Chapter One

1.1 Introduction to programming

A **Computer** is an electronic device that accepts data, performs computations, and makes logical decisions according to instructions that have been given to it; then produces meaningful information in a form that is useful to the user. In current world we live in, computers are almost used in all walks of life for different purposes. They have been deployed to solve different real life problems, from the simplest game playing up to the complex nuclear energy production. Computers are important and widely used in our society because they are cost-effective aids to problem solving in business, government, industry, education, etc.

In order to solve a given problem, computers must be given the correct instruction about how they can solve it. The terms **computer programs**, **software programs**, or just **programs** are the instructions that tells the computer what to do. Computer requires programs to function, and a computer programs does nothing unless its instructions are executed by a CPU. **Computer programming** (often shortened to **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. Each instruction is relatively simple, yet because of the computer's speed, it is able to run millions of instructions in a second. A computer program usually consists of two elements:

- Data – characteristics
- Code – action

Computer programs (also know as **source code**) is often written by professionals known as **Computer Programmers** (simply programmers). Source code is written in one of programming languages. A **programming language** is an artificial language that can be used to control the behavior of a machine, particularly a computer. Programming languages, like natural language (such as Amharic), are defined by syntactic and semantic rules which describe their structure and meaning respectively. The syntax of a language describes the possible combinations of symbols that form a syntactically correct program. The meaning given to a combination of symbols is handled by semantics. Many programming languages have some form of written specification of their syntax and semantics; some are defined only by an official implementation. In general, programming languages allow humans to communicate instructions to machines.

A main purpose of programming languages is to provide instructions to a computer. As such, programming languages differ from most other forms of human expression in that they require a greater degree of precision and completeness. When using a natural language to communicate with other people, human authors and speakers can be ambiguous and make small errors, and still expect their intent to be understood. However, computers do exactly what they are told to do, and cannot understand the code the programmer "intended" to write. So computers need to be instructed to perform all the tasks. The combination of the language definition, the program, and the program's inputs must fully specify the external behavior that occurs when the program is executed. Computer languages have relatively few, exactly defined, rules for composition of programs, and strictly controlled vocabularies in which unknown words must be defined before they can be used.

Available programming languages come in a variety of forms and types. Thousands of different programming languages have been developed, used, and discarded. Programming languages can be divided in to two major categories: low-level and high-level languages.

Low-level languages

Computers only understand one language and that is binary language or the language of 1s and 0s. Binary language is also known as **machine language**, one of low-level languages. In the initial years of computer programming, all the instructions were given in binary form. Although the computer easily understood these programs, it proved too difficult for a normal human being to remember all the instructions in the form of 0s and 1s. Therefore, computers remained mystery to a common person until other languages such as assembly language was developed, which were easier to learn and understand. **Assembly language** correspondences symbolic instructions and executable machine codes and was created to use letters (called mnemonics) to each machine language instructions to make it easier to remember or write. For example:

ADD A, B – adds two numbers in memory location A and B

Assembly language is nothing more than a symbolic representation of machine code, which allows symbolic designation of memory locations. However, no matter how close assembly language is to machine code, computers still cannot understand it. The assembly language must be translated to machine code by a separate program called *assembler*. The machine instruction created by the assembler from the original program (source code) is called *object code*. Thus assembly languages are unique to a specific computer (machine). Assemblers are written for each unique machine language.

High-level languages

Although programming in assembly language is not as difficult and error prone as stringing together ones and zeros, it is slow and cumbersome. In addition it is hardware specific. The lack of portability between different computers led to the development of high-level languages—so called because they permitted a programmer to ignore many low-level details of the computer's hardware. Further, it was recognized that the closer the syntax, rules, and mnemonics of the programming language could be to "natural language" the less likely it became that the programmer would inadvertently introduce errors (called "bugs") into the program. High-level languages are more English-like and, therefore, make it easier for programmers to "think" in the programming language. High-level languages also require translation to machine language before execution. This translation is accomplished by either a compiler or an interpreter. Compilers translate the entire source code program before execution. Interpreters translate source code programs one line at a time. Interpreters are more interactive than compilers. FORTRAN (FORmula TRANslator), BASIC (Bingers

All Purpose Symbolic Instruction Code), PASCAL, C, C++, Java are some examples of high-level languages.

