**Chapter One**

1. **Introduction to programming**
   1. **Introduction to Computer programming**

**What is Computer?**

* It is an Electronic devices that allows to accept an input(data) from the user or from the environment, process it(i.e. perform Mathematical Expression , Make logical Decision ) based on the instruction given to it , and display an output as a meaningful information or store information for later use

**What is Computer program/software program (in short program)?**

* It is set of instruction that tells the computer the computer what to do. Computer requires  
  programs to function, and a computer programs does nothing unless its instructions are  
  executed by a CPU.

**What is Computer Programming (in short programming or Coding)?**

* Is the process of writing, testing, debugging/troubleshooting, and maintaining the  
  source code of computer programs? Writing computer programs means writing instructions  
  that will make the computer follow and run a program based on those instructions.

**What is Computer Programmer?**

* Is a professional person who write a program?

**Programming Language**

* A language is a systematic set of rules for communicating ideas. With a natural language, like English, the communication is between people and the language is used in both spoken and written. A programming language is a language for communication between a person and computer. The content of communication is known as a program. Programs are set of instructions, which enable a computer to perform a required operation.
* Several hundred programming languages are in use today, each varying in their specific syntax, but showing similarities in purpose. Input/output instructions, arithmetic instructions, logic instructions, control instructions, data movement instructions and specification instructions are such common purpose instructions. Together with the improvement of hardware and processing speed and memory size increase computer languages changed to languages that were easier for humans to understand. So generations of programming languages are developed to correspond with the generations of computer hardware.

**1.2. What skills do we need to be a programmer?**

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**.

**1.3. Generations of Programming Language**

Thus, programming languages fit into five generations: First generation languages, second-generation languages, third generation languages, fourth-generation languages and fifth-generation languages. These generations of programming languages can also be categorized into two broad categories: *-low and high-level languages*. Low-level languages are machine-dependent; that is, they are designed to be run on a particular computer. In contrast, high-level languages (for example, COBOL and BASIC) are machine-independent and can be run on a variety of computers.

The first two generations were low-level and the rest high-level of programming languages. The higher-level languages do not provide us with greater programming capabilities, but they do provide a more sophisticated program/computer interaction with simple instructions. In short, the higher the level of the language, the easier it is to understand and use. For example, in a fourth-generation language you need only instruct the computer system *what to do, not necessarily how to do it.*

When programming in one of the first three generations of languages, you have to tell the computer what to do and how to do it. What comprises a new generation is less clear; therefore, languages after the fourth generation are referred to as *very high-level languages.*

With each new level, **fewer instructions** are needed to tell the computer to perform a particular task. A program written in a second-generation language that computes the total sales for each sales representative, and then lists those over quota, may require 100 or more instructions; the same program in a fourth-generation language may have fewer than 10 instructions.

The ease with which the later generations can be used is certainly appealing, but the earlier languages also have their advantages. All generations of languages are in use today.

**First generation (Machine languages, 1940’s):** Difficult to program in; they are dependent on machine languages of the type of computer being used. However, machine language allows the programmer to interact directly with the hardware, and it can be executed by the computer without the need for a translator.

**Machine language** is the ultimate low-level language. To communicate with the first generation computer programmers had to *write programs in machine language*: - the 0’s and 1’s of binary code. Programming in 0s and 1s needed reducing statements such as add, subtract and divide into a series of 0’s and 1’s. Programs written in this way are machine-dependent. There is no universal machine language as different hardware designs dictate different data and memory locations, and hence different bit strings. Programmer must remember exact memory addresses used to store data and instructions. They *execute very quickly and use memory very efficiently, as no conversion from source code is necessary*. *Writing the programs is difficult, tedious, time consuming and hard to debug*. A high *level of understanding of hardware specifics is required*. All programs written using other languages are translated into machine language before they can be executed.

End users who wanted applications have to work with *specialized programmers who could understand, think and work directly in the machine language* of a particular computer. So *programming in machine is slow and labor intensive process.*

**Second generation (Assembly languages, appeared in the early 1950’s):** Use *symbolic names for operations and storage locations*. Assembly language is *easier to use than machine language but it is still difficult to understand*. A systems program called an **assembler** translates it into machine language. Different computer architectures have their own machine and assembly languages, which means that *programs written in these languages are not portable to other, incompatible systems*. Many programmers still use assembly languages because they still give them close control over the hardware.

* Instead of using 0’s and 1’s programmers substituted language like *acronyms and*

*Words such as add, sub (subtract) in programming statements*.

§        *Costly in terms of programmer time*

§        *Difficult to read and debug*, and

§        *Difficult to learn*

§        *Highly used in system software development*

Programs are *easier to create and to debug than machine language*, but are still *prone to errors and tedious and execute faster than high-level languages*.

