Chapter 1

INTRODUCTION TO COMPUTERS AND PROGRAMMING

This chapter discusses what computers are, how they work and how they are programmed. This chapter also includes an introduction to problem solving and program development.

1.1 HARDWARE AND SOFTWARE

A computer is a device capable of performing computations and making logical decisions at speeds millions and even billions of times faster than human beings can. For example, many of today's personal computers can perform hundreds of millions of additions per second.

Computers process data under the control of sets of instructions called *computer programs*. These computer programs guide the computer through orderly sets of actions specified by people called *computer programmers*.

A computer is comprised of various devices (such as the keyboard, screen, "mouse", disks, memory, CD-ROM and processing units) that are referred to as *hardware*. The computer programs that run on a computer are referred to as *software*.

1.1.1 Computer Hardware

Almost every computer may be seen as being divided into six logical units. Figure 1.1 illustrates the main computer components.

Input Unit.

This unit obtains information from various input devices and places this information at the disposal of the other units so that the information may be processed. The information is entered into computers today through keyboards and mouse devices.

Output Unit

This unit takes information that has been processed by the computer and places it on various output devices to make information available for use outside the computer. Most output from computer today is displayed on screens, printed on paper, or used to control other devices.

Memory Unit

The memory unit stores information. Each computer contains memory of two main types: RAM and ROM.

RAM (*random access memory*) is volatile. Your program and data are stored in RAM when you are using the computer.

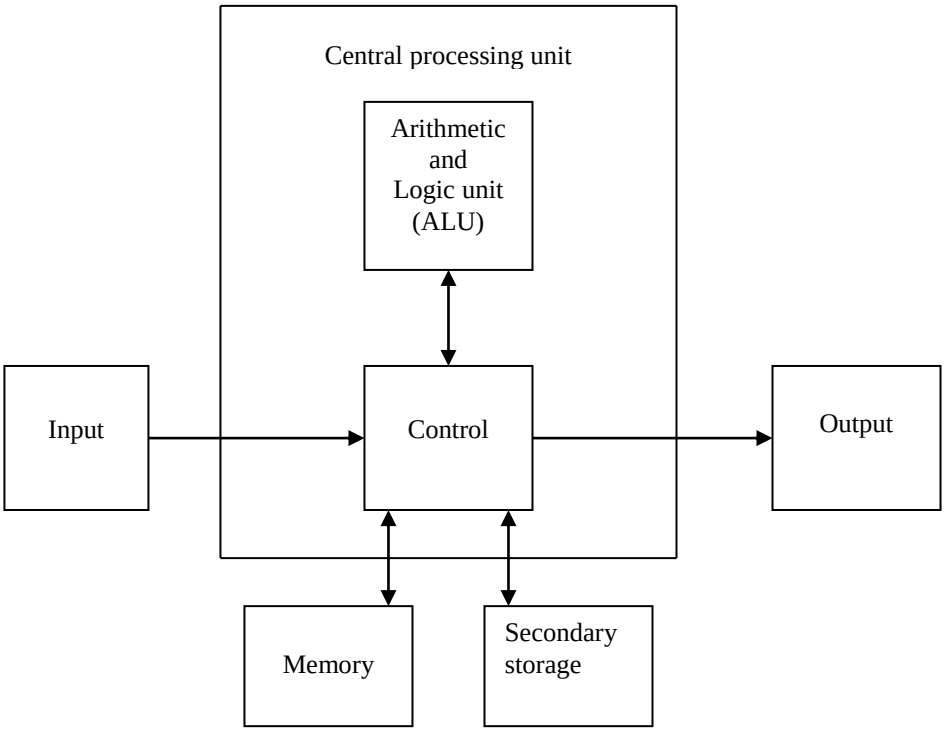

Figure 1.1 Basic hardware units of a computer

ROM (*read only memory*) contains fundamental instructions that cannot be lost or changed by the user. ROM is non-volatile.

Arithmetic and Logic Unit (ALU)

The ALU performs all the arithmetic and logic operations. Ex: addition, subtraction, comparison, etc.

Central Processing Unit (CPU)

The unit supervises the overall operation of the computer. The CPU tells the input unit when information should be read into the memory unit, tell the ALU when information from the memory should be used in calculations and tells the output unit when to send information from the memory unit to certain output devices.

Secondary Storage.

Secondary storage devices are used to be permanent storage area for programs and data. Virtually all secondary storage is now done on magnetic tapes, magnetic disks and CD-ROMs.