The question of which language is best is one that consumes a lot of time and energy among computer professionals. Every language has its strengths and weaknesses. For example, FORTRAN is a particularly good language for processing numerical data, but it does not lend itself very well to organizing large programs. Pascal is very good for writing well-structured and readable programs, but it is not as flexible as the C programming language. C++ embodies powerful object-oriented features

As might be expected in a dynamic and evolving field, there is no single standard for classifying programming languages. Another most fundamental ways programming languages are characterized (categorized) is by programming paradigm. A **programming paradigm** provides the programmer's view of code execution. The most influential paradigms are examined in the next three sections, in approximate chronological order.

Procedural Programming Languages

Procedural programming specifies a list of operations that the program must complete to reach the desired state. Each program has a starting state, a list of operations to complete, and an ending point. This approach is also known as imperative programming. Integral to the idea of procedural programming is the concept of a procedure call.
Procedures, also known as functions, subroutines, or methods, are small sections of code that perform a particular function. A procedure is effectively a list of computations to be carried out. Procedural programming can be compared to unstructured programming, where all of the code resides in a single large block. By splitting the programmatic tasks into small pieces, procedural programming allows a section of code to be re-used in the program without making multiple copies. It also makes it easier for programmers to understand and maintain program structure.
Two of the most popular procedural programming languages are FORTRAN and BASIC.

Structured Programming Languages

Structured programming is a special type of procedural programming. It provides additional tools to manage the problems that larger programs were creating. Structured programming requires that programmers break program structure into small pieces of code that are easily understood. It also frowns upon the use of global variables and instead uses variables local to each subroutine. One of the well-known features of structural programming is that it does not allow the use of the GOTO statement. It is often associated with a "top-down" approach to design. The top-down approach begins with an initial overview of the system that contains minimal details about the different parts. Subsequent design iterations then add increasing detail to the components until the design is complete.
The most popular structured programming languages include C, Ada, and Pascal.

Object-Oriented Programming Languages

Object-oriented programming is one the newest and most powerful paradigms. In object-oriented programs, the designer specifies both the data structures and the types of operations that can be applied to those data structures. This pairing of a piece of data with the operations that can be performed on it is known as an object. A program thus becomes a collection of cooperating objects, rather than a list of instructions. Objects can store state information and interact with other objects, but generally each object has a distinct, limited role.

1.2 Problem solving Techniques

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. A program is not needed only to solve a problem but also it should be reliable, (maintainable) portable and efficient. In computer programming two facts are given more weight:

- The first part focuses on defining the problem and logical procedures to follow in solving it.
- The second introduces the means by which programmers communicate those procedures to the computer system so that it can be executed.

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 use the three methods here. Generally, flowcharts work well for small problems but Pseudocode is used for larger problems.

1.2.1 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 detailes 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 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, as a program in pseudocode is not an executable program. As the name suggests, pseudocode generally does not actually obey the synatx 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, of course, Then you 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**:**

Original Program Specification:

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

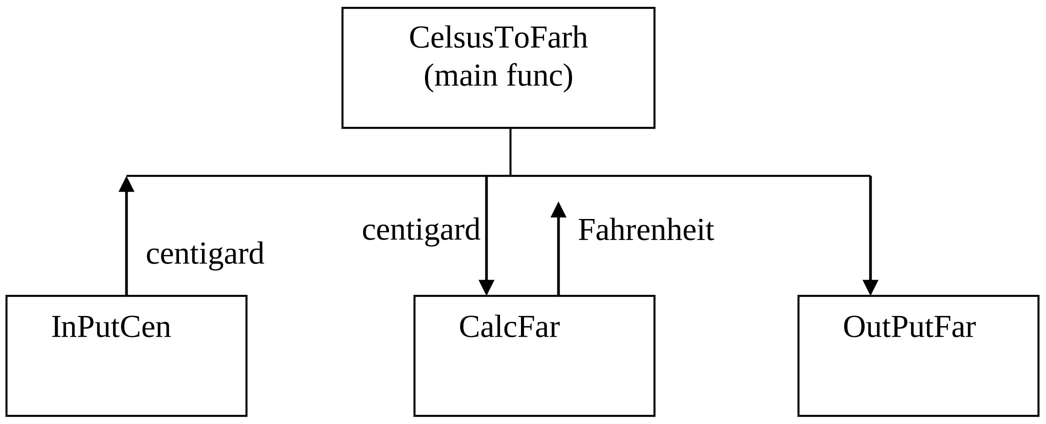
Pseudocode:

```
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
```

1.2.2 Structured Charts

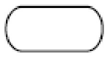
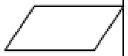
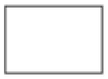
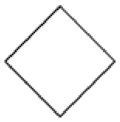
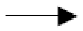
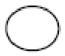
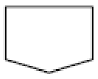
Structured chart depicts the logical functions to the solution of the problem using a chart. It provides an overview that confirms the solution to the problem without excessive consideration to detail. It is high-level in nature.
Example: Write a program that asks the user to enter a temperature reading in centigrade and then prints the equivalent Fahrenheit value.

Input	Process	Output
Centigrade	• Prompt for centigrade value • Read centigrade value • Compute Fahrenheit value • Display Fahrenheit value	Fahrenheit



1.2.3 Flowchart

A **flowchart** (also spelled **flow-chart** and **flow chart**) is a schematic representation of an algorithm or a process . The advantage of flowchart is it doesn't depend on any particular programming language, so that it can used, to translate an algorithm to more than one programming language. Flowchart uses different symbols (geometrical shapes) to represent different processes. The following table shows some of the common symbols.

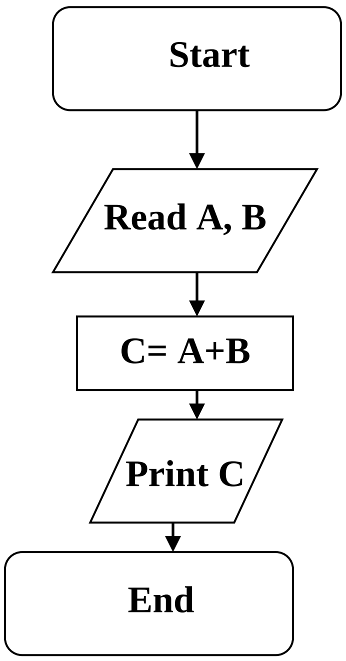
Symbol	Name	Function
	**Terminal**	Used to represent the start of the end of a program
	**Input/ output**	Used to represent data input or data output from a computer
	**Processing**	Usually encloses operations or (command black) a group of operations(a process)
	**Decision block**	it usually contains a question within it there are typically two output paths: one if the answer to the question is yes (true) , and the other 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**	is used for connecting two points in connector a flow chart without drawing flow lines In one page.
	**Off page connector**	It is used 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