**Third generation (High-level languages, emerged from 1950’s - 1970’s):** *Use English-like instructions* like *print* and mathematicians were able to define variables with statements such as Z=A+B. Many high-level languages are **portable**. Each *has syntax rules that must be followed*. Such languages are *much easier to use than assembly language* but they can take considerable time to learn. Two types of systems programs exist for translating them into machine language: *interpreters and compilers.*

In High-level languages, *instructions closely resemble human language and mathematical notation.* Programmers do not require detailed knowledge of hardware specifics as these languages extensively use symbolic representations of memory addresses and library functions. The use of common words (**reserved words**) within instructions makes them easier to learn. Assembly language code segments may be added to most high-level programs, where faster execution speed is required by the programmer. There are two types of language-translator programs used to translate into machine code, Interpreters and Compilers.

**Fourth-generation (Since late 70’s):** Have simple, English-like syntax rules; commonly used to access **databases**. As we mentioned earlier, the third-generation programming languages discussed in the previous section are all **procedural languages** because the programmer must list each step and must use logical control structures to indicate the order in which instructions are to be executed. Fourth-generation languages (4GLs), on the other hand, are **nonprocedural languages**. These languages can be compared to the way in which you might instruct someone to cook a meal. The **nonprocedural method is simply** to state the needed output: fix a meal of chicken, rice, and salad. Using the procedural method, on the other hand, involves specifying each step - from preparing the shopping list to washing the dishes.

Obviously, the nonprocedural method is easier to write, but you have less control over how each task is actually performed. For example, the dishes might be washed by hand or in a dishwasher. When using nonprocedural languages, the methods used and the order in which each task is carried out are left to the language itself; the user does not have any control over it. In addition, 4GLs sacrifice computer efficiency in order to make programs easier to write; hence they require more computer power and processing time. As the power and speed of hardware have increased and its cost has decreased, the use of 4GLs has spread.

Because fourth-generation languages have a minimum number of syntax rules, people who have not been trained as programmers can use such languages to write application programs, as they need them. This saves time and fees professional programmers for more complex tasks. There are several categories of 4GLs languages. The most common once are: *query languages, report generators, and application generators and graphic languages.*

**Query languages** allow the user to retrieve information from databases by following simple syntax rules. For example, you might ask the database to locate all customer accounts that are more than 90 days overdue. Example of query languages is SQL, which has become a de facto standard.

**Report generators** produce customized reports using data stored in a database. The user specifies the data to be in the report, how the report should be formatted, and whether any subtotals and totals are needed. For example, you might ask the system to create a list, arranged by account number, of all the company customers located in Debre Markos.

The user can access the database when using query languages and report generators, but the database cannot be altered with these tools. With **application generators**, however, the user writes programs to allow data to be entered into the database. The program prompts the user to enter the needed data. It also checks the data for validity.

Application generators contain preprogrammed modules that can generate entire applications, greatly seeding development. A user can specify what needs to be done, and the application generator will create the appropriate code for input, validation, update, processing, and reporting.

**Graphic languages** that retrieve data from files or databases and display them in graphic format. Users can ask for data and specify how they are to be charted. Some graphic software can perform arithmetic or logic operations on data as well. SAS, Harvard Graphics, and Lotus Freelance Graphics are popular graphics tools.

**Fifth-generation (1990’s):** Used in artificial intelligence and expert systems; also used for accessing databases. Fifth-generation languages (5GLs) are also nonprocedural languages and are also commonly used to query databases. Because these languages are still in their infancy, only a few are currently commercially available. They are closely tied to artificial intelligence and expert systems.

Fifth-generation languages are the “natural” languages whose instructions closely resemble human speech "Get me Debela’s sales figures for the 1997 financial year" is a typical instruction. Very powerful hardware and software are required to execute such programs because of the complexity involved in interpreting commands entered in human language. This means computers can in the future have the ability to think for themselves and draw their own inferences using programmed information in large databases. Complex processes like understanding speech would appear to be trivial using these fast inferences and would make the software seem highly intelligent.

* 1. ***Problem solving using computers***

**Problems**: Undesirable situations that prevent an organization from fully achieving its purpose, goals and objectives

* True problem situations, either real or anticipated, that require corrective action
* Unexploited Opportunities to improve a situation despite the absence of complaints
* Directives to change a situation regardless of whether anyone has complained about the current situation

**Problem solving** is the process of transforming the description of a problem into the solution of that problem by using our knowledge of the problem domain and by relying on our ability to select and use appropriate problem-solving strategies, techniques, and tools.