A magnetic hard disk consists of either a single rigid platter or several platters that spin together on a common spindle. A movable access arm positions the read and write mechanisms over, but not quite touching, the recordable surfaces. Such a configuration is shown in Figure 1.2.

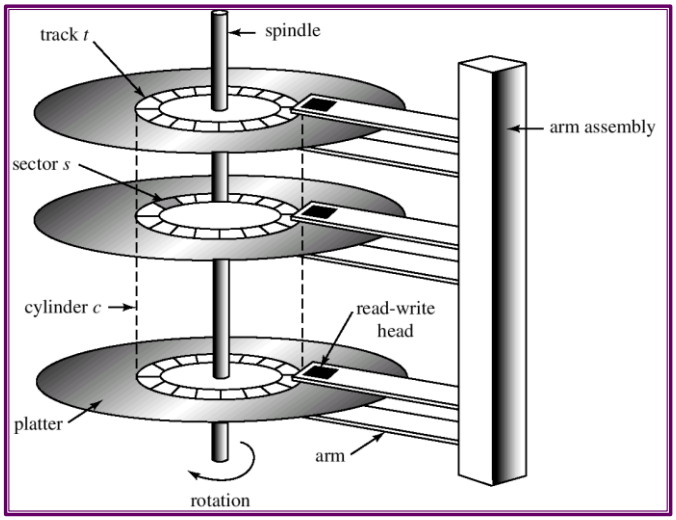

Figure 1.2 The internal structure of a magnetic hard disk drive

1.1.2 Computer Software

A *computer program* is a set of instructions used to operate a computer to produce a specific result.

Another term for a program or a set of programs is *software*, and we use both terms interchangeably throughout the text.

Writing computer programs is called *computer programming*.

The languages used to create computer programs are called *programming languages*.

To understand C++ programming, it is helpful to know a little background about how current programming languages evolved.

Machine and Assembly Languages

• Machine languages are the lowest level of computer languages. Programs written in machine language consist of entirely of 1s and 0s.

Programs in machine language can control directly to the computer's hardware.

```
00101010  000000000001  000000000010
10011001  000000000010  000000000011
```

A machine language instruction consists of two parts: an instruction part and an address part.

The instruction part (*opcode*) is the leftmost group of bits in the instruction and tells the computer the operation to be performed.

The *address part* specifies the memory address of the data to be used in the instruction.

• Assembly languages perform the same tasks as machine languages, but use symbolic names for opcodes and operands instead of 1s and 0s.

LOAD BASEPAY
ADD OVERPAY
STORE GROSSPAY

Since computers can only execute machine language programs, an assembly language program must be translated into a machine language program before it can be executed on a computer.

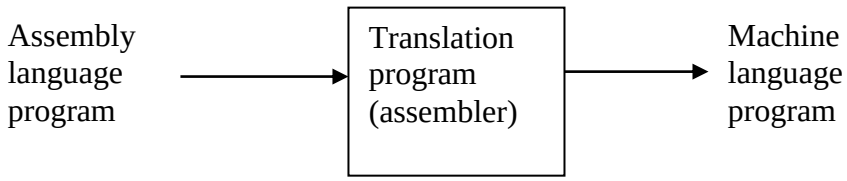

Figure 1.3 Assembly language program must be translated.

Machine languages and assembly languages are called low-level languages since they are closest to computer hardware.

High-level Programming Languages

High-level programming languages create computer programs using instructions that much easier to understand than machine or assembly language instructions.
Programs written in a high-level language must be translated into a low level language using a program called a *compiler*.

A compiler translates programming code into a low-level format.

High-level languages allow programmers to write instructions that look like every English sentences and commonly used mathematical notations.

Each line in a high-level language program is called a *statement*.

Ex: Result = (First + Second)*Third.

Once a program is written in a high-level language, it must also be translated into the machine language of the computer on which it will be run. This translation can be accomplished in two ways.

When each statement in a high-level source program is translated individually and executed immediately upon translation, the programming language used is called an *interpreted language*, and the program doing the translation is called an *interpreter*.

When all of the statements in a high-level source program are translated as a complete unit before any one statement is executed, the programming language used is called is called a *compiled language*. In this case, the program doing the translation is called a *compiler*.

Application and System Software

Two types of computer programs are: application software and system software.

Application software consists of those programs written to perform particular tasks required by the users.

System software is the collection of programs that must be available to any computer system for it to operate.

The most important system software is the *operating system*. Examples of some well-known operating systems include MS-DOS, UNIX, and MS WINDOWS. Many operating systems allow user to run multiple programs. Such operating systems are called *multitasking systems*.