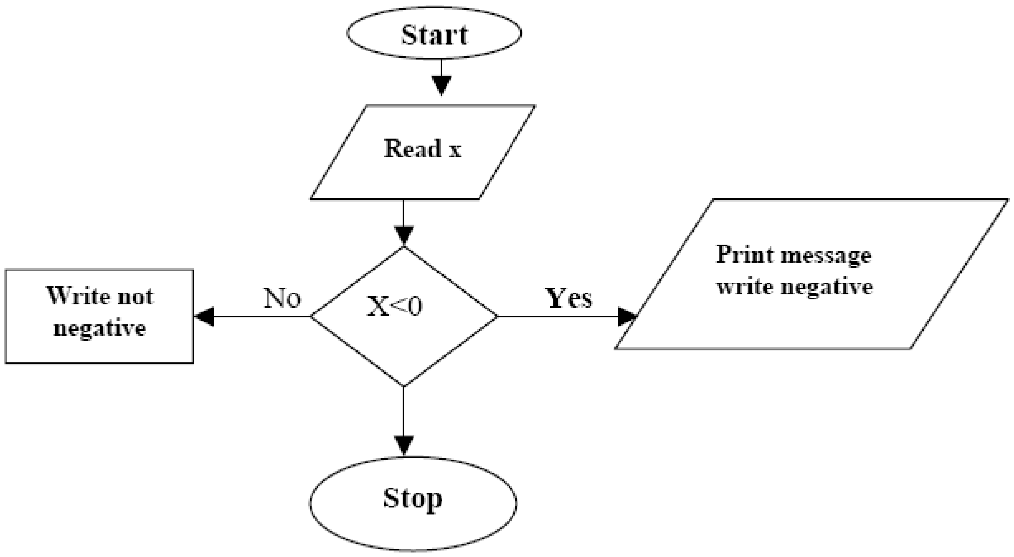
- Read the rules of the two numbers (A and B)
- Add A and B
- Assign the sum of A and B to C
- Display the result (c)

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.
1/ Read a number x
2/ If x is less than zero write a message negative
else write a message not negative



Some times there are conditions in which it is necessary to execute a group of statements repeatedly. Until some condition is satisfied. This condition is called a loop. Loop is a sequence of instructions, which is repeated until some specific condition occurs. A loop normally consists of four parts. These are:

Initialization: - Setting of variables of the computation to their initial values and setting the counter for determining to exit from the loop.

Computation: - Processing

Test: - Every loop must have some way of exiting from it or else the program would endlessly remain in a loop.

Increment: - Re-initialization of the loop for the next loop.

Example 3: - Write the algorithmic description and draw a flow chart to find the following sum.

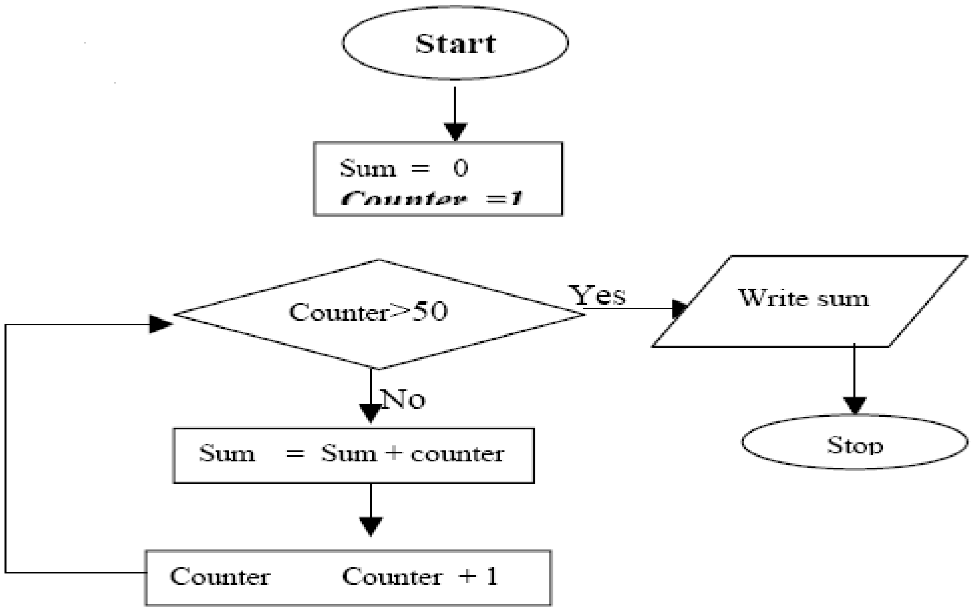
$$Sum = 1+2+3+\ldots + 50$$

Algorithmic description

```
1. Initialize sum too and counter to 1
   1.1.If the counter is less than or equal to 50

            • Add counter to sum
            • Increase counter by 1
            • Repeat step 1.1
   1.2.Else
      • Exit

2. Write sum
```