**Problem Solving**: is basic intellectual process that has been refined and systemized for the various challenges people face.

There are two approaches of problem solving:

*Top down design*: is a systematic approach based on the concept that the structure of the problem should determine the structure of the solution and what should be done in lower level. This approach will try to disintegrate a larger problem into more smaller and manageable problems to narrow the problem domain.

*Bottom up design*: is the reverse process where the lowest level component are built first and the system builds up from the bottom until the whole process is finally completed.

* + 1. ***Basics of program development***

 The vehicle for the computer solution to problem is a set of explicit and unambiguous instructions called programs expressed in programming language.

 Quality programming is necessary for the economic and correct solution of problems. This only comes from good program design. The programs we design need to be:

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

**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.

* + 1. ***Software Engineering***

"Software Engineering" has been used to describe "building of software systems which are so large or so complex that they are built by a team or teams of engineers".

   Programming Life cycle

The method (approach) that software Engineers use in solving problems in computer science is called the **software development method**. Another name that is commonly used for the software development method is **software life cycle**.

The software life cycle has the following components

1. Preliminary Investigation
2. Analysis
3. Design
4. Implementation
5. Testing
6. Maintenance

**Preliminary Investigation**

* ***Defining the problem*** (examines documents, work papers, and procedures; observe system operations; interview key users of the system). Suggesting a solution (often improving an existing one or building a new information system)
* ***Feasibility study*** (determine whether the solution is feasible)
  + Technical ***feasibility:*** whether implementation is possible with the available or affordable hardware, software and other technical resources.
  + Economic ***feasibility:*** whether the benefits of the proposed solution outweigh the costs
  + Operational ***feasibility:*** whether the proposed solution is desirable within the existing managerial and organizational framework.

**Analysis *(Requirement gathering)***

* Try to understand the business in general (activities done, how it is done, etc)
* Define the specific information requirements: who needs what information, where, when, and how. Present a detail description of the functions the new system must perform
* In this phase we identify:

     Inputs to the problem and their form (Filing status: Single, Married, divorced)

     Outputs expected from the solution and their form (Intermediate results or Final results Completed tax information)

     Constraints (What are the limits on the data? Income cannot be a negative number)

     Assumptions (Problem deals in $US, Only for 2005)

     Formulas (For determining the area, it is length x width)

**Design**

Based on the requirements specified in the analysis phase algorithms are developed at this stage. Develop a series of steps with a logical order which, when applied to the input would produce the specified output.

* Consider alternative technology configurations to develop the system (hardware, software, security capability of the system, network alternatives etc.)
* Management and control of the technical realization of the system: detailed program specification,
* Detail system specifications for the functions identified in the analysis: managerial, organizational and technological components of the system solution (input, output, user interface, database design, processing, manual procedures, controls: input, processes, output and procedural controls; security, documentation, training, organizational change)

##### Implementation

During implementation, we translate (code) each step of the algorithm in the design specifications into a statement in the particular language selected, and end up with a computer program.

**Testing** (Will the system produce the desired results?)

## Unit testing (Program testing): test each unit of program in the system separately

***System testing:*** test the functions of the system as a whole (performance, capacity for file storage, recovery and restart capabilities)

***Acceptance testing:*** system is evaluated by users and management

##### Maintenance

Changes in hardware, software, documentation, or procedures to correct errors, meet new requirements, or improve processing efficiency.

– Fix bugs found by the customer

– Make modifications

– Add new features

**Documentation:** Written descriptions of, specifications, development, and actual code of program. Documentation is valuable because:

  1.     It is probable that you may come back to this program sometime in the future in order to use it again, or you may want to use part of it in developing the solution for another similar problem.

 2.     If you have developed this program for a client's use, he or she will need some information so that in your absence the program can be used.

 3.     If you are working for a company and have developed this program as an addition to the company's software library, some other programmer may be assigned to maintain it or to make additions to it.

 4. It is possible that sometime after you have developed the program, you may discover some errors in it and you may be required to correct them.

The significance of proper program documentation in the software life cycle cannot be overemphasized. Program documentation consists of

1.     A concise problem specification;

2.     Descriptions of inputs, expected outputs, constraints, and applicable formulas;

3.     A pseudo code or flowchart for its algorithm;

4.     A source program listing;

5.     A self-documenting code: a program containing meaningful identifiers as well as clarifying comments;

6.     A hard copy of a sample of data and output of test run of the program; and

7.     A user's guide explaining to nonprogrammer users how the program should be used.

* + 1. **Algorithm development**

An algorithm is procedure for solving a problem in terms of

* the action to execute (what to do) and
* the order in which these actions are executed(done)

 An algorithm needs to be

* Precise and unambiguous (no ambiguity in any instruction and in the order of execution)
* Simple
* Correct
* Finite (has to have an end)
* Produce expected output
* Efficient: in time, memory and other resources

An algorithm can be expressed in many ways. Some of these methods are ***narrative, flowchart and pseudo-code:***

***Narrative***: often used to narrate the algorithm, can be understood by any user who may not have any knowledge of computer programming. Too wordy, too unambiguous and can be interpreted in different ways.