Beside operating systems, language translators are also system softwares.

1.2 HIGH-LEVEL PROGRAMMING LANGUAGES

Because of the difficulty of working with low-level languages, high-level languages were developed to make it easier to write computer programs. High level programming languages create computer programs using instructions that are much easier to understand than machine or assembly language code because you can use words that more clearly describe the task being performed. Examples of high-level languages include FORTRAN, COBOL, BASIC, PASCAL, C, C++ and JAVA.

C and C++ are two separate, but related programming languages. In the 1970s, at Bell Laboratories, Dennis Ritchie and Brian Kernighan designed the C programming language. In 1985, at Bell Laboratories, Bjarne Stroutrup created C++ based on the C language. C++ is an extension of C that adds object-oriented programming capabilities.

1.2.1 What is Syntax?

A programming language's *syntax* is the set of rules for writing grammatically correct language statements. In practice this means a C statement with correct syntax has a proper form specified for the compiler. As such, the compiler accepts the statement and does not generate an error message.

1.2.2 The C Programming Language

C was used exclusively on UNIX and on mini-computers. During the 1980s, C compilers were written for other platforms, including PCs.

To provide a level of standardization for C language, in 1989, ANSI created a standard version of C that is called ANSI C.

One main benefit of the C language is that it is much closer to assembly language other than other types of high-level programming languages.

The programs written in C often run much faster and more efficiently than programs written in other types of high-level programming language.

1.2.3 The C++ Programming Language

C++ is an extension of C that adds object-oriented programming capabilities. C++ is a popular programming language for writing graphical programs that run on Windows and Macintosh.

The standardized version of C++ is commonly referred to as ANSI C++.

The ANSI C and ANSI C++ standards define how C/C++ code can be written.

The ANSI standards also define *run-time libraries*, which contains useful functions, variables, constants, and other programming items that you can add to your programs.

The ANSI C++ run-time library is also called the Standard Template Library or Standard C++ Library.

1.2.4 Structured Programming and Object Oriented Programming

During the 1960s, many large software development effects encountered severe difficulties. Software schedules were typically late, costs greatly exceeded budgets and finished products were unreliable. People began to realize that software development was a far more complex activity than they had imagined. Research activity in the 1960s resulted in the evolution of structured programming – a discipline approach to writing programs that are clearer than unstructured programs, easier to test and debug and easier to modify. Chapter 5 discusses the principles of structured programming. Chapters 2 through 6 develop many structured programs.

One of the more tangible results of this research was the development of the Pascal programming language by Niklaus Wirth in 1971. Pascal was designed for teaching structured programming in academic environments and rapidly became the preferred programming languages in most universities.

In the 1980s, there is a revolution brewing in the software community: *object- oriented programming*. Objects are essentially reusable software components that model items in the real world. Software developers are discovering that using a modular, object-

oriented design and implementation approach can make software development groups much more productive than with previous popular programming techniques such as structured programming.

Object-oriented programming refers to the creation of reusable software objects that can be easily incorporated into another program. An *object* is programming code and data that can be treated as an individual unit or component. *Data* refers to information contained within variables, constants, or other types of storage structures. The procedures associated with an object are referred as *functions* or *methods*. Variables that are associated with an object are referred to as *properties* or *attributes*. Object-oriented programming allows programmers to use programming objects that they have written themselves or that have been written by others.

1.3 PROBLEM SOLUTION AND SOFTWARE DEVELOPMENT

No matter what field of work you choose, you may have to solve problems. Many of these can be solved quickly and easily. Still others require considerable planning and forethought if the solution is to be appropriate and efficient.

Creating a program is no different because a program is a solution developed to solve a particular problem. As such, writing a program is almost the last step in a process of first determining what the problem is and the method that will be used to solve the problem.

One technique used by professional software developers for understanding the problem that is being solved and for creating an effective and appropriate software solution is called the software development procedure. The procedure consists of three overlapping phases

- Development and Design
- Documentation
- Maintenance

As a discipline, software engineering is concerned with creating readable, efficient, reliable, and maintainable programs and systems.

Phase I: Development and Design

The first phase consists of four steps:

1. *Analyze the problem*
This step is required to ensure that the problem is clearly defined and understood. The person doing the analysis has to analyze the problem requirements in order to understand what the program must do, what outputs are required and what inputs are needed. Understanding the problem is very important. Do not start to solve the problem until you have understood clearly the problem.

2. *Develop a Solution*

Programming is all about solving problems. In this step, you have to develop an algorithm to solve a given problem. *Algorithm* is a sequence of steps that describes how the data are to be processed to produce the desired outputs.

