Programming in C

As per the latest AICTE syllabus

Reema Thareja

Assistant Professor
Department of Computer Science
Shyama Prasad Mukherji College for Women
University of Delhi

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide. Oxford is a registered trade mark of
Oxford University Press in the UK and in certain other countries.

Published in India by
Oxford University Press
Ground Floor, 2/11, Ansari Road, Daryaganj, New Delhi 110002, India

First published in 2018

ISBN-13: 978-0-19-949228-2
ISBN-10: 0-19-949228-X

Typeset in Times New Roman PS
by Ideal Publishing Solutions, Delhi
Printed in India by Magic International (P) Ltd., Greater Noida

Cover image: BEST-BACKGROUNDS / Shutterstock

Preface

C is one of the most popular and successful programming languages of all time and considered to be the origin of all modern-day computer languages. Many of the popular cross-platform programming languages, such as C++, Java, Python, Objective-C, Perl, and Ruby, and scripting languages, such as PHP, Lua, and Bash, borrow syntaxes and functions from C.

C is also used for programming embedded microprocessors and device drivers. As many embedded systems do not support C++, learning to develop programs using a strict C, without advanced C++ features, is critical for many applications including interface to hardware.

Thus, studying C provides a good foundation to learn advanced programming skills such as object-oriented programming, event-driven programming, multi-thread programming, real-time programming, embedded programming, network programming, parallel programming, other programming languages, as well as new and emerging computing paradigms such as grid computing and cloud computing.

ABOUT THE BOOK

The objective of this book is to provide readers with a sound understanding of the fundamentals of C and how to apply them effectively. Every effort has been made to acquaint readers with the techniques and applications in the area. After learning the rudiments of program writing, readers will find a number of examples and exercises that would help them to design efficient programs.

The salient features of the book include:

- *Lucid style of presentation* that makes the concepts easy to understand
- *Plenty of illustrations* to support the explanations, which help clarify the concepts in a clear manner
- *Programming tips* in between the text educating readers about common programming errors and how to avoid them
- *Notes* highlighting important terms and concepts
- *More than 240 programs* that have been tested on *GCC compiler version 4.6.3*
- *Glossary* of important terms at the end of each chapter for recapitulation of the important concepts learnt
- *Comprehensive exercises* at the end of each chapter to facilitate revision

CONTENT AND COVERAGE

The book is organized into 12 chapters.

Chapter 1 provides an introduction to computer hardware and software. It also provides an insight into different programming languages and the generations through which these languages have evolved. The chapter also discusses program design tools such as algorithms and flowcharts.

Annexure 1 covers examples of different problems solved using both algorithms and flowcharts.

Chapter 2 discusses the building blocks of the C programming language. The chapter discusses keywords,

identifiers, basic data types, constants, variables, and operators supported by the language. It also discusses the storage classes as well as variable scope in C.

Chapter 3 explains decision control and iterative statements as well as special statements such as break statement, control statement, and jump statement.

Chapter 4 deals with declaring, defining, and calling functions. A table listing all the built-in functions is also provided in the chapter.

Chapter 5 explains the important concept of recursion and how different problems such as Fibonacci series and Ackermann function can be solved using recursive functions in C.

Chapter 6 focuses on the concept of arrays, including one-dimensional, two-dimensional, and multidimensional arrays. Finally, the operations that can be performed on such arrays are also explained.

Chapter 7 discusses the concept of strings which are also known as character arrays. The chapter not only focuses on operations that can be used to manipulate strings but also explains various operations that can be used to manipulate the character arrays.

Chapter 8 introduces algorithms that serve as building blocks for creating efficient programs. The chapter explains how to calculate the time complexity which is a key concept for evaluating the performance of algorithms. Searching algorithms such as linear search and binary search are also explained in the chapter. It provides an introduction to sorting and various sorting techniques such as bubble sort, insertion sort, selection sort, merge sort, and quick sort.

Chapter 9 presents a detailed overview of pointers, pointer variables, and pointer arithmetic. The chapter also relates the use of pointers with arrays, strings, and functions. This helps readers to understand how pointers can be used to write better and efficient programs.

Chapter 10 introduces two user-defined data types. The first is a structure and the second is a union. The chapter includes the use of structures and unions with pointers, arrays, and functions so that the inter-connectivity between the programming techniques can be well understood.

Chapter 11 explains how data can be stored in files. The chapter deals with opening, processing, and closing of files through a C program. These files are handled in text mode as well as binary mode for better clarity of the concepts.

Chapter 12 discusses singly linked lists. As a linked list is a preferred data structure when memory needs to be allocated dynamically for the data, the chapter gives the techniques to insert and delete data from the linked list.

Appendix A provides answers to all the objective-type questions given in the exercises section in each chapter.

ONLINE RESOURCES

The following resources are available at Oxford University Press India's Higher Education Companion Site (https://india.oup.com/digital/resources/Instructors-students) to support the faculty and students using this text:

For Faculty

- Solutions Manual
- PowerPoint Slides

For Students

- Multiple Choice Questions
- Supplementary Reading: Data Structures

ACKNOWLEDGMENTS

I am grateful to my family, friends, and fellow members of the teaching staff at the Institute of Information Technology and Management.

My special thanks would always go to my parents, brother Pallav, sisters Kimi and Rashi, and son Goransh. My sincere thanks goes to my uncle Mr B. L. Theraja for his inspiration and guidance in writing this book. Finally, I would like to acknowledge the technical assistance provided to me by Mr Udit Chopra. I would like to thank him for sparing his precious time to help me design and test the programs.

Last but not the least, my acknowledgements will remain incomplete if I do not thank the editorial staff at Oxford University Press, India, for their help and support.

Reema Thareja

Brief Contents

Detailed Contents

Roadmap to Syllabus

Unit No.	Topics	Chapter
1	Introduction to programming Introduction to components of a computer system (disks, memory, processor, where a program is stored and executed, operating system, compilers, etc.)	1
	Idea of algorithm: steps to solve logical and numerical problems; Representation of algorithm: Flowchart/Pseudocode with examples; From algorithms to programs Source code; Variables (with data types); Variables and memory locations; Syntax and logical errors in compilation; Object and executable code	1 Annexure 1 1+2
2	Arithmetic expressions and precedence	2
	Conditional branching and loops; Writing and evaluation of conditionals and consequent branching; Iteration and loops	3
3	Arrays (1-D, 2-D); Character arrays Strings	6 7
4	Basic algorithms Searching; Basic sorting algorithms (bubble, insertion and selection); Finding roots of equations Notion of order of complexity through example programs (no formal definition required)	8 Annexure 1
5	Functions (including using built-in libraries); Parameter passing in functions; Call by value Passing arrays to functions: idea of call by reference	4 6
6	Recursion, as a different way of solving problems. Example programs, such as finding factorial, Fibonacci series, Ackerman function, etc. Quick sort or Merge sort	5 8
7	Structures; Defining structures and Array of Structures	10
8	Idea of pointers; Defining pointers; Use of pointers in self-referential structures Notion of linked list (no implementation)	9 12
9	File handling	11

1 Introduction to Programming

Takeaways

- Hardware
- Application software
- Assembly language
- System software
- Stored program concept
- Generation of programming languages
- Procedural and non-procedural languages
- Compiler, interpreter, linker, loader
- Machine language
- Design of efficient programs

1.1 WHAT IS A COMPUTER?

A computer, in simple terms, can be defined as an electronic device that is designed to accept data, perform the required mathematical and logical operations at high speed, and output the result. A computer accepts data, processes it, and produces information (see Figure 1.1). Here, data refers to some raw facts or figures, and information implies the processed data. For example, if 12-12-92 is the date of birth of a student, then it is data (a raw fact/figure). However, when we process this data (subtract it from the present date) and say that the age of the student is 18 years, then the outcome is information.

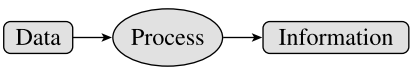


Figure 1.1 Functions of computers

Today, computers have become a crucial part of our everyday lives, and we need computers just like we need the television, telephones, or other electronic devices at home. Computers are basically meant to solve problems quickly and accurately.

Basic Computer Organization

A computer is an electronic device that basically performs five major operations:

- Accepting data or instructions (input)
- Storing data
- Processing data
- Displaying results (output)
- Controlling and coordinating all operations inside a computer

Refer to Figure 1.2, which shows the interaction between the different units of a computer system.

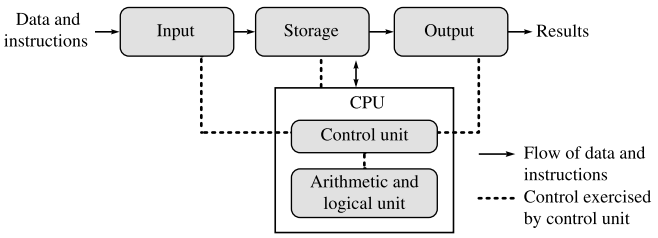


Figure 1.2 Block diagram of a computer

Input This is the process of entering data and instructions (also known as *programs*) in to the computer system. The data and instructions can be entered by using different input devices such as keyboard, mouse, scanner, and trackball. Note that computers understand binary language, which consists of only two symbols (0 and 1), so it is the responsibility of the input devices to convert the input data into binary codes.

Storage It is the process of saving data and instructions permanently in the computer so that they can be used for processing. The computer storage space not only stores the data and programs that operate on that data but also stores the intermediate results and the final results of processing.

A computer has two types of storage areas: primary storage and secondary storage.

Processing The process of performing operations on the data as per the instructions specified by the user (program) is called *processing*. Data and instructions are taken from the primary memory and transferred to the arithmetic and logical unit (ALU), which performs all sorts of calculations. The intermediate results of processing may be stored in the main memory, as they might be required again. When the processing completes, the final result is then transferred to the main memory. Hence, the data may move from main memory to the ALU multiple times before the processing is over.

Output Output is the process of giving the result of data processing to the outside world (external to the computer system). The results are given through output devices such as monitor and printer. Since the computer accepts data only in the binary form and the result of processing is also in the binary form, the result cannot be directly given to the user. The output devices, therefore, convert the results available in binary codes into a human-readable language before displaying it to the user.

Control The control unit (CU) is the central nervous system of the entire computer system. It manages and controls all the components of the computer system. The CU decides the manner in which instructions will be executed and operations performed. It takes care of the step-by-step processing of all operations that are performed in the computer.

1.2 COMPONENTS OF A COMPUTER SYSTEM

The components of a computer system consist of: Hardware and Software.

1.2.1 Hardware

Hardware of a computer system includes:

- Memory
- Disks
- Processor
- Peripheral Devices/Input and Output Devices

Memory

Memory is an internal storage area in the computer, which is used to store data and programs either temporarily or permanently. Computer memory can be broadly divided into two groups—primary memory and secondary memory.

While the main memory holds instructions and data when a program is executing, the auxiliary or the secondary memory holds data and programs that are not currently in use and provides long-term storage.

To execute a program, all the instructions or data that has to be used by the CPU has to be loaded into the main memory. However, the primary memory is volatile and so it can retain data only when the power is on. Moreover, it is very expensive and therefore limited in capacity.

On the contrary, the secondary memory stores data or instructions permanently, even when the power is turned off. It is cheap and can store large volumes of data. Moreover, data stored in auxiliary memory is highly portable, as the users can easily move it from one computer to another. The only drawback of secondary memory is that data can be accessed from it only at very low speeds as compared with the data access speed of the primary memory.

Random access memory (RAM) and read only memory (ROM) are the two types of primary memory.

Random access memory RAM is a volatile storage area within the computer that is typically used to store data temporarily, so that it can be promptly accessed by the processor. The information stored in the RAM is basically loaded from the computer's hard disk, and includes data related to the operating system and applications that are currently being executed by the processor.

RAM is considered *random access* because any memory cell can be directly accessed if its address is known. When the RAM gets full, the computer system operates at a slow speed. When multiple applications are being executed simultaneously and the RAM gets fully occupied by the application's data, it is searched to identify memory portions that have not been utilized. The contents of those locations are then copied onto the hard drive. This action frees up RAM space and enables the system to load other pieces of required data.

These days, the applications' and operating system's demand for system RAM has drastically increased. For example, in the year 2000, a personal computer (PC) had

only 128 MB of RAM, but today PCs have 1–2 GB of RAM installed, and may include graphics cards with their own additional 512 MB or more of RAM. There are two types of RAM—static RAM (SRAM) and dynamic RAM (DRAM).