1.3 System Development Life Cycle (SDLC)

The Systems Development Life Cycle (SDLC) is a conceptual model used in project management that describes the stages involved in a computer system development project from an initial feasibility study through maintenance of the completed application. The phases of SDLC is discussed below briefly.

1.3.1 Feasibility study

The first step is to identify a need for the new system. This will include determining whether a business problem or opportunity exists, conducting a feasibility study to determine if the proposed solution is cost effective, and developing a project plan.

This process may involve end users who come up with an idea for improving their work or may only involve IS people. Ideally, the process occurs in tandem with a review of the organization's strategic plan to ensure that IT is being used to help the organization achieve its strategic objectives. Management may need to approve concept ideas before any money is budgeted for its development.

A preliminary analysis, determining the nature and scope of the problems to be solved is carried out. Possible solutions are proposed, describing the cost and benefits. Finally, a preliminary plan for decision making is produced.

The process of developing a large information system can be very costly, and the investigation stage may require a preliminary study called a *feasibility study*, which includes e.g. the following components:

a. Organisational Feasibility

- How well the proposed system supports the strategic objectives of the organization.

b. Economic Feasibility

- Cost savings
- Increased revenue
- Decreased investment
- Increased profits

c. Technical Feasibility

- Hardware, software, and network capability, reliability, and availability

d. Operational Feasibility

- End user acceptance
- Management support

- Customer, supplier, and government requirements

1.3.2 Requirements analysis

Requirements analysis is the process of analyzing the information needs of the end users, the organizational environment, and any system presently being used, developing the functional requirements of a system that can meet the needs of the users. Also, the requirements should be recorded in a document, email, user interface storyboard, executable prototype, or some other form. The requirements documentation should be referred to throughout the rest of the system development process to ensure the developing project aligns with user needs and requirements.

End users must be involved in this process to ensure that the new system will function adequately and meets their needs and expectations.

1.3.3 Designing solution

After the requirements have been determined, the necessary specifications for the hardware, software, people, and data resources, and the information products that will satisfy the functional requirements of the proposed system can be determined. The design will serve as a blueprint for the system and helps detect problems before these errors or problems are built into the final system.

The created system design, but must reviewed by users to ensure the design meets users' needs.

1.3.4 Testing designed solution

A smaller test system is sometimes a good idea in order to get a “proof-of-concept” validation prior to committing funds for large scale fielding of a system without knowing if it really works as intended by the user.

1.3.5 Implementation

The real code is written here. Systems implementation is the construction of the new system and its delivery into production or day-to-day operation.The key to understanding the implementation phase is to realize that there is a lot more to be done than programming. Implementation requires programming, but it also requires database creation and population, and network installation and testing. You also need to make sure the people are taken care of with effective training and documentation. Finally, if you expect your development skills to improve over time, you need to conduct a review of the lessons learned.

1.3.6 Unit testing

Normally programs are written as a series of individual modules, these subject to separate and detailed test.

1.3.7 Integration and System testing

Brings all the pieces together into a special testing environment, then checks for errors, bugs and interoperability. The system is tested to ensure that interfaces between modules work (integration testing), the system works on the intended platform and with the expected volume of data (volume testing) and that the system does what the user requires (acceptance/beta testing).

1.3.8 Maintenance

What happens during the rest of the software's life: changes, correction, additions, moves to a different computing platform and more. This, the least glamorous and perhaps most important step of all, goes on seemingly forever.