An algorithm should be (at least)

–complete (i.e. cover all the parts)
–unambiguous (no doubt about what it does)
–finite (it should finish)

3. *Code the solution*
This step consists of translating the algorithm into a computer program using a programming language.

4. *Test and correct the program*
This step requires testing of the completed computer program to ensure that it does, in fact, provide a solution to the problem. Any errors that are found during the tests must be corrected.
Table 1.1 lists the relative amount of effort that is typically expended on each of these four development and design steps in large commercial programming projects.

Table 1.1 Effort expended in Phase 1

Step	Effort
Analyze the problem	10%
Develop a solution	20%
Code the solution	20%
Test and correct the program	50%

Phase II: Documentation

Documentation requires collecting critical documents during the analysis, design, coding, and testing.
There are five documents for every program solution:

- Program description
- Algorithm development and changes
- Well-commented program listing
- Sample test runs
- User's manual

Phase III: Maintenance

This phase is concerned with the ongoing correction of problems, revisions to meet changing needs and the addition of new features. Maintenance is often the major effort, and the longest lasting of the three phases. While development may take days or months, maintenance may continue for years or decades.

1.4 ALGORITHMS

An algorithm is defined as a step-by-step sequence of instructions that describes how the data are to be processed to produce the desired outputs. In essence, an algorithm answers the question: "What method will you use to solve the problem?".

You can describe an algorithm by using flowchart symbols. By that way, you obtain a flowchart which is an outline of the basic structure or logic of the program.

Flowchart Symbols

To draw flowchart, we employ the symbols shown in the Figure 1.3. The meaning of each flowchart symbol is given in Table 1.2.

To illustrate an algorithm, we consider the simple program that computes the pay of a person. The flowchart for this program is given in Figure 1.4.

Note: Name, Hours and Pay are *variables* in the program.

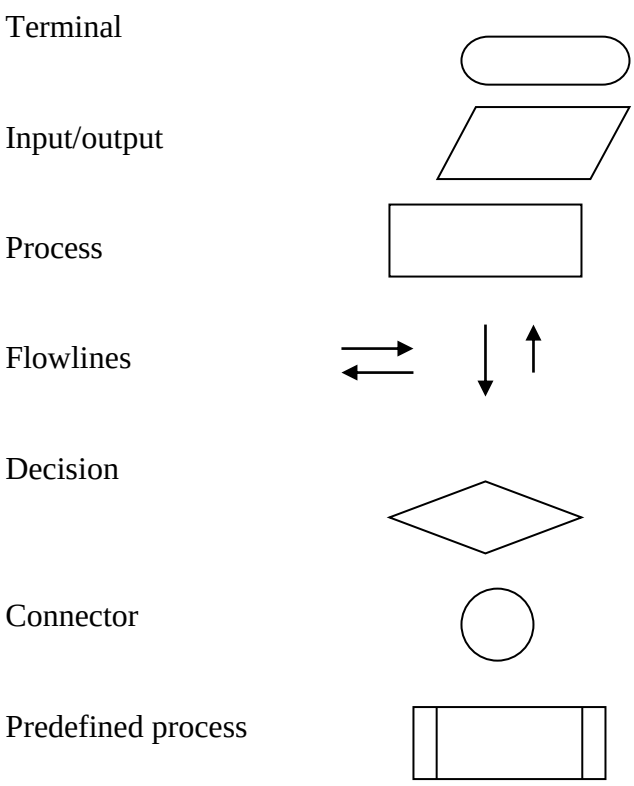

Figure 1.3 Flowchart symbols

Table 1.2 Meaning of flowchart symbols

Symbol name	Description
Terminal	Indicates the beginning or end of an algorithm
Input/Output	Indicates an input or output operation
Process	Indicates computation or data manipulation
Flow lines	Connects the flowchart symbols and indicates the logic flow.
Decision	Indicates a program branch point
Connector	Indicates an entry to, or exit from another part of the flowchart or a connection point
Predefined process	Indicates a predefined process, as in calling a function

Algorithms in pseudo-code

You also can use English-like phases to describe an algorithm. In this case, the description is called *pseudocode*. Pseudocode is an artificial and informal language that helps programmers develop algorithms. Pseudocode has some ways to represent sequence, decision and repetition in algorithms. A carefully prepared pseudocode can be converted easily to a corresponding C++ program.

