

CHAPTER - TWO

INTRODUCTION

what is programming?

- ❖ **Programming:** is a skill that can be acquired by a computer professional that gives him/her the knowledge of making the computer perform the required operation or task. Why do we need to learn computer programming?
- ❖ **Computer programming:** is critical if one wants to know how to make the computer perform a task. Most users of a computer only use the available applications on the computer. These applications are produced by computer programmers. Thus if someone is interested to make such kind of applications, he/she needs to learn how to talk to the computer, which is learning computer programming. What is programming language?
- ❖ **Programming Language:** is a set different category of written symbols that instruct computer hardware to perform specified operations required by the designer. For someone to be a programmer, in addition to basic skills in computer, needs to have the following major skills:
 - **Programming Language Skill:** knowing one or more programming language to talk to the computer and instruct the machine to perform a task.
 - **Problem Solving Skill:** skills on how to solve real world problem and represent the solution in understandable format.
 - **Algorithm Development:** skill of coming up with sequence of simple and human understandable set of instructions showing the step of solving the problem. Those set of steps should not be dependent on any programming language or machine.

In every programming Language, there are sets of rules that govern the symbols used in a programming language. These set of rules determine how the programmer can make the computer hardware to perform a specific operation. These sets of rules are called **syntax**.

Generations of programming language

Programming languages are categorized into five generations: first, second, third, fourth and fifth generation languages. These programming languages can also be categorized into two broad categories: low level and high level languages.

- ❖ Low level languages are machine specific or dependent.

- ❖ High level languages like COBOL, BASIC are machine independent and can run on variety of computers.

From the five categories of programming languages, first and second generation languages are low level languages and the rest are high level programming languages. The higher the level of a language, the easier it is to understand and use by programmers. Languages after the fourth generation are referred to as very high level languages.

First Generation (Machine languages, 1940's):

Difficult to write applications with this language. It's dependent on machine languages of the specific computer being used. Machine languages allow the programmer to interact *directly with the hardware*, and it can be executed by the computer without the need for a translator. It's more powerful in utilizing resources of the computer since it gives power to the programmer. They execute very quickly and use memory very efficiently.

Second Generation (Assembly languages, early 1950's):

Uses mnemonics (symbolic names) for operations and storage locations. A system program called an *assembler* translates a program written in assembly language to machine language. Programs written in assembly language *are not portable*. i.e., different computer architectures have their own machine and assembly languages. They are easier than machine languages. They are *highly used in system software development*.

Third Generation (High level languages, 1950's to 1970's):

Uses English like instructions and mathematicians were able to define variables with statements such as $Z = A + B$. Such languages are much easier to use than assembly language. Programs, written in high level languages, need to be translated into machine language in order to be executed. The use of common words (reserved words) within instructions makes them easier to learn. All third generation programming languages are procedural languages. In procedural languages, the programmer is expected to specify what is required and how to perform it.

Examples: FORTRAN, ALGOL, COBOL, C++, C

Fourth Generation (since late 1970's):

Have a simple, English like syntax rules; commonly used *to access databases*. Fourth generation languages are *non-procedural* languages. The non-procedural method is easier to write, but you have less control over how each task is actually performed. In non-procedural languages, the programmer is

not required to write traditional programming logic. Programmers concentrate on defining the input and output rather than the program steps required.

- For example, a command, such as LIST, might display all the records in a file on screen, separating fields with a blank space. In a procedural language, all the logic for inputting each record, testing for end of file and formatting each column on screen has to be explicitly programmed.

Fourth generation languages have a minimum number of syntax rules. This saves time and free professional programmers for more complex tasks. Some examples of 4GL are structured query languages (SQL), report generators, application generators and graphics languages.

Fifth Generation (1990's):

These are used in artificial intelligence (AI) and expert systems; also used for accessing databases. 5GLs are “natural” languages whose instruction *closely resembles* human speech. E.g. “get me Jones Brown’s sales figure for the 1997 financial year”. 5GLs require very powerful hardware and software because of the complexity involved in interpreting commands in human language.

