG= clause will explain in words what went wrong. The program should be designed to display this message to the user.

If no IOSTAT= clause is present in a READ statement, any attempt to read a line beyond the end of a file will abort the program. This behaviour is unacceptable in a well-designed program. We often want to read all of the data from a file until the end is reached, and then perform some sort of processing on that data. This is where the IOSTAT= clause comes in: If an IOSTAT= clause is present, the program will not abort on an attempt to read a line beyond the end of a file. Instead, the READ will complete with the IOSTAT variable set to a negative number. We can then test the value of the variable, and process the data accordingly.

10.2.5 File Positioning

As we stated previously, ordinary Fortran files are sequential—they are read in order from the first record in the file to the last record in the file. However, we sometimes need to read a piece of data more



than once, or to process a whole file more than once during a program. How can we skip around within a sequential file? Fortran provides two statements to help us move around within a sequential file. They are the BACKSPACE statement, which moves back one record each time it is called, and the REWIND statement, which restarts the file at its beginning. The forms of these statements are

BACKSPACE (UNIT=unit)

and

REWIND (UNIT=unit)

where unit is the i/o unit number associated with the file that we want to work with. Both statements can also include IOSTAT= and IOMSG= clauses to detect errors during the backspace or rewind operation without causing the program to abort.

Example:

Using File Positioning Commands:

We will now illustrate the use of scratch files and file positioning commands in a simple example problem. Write a program that accepts a series of nonnegative real values and stores them in a scratch file. After the data is input, the program should ask the user what data record he or she is interested in, and then recover and display that value from the disk file.

Solution

Since the program is expected to read only positive or zero values, we can use a negative value as a flag to terminate the input to the program. This program opens a scratch file, and then reads input values from the user. If a value is nonnegative, it is written to the scratch file. When a negative value is encountered, the program asks the user for the record to display. It checks to see if a valid record number was entered. If the record number is valid, it rewinds the file and reads forward to that record number. Finally, it displays the contents of that record to the user.

Example

Sample program illustrating the use of file positioning commands.

```
PROGRAM scratch_file
```



```
!  
!  
! Purpose:  
!  
! To illustrate the use of a scratch file and positioning  
! commands as follows:  
!  
! 1. Read in an arbitrary number of positive or zero  
! values, saving them in a scratch file. Stop  
! reading when a negative value is encountered.  
!  
! 2. Ask the user for a record number to display.  
!  
! 3. Rewind the file, get that value, and display it.  
!  
!  
!  
IMPLICIT NONE  
!  
! Data dictionary: declare constants  
INTEGER, PARAMETER :: LU = 8 ! i/o unit for scratch file  
!  
! Data dictionary: declare variable types, definitions, & units  
REAL :: data ! Data value stored in a disk file  
!  
INTEGER :: icount = 0 ! The number of input data records  
INTEGER :: irec ! Record number to recover and display  
!  
INTEGER :: j ! Loop index  
!  
! Open the scratch file  
OPEN (UNIT=LU, STATUS='SCRATCH')  
!  
! Prompt user and get input data.  
WRITE (*, 100)
```



```
100 FORMAT ('Enter positive or zero input values. ',/, &
'A negative value terminates input. ')
! Get the input values, and write them to the scratch file
DO
WRITE (*, 110) icount + 1 ! Prompt for next value
110      FORMAT ('Enter sample ',I4,':' )
READ (*,*) data ! Read value
IF ( data< 0. ) EXIT ! Exit on negative numbers
icount = icount + 1 ! Valid value: bump count
WRITE (LU,120) data ! Write data to scratch file
120 FORMAT (ES16.6)
END DO
! Now we have all of the records. Ask which record to see.
! icount records are in the file.
WRITE (*,130) icount
130      FORMAT ('Which record do you want to see (1 to ',I4, ')? ')
READ (*,*) irec
! Do we have a legal record number? If so, get the record.
! If not, tell the user and stop.
IF ( (irec>= 1) .AND. (irec<= icount) ) THEN
! This is a legal record. Rewind the scratch file.
REWIND (UNIT=LU)
! Read forward to the desired record.
```



```
DO j = 1, irec
READ (LU,*) data
END DO
! Tell user.
WRITE (*,140) irec, data
140 FORMAT ('The value of record ', I4, ' is ', ES14.5 )
ELSE
! We have an illegal record number. Tell user.
WRITE (*,150) irec
150 FORMAT ('Illegal record number entered: ', I8)
! Close file
CLOSE(LU)
END PROGRAM scratch_file
```

Let us test the program with valid data:

```
C:\book\fortran>scratch_file
```

Enter positive or zero input values.

A negative input value terminates input.

Enter sample 1:

234.

Enter sample 2:

12.34

Enter sample 3:

0.



Enter sample 4:

16.

Enter sample 5:

11.235

Enter sample 6:

2.

Enter sample 7:

-1

Which record do you want to see (1 to 6)?

5

The value of record 5 is 1.12350E+01

Next, we should test the program with an invalid record number to see that the error condition is handled properly.

```
C:\book\fortran>scratch_file
```

Enter positive or zero input values.

A negative input value terminates input.

Enter sample 1:

234.

Enter sample 2:

12.34

Enter sample 3:

0.

Enter sample 4:



16.

Enter sample 5:

11.235

Enter sample 6:

2.

Enter sample 7:

-1

Which record do you want to see (1 to 6):

7

Illegal record number entered: 7

The program appears to be functioning correctly.

10.3 Reading from and Writing into a File:

The read and write statements respectively are used for reading from and writing into a file respectively.

They have the following syntax –

```
read ([UNIT = ]u, [FMT = ]fmt, IOSTAT = ios, ERR = err, END = s)
```

```
write([UNIT = ]u, [FMT = ]fmt, IOSTAT = ios, ERR = err, END = s)
```

Most of the specifiers have already been discussed.

The END = s specifier is a statement label where the program jumps, when it reaches end-of-file.

Example

This example demonstrates reading from and writing into a file.

In this program we read from the file, we created in the last example, data1.dat, and display it on screen.

program outputdata



implicit none

```
real, dimension(100) :: x, y
```

```
real, dimension(100) :: p, q
```

```
integer :: i
```

```
! data
```

```
do i = 1,100
```

```
    x(i) = i * 0.1
```

```
    y(i) = sin(x(i)) * (1-cos(x(i)/3.0))
```

```
end do
```

```
! output data into a file
```

```
open(1, file = 'data1.dat', status='new')
```

```
do i = 1,100
```

```
write(1,*) x(i), y(i)
```

```
end do
```

```
close(1)
```

```
! opening the file for reading
```

```
open (2, file = 'data1.dat', status = 'old')
```

```
do i = 1,100
```

```
read(2,*) p(i), q(i)
```



```
end do  
close(2)  
do i = 1,100  
write(*,*) p(i), q(i)  
end do  
end program outputdata
```

When the above code is compiled and executed, it produces the following result –

Example:

It is very common to read a large data set into a program from a file, and then to process the data in some fashion. Often, the program will have no way of knowing in advance just how much data is present in the file. In that case, the program needs to read the data in a while loop until the end of the data set is reached, and then must detect that there is no more data to read. Once it has read in all of the data, the program can process it in whatever manner is required.

Let's illustrate this process by writing a program that can read in an unknown number of real values from a disk file, and detect the end of the data in the disk file.

Solution

This program must open the input disk file, and then read the values from it using the IOSTAT= clause to detect problems. If the IOSTAT variable contains a negative number after a READ, then the end of the file has been reached. If the IOSTAT variable contains 0 after a READ, then everything was ok. If the IOSTAT variable contains a positive number after a READ, then a READ error occurred. In this example, the program should stop if a READ error occurs.

1. State the problem.

The problem may be succinctly stated as follows:



Write a program that can read an unknown number of real values from a user-specified input data file, detecting the end of the data file as it occurs.

2. Define the inputs and outputs.

The inputs to this program consist of:

- (a) The name of the file to be opened.
- (b) The data contained in that file.

The outputs from the program will be the input values in the data file. At the end of the file, an informative message will be written out telling how many valid input values were found.

3. Describe the algorithm.

This pseudocode for this program is

Initialize nvals to 0

Prompt user for file name

Get the name of the input file

OPEN the input file

Check for errors on OPEN

If no OPEN error THEN

! Read input data

WHILE

READ value

IF status /= 0 EXIT

nvals ← nvals + 1

WRITE valid data to screen

END of WHILE



```
! Check to see if the WHILE terminated due to end of file
! or READ error
IF status > 0
WRITE 'READ error occurred on line', nvals
ELSE
WRITE number of valid input values nvals
END of IF ( status> 0 )
END of IF (no OPEN error)
END PROGRAM
```

4. Turn the algorithm into Fortran statements.

Program to read an unknown number of values from a user-specified input disk file.

```
PROGRAM read_file
!
! Purpose:
! To illustrate how to read an unknown number of values from
! an input data file, detecting both any formatting errors and
! the end of file.
!
! Record of revisions:
!
IMPLICIT NONE
! Data dictionary: declare variable types, definitions, & units
CHARACTER(len=20) :: filename ! Name of file to open
```



```
CHARACTER(len=80) :: msg ! Error message
INTEGER :: nvals = 0 ! Number of values read in
INTEGER :: status ! I/O status
REAL :: value ! The real value read in
! Get the file name, and echo it back to the user.
WRITE (*,*) 'Please enter input file name: '
READ (*,*) filename
WRITE (*,1000) filename
1000 FORMAT ('The input file name is: ', A)
! Open the file, and check for errors on open.
OPEN (UNIT=3, FILE=filename, STATUS='OLD', ACTION='READ', &
      IOSTAT=status, IOMSG=msg )
openif: IF ( status == 0 ) THEN
  ! OPEN was ok. Read values.
readloop: DO
  READ (3,*,IOSTAT=status) value ! Get next value
  IF ( status /= 0 ) EXIT ! EXIT if not valid.
  nvals = nvals + 1 ! Valid: increase count
  WRITE (*,1010) nvals, value ! Echo to screen
  1010 FORMAT ('Line ', I6, ': Value = ',F10.4 )
END DO readloop
! The WHILE loop has terminated. Was it because of a READ
! error or because of the end of the input file?
```



readif: IF (status > 0) THEN ! a READ error occurred. Tell user.

(concluded)

```
WRITE (*,1020) nvals + 1
1020 FORMAT ('An error occurred reading line ', I6)
ELSE ! the end of the data was reached. Tell user.
WRITE (*,1030) nvals
1030 FORMAT ('End of file reached. There were ', I6, &
' values in the file.')
END IF readif
ELSE openif
WRITE (*,1040) status
1040 FORMAT ('Error opening file: IOSTAT = ', I6 )
WRITE (*,1050) TRIM(msg)
1050 FORMAT (A)
END IF openif
! Close file
CLOSE ( UNIT=3 )
END PROGRAM read_file
```

Note that the input file is opened with STATUS='OLD', since we are reading from the file, and the input data must already exist before the program is executed.

5. Test the program.

To test this program, we will create two input files, one with valid data and one with an input data error. We will run the program with both input files, and verify that it works correctly both for valid data and



for data containing input errors. Also, we will run the program with an invalid file name to show that it can properly handle missing input files.

The valid input file is called READ1.DAT. It contains the following lines:

```
-17.0  
30.001  
1.0  
12000.  
-0.012
```

The invalid input file is called READ2.DAT. It contains the following lines:

```
-17.0  
30.001  
ABCDEF  
12000.  
-0.012
```

Running these files through the program yields the following results:

```
C:\book\fortran>read_file
```

```
Please enter input file name:
```

```
read1.dat
```

```
The input file name is: read1.dat
```

```
Line 1: Value = -17.0000
```

```
Line 2: Value = 30.0010
```

```
Line 3: Value = 1.0000
```

```
Line 4: Value = 12000.0000
```



Line 5: Value = -.0120

End of file reached. There were 5 values in the file.

```
C:\book\fortran>read_file
```

Please enter input file name:

```
read2.dat
```

The input file name is: read2.dat

Line 1: Value = -17.0000

Line 2: Value = 30.0010

An error occurred reading line 3

Finally, let's test the program with an invalid input file name.

```
C:\book\fortran>read_file
```

Please enter input file name:

```
xxx
```

The input file name is: xxx

Error opening file: IOSTAT = 29

file not found, unit 3, file C:\Data\book\fortran\xxx

The number of the IOSTAT error reported by this program will vary from processor to processor, but it will always be positive. You must consult a listing of the runtime error codes for your particular compiler to find the exact meaning of the error code that your computer reports. For the Fortran compiler used here, IOSTAT = 29 means "File not found." Note that the error message returned from the IOMSG clause is clear to the user, without having to look up the meaning of status 29! This program correctly read all of the values in the input file, and detected the end of the data set when it occurred.

Check your Progress A

1 Before using a file you must the file.



2 Open command is used to open files for

3 statement closes a file and releases the i/o unit number associated with it.

10.4 SUMMARY:

Fortran allow you to read and write data from the files. Before reading and writing into the file we must have to open the file. If the file is not opened we cannot read and write into the file. Once the reading and writing operation are done file should be closed. Computers have a standard structure for holding data that we will be able to use in our programs. This structure is called a file. A file consists of many lines of data that are related to each other, and that can be accessed as a unit. Each line of information in a file is called a record. Fortran can read information from a file or write information to a file one record at a time.

10.5 KEYWORD:

File: File allow you to store the data into the file.

Open/ Close: Before inserting the data we must have to open the file after adding or removing the data we must have to close it.

Read/ Write: We have to open a file either for reading or either for writing purpose.

Answer to check your Progress

Check Your Progress A

1 OPEN

2 reading/ writing

3 CLOSE

10.6 SELF ASSESSMENT QUESTION:

1)What is File?

2) How to open and close a file?

3) How to read and write into a file?



10.7 SUGGESTED READINGS

1. Computer Programming in Fortran 90 and 95 by V Rajaraman
2. Introduction to Programming with Fortran by Chivers, Ian, Sleightholme, Jane
3. Fortran For Scientists and Engineers by Stephen J. Chapman, Mcgraw-Hill