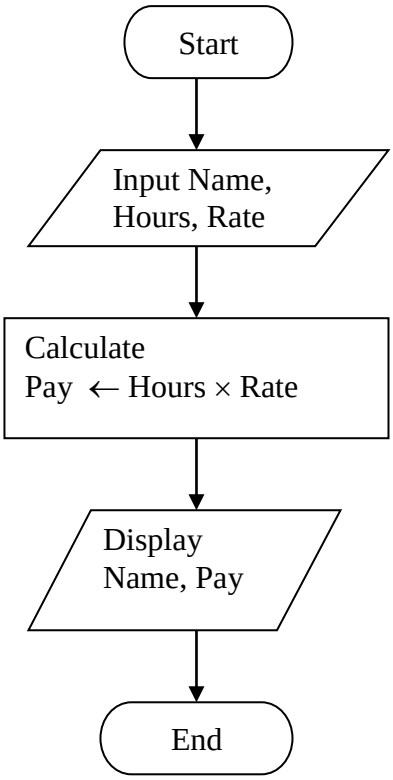

Figure 1.4 Flowchart for calculating the payment of a person

Example: The following set of instructions forms a detailed algorithm in pseudocode for calculating the payment of person.

Input the three values into the variables Name, Hours, Rate.
Calculate Pay = Hours × Rate.
Display Name and Pay.

Loops in Algorithms

Many problems require repetition capability, in which the same calculation or sequence of instructions is repeated, over and over, using different sets of data.

Example 1.1. Write a program to do the task: Print a list of the numbers from 4 to 9, next to each number, print the square of the number.

The flowchart for the algorithm that solves this problem is given in Figure 1.5. You will see in Figure 1.5 the flowchart symbol for decision and the flowline that can connect backward to represent a loop.

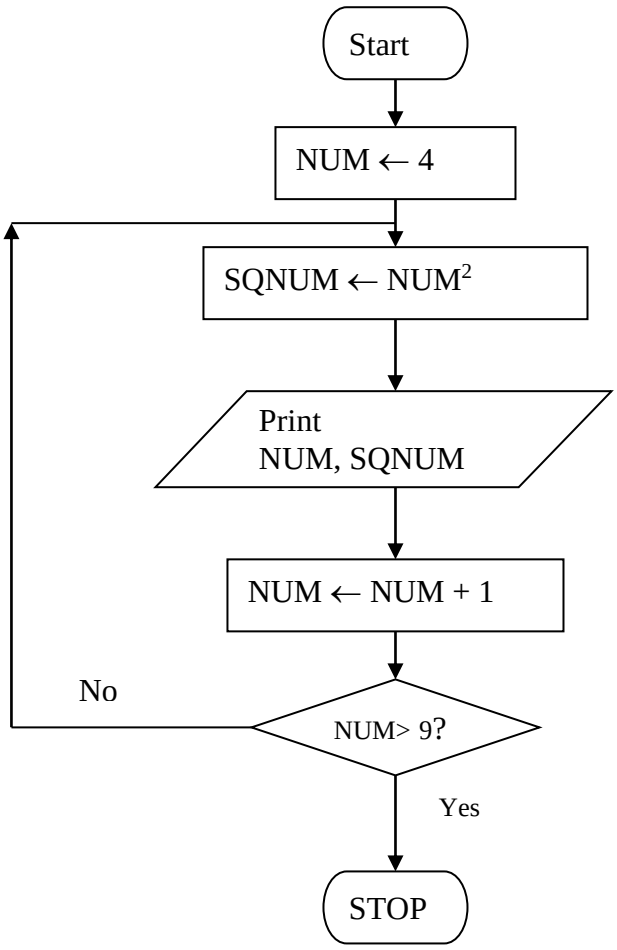

Figure 1.5 Flowchart for calculating the squares of numbers from 4 to 9.

Note:

1. In the flowchart, the statement
   NUM ← NUM + 1
   means "old value of NUM + 1 becomes new value of NUM ".

The above algorithm can be described in pseudocode as follows:

```
NUM ← 4
do
  SQNUM← NUM*NUM
  Print NUM, SQNUM
  NUM ← NUM + 1
while (NUM <= 9)
```

You can compare the pseudo-code and the flowchart in Figure 1.5 to understand the meaning of the *do.. while* construct used in the pseudo-code.

Flowchart versus pseudocode

Since flowcharts are inconvenient to revise, they have fallen out of favor by programmers. Nowadays, the use of pseudocode has gained increasing acceptance.

Only after an algorithm has been selected and the programmer understands the steps required can the algorithm be written using computer-language statements. The writing of an algorithm using computer-language statements is called *coding* the algorithm, which is the third step in our program development process.