- *Static RAM* This is a type of RAM that holds data without an external refresh as long as it is powered.
- *Dynamic RAM* This is the most common type of memory used in personal computers, workstations, and servers today. A DRAM chip contains millions of tiny memory cells. Each cell is made up of a transistor and a capacitor, and can contain 1 bit of information—0 or 1. To store a bit of information in a DRAM chip, a tiny amount of power is put into the cell to charge the capacitor. Hence, while reading a bit, the transistor checks for a charge in the capacitor. If a charge is present, then the reading is 1; if not, the reading is 0.

Read only memory ROM refers to computer memory chips containing permanent or semi-permanent data. Unlike RAM, ROM is non-volatile; that is, the data is retained in it even after the computer is turned off.

Most computers contain a small amount of ROM that stores critical programs such as the basic input/output system (BIOS), which is used to boot up the computer when it is turned on. The BIOS consists of a few kilobytes of code that tells the computer what to do when it starts up, such as running hardware diagnostics and loading the operating system into the RAM. Moreover, ROMs are used extensively in calculators and peripheral devices such as laser printers, whose fonts are often stored in ROMs.

Originally, ROMs were read-only. So, in order to update the programs stored in them, the ROM chip had to be removed and physically replaced by another that had a newer version of the program. However, today ROM chips are not literally read-only, as updates to the chip are possible. The process of updating a ROM chip is a bit slower, as memory must be erased in large portions before it can be rewritten. Rewritable ROM chips include PROMs, EPROMs, and EEPROMs.

- *Programmable read-only memory (PROM)* It is also called one-time programmable ROM, and can be written to or programmed using a special device called a PROM programmer.
- *Erasable programmable read-only memory (EPROM)* It is a type of ROM that can be erased and re-programmed. The EPROM can be erased by exposing the chip to strong ultraviolet light, typically for 10 minutes or longer, and can then be rewritten with a process that again needs the application of a higher voltage. Repeated exposure to ultraviolet light wears out the chip.
- *Electrically erasable programmable read-only memory (EEPROM)* It is based on a semiconductor structure similar to the EPROM, but allows the entire or selected contents to be electrically erased, then rewritten electrically, so that they need not be removed from the computer (or camera, MP3 player, etc.). The process of writing an EEPROM is also known as *flashing*.

Secondary storage devices Secondary storage (also known as external memory or auxiliary storage) differs from main memory in that it is not directly accessible by the CPU. The secondary storage devices hold data even when the computer is switched off. An example of such a device is the hard disk.

The computer usually uses its input/output channels to access data from the secondary storage devices to transfer the data to an intermediate area in the main memory. Secondary storage devices are non-volatile in nature, cheaper than the primary memory, and thus can be used to store huge amounts of data. While the CPU can read the data stored in the main memory in nanoseconds, the data from the secondary storage devices can be accessed in milliseconds.

The secondary storage devices are basically formatted according to a file system that organizes the data into files and directories. The file system also provides additional information to describe the owner of a certain file, the access time, the access permissions, and other information.

Hard Disks

The hard drive is a part of the computer that stores all the programs and files, so if the drive is damaged for some reason, all the data stored on the computer is lost. The hard disk provides relatively quick access to large amounts of data stored on an electromagnetically charged surface or a set of surfaces. Today, PCs come with hard disks that can store gigabytes of data.

A hard disk is basically a set of disks, stacked together like phonograph records, that has data recorded electromagnetically in concentric circles known as *tracks*.

A single hard disk includes several *platters* (or disks) that are covered with a magnetic recording medium. Each platter requires two read/write (R/W) heads, one for each side. Note that in the figure, all the R/W heads are attached to a single access arm and so they cannot move independently. The parts of the hard disk are shown in Figure 1.3.

The R/W head can pivot back and forth over the platters to read or write data on them. Data is actually stored on the surface of a platter in *sectors* and *tracks.*

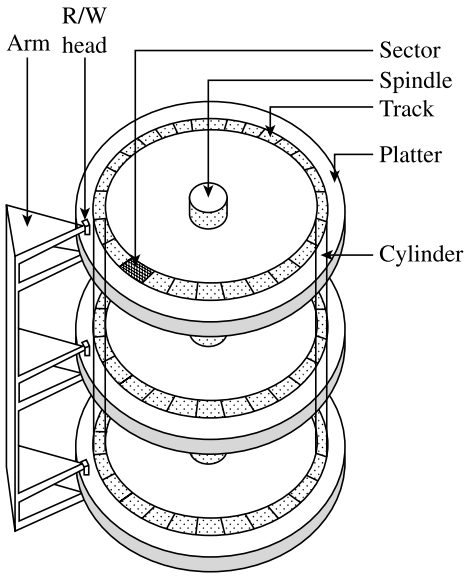


Figure 1.3 Schematic diagram of a hard disk

The performance of a hard disk depends on its access time, which is the time required to read or write on the disk. Access time is a combination of the following three components:

- *Seek time* This is the time taken to position the R/W head over the appropriate cylinder (usually around 8 ms on an average). Seek time varies depending on the position of the access arm when the R/W command is received. Its value will be maximum when the access arm is positioned over the innermost track while the data that has to be accessed is stored on the outermost track. Similarly, it will be zero if the access arm is already positioned over the desired track. On an average, the seek time varies from 10 to 100 milliseconds.
- *Rotational delay* This is the time taken to bring the target sector to rotate under the R/W head. Assuming that the hard disk has 7,200 rpm, or 120 rotations per second, a single rotation is done in approximately 8 ms. The average rotational delay is around 4 ms.
- *Transfer time* The time to transfer data or read/write to a disk is called the transfer rate.

Thus, the overall time required to access data = seek time + rotational delay + transfer time.

The sum of the seek time and the rotational delay is also known as *disk latency*. Disk latency is the time taken to initiate a transfer.

Processor

A basic processor consists of two main parts—ALU and control unit (CU). Besides these components, there are also registers, an execution unit, and a bus interface unit (BIU) as shown in Figure 1.4.

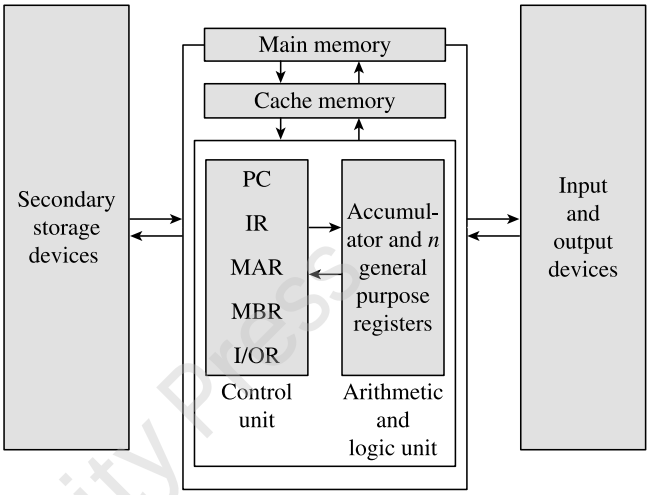


Figure 1.4 Basic computer organization

Execution unit The execution unit mainly consists of the CU, ALU, and registers.

- *Control unit* The main function of the CU is to direct and coordinate the computer operations. It interprets the instructions (program) and initiates action to execute them. The CU controls the flow of data through the computer system and directs the ALU, registers, buses, and input/output (I/O) devices. It is, therefore, called the brain of the computer system. Similar to the human brain, the CU controls all operations within the processor, which in turn controls all other parts of the computer system. In addition, the CU is responsible for fetching, decoding, executing instructions, and storing results.
- *Arithmetic and logic unit* The ALU performs arithmetic (add, subtract, multiply, divide, etc.), comparison (less than, greater than, or equal to), and other operations.

Registers A processor register is a computer memory that provides quick access to the data currently being used for processing. The ALU stores all temporary results and the final result in the processor registers. As mentioned earlier, registers are at the top of memory hierarchy and are always preferred to speed up program execution. Registers are also used to store the instructions of the program currently being executed. There are different types of registers, each with a specific storage function.

- *Accumulator and general purpose registers* These are frequently used to store the data brought from the main memory and the intermediate results during program execution. The number of general purpose registers present varies from processor to processor.
- *Special purpose registers* These include the memory address register (MAR) that stores the address of the data or instruction to be fetched from the main memory, the memory buffer register (MBR) that stores the data or instruction fetched from the main memory, the instruction register (IR) that stores the instructions currently being executed, the I/O register that is used to transfer data or instructions to or from an I/O device, and the program counter that stores the address of the next instruction to be executed.
- *Instruction cycle* To execute an instruction, a processor normally follows a set of basic operations that are together known as an instruction cycle. The operations performed in an instruction cycle involve fetch, decode, execute, and store instructions.

Bus interface unit The BIU provides functions for transferring data between the execution unit of the CPU and other components of the computer system that lie outside the CPU. Every computer system has three different types of busses to carry information from one part to the other. These are the data bus, control bus, and address bus (Figure 1.5).

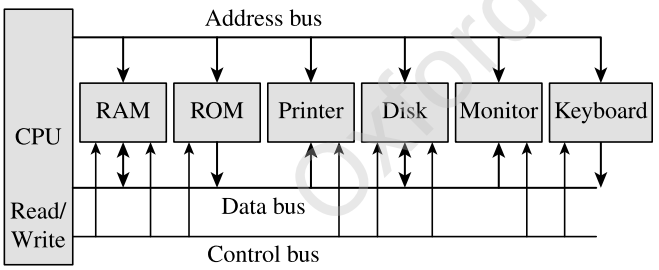


Figure 1.5 Buses with a computer system

The BIU puts the contents of the program counter on the address bus. Note that the content of the program counter is the address of the next instruction to be executed. Once the memory receives an address from the BIU, it places the contents at that address on the data bus, which is then transferred to the IR of the processor through the MBR. At this time, the contents of the program counter are modified (e.g., incremented by 1) so that it now stores the address of the next instruction.

Instruction set The instruction set is a set of commands that instructs the processor to perform specific tasks. It tells the processor what it needs to do, from where to find data (register, memory, or I/O device), from where to find instruction, and so on. Nowadays, computers come with a large set of instructions, and each processor supports its own instruction set. Although the instructions across processors are almost the same, they may vary in their internal design.

System clock A small quartz crystal circuit called the system clock controls the timing of all operations within the computer system. The system clock regularly generates ticks to control the functioning of the computer. Every processor has a system clock to synchronize various operations that take place within the computer system. Many modern computers even have multiple system clocks that vibrate at a specific frequency.

Processor speed The speed of PCs and minicomputers is usually specified in MHz or GHz. However, the speed of a mainframe computer is measured in MIPS (millions of instructions per second) or BIPS (billions of instructions per second) and that of a supercomputer is measured in MFLOPS (millions of floating-point operations per second), GFLOPS (giga or billions of floating-point operations per second). The reason for the variations in speed is that personal or minicomputers use a single processor to execute instructions, whereas mainframes and supercomputers employ multiple processors to speed up their overall performance.

Pipelining and parallel processing Most of the modern PCs support pipelining. Pipelining is a technique with which the processor can fetch the second instruction before completing the execution of the first instruction. Initially, a processor had to wait for an instruction to complete all stages before it could fetch the next instruction, thereby wasting its time. However, with pipelining, processors can operate at a faster pace as they no longer have to wait for one instruction to complete before fetching the next instruction. Such processors that can execute more than one instruction per clock cycle are called superscalar processors.

Peripheral Devices/Input and Output Devices

In order to accomplish tasks, a computer must be able to interact with its users. For this purpose, we need input and output devices, which are also known as peripheral devices. There are different types of input/output devices, and each device has capabilities that differentiate it from the others.