Example: Accept salary of the employee. Calculate bonus as 10% of salary and add it to salary. Accept service year of employee. If the service year is greater than 10, give additional 100 birr as bonus. Display the bonus of the employee.

***Flowcharts:*** a diagram consisting of labeled symbols, together with arrows connecting one symbols to another.

Basic flowcharting symbols are:

***Process:*** is used to compute/calculate a process.

***Terminal point:*** marks the beginning or end of a program

***Process:*** indicates any arithmetic computations or data movement

***Flow line:*** is used to show the direction of logical flow

***Input/output:*** is used to show input or output data

***Decision:*** Indicates a yes/no decision to be made by the program

***On-page Connector:*** used to connect two points without drawing a flow line

***Annotation flag:*** used to add clarifying comments or descriptions

***Inter-page Connector:*** used as exit or entry from a flowchart on one page to a flowchart in another page

***Predefined Process:*** indicates a process defined elsewhere

**Example**: A certain company plan to give a 10% bonus to each of its employees at the end of every year. If an employee has been working 10/more years at the company, she/he is to get an additional birr 100. Draw a flowchart of an algorithm to calculate and print the bonus for a given employee.

**Solution:**

***Analysis***: the problem is to compute bonus for employee. To do so, the salary and number of service years of the employee must be known. Let the salary be denoted by SAL, service be denoted by YEAR and bonus by BONUS. To compute the bonus, we multiply the salary by 10% and assign the result to BONUS. Thus, BONUS = SAL \* 0.10. If the employee has served at least 10 years, birr 100 is added to the computed bonus of the employee and the result is displayed. Otherwise the originally computed bonus is displayed.

***Algorithm:*** uses English to write operations in a group.

1.   Read employee’s salary and year of service

2.   Calculate the employee’s bonus

3. If years of service is greater than or equal to 10 then increase the bonus.

Otherwise only print the bonus

* Most algorithms incorporate three programming conventions

     Sequence

     Decision and

     Repetition

* **Sequence**

STOCK=STOCK+ QUANTITY

* **Decision**

IF HOURS>40

THEN

PERFORM OVERTIME

ELSE

PERFORM REGULAR

ENDIF

* **Repetition**

 INITIALIZE SUM TO ZERO

WHILE SUM<100

GET DATA

ADD DATA TO SUM

ENDWHILE

* 1. **Compiler and Interpreter**
* Any program written in a language other than machine language need to be translated to machine language. The set of instructions that do this task are known as translators.
* There are different kinds of translator software among which Compilers and interpreters are of interest for most programmers.
* ***Compilers***: a compiler is a computer program that translates a series of statements written in source code (a collection of statements in a specific programming language) into a resulting object code (translated instructions of the statements in a programming language). A compiler changes or translates *the whole* source code into executable machine code (also called *object code*) which is *output to a file* for latter execution. E.g.  
  C++, Pascal, FORTRAN, etc
* ***Interpreters:*** is a computer program that translates a single high level statement and executes it and then goes to the next high level language line etc. E.g. QBASIC, Lisp etc.

***Worksheet 1***

For each of the problems below, develop a flow chart

1. Assign your full name and display for the user.
2. Accept full name from the user and display for the user.
3. Receive two number form the user and
   1. Display the largest or the smallest or both
   2. Display their sum , product , reminder , quotient , average
4. Receive a number and determine whether it is odd or even.
5. Obtain two numbers from the keyboard, and determine and display which (if either) is the larger of the two numbers.
6. Receive 3 numbers and display them in ascending order from smallest to largest
7. Add the numbers from 1 to 100 and display the sum
8. Add the even numbers between 0 and any +ve integer number given by the user.
9. Find the average, maximum, minimum, and sum of 3 numbers given by the user.
10. Find the area of a circle where the radius is provided by the user.
11. Swap the contents of two variables using a third variable.
12. Swap the content of two variables without using a third variable.
13. 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.
14. Read N- Number of integer from the keyboard, and display prime number only.
15. Take an integer from the user and display the factorial of that number
16. Read any natural number from the user and count the number of digit.
17. Read any natural number from the user and determine whether it is Armstrong number or not. (153 = 13 + 23 + 53 = 153)
18. Read two number from the user and display the result of ab?