Programming in C

Second Edition

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 trademark of
Oxford University Press in the UK and in certain other countries.

Published in India by
Oxford University Press
YMCA Library Building, 1 Jai Singh Road, New Delhi 110001, India

First edition published in 2011
Second edition published in 2015

ISBN-13: 978-0-19-945614-7
ISBN-10: 0-19-945614-3

Typeset in Times New Roman
by Pee-Gee Graphics, New Delhi
Printed in India by Magic International (P) Ltd., Greater Noida

Features of the Book

Introduction to Programming	**Linked Lists**
Introduction to C	**Stacks and Queues**
Decision Control and Looping Statements	**Trees**
Functions	**Graphs**

Comprehensive Coverage

The book provides comprehensive coverage of topics ranging from fundamental concepts of C programming to data structures

Notes

These elements highlight the important terms and concepts discussed in