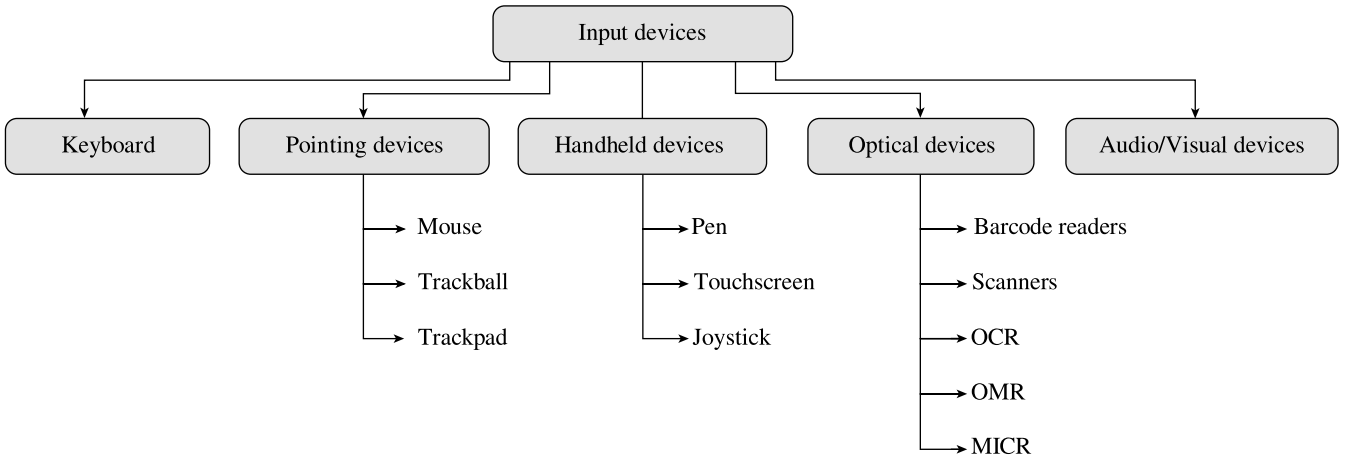


Figure 1.6 Categories of input devices

Input Devices An input device is used to feed data and instructions into the computer. Figure 1.6 categorizes input devices into different groups.

- *Keyboard* The keyboard is the main input device for computers. Computer keyboards look very similar to the keyboards on it and is used to access the options available by pressing the right mouse button.
- *Pointing Devices* A pointing input device enables the users to easily control the movement of the pointer to select items on a display screen, to select commands from commands menu, to draw graphs, etc. Some examples of pointing devices include mouse, trackball, light pen, joystick, and touchpad.
- *Handheld Devices* A handheld device is a pocket-sized computing device with a display screen and touch input and/or a miniature keyboard. Some common examples of handheld devices include smartphones, PDAs, handheld game consoles, and portable media players (such as iPod).
- *Optical Devices* Optical devices, also known as data-scanning devices, use light as a source of input for detecting or recognizing different objects such as characters, marks, codes, and images. The optical device converts these objects into digital data and sends it to the computer for further processing. Some optical devices include barcode readers, image scanners, optical character recognition (OCR) devices, optical mark readers (OMR), and magnetic ink character recognition (MICR) devices.
- *Audio-visual Input Devices* Today, all computers are multimedia-enabled, that is, computers not only allow one to read or write text, but also enable the user to record songs, view animated movies, etc. Hence, in addition to having a keyboard and a mouse, audio–video devices have become a necessity today.

Output Devices Any device that outputs/gives information from a computer can be called an output device. Basically, output devices are electromechanical devices that accept digital data (in the form of 0s and 1s) from the computer and convert them into human-understandable language. Monitors and speakers are two widely used output devices. These devices provide instant feedback to the user's input. Output devices are classified based on whether they give a hard copy or soft copy output (refer to Figure 1.7).

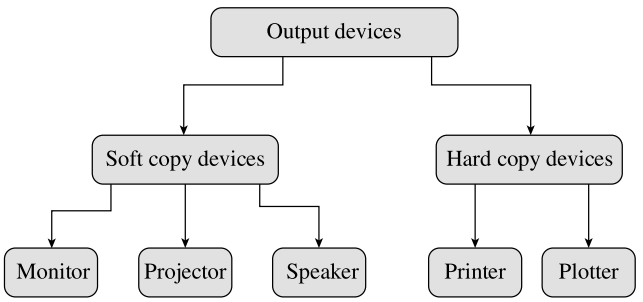


Figure 1.7 Classification of output devices

- *Soft copy devices* These devices produce an electronic version of an output—for example, a file that is stored on a hard disk, CD, or pen drive—and is displayed on the computer screen.
- *Hard copy devices* These devices produce a physical form of output. For example, the content of a file printed on paper is a form of hard copy output.

1.2.2 Computer Software

Computer software can be broadly classified into two groups: system software and application software.

- System software [according to Nutt, 1997] provides a general programming environment in which programmers can create specific applications to suit their needs. This environment provides new functions that are not available at the hardware level and performs tasks related to execution of application programs.

  System software represents programs that allow the hardware to run properly. System software is transparent to the user and acts as an interface between the hardware of the computer and the application software that users need to run on the computer. Figure 1.8 illustrates the relationship between application software, system software, and hardware.

- Application software is designed to solve a particular problem for users. It is generally what we think of when we say the word 'computer programs'. Examples of application software include spreadsheets, database systems, desktop publishing systems, program development software, games, web browsers, among others. Simply put, application software represents programs that allow users to do something besides simply running the hardware.

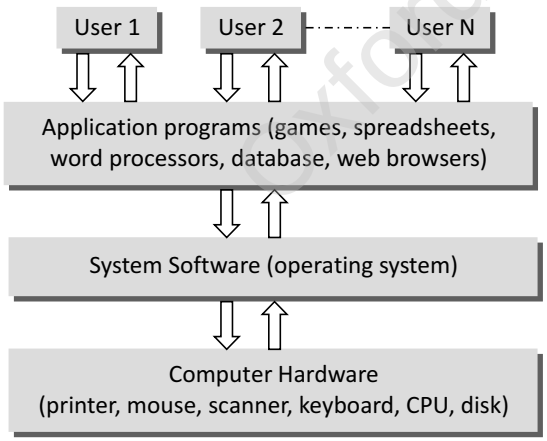


Figure 1.8 Relationship between hardware, system software, and application software

System Software

System software is software designed to operate the computer hardware and to provide and maintain a platform for running application software.

The most widely used system software are discussed in the following sections:

Computer BIOS and Device Drivers The computer BIOS and device drivers provide basic functionality to operate and control the hardware connected to or built into the computer.

BIOS or Basic Input/Output System is a *de facto* standard defining a firmware interface. BIOS is built into the computer and is the first code run by the computer when it is switched on. The key role of BIOS is to load and start the operating system.

When the computer starts, the first function that BIOS performs is to initialize and identify system devices such as the video display card, keyboard and mouse, hard disk, CD/DVD drive, and other hardware. In other words, the code in the BIOS chip runs a series of tests called POST (Power On Self Test) to ensure that the system devices are working correctly.

BIOS then locates software held on a peripheral device such as a hard disk or a CD, and loads and executes that software, giving it control of the computer. This process is known as *booting*.

BIOS is stored on a ROM chip built into the system and has a user interface like that of a menu (Figure 1.9) that can be accessed by pressing a certain key on the keyboard when the computer starts. The BIOS menu can enable the user to configure hardware, set the system clock, enable or disable system components, and most importantly, select which devices are eligible to be a potential boot device and set various password prompts.

To summarize, BIOS performs the following functions:

- Initializes the system hardware
- Initializes system registers
- Initializes power management system
- Tests RAM
- Tests all the serial and parallel ports
- Initializes CD/DVD drive and hard disk controllers
- Displays system summary information

Operating System The primary goal of an operating system is to make the computer system (or any other device in which it is installed like a cell phone) *convenient* and *efficient to use*. An operating system offers generic services to support user applications.

From users' point of view the primary consideration is always the convenience. Users should find it easy to launch an application and work on it. For example, we use icons

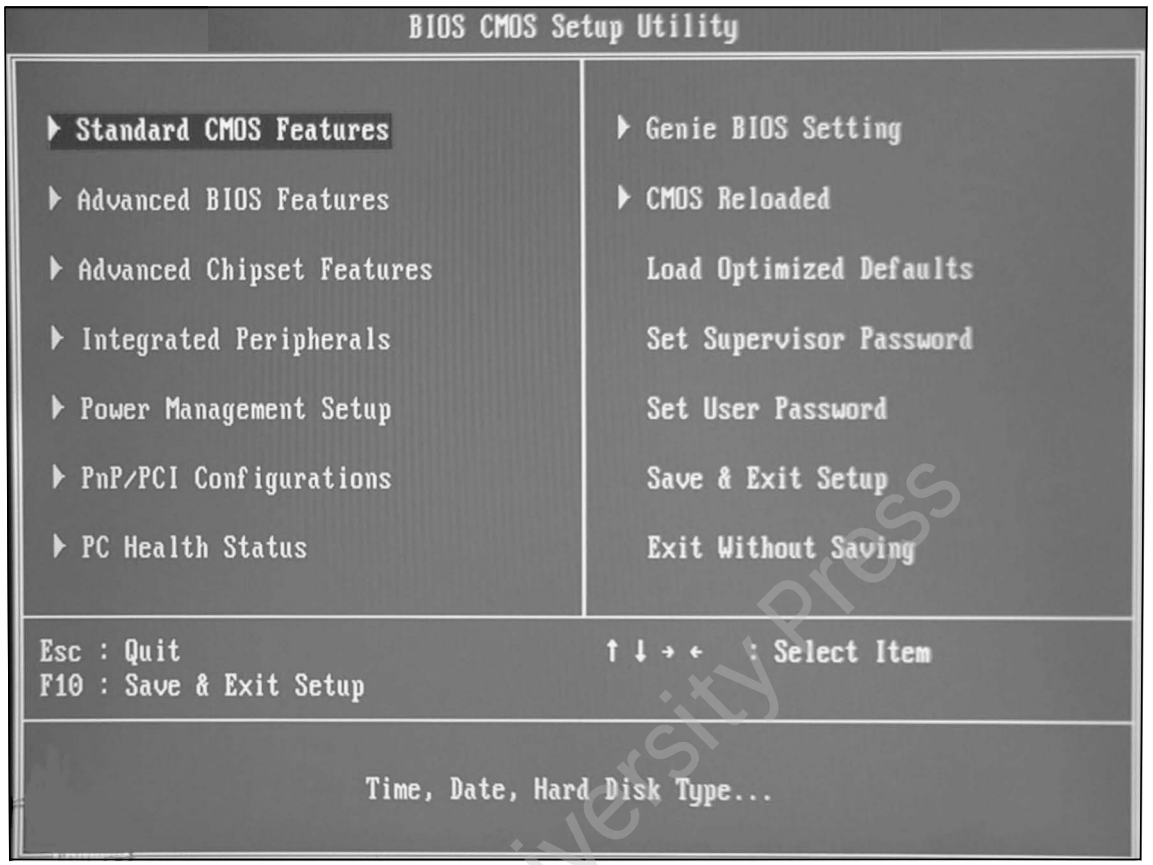


Figure 1.9 BIOS menu

which give us clues about applications. We have a different icon for launching a web browser, e-mail application, or even a document preparation application. In other words, it is the human–computer interface which helps to identify and launch an application. The interface hides a lot of details of the instructions that perform all these tasks.

Similarly, if we examine the programs that help us in using input devices like keyboard/mouse, all the complex details of the character reading programs are hidden from users. We as users simply press buttons to perform the input operation regardless of the complexity of the details involved.

An operating system ensures that the system resources (such as CPU, memory, I/O devices) are utilized efficiently. For example, there may be many service requests on a web server and each user request needs to be serviced. Moreover, there may be many programs residing in the main memory. Therefore, the system needs to determine which programs are currently being executed and which programs need to wait for some I/O operation. This information is necessary because the programs that need to wait can be suspended temporarily from engaging the processor. Hence, it is important for an operating system to have a control policy and algorithm to allocate the system resources.

Utility Software Utility software is used to analyse, configure, optimize, and maintain the computer system. Utility programs may be requested by application programs during their execution for multiple purposes. Some of them are as follows:

- *Disk defragmenters* can be used to detect computer files whose contents are broken across several locations on the hard disk, and move the fragments to one location in order to increase efficiency.
- *Disk checkers* can be used to scan the contents of a hard disk to find files or areas that are either corrupted in some way, or were not correctly saved, and eliminate them in order to make the hard drive operate more efficiently.
- *Disk cleaners* can be used to locate files that are either not required for computer operation, or take up considerable amounts of space. Disk cleaners help users to decide what to delete when their hard disk is full.
- *Disk space analysers* are used for visualizing the disk space usage by getting the size for each folder (including subfolders) and files in a folder or drive.