The evolution of Operating Systems

Early computers were capable of performing only *one job at a time*. This form of computer operation is often called *single-user batch processing*. The computer runs a single program at a time while processing data in groups or batches. As computers became more powerful, it became evident that single-user batch processing rarely utilizes the computer’s resources. Therefore, computer developers thought that many jobs or tasks from different programs or applications could be made to share the resources of the computer to achieve better utilization. This concept is called *multiprogramming*. Multiprogramming involving the “simultaneous” operation of many jobs on the computer. In the 1960’s, many groups, pioneered *timesharing operating systems*. Timesharing is a special case of multiprogramming, in which users access the computers through terminals. Here, the computer does not run the users job paralleling, but shares the CPU’s time.

Basic Program development tips

The program we design in any programming language, need to be:

- **Reliable:** the program should always do what it is expected to do and handle all types of exception.

- **Maintainable:** the program should be in a way that it could be modified and upgraded when the need arises.
- **Portable:** It needs to be possible to adapt the software written for one type of computer to another with minimum modification.
- **Efficient:** the program should be designed to make optimal use of time, space and other resources of the computer.

Algorithm designing and modeling the logic (using flow chart)

Computer solves varieties of problems that can be expressed in a finite number of steps leading to a precisely defined goal by writing different programs. There are system analysis and design tools, particularly *flowchart* and *structure chart*, that can be used to define the problem in terms of the steps to its solution. The programmer uses programming language to communicate the logic of the solution to the computer. Before a program is written, the programmer must clearly understand what data are to be used, the desired result, and the procedure to be used to produce the result. The procedure, or solution, selected is referred to as an algorithm. An algorithm is defined as a step-by-step sequence of instructions that must terminate and describe how the data is to be processed to produce the desired outputs. Simply, algorithm is a sequence of instructions. Algorithms are a fundamental part of computing. There are three commonly used tools to help to document program logic (the algorithm). These are flowcharts, structured chart, and Pseudocode. We will see flowchart and pseudocode.

The algorithm should be:

- ✓ **Precise and unambiguous**
- ✓ **Simple**
- ✓ **Correct**
- ✓ **Efficient**

Pseudocode

Pseudocode (derived from pseudo and code) is a compact and informal high-level description of a computer algorithm that uses the structural conventions of programming languages, but typically omits details such as subroutines, variables declarations and system-specific syntax. The programming language is augmented with natural language descriptions of the details, where convenient, or with compact mathematical notation. The purpose of using pseudocode is that it may be easier for humans to

read it than conventional programming languages, and that it may be a compact and environment-independent generic description of the key principles of an algorithm. No standard for pseudocode syntax exists, since, as a program, pseudocode is not an executable program. As the name suggests, pseudocode generally does not actually obey the syntax rules of any particular language; there is no systematic standard form, although any particular writer will generally borrow the appearance of a particular language.

The programming process is a complicated one. You must first understand the program specifications, and then need to organize your thoughts and create the program. This is a difficult task when the program is not trivial (i.e. easy). You must break the main tasks that must be accomplished into smaller ones in order to be able to eventually write fully developed code. Writing pseudocode will save you time later during the construction & testing phase of a program's development.

Example:

Write a program that obtains two integer numbers from the user. It will print out the sum of those numbers.

Pseudocode:

- Prompt the user to enter the first integer
- Prompt the user to enter a second integer
- Compute the sum of the two user inputs
- Display an output prompt that explains the answer as the sum
- Display the result

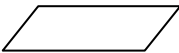
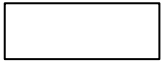
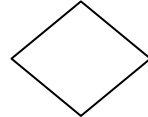
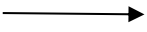
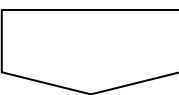
Modeling a programs logic using flow chart

Algorithm could be designed using many techniques and tools. One tool of designing algorithm is by using flowcharts. Flowchart is a graphical way of expressing the steps needed to solve a problem.

A flow chart is a schematic (*diagrammatic description*) representation of a process.