- *Disk partitions utilities* are used to divide an individual drive into multiple logical drives, each with its own file system. Each partition is then treated as an individual drive.
- *Backup utilities* can be used to make a copy of all information stored on a disk. In case a disk failure occurs, backup utilities can be used to restore the entire disk. Even if a file gets deleted accidentally, the backup utility can be used to restore the deleted file.
- *Disk compression utilities* can be used to enhance the capacity of the disk by compressing/decompressing the contents of a disk.
- *File managers* can be used to provide a convenient method of performing routine data management tasks such as deleting, renaming, cataloguing, moving, copying, merging, generating, and modifying data sets.
- *System profilers* can be used to provide detailed information about the software installed and hardware attached to the computer.
- *Anti-virus* utilities are used to scan for computer viruses.
- *Data compression* utilities can be used to output a file with reduced file size.
- *Cryptographic utilities* can be used to encrypt and decrypt files.
- *Launcher applications* can be used as a convenient access point for application software.
- *Registry cleaners* can be used to clean and optimize the Windows operating system registry by deleting the old registry keys that are no longer in use.
- *Network utilities* can be used to analyse the computer's network connectivity, configure network settings, check data transfer, or log events.
- *Command line interface (CLI)* and *Graphical user interface (GUI)* can be used to make changes to the operating system.

Compiler, Interpreter, Linker, and Loader

- *Compiler* It is a special type of program that transforms the source code written in a programming language (the *source language*) into machine language comprising just two digits, 1s and 0s (the *target language*). The resultant code in 1s and 0s is known as the *object code*. The object code is the one which will be used to create an executable program.

  Therefore, a compiler is used to translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code).

  If the source code contains errors then the compiler will not be able to perform its intended task. Errors resulting from the code not conforming to the syntax of the programming language are called *syntax errors*. Syntax errors may be spelling mistakes, typing mistakes, etc. Another type of error is *logical error* which occurs when the program does not function accurately. Logical errors are much harder to locate and correct.

  The work of a compiler is simply to translate human readable source code into computer executable machine code. It can locate syntax errors in the program (if any) but cannot fix it. Until and unless the syntactical errors are rectified the source code cannot be converted into the object code.
- *Interpreter* Like the compiler, the interpreter also executes instructions written in a high-level language. Basically, a program written in a high-level language can be executed in any of the two ways. First by compiling the program and second, to pass the program through an interpreter.

  While the compiler translates instructions written in high-level programming language directly into the machine language, the interpreter, on the other hand, translates the instructions into an intermediate form, which it then executes. This clearly means that the interpreter interprets the source code line by line. This is in striking contrast with the compiler which compiles the entire code in one go.

  Usually, a compiled program executes faster than an interpreted program. However, the big advantage of an interpreted program is that it does not need to go through the compilation stage during which machine instructions are generated. This process can be time consuming if the program is long. Moreover, the interpreter can immediately execute high-level programs.

  All in all, compilers and interpreters both achieve similar purposes, but inherently different as to how they achieve that purpose.
- *Linker* (*link editor binder*) It is a program that combines object modules to form an executable program. Generally, in case of a large program, the programmers prefer to break a code into smaller modules as this simplifies the programming task. Eventually, when the source code of all the modules has been converted into object code, we need to put all the modules together. This is the job of the linker. Usually, the compiler automatically invokes the linker as the last step in compiling a program.

- *Loader* It is a special type of program that copies programs from a storage device to main memory, where they can be executed. However, in this book we will not go into the details of how a loader actually works. This is because the functionality of a loader is generally hidden from the programmer. As a programmer, it suffices to learn that the task of a loader is to bring the program and all its related files into the main memory from where it can be executed by the CPU.

Application Software

Application software is a type of computer software that employs the capabilities of a computer directly to perform a user-defined task. This is in contrast with system software which is involved in integrating a computer's capabilities, but typically does not directly apply them in the performance of tasks that benefit users.

To better understand application software consider an analogy where hardware would depict the relationship of an electric light bulb (an application) to an electric power generation plant (a system) that depicts the software.

The power plant merely generates electricity which is not by itself of any real use until harnessed to an application like the electric light that performs a service which actually benefits users.

Typical examples of software applications are word processors, spreadsheets, media players, education software, CAD, CAM, data communication software, and statistical and operational research software. Multiple applications bundled together as a package are sometimes referred to as an application suite.

1.3 STORED PROGRAM CONCEPT

All digital computers are based on the principle of stored program concept,which was introduced by Sir John von Neumann in the late 1940s. The following are the key characteristic features of this concept:

- Before any data is processed, instructions are read into memory.
- Instructions are stored in the computer's memory for execution.
- Instructions are stored in binary form (using binary numbers—only 0s and 1s).
- Processing starts with the first instruction in the program, which is copied into a control unit circuit. The control unit executes the instructions.
- Instructions written by the users are performed sequentially until there is a break in the current flow.
- Input/Output and processing operations are performed simultaneously. While data is being read/written, the central processing unit (CPU) executes another program in the memory that is ready for execution.

Note

A stored program architecture is a fundamental computer architecture wherein the computer executes the instructions that are stored in its memory.

John W. Mauchly, an American physicist, and J. Presper Eckert, an American engineer, further contributed to the stored program concept to make digital computers much more flexible and powerful. As a result, engineers in England built the first stored-program computer, Manchester Mark I, in the year 1949. They were shortly followed by the Americans who designed EDVAC in the very same year.

Today, a CPU chip can handle billions of instructions per second. It executes instructions provided both the data and instructions are valid. In case either one of them or both are not valid, the computer stops the processing of instructions.

1.3.1 Types of Stored Program Computers

A computer with a Von Neumann architecture (refer to Figure 1.10) stores data and instructions in the same memory. There is a serial machine in which data and instructions are selected one at a time. Data and instructions are transferred to and from memory through a shared data bus. Since there is a single bus to carry data and instructions, process execution becomes slower.

Later Harvard University proposed a stored program concept in which there was a separate memory to store data and instructions. Instructions are selected serially from the instruction memory and executed in the processor. When an instruction needs data, it is selected from the data memory. Since there are separate memories, execution becomes faster.

1.4 PROGRAMMING LANGUAGES

A programming language is a language specifically designed to express computations that can be performed by the computer. Programming languages are used to create programs that control the behaviour of a system, to express algorithms, or as a mode of human–computer communication.

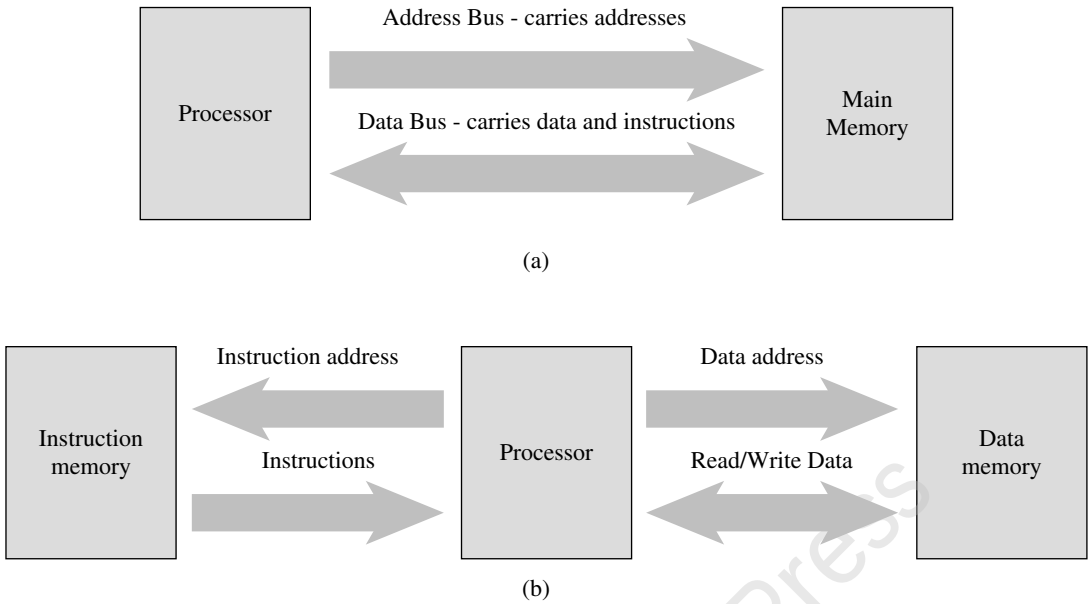


Figure 1.10 Von Neumann architecture (a) Shared memory for instructions and data (b) Separate memories for instructions and data

Usually, programming languages have a vocabulary of syntax and semantics for instructing a computer to perform specific tasks. The term *programming language* usually refers to high-level languages, such as BASIC, C, C++, COBOL, FORTRAN, Ada, and Pascal to name a few. Each of these languages has a unique set of keywords (words that it understands) and a special syntax for organizing program instructions.

While high-level programming languages are easy for humans to read and understand, the computer actually understands the machine language that consists of numbers only. Each type of CPU has its own unique machine language.

In between the machine languages and high-level languages, there is another type of language known as assembly language. Assembly languages are similar to machine languages, but they are much easier to program because they allow a programmer to substitute names for numbers.

However, irrespective of what language the programmer uses, the program written using any programming language has to be converted into machine language so that the computer can understand it. There are two ways to do this: *compile* the program or *interpret* the program

The question of which language is the best depends on the following factors:

- The type of computer on which the program has to be executed
- The type of program
- The expertise of the programmer

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 can be used for writing well-structured and readable programs, but it is not as flexible as the C programming language. C++ goes one step ahead of C by incorporating powerful object-oriented features, but it is complex and difficult to learn.

1.5 GENERATION OF PROGRAMMING LANGUAGES

We now know that programming languages are the primary tools for creating software. As of now, hundreds of programming languages exist in the market, some more used than others, and each claiming to be the best. However, back in the 1940s when computers were being developed there was just one language—the machine language.

The concept of generations of programming languages (also known as levels) is closely connected to the advances in technology that brought about computer generations. The four generations of programming languages include:

- Machine language
- Assembly language

- High-level language (also known as third generation language or 3GL)
- Very high-level language (also known as fourth generation language or 4GL)

1.5.1 First Generation: Machine Language

Machine language was used to program the first stored program on computer systems. This is the lowest level of programming language. The machine language is the only language that the computer understands. All the commands and data values are expressed using 1 and 0s, corresponding to the 'on' and 'off' electrical states in a computer.

In the 1950s each computer had its own native language, and programmers had primitive systems for combining numbers to represent instructions such as *add* and *subtract*. Although there were similarities between each of the machine languages, a computer could not understand programs written in another machine language (Figure 1.11).

MACHINE LANGUAGE
This is an example of a machine language program that will add two numbers and find their average. It is in hexadecimal notation instead of binary notation because this is how the computer presented the code to the programmer.

```
                                        D000000A    D000
                                        D000000F    D009
                                        D000000B    D009
                                                    D009
                                                    D009
                                                    D0Θ0
FF55   CF      FF54   CF      FF53   CF   C1      DOD0
       FF24    CF     FF27    CF     D2   C7      D00C
                                                    D0E4
                                                    Dd0D
                                                    Dd3D
```

Figure 1.11 A machine language program

In machine language, all instructions, memory locations, numbers, and characters are represented in strings of 1s and 0s. Although machine-language programs are typically displayed with the binary numbers represented in octal (base 8) or hexadecimal (base 16), these programs are not easy for humans to read, write, or debug.

The main advantage of machine language is that the code can run very fast and efficiently, since it is directly executed by the CPU.

However, on the downside, the machine language is difficult to learn and is far more difficult to edit if errors occur. Moreover, if you want to add some instructions into memory at some location, then all the instructions after the insertion point would have to be moved down to make room in memory to accommodate the new instructions.

Last but not the least, the code written in machine language is not portable across systems and to transfer the code to a different computer it needs to be completely rewritten since the machine language for one computer could be significantly different from another computer. Architectural considerations made portability a tough issue to resolve.

1.5.2 Second Generation: Assembly Language

The second generation of programming language includes the assembly language. Assembly languages are symbolic programming languages that use symbolic notation to represent machine-language instructions. These languages are closely connected to machine language and the internal architecture of the computer system on which they are used. Since they are close to the machine, assembly language is also called low-level language. Nearly all computer systems have an assembly language available for use.

Assembly language developed in the mid 1950s was a great leap forward. It used symbolic codes also known as *mnemonic codes* that are easy-to-remember abbreviations, rather than numbers. Examples of these codes include ADD for add, CMP for compare, MUL for multiply, etc.

Assembly language programs consist of a series of individual statements or instructions that instruct the computer what to do. Basically, an assembly language statement consists of a *label*, an *operation code*, and one or more *operands*.

Labels are used to identify and reference instructions in the program. The operation code (opcode) is a mnemonic that specifies the operation that has to be performed such as *move, add, subtract,* or *compare*. The operand specifies the register or the location in main memory where the data to be processed is located.

However, like the machine language, the statement or instruction in the assembly language will vary from

machine to another because the language is directly related to the internal architecture of the computer and is not designed to be machine independent. This makes the code written in assembly language less portable as the code written for one machine will not run on machines from a different or sometimes even the same manufacturer.

No doubt, the code written in assembly language will be very efficient in terms of execution time and main memory usage as the language is also close to the computer.

Programs written in assembly language need a *translator* often known as *assembler* to convert them into machine language. This is because the computer will understand only the language of 1s and 0s and will not understand mnemonics like ADD and SUB.

The following instructions are a part of assembly language code to illustrate addition of two numbers:

```
MOV AX,4     Stores value 4 in the AX
             register of CPU
MOV BX,6     Stores value 6 in the BX
             register of CPU
ADD AX,BX    Adds the contents of AX and BX
             registers. Stores the result in
             AX register
```

Although assembly languages are much better to program as compared to the machine language, they still require the programmer to think on the machine's level. Even today, some programmers still use assembly language to write parts of applications where speed of execution is critical, such as video games but most programmers today have switched to third or fourth generation programming languages.

1.5.3 Third Generation Programming Languages

A third generation programming language (3GL) is a refinement of the second-generation programming language. The 2GL languages brought logical structure to software. The third generation was introduced to make the languages more programmer friendly.

Third Generation Programming Languages spurred the great increase in data processing that occurred in the 1960s and 1970s. In these languages, the program statements are not closely related to the internal architecture of the computer and is therefore often referred to as high-level languages.

Generally, a statement written in a high-level programming language will expand into several machine language instructions. This is in contrast to assembly languages, where one statement would generate one machine language instruction. Third Generation Programming Languages made programming easier, efficient, and less prone to errors.

High-level languages fall somewhere between natural languages and machine languages. Third Generation Programming Languages include languages such as FORTRAN (FORmula TRANslator) and COBOL (COmmon Business Oriented Language) that made it possible for scientists and business people to write programs using familiar terms instead of obscure machine instructions.

The first widespread use of high-level languages in the early 1960s changed programming into something quite different from what it had been. Programs were written in statements like English language statements, making them more convenient to use and giving the programmer more time to address a client's problems.

Although 3GLs relieve the programmer of demanding details, they do not provide the flexibility available in low-level languages. However, a few high-level languages like C and FORTRAN combine some of the flexibility of assembly language with the power of high-level languages, but these languages are not well suited to an amateur programmer.

While some high-level languages were designed to serve a specific purpose (such as controlling industrial robots or creating graphics), other languages were flexible and considered to be general-purpose languages. Most of the programmers preferred to use general-purpose high-level languages like BASIC (Beginners' All-purpose Symbolic Instruction Code), FORTRAN, PASCAL, COBOL, C++, or Java to write the code for their applications.

Again, a *translator* is needed to translate the instructions written in high-level language into computer-executable machine language. Such translators are commonly known as interpreters and compilers. Each high-level language has many compilers.

For example, the machine language generated by one computer's C compiler is not the same as the machine language of some other computer. Therefore, it is necessary to have a C compiler for each type of computer on which the C program has to be executed.

Third generation programming languages have made it easier to write and debug programs, which gives programmers more time to think about its overall logic. The programs written in such languages are portable between machines. For example, a program written in standard C can be compiled and executed on any computer that has a standard C compiler.

1.5.4 Fourth Generation: Very High-Level Languages

With each generation, programming languages started becoming easier to use and more like natural languages. However, fourth generation programming languages (4GLs) are a little different from their its prior generation because they are basically non-procedural. When writing code using a procedural language, the programmer has to tell the computer how a task is done—add this, compare that, do this if the condition is true, and so on, in a very specific step-by-step manner. In striking contrast, while using a non-procedural language the programmers define only what they want the computer to do, without supplying all the details of how it has to be done.

There is no standard rule that defines what a 4GL is but certain characteristics of such languages include:

- the code comprising instructions are written in English-like sentences;
- they are non-procedural, so users concentrate on 'what' instead of the 'how' aspect of the task;
- the code is easier to maintain;
- the code enhances the productivity of the programmers as they have to type fewer lines of code to get something done. It is said that a programmer becomes 10 times more productive when he writes the code using a 4GL than using a 3GL.

A typical example of a 4GL is the *query language* that allows a user to request information from a database with precisely worded English-like sentences. A query language is used as a database user interface and hides the specific details of the database from the user. For example, when working with structured query language (SQL), the programmer just needs to remember a few rules of *syntax* and *logic*, and it is easier to learn than COBOL or C.

Let us take an example in which a report has to be generated that displays the total number of students enrolled in each class and in each semester. Using a 4GL, the request would look similar to one that follows:

```
TABLE FILE ENROLLMENT
SUM STUDENTS BY SEMESTER BY CLASS
```

So we see that a 4GL is much simpler to learn and work with. The same code if written in C language or any other 3GL would require multiple lines of code to do the same task.

Fourth generation programming languages are still evolving, which makes it difficult to define or standardize them. The only downside of a 4GL is that it does not make efficient use of the machine's resources. However, the benefit of executing a program fast and easily, far outweighs the extra costs of running it.

1.5.5 Fifth Generation Programming Languages

Fifth generation programming languages (5GLs) are centred on solving problems using constraints given to the program, rather than using an algorithm written by a programmer. Most constraint-based and logic programming languages and some declarative languages form a part of the fifth-generation languages. Fifth generation programming languages are widely used in artificial intelligence research. Typical examples of 5GLs include Prolog, OPS5, and Mercury.

Another aspect of a 5GL is that it contains visual tools to help develop a program. A good example of a fifth generation language is Visual Basic.

So taking a forward leap than the 4GLs, 5GLs are designed to make the computer solve a given problem without the programmer. While working with a 4GL, the programmer had to write specific code to do a work but with 5GL, the programmer only needs to worry about what problems need to be solved and what conditions need to be met, without worrying about how to implement a routine or algorithm to solve them.

Generally, 5GLs were built upon Lisp, many originating on the Lisp machine, such as ICAD. Then, there are many frame languages such as KL-ONE.

In the 1990s, 5GLs were considered to be the wave of the future, and some predicted that they would replace all other languages for system development (except the low-level languages). In 1982 to 1993 Japan had put much research and money into their fifth generation computer systems project, hoping to design a massive computer network of machines using these tools. But when larger programs were built, the flaws of the approach became more apparent. Researchers began to observe that starting from a set of constraints for defining a particular problem, then deriving an efficient algorithm to solve the problem is a very difficult task. All these things could not be automated and still requires the insight of a programmer.

However, today the fifth-generation languages are back as a possible level of computer language. Software vendors across the globe currently claim that their software meets the visual 'programming' requirements of the 5GL concept.

1.6 DESIGN AND IMPLEMENTATION OF EFFICIENT PROGRAMS

The design and development of correct, efficient, and maintainable programs depends on the approach adopted by the programmer to perform various activities that need to be performed during the development process. The entire program or software (collection of programs) development process is divided into a number of phases where each phase performs a well-defined task. Moreover, the output of one phase provides the input for its subsequent phase.

The phases in software development process (as shown in Figure 1.12) can be summarized as below:

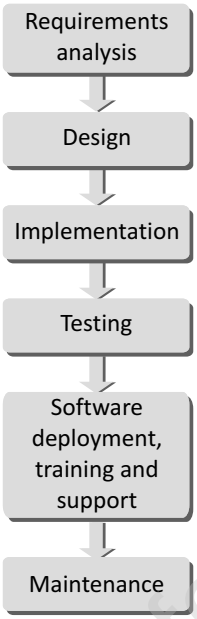


Figure 1.12 Phases in software development life cycle

1.6.1 Requirements Analysis

In this phase, users' expectations are gathered to know why the program/software has to be built. Then all the gathered requirements are analysed to pen down the scope or the objective of the overall software product. The last activity in this phase includes documenting every identified requirement of the users in order to avoid any doubts or uncertainty regarding the functionality of the programs. The functionality, capability, performance, availability of hardware and software components are all analysed in this phase.

1.6.2 Design

The requirements documented in the previous phase acts as an input to the design phase. In the design phase, a plan of actions is made before the actual development process could start. This plan will be followed throughout the development process. Moreover, in the design phase the core structure of the software/program is broken down into modules. The solution of the program is then specified for each module in the form of algorithms, flowcharts, or pseudocodes. The design phase, therefore, specifies how the program/software will be built.

1.6.3 Implementation

In this phase, the designed algorithms are converted into program code using any of the high level languages. The particular choice of language will depend on the type of program like whether it is a system or an application program. While C is preferred for writing system programs, Visual Basic might be preferred for writing an application program. The program codes are tested by the programmer to ensure their correctness.

This phase is also called construction or code generation phase as the code of the software is generated in this phase. While constructing the code, the development team checks whether the software is compatible with the available hardware and other software components that were mentioned in the Requirements Specification Document created in the first phase.

1.6.4 Testing

In this phase, all the modules are tested together to ensure that the overall system works well as a whole product. Although individual pieces of codes are already tested by the programmers in the implementation phase, there is always a chance for bugs to creep in the program when the individual modules are integrated to form the overall program structure. In this phase, the software is tested using a large number of varied inputs also known as test data to ensure that the software is working as expected by the users' requirements that were identified in the requirements analysis phase.

1.6.5 Software Deployment, Training, and Support

After the code is tested and the software or the program has been approved by the users, it is then installed or deployed in the production environment. Software Training and Support is a crucial phase which is often ignored by most of the developers. Program designers and developers spend a lot of time to create software but if nobody in

an organization knows how to use it or fix up certain problems, then no one would like to use it. Moreover, people are often resistant to change and avoid venturing into an unfamiliar area, so as a part of the deployment phase, it has become very crucial to have training classes for the users of the software.

1.6.6 Maintenance

Maintenance and enhancements are ongoing activities which are done to cope with newly discovered problems or new requirements. Such activities may take a long time to complete as the requirement may call for addition of new code that does not fit the original design or an extra piece of code required to fix an unforeseen problem. As a general rule, if the cost of the maintenance phase exceeds 25% of the prior-phases cost then it clearly indicates that the overall quality of at least one prior phase is poor. In such cases, it is better to re-build the software (or some modules) before maintenance cost is out of control.

1.7 PROGRAM DESIGN TOOLS: ALGORITHMS, FLOWCHARTS, PSEUDOCODES

This section will deal about different tools which are used to design solution(s) of a given problem at hand.

1.7.1 Algorithms

In general terms, an algorithm provides a blueprint to writing a program to solve a particular problem. It is considered to be an effective procedure for solving a problem in a finite number of steps. That is, a well-defined algorithm always provides an answer, and is guaranteed to terminate.

Algorithms are mainly used to achieve *software re-use*. Once we have an idea or a blueprint of a solution, we can implement it in any high-level language, such as C, C++, Java, and so on. In order to qualify as an algorithm, a sequence of instructions must possess the following characteristics:

- Be precise
- Be unambiguous
- Not even a single instruction must be repeated infinitely
- After the algorithm gets terminated, the desired result must be obtained

Control Structures used in Algorithms

An algorithm has a finite number of steps and some steps may involve decision-making and repetition. Broadly speaking, an algorithm can employ any of the three control structures, namely, sequence, decision, and repetition.

Sequence

Sequence means that each step of the algorithm is executed in the specified order. An algorithm to add two numbers is given in Figure 1.13. This algorithm performs the steps in a purely sequential order.

```
Step 1: Input the first number as A
Step 2: Input the second number as B
Step 3: SET SUM = A + B
Step 4: PRINT SUM
Step 5: END
```

Figure 1.13 Algorithm to add two numbers

Decision

Decision statements are used when the outcome of the process depends on some condition. For example, `if x=y`, then print "`EQUAL`". Hence, the general form of the `if` construct can be given as

```
if condition then process
```

An algorithm to check the equality of two numbers is shown in Figure 1.14.

```
Step 1: Input the first number as A
Step 2: Input the second number as B
Step 3: IF A = B
            Then PRINT "EQUAL"
        ELSE
             PRINT "NOT EQUAL"
Step 4: END
```

Figure 1.14 Algorithm to test for equality of two numbers

Repetition

Repetition, which involves executing one or more steps for a number of times, can be implemented using constructs such as while, do-while, and for loops. These loops execute one or more steps until some condition is true. Figure 1.15 shows an algorithm that prints the first 10 natural numbers.

```
Step 1: [INITIALIZE] SET I = 0, N = 10
Step 2: Repeat Step while I<=N
Step 3:   PRINT I
Step 4: SET I = I + 1
Step 5: END
```

Figure 1.15 Algorithm to print first 10 natural numbers

Selecting the Most Efficient Algorithm

Many a times, you may formulate more than one algorithm for a problem. In such cases, you must always analyse all the alternatives and try to choose the most efficient algorithm.

Analysing an algorithm means determining the amount of resources (such as time and memory) needed to execute it. Algorithms are generally designed to work with an arbitrary number of inputs, so the efficiency or complexity of an algorithm is stated in terms of time and space complexity.

The *time complexity* of an algorithm is basically the running time of a program as a function of the input size. Similarly, the *space complexity* of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.

In other words, the number of machine instructions which a program executes is called its time complexity. This number is primarily dependent on the size of the program's input and the algorithm used.

Generally, the space needed by a program depends on the following two parts:

- *Fixed part*: It varies from problem to problem. It includes the space needed for storing instructions, constants, variables, and structured variables (like arrays and structures).
- *Variable part*: It varies from program to program. It includes the space needed for recursion stack, and for structured variables that are allocated space dynamically during the runtime of a program.

However, running time requirements are more critical than memory requirements. Therefore, choose the algorithm that has less running time complexity.

1.7.2 Flowcharts

A flowchart is a graphical or symbolic representation of a process. It is basically used to design and document virtually complex processes to help the viewers to visualize the logic of the process, so that they can gain a better understanding of the process and find flaws, bottlenecks, and other less obvious features within it. When designing a flowchart, each step in the process is depicted by a different symbol and is associated with a short description. The symbols in the flowchart (refer Figure 1.16) are linked together with arrows to show the flow of logic in the process.

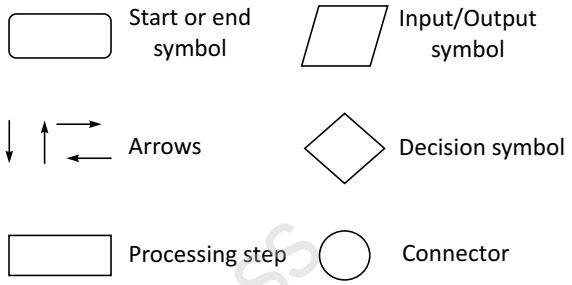


Figure 1.16 Symbols of flowchart

The symbols of a flowchart include:

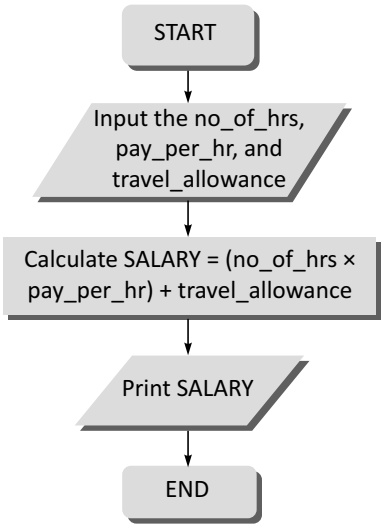
- *Start* and *end symbols* are also known as the terminal symbols and are represented as circles, ovals, or rounded rectangles. Terminal symbols are always the first and the last symbols in a flowchart.
- *Arrows* depict the flow of control of the program. They illustrate the exact sequence in which the instructions are executed.
- *Generic processing step*, also called as an activity, is represented using a rectangle. Activities include instructions such as `add a to b`, `save the result`. Therefore, a processing symbol represents arithmetic and data movement instructions. When more than one process has to be executed simultaneously, they can be placed in the same processing box. However, their execution will be carried out in the order of their appearance.
- *Input/output symbols* are represented using a parallelogram and are used to get inputs from the users or display the results to them.
- A *conditional or decision symbol* is represented using a diamond. It is basically used to depict a Yes/No question or a True/False test. The two arrows coming out of it, one from the bottom vertex and the other from the right vertex, correspond to Yes or True, and No or False, respectively. The arrows should always be labelled. A decision symbol in a flowchart can have more than two arrows, which indicate that a complex decision is being taken.
- *Labelled connectors* are represented by an identifying label inside a circle and are used in complex or multi-

sheet diagrams to substitute for arrows. For each label, the 'outflow' connector must have one or more 'inflow' connectors. A pair of identically labelled connectors issued to indicate a continued flow when the use of lines becomes confusing.

Example 1.1

Draw a flowchart to calculate the salary of a daily wager.

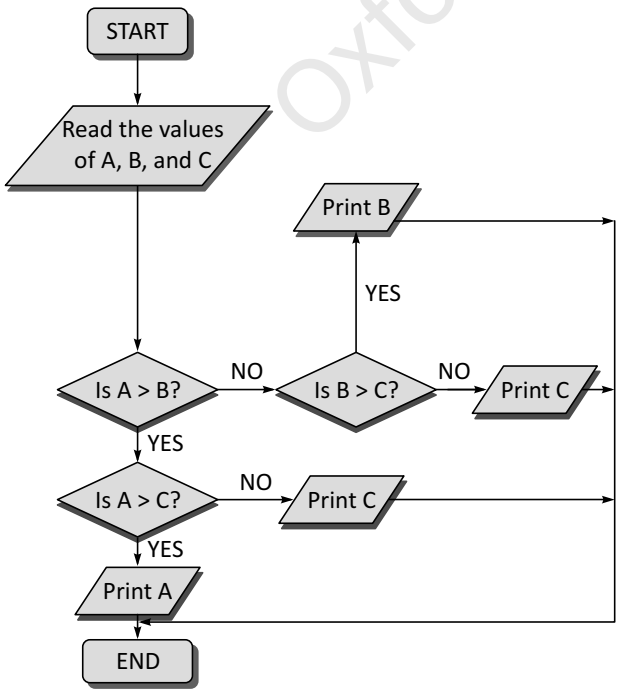
Solution



Example 1.2

Draw a flowchart to determine the largest of three numbers.

Solution



1.7.3 Pseudocodes

Pseudocode is a compact and informal high-level description of an algorithm that uses the structural conventions of a programming language. It is basically meant for human reading rather than machine reading, so it omits the details that are not essential for humans. Such details include variable declarations, system-specific code, and sub-routines. Pseudocodes are an outline of a program that can be easily converted into programming statements. They consist of short English phrases that explain specific tasks within a program's algorithm. They should not include keywords in any specific computer language. The sole purpose of pseudocodes is to enhance human understandability of the solution. They are commonly used in textbooks and scientific publications for documenting algorithms, and for sketching out the program structure before the actual coding is done. This helps even non-programmers to understand the logic of the designed solution. There are no standards defined for writing a pseudocode, because a pseudocode is not an executable program. Flowcharts can be considered as graphical alternatives to pseudocodes, but require more space on paper.

Example 1.3

Write a pseudocode for calculating the price of a product after adding sales tax to its original price.

Solution

```
1. Read the price of the product
2. Read the sales tax rate
3. Calculate sales tax = price of the item ×
   sales tax rate
4. Calculate total price = price of the product
   + sales tax
5. Print total price
6. End
Variables: price of the product, sales tax
   rate, sales tax, total price
```

Example 1.4

Write a pseudocode to read the marks of 10 students. If marks are greater than 50, the student passes, else the student fails. Count the number of students who pass and the number who fail.

Solution

```
1. Set pass to 0
2. Set fail to 0
3. Set no of students to 0
```

```
4. WHILE no of students < 10
      a. input the marks
      b. IF marks >= 50 then
            Set pass = pass + 1
         ELSE
            Set fail = fail + 1
         ENDIF
   ENDWHILE
5. End
Variables: pass, fail, no of students, marks
```

1.8 TYPES OF ERRORS

While writing programs, very often we get errors in our program. These errors if not removed will either give erroneous output or will not let the compiler to compile the program. These errors are broadly classified under four groups as shown in Figure 1.17.

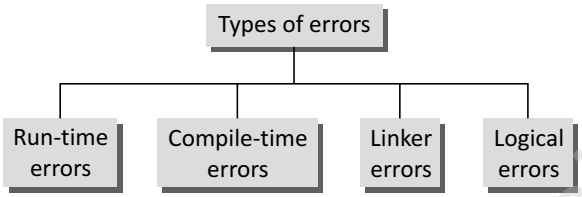


Figure 1.17 Types of Errors

Run-time Errors As the name suggests, run-time errors occur when the program is being run executed. Such errors occur when the program performs some illegal operations like

- Dividing a number by zero
- Opening a file that already exists
- Lack of free memory space
- Finding square or logarithm of negative numbers

Run-time errors may terminate program execution, so the code must be written in such a way that it handles all sorts of unexpected errors rather terminating it unexpectedly. This ability to continue operation of a program despite of run-time errors is called robustness.

Compile-time Errors Again as the name implies, compile-errors occur at the time of compilation of the program. Such errors can be further classified as follows:

Syntax Errors Syntax error is generated when rules of C programming language are violated. For example, if we write int a: then a syntax error will occur since the correct statement should be int a;

Semantic Errors Semantic errors are those errors which may comply with rules of the programming language but are not meaningful to the compiler. For example, if we write, a * b = c; it does not seem correct. Rather, if written like c = a * b would have been more meaningful.

Logical Errors Logical errors are errors in the program code that result in unexpected and undesirable output which is obviously not correct. Such errors are not detected by the compiler, and programmers must check their code line by line or use a debugger to locate and rectify the errors. Logical errors occur due to incorrect statements. For example, if you meant to perform c = a + b; and by mistake you typed c = a * b; then though this statement is syntactically correct, it is logically wrong.

Linker Errors These errors occur when the linker is not able to find the function definition for a given prototype. For example, if you write clrscr(); but do not include conio.h then a linker error will be shown. Similarly, even if you have defined a function display_data() but while calling if you mistakenly write displaydata() then again a linker error will be generated.

1.8.1 Testing and Debugging Approaches

Testing is an activity that is performed to verify correct behaviour of a program. It is specifically carried out with an intent to find errors. Ideally testing should be conducted at all stages of program development. However, in the implementation stage, three types of tests can be conducted:

Unit Tests Unit testing is applied only on a single unit or module to ensure whether it exhibits the expected behaviour.

Integration Tests These tests are a logical extension of unit tests. In this test, two units that have already been tested are combined into a component and the interface between them is tested. The guiding principle is to test combinations of pieces and then gradually expanding the component to include other modules as well. This process is repeated until all the modules are tested together. The main focus of integration testing is to identify errors that occur when the units are combined.

System Tests System testing checks the entire system. For example, if our program code consists of three modules then each of the module is tested individually using unit

tests and then system test is applied to test this entire system as one system.

Note

Testing should not be restricted to just execution testing.

Debugging, on the other hand, is an activity that includes execution testing and code correction. The main aim of debugging is locating errors in the program code. Once the errors are located, they are then isolated and fixed to produce an error-free code. Different approaches applied for debugging a code includes:

Brute-Force Method In this technique, a printout of CPU registers and relevant memory locations is taken, studied, and documented. It is the least efficient way of debugging a program and is generally done when all the other methods fail.

Backtracking Method It is a popular technique that is widely used to debug small applications. It works by locating the first symptom of error and then trace backward across the entire source code until the real cause of error is detected. However, the main drawback of this approach is that with increase in number of source code lines, the possible backward paths become too large to manage.

Cause Elimination In this approach, a list of all possible causes of an error is developed. Then relevant tests are carried out to eliminate each of them. If some tests indicate that a particular cause may be responsible for an error then the data are refined to isolate the error.

Example 1.5

Let us take a problem, collect its requirement, design the solution, implement it in C and then test our program.

Problem Statement To develop an automatic system that accepts marks of a student and generates his/her grade.

Requirements Analysis Ask the users to enlist the rules for assigning grades. These rules are:

Marks	Grade
Above 75	O
60-75	A
50-60	B
40-50	C
Less than 40	D

Design In this phase, write an algorithm that gives a solution to the problem.

Step 1: Enter the marks obtained as M
Step 2: If M > 75 then print "O"
Step 3: If M >= 60 and M < 75 then print "A"
Step 4: If M >= 50 and M < 60 then print "B"
Step 5: If M >= 40 and M < 50 then print "C"
else
print "D"
Step 6: End

Implementation Write the C program to implement the proposed algorithm.

```
#include <stdio.h>
int main()
{  int marks;
   char grade;
   printf("\n Enter the marks of the student
   : ");
   scanf("%d", &marks);
   if (marks<0 || marks >100)
   {    printf("\n Not Possible");
        exit(1);
   }
   if(marks>=75)
      grade = 'O';
   else if(marks>=60 && marks<75)
      grade = 'A';
   else if(marks>=50 && marks<60)
      grade = 'B';
   else if(marks>=40 && marks<50)
      grade = 'C';
   else
      grade = 'D';
   printf("\n GRADE = %c", grade);
}
```

Test The above program is then tested with different test data to ensure that the program gives correct output for all relevant and possible inputs. The test cases are shown in the table given below.

Test Case ID	Input	Expected Output	Actual Output
1	-12	Not Possible	Not Possible
2	112	Not Possible	Not Possible
3	32	D	D
4	46	C	C
5	54	B	B
6	68	A	A

Test Case ID	Input	Expected Output	Actual Output
7	91	O	O
8	40	C	C
9	50	B	B
10	60	A	A
11	75	O	O
12	100	O	O
13	0	D	D

Note in the above table, we have identified test cases for the following,

1. "Not Possible" Combinations
2. A middle value from each range
3. Boundary values for each range

POINTS TO REMEMBER

- A computer has two parts—computer hardware which does all the physical work and computer software which tells the hardware what to do and how to do it.
- A program is a set of instructions that are arranged in a sequence to guide a computer to find a solution for a given problem. The process of writing a program is called programming.
- Computer software is written by computer programmers using a programming language.
- Modern-day computers are based on the principle of the stored program concept, which was introduced by sir John van Neumann in the late 1940s.
- Application software is designed to solve a particular problem for users.
- System software represents programs that allow the hardware to run properly. It acts as an interface between the hardware of the computer and the application software that users need to run on the computer.
- The key role of BIOS is to load and start the operating system. The code in the BIOS chip runs a series of tests called POST (Power On Self Test) to ensure that the system devices are working correctly. BIOS is stored on a ROM chip built into the system.
- Utility software is used to analyse, configure, optimize, and maintain the computer system.
- A compiler is a special type of program that transforms source code written in a programming language (the *source language*) into machine language comprising of just two digits—1s and 0s (the *target language*). The resultant code in 1s and 0s is known as the object code.
- Linker is a program that combines object modules to form an executable program.
- A loader is a special type of program that copies programs from a storage device to main memory, where they can be executed.
- The fourth generations of programming languages are: machine language, assembly language, high-level language, and very high-level language.
- Machine language is the lowest level of programming language that a computer understands. All the instructions and data values are expressed using 1s and 0s.
- Assembly language is a low-level language that uses symbolic notation to represent machine language instructions.
- Third-generation languages are high-level languages in which instructions are written in statements like English language statements. Each instruction in this language expands into several machine language instructions.
- Fourth-generation languages are non-procedural languages in which programmers define only what they want the computer to do, without supplying all the details of how it has to be done.
- In requirements elicitation phase, users' expectations are gathered to know why the program/software has to be built.
- Course of actions is planned in the design phase.
- An algorithm provides a blueprint to writing a program to solve a particular problem.
- The *time complexity* of an algorithm is basically the running time of a program as a function of the input size. Similarly, the *space complexity* of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.
- A flowchart is a graphical or symbolic representation of a process.
- A pseudocode is a compact and informal high-level description of an algorithm that uses the structural conventions of a programming language.
- In the implementation phase, the designed algorithms are converted into program code using any of the high level languages.

- In the testing phase, all the modules are tested together to ensure that the overall system works well as a whole product.
- After the code is tested and the software or the program has been approved by the users, it is then installed or deployed in the production environment.
- Maintenance and enhancements are ongoing activities which are done to cope with newly discovered problems or new requirements.

GLOSSARY

Backtracking method The technique used to debug small applications which works by locating the first symptom of error and then tracing backward across the entire source code until the real cause of error is detected.

Cause elimination The technique in which list of all possible causes of an error is developed.

Compile errors Errors that occur at the time of compilation of the program

Debugging An activity that includes execution testing and code correction. The main aim of debugging is locating errors in the program code.

Integration testing A testing technique in which two units that have already been tested are combined into a component and the interface between them is tested.

Linker error Errors that may occur when the linker is not able to find the function definition for a given prototype

Logical errors Errors that result in unexpected and undesirable output which is obviously not correct. Such errors are not detected by the compiler.

Runtime errors Errors that occur when the program is being executed

Semantic errors Errors which may comply with rules of the programming language but are not meaningful to the compiler

Syntax Errors Errors that are generated when rules of C programming language are violated

System testing A testing technique that checks the entire system

Testing An activity performed to verify correct behaviour of a program. It is specifically carried out with the intent to find errors.

Unit testing A testing technique applied only on a single unit or module to ensure whether it exhibits the expected behaviour.

EXERCISES

Fill in the Blanks

1. _____ tells the hardware what to do and how to do it.
2. The hardware needs a _____ to instruct what has to be done.
3. The process of writing a program is called _____.
4. _____ is used to write computer software.
5. _____ transforms the source code into binary language.
6. _____ allows a computer to interact with additional hardware devices such as printers, scanners, and video cards.
7. _____ helps in coordinating system resources and allows other programs to execute.
8. _____ provides a platform for running application software.
9. _____ can be used to encrypt and decrypt files.
10. An assembly language statement consists of a _____, an _____, and _____.
11. In _____ phase user's expectations are collected and documented.
12. The documented requirements act as an input to the _____ phase.
13. Modularization of the program is done in _____ phase.
14. A conditional or decision symbol is represented using a _____.
15. _____ phase is also called construction or code generation phase.
16. _____ errors may terminate program execution.
17. Debugging is an activity that includes _____ and _____.
18. Structured programming follows a _____ approach for problem solving.
19. _____ concept was introduced by sir John van Neumann in the late 1940s.

Multiple Choice Questions

1. BIOS is stored in
   (a) RAM (b) ROM
   (c) Hard disk (d) None of these
2. Which language should not be used for organizing large programs?
   (a) C (b) C++
   (c) COBOL (d) FORTRAN
3. Which language is a symbolic language?
   (a) Machine language (b) C
   (c) Assembly language (d) All of these
4. Which language is a 3GL?
   (a) C (b) COBOL
   (c) FORTRAN (d) All of these
5. Which language does not need any translator?
   (a) Machine language (b) 3GL
   (c) Assembly language (d) 4GL
6. Choose the odd one out.
   (a) Compiler (b) Interpreter
   (c) Assembler (d) Linker
7. Which one is a utility software?
   (a) Word processor
   (b) Antivirus
   (c) Desktop publishing tool
   (d) Compiler
8. POST is performed by
   (a) Operating system (b) Assembler
   (c) BIOS (d) Linker
9. Printer, monitor, keyboard, and mouse are examples of
   (a) Operating system
   (b) Computer hardware
   (c) Firmware
   (d) Device drivers
10. Windows VISTA, Linux, Unix are examples of
    (a) Operating system
    (b) Computer hardware
    (c) Firmware
    (d) Device drivers
11. The functionality, capability, performance, availability of hardware and software components are all analysed in which phase?
    (a) Requirements analysis
    (b) Design
    (c) Implementation
    (d) Testing
12. In which phase are algorithms, flowcharts, pseudocodes prepared?
    (a) Requirements analysis
    (b) Design
    (c) Implementation
    (c) Testing
13. Algorithms should be
    (a) Precise (b) Unambiguous
    (c) Clear (d) All of these
14. To check whether a given number is even or odd, you will use which type of control structure?
    (a) Sequence (b) Decision
    (c) Repetition (d) All of these
15. Which among the following is represented using a rectangle?
    (a) Terminal symbols (b) Processing steps
    (c) Input/output symbols (d) Decision symbol
16. Trying to open a file that already exists, will result in which type of error?
    (a) Run time (b) Compile time
    (c) Linker error (d) Logical error
17. Which among the following is an ongoing activity in software development?
    (a) Requirements analysis
    (b) Implementation
    (c) User training
    (d) Maintenance

State True or False

1. Computer hardware does all the physical work.
2. The computer hardware cannot think and make decisions on its own.
3. A software is a set of instructions that are arranged in a sequence to guide a computer to find a solution for the given problem.
4. Word processor is an example of educational software.
5. Desktop publishing system is a system software.
6. BIOS defines firmware interface.
7. Pascal cannot be used for writing well-structured programs.
8. Assembly language is a low-level programming language.
9. Operation code is used to identify and reference instructions in the program.
10. 3GLs are procedural languages.
11. Each phase in software development has a well-defined set of tasks to be performed.
12. Course of actions is planned in requirements analysis phase.

13. Start and end symbols are also known as the terminal symbols.
14. A decision symbol in a flowchart cannot have more than two arrows.
15. Variable declarations are not included in the psuedocode.
16. Logical errors are detected by the compiler.

Review Questions

1. Broadly classify the computer system into two parts. Also make a comparison between a human body and the computer system thereby explaining what each part does.
2. Differentiate between computer hardware and software.
3. Define programming.
4. Define source code.
5. What is booting?
6. What criteria are used to select the language in which the program will be written?
7. Explain the role of operating system.
8. Give some examples of computer software.
9. Differentiate between the source code and the object code.
10. Why are compilers and interpreters used?
11. Is there any difference between a compiler and an interpreter?
12. What is application software? Give examples.
13. What is BIOS?
14. What do you understand by utility software? Is it a must to have it?
15. Differentiate between syntax errors and logical errors.
16. Can a program written in a high-level language be executed without a linker?
17. Give a brief description of generation of programming languages. Highlight the advantages and disadvantages of languages in each generation.
18. What are the advantages of modularization?
19. Briefly explain the phases in software development project.
20. Explain the significance of an algorithm.
21. What are the features of an ideal algorithm?
22. What are the different control structures that are frequently used while writing algorithms?
23. Why is it recommended to draw a flowchart before implementing a program?
24. Explain the significance of different symbols used in a flowchart.
25. What do you understand by a pseudocode?
26. Is it permissible to use keywords in a pseudocode? Justify your answer.
27. What factors determine the efficiency of an algorithm?
28. Testing is an unavoidable phase in software development life cycle. Comment.
29. What are the different types of errors which frequently occur in programs?
30. Differentiate between syntax and semantic errors.
31. Differentiate between unit test, integration test, and system test.
32. What are the different techniques for debugging a computer program?
33. Suppose you are given a problem to find all composite numbers in range provided by users. Perform all the phases of software development on this problem.

ANNEXURE 1

Examples for Program Design Tools

Example 1 Swap Two Variables

To swap two variables we will use a temporary variable. First, we copy the value of any one variable in the temporary variable. Then, we copy the value of second variable in first. Finally, the value of the temporary variable is copied in the second variable. Look at Figure 1 given below to understand this concept.

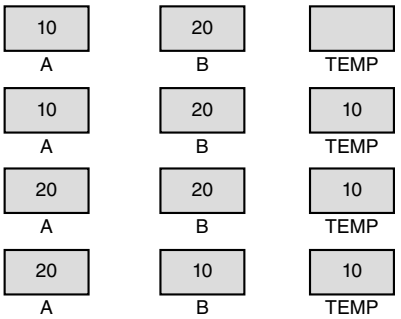


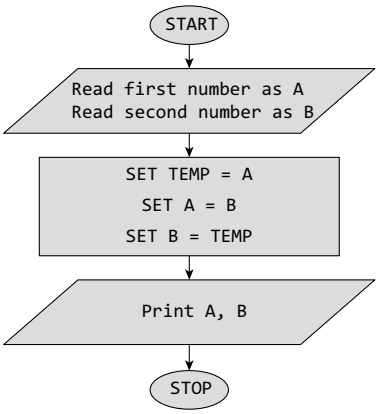
Figure 1 Steps involved in swapping two variables

Algorithm 1 and Flowchart 1 demonstrate the step-wise solution for swapping two variables.

Algorithm 1

```
Step 1: Start
Step 2: Read the first number as A
Step 3: Read the second number as B
Step 4: SET TEMP = A
Step 5: SET A = B
Step 6: SET B = TEMP
Step 7: PRINT A, B
Step 8: End
```

Flowchart 1



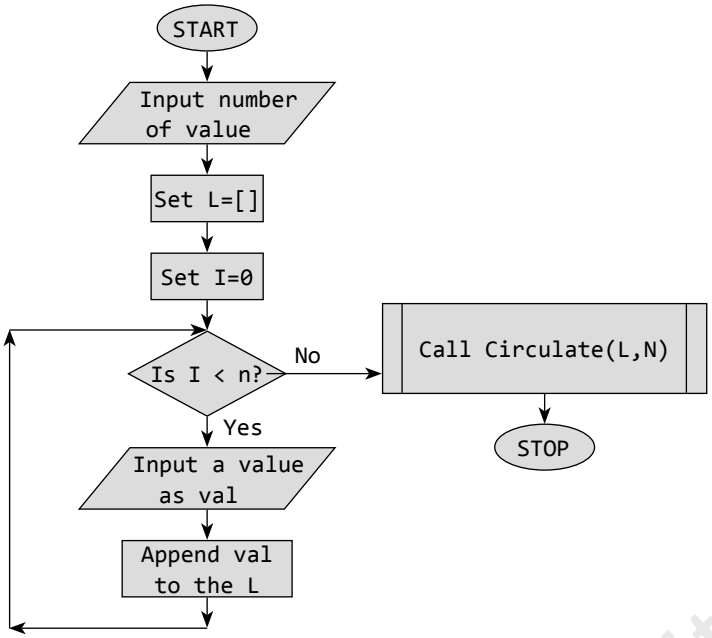
Example 2 Circulate the Values of N Variables

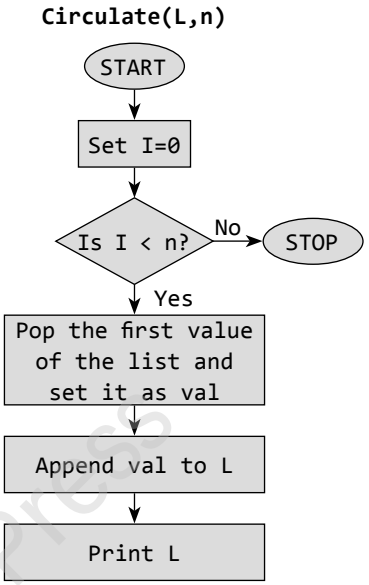
Algorithm 2

```
Step 1: Start
Step 2: Enter the number of elements in the
        list as n
Step 3: SET I = 0
Step 4: WHILE I < n
        Read an element
        Append the element to the list
        Print the list
        Calculate I = I + 1
Step 5: CALL CIRCULATE (list, n)
Step 6: End

CIRCULATE(list, n)
Step 1: Start
Step 2: Set I = 0
Step 3: WHILE I < n
        Pop the first element from the list
        Append it to the list (it now becomes
        the last element)
        Print the list
        Calculate I = I + 1
Step 4: End
```

Flowchart 2



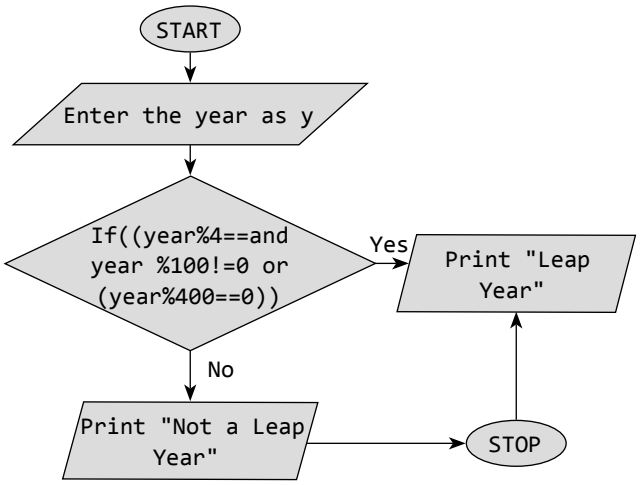


Example 3 Test for Leap Year

Algorithm 3

```
Step 1: Start
Step 2: Enter a year as year
Step 3: Check IF((year%4==0 and year %100!=0)
or (year%400 == 0)),
           Then PRINT "Leap Year"
        ELSE
           PRINT "Not a Leap Year"
Step 4: End
```

Flowchart 3

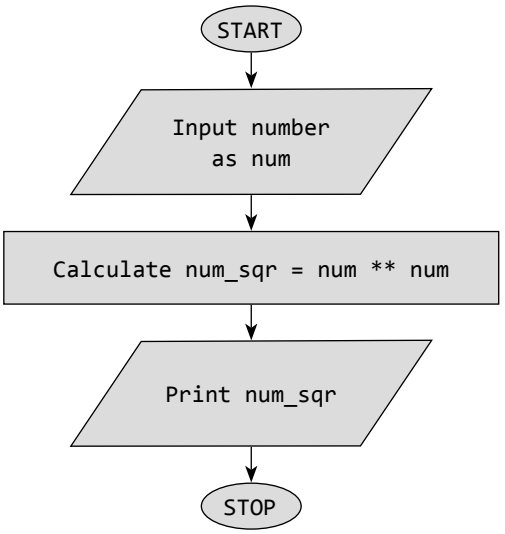


Example 4 Square root of a number

Algorithm 4

```
Step 1: Start
Step 2: Input the number as num
Step 3: Calculate square as num ** num
Step 4: Print square as calculated in Step 3
Step 5: End
```

Flowchart 4

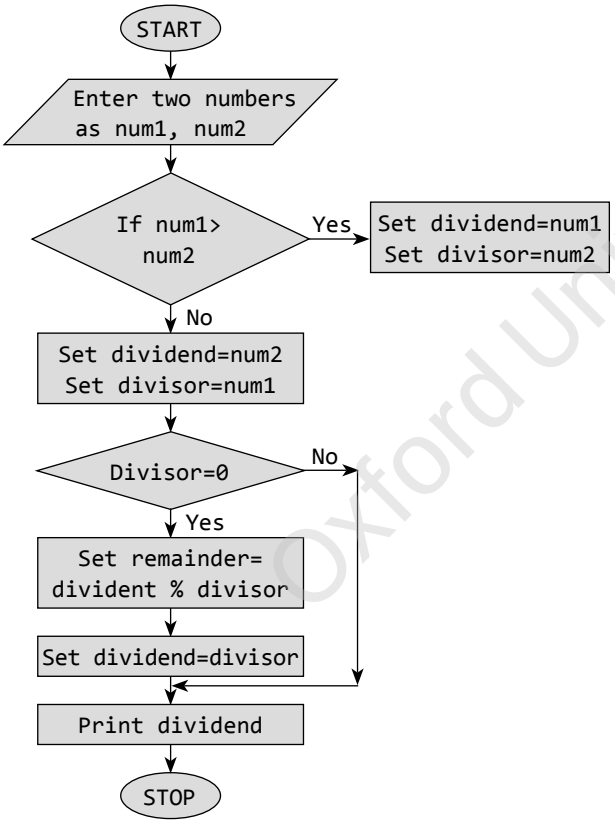


Example 5 GCD of Two Numbers

Algorithm 5

```
Step 1: Start
Step 2: Enter the two numbers as num1 and num2
Step 3: Set larger of the two numbers as
        dividend
Step 4: Set smaller of the two numbers as
        divisor
Step 5: Repeat Steps 6-8 while divisor != 0
Step 6: Set remainder = dividend % divisor
Step 7: Set dividend = divisor
Step 8: Set divisor = remainder
Step 9: Print dividend
Step 10: End
```

Flowchart 5

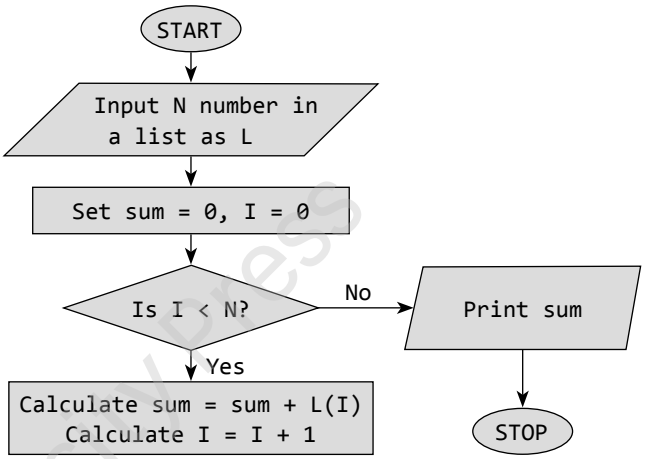


Example 6 Sum an Array of Numbers

Algorithm 6

```
Step 1: Start
Step 2: Read a list of N numbers from the user
        as L
Step 3: Set sum = 0, I = 0
Step 4: WHILE I < N
        Calculate sum = sum + L[I]
        Calculate I = I + 1
Step 5: Print sum
Step 6: End
```

Flowchart 6



Example 7 Counting Words in a File

We know that two consecutive words are separated from each other with a space ' '. So to count the number of words written in the file, we will read the text from the file and count the number of spaces. Number of spaces + 1 gives the count of words as illustrated below using Algorithm 7, Flowchart 7, and Program 3.

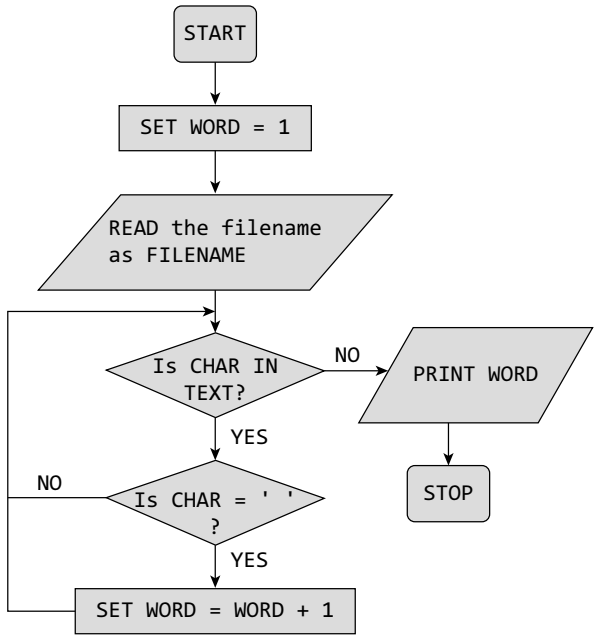
```
HELLO ALL
  WELCOME TO THE WORLD OF PROGRAMMING
```

Here, number of spaces including the new line is 7 and thus, the number of words is 8.

Algorithm 7

```
Step 1: Start
Step 2: SET WORD = 1
Step 3: READ the filename as FILENAME
Step 4: Open the file
Step 5: READ contents of the file in TEXT
Step 6: REPEAT Step 7 WHILE CHAR in TEXT
Step 7:    IF CHAR == ' '
           THEN WORD = WORD + 1
Step 8: PRINT WORD
Step 9: End
```

Flowchart 7

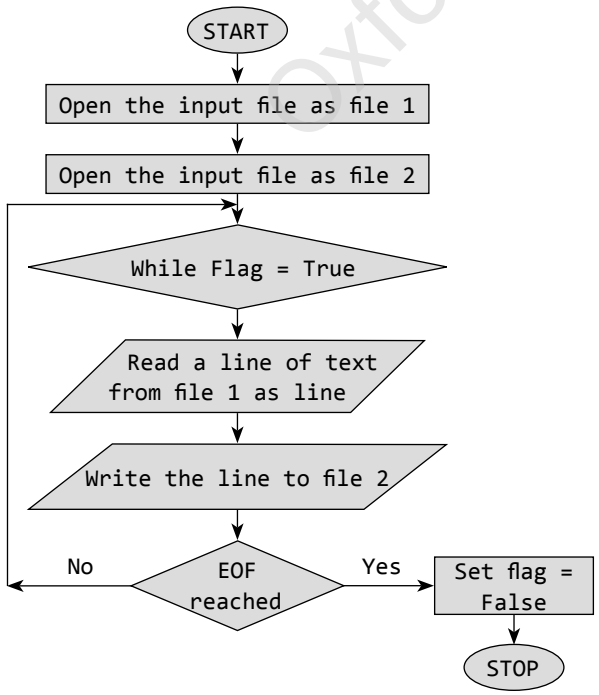


Example 8 Copy a File

Algorithm 8

```
Step 1: Start
Step 2: Open the input file as file1
Step 3: Open the output file as file2
Step 4: WHILE End of file1 is not reached
        Read a line from file1 and write it in
        file2
Step 5: End
```

Flowchart 8



Example 9 Finding Roots of Equations

Algorithm 9

```
Step 1: Start
Step 2: Input three values as a, b and c
Step 3: Set D = (b X b) - (4 X a X c)
Step 4: If D < 0,
              Print "Imaginary Roots".
        Else If D = 0,
              Set x = -b / (2a)
              Print x
        Else
              Set x1 = (-b + sqrt(D))/2a
        Set x2 = (-b - sqrt(D))/2a
              Print x1, x2
Step 5: End
```

Flowchart 9