Table 1. Basic flowcharting symbols

Symbol	Name	Function
	Terminal point	Used to represent the start and the end of a program

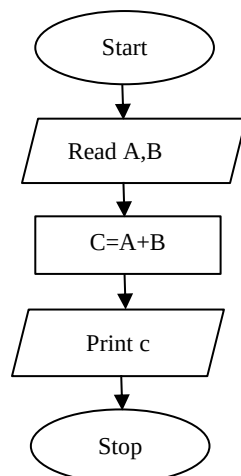
	Input/ Output	Used to represent data input or data output from a computer
	Process	Usually encloses operations (command block) or a group of operations (process)
	Decision	It usually contains a question within it, there are typically two output paths: one if the answer to the question is YES (true), and the other is if the answer is NO (false)
	Flow line	Is used to indicate the direction of logical flow (a path from one operation to another)
	On-page connector	Is used for connection two points in a flow chart without drawing flow lines in one page
	Inter-page connector	It is used as an exit to or any entry from another part of the flowchart on another page

Example 1: - Draw flow chart of an algorithm to add two numbers and display their result.

Algorithm description

- ✓ Read the values of the two numbers (A and B)
- ✓ Add A and B
- ✓ Assign the sum of A and B to C
- ✓ Display the result (c)

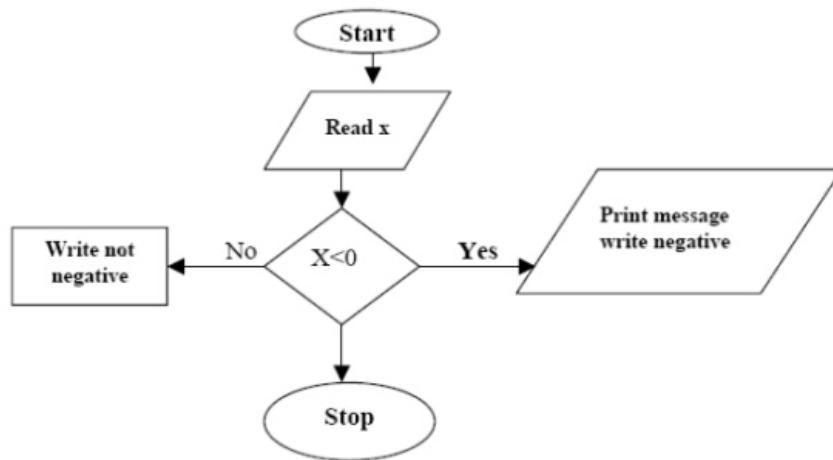
The flow chart is:



Example 2: Write an algorithm description and draw a flow chart to check a number is negative or not.

Algorithm description.

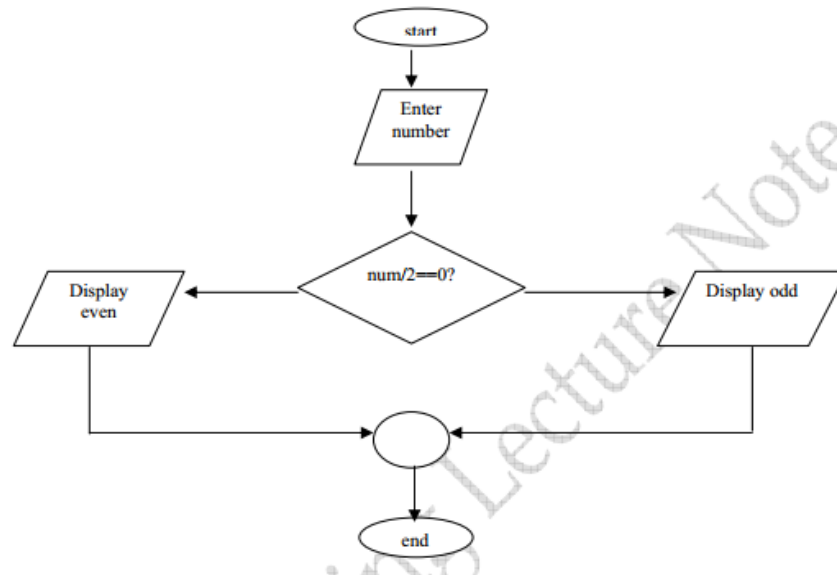
- ✓ Read a number x
- ✓ If x is less than zero write a message negative
- ✓ else write a message not negative



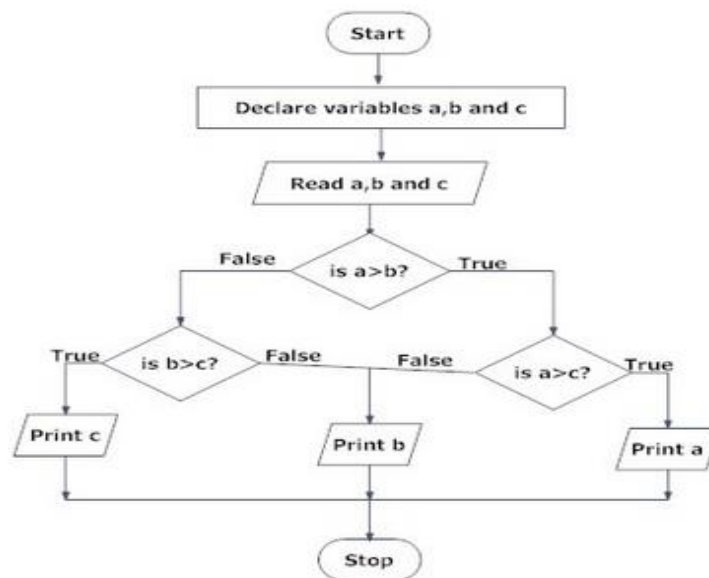
Example 3: Check whether the number is odd or even

Algorithm description.

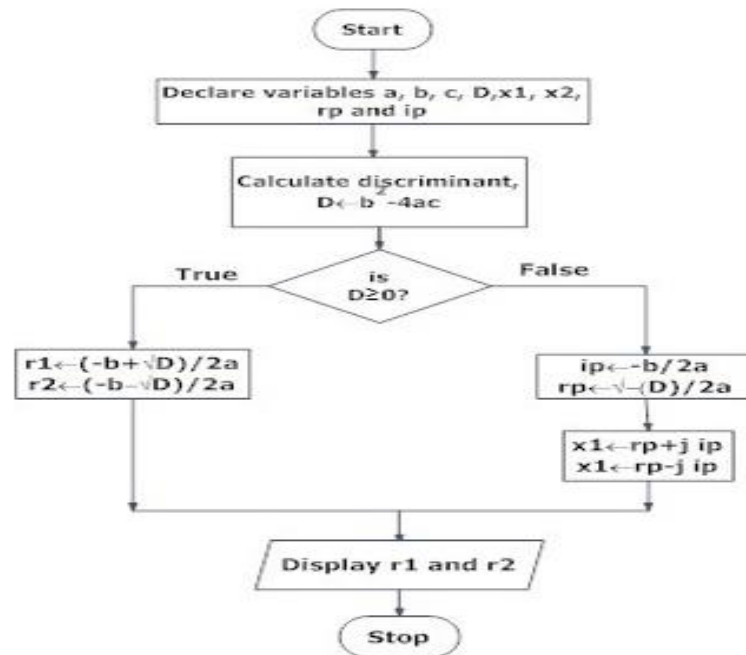
- ✓ Read a number
- ✓ Check the number is divisible by 2 without remainder
- ✓ If the number has no remainder (remainder = 0) display even
- ✓ else display odd



Draw flowchart to find the largest among three different numbers entered by user



Draw a flowchart to find all the roots of a quadratic equation $ax^2+bx+c=0$



Exercise

For each of the problems below, develop a flow chart

- 1) Obtain two numbers from the keyboard, and determine and display which (if either) is the larger of the two numbers.
- 2) Receive 3 numbers and display them in ascending order from smallest to largest.
- 3) Add the numbers from 1 to 100 and display the sum.
- 4) Add the even numbers between 0 and any positive integer number given by the user.
- 5) Find the average of two numbers given by the user.
- 6) Find the average, maximum, minimum, and sum of three numbers given by the user.
- 7) Find the area of a circle where the radius is provided by the user.
- 8) Read 10 integers from the keyboard in the range 0 - 100, and count how many of them are larger than 50, and display this result.
- 9) Take an integer from the user and display the factorial of that number.
- 10) Displays the sum of five numbers entered from user.
- 12) Prints hello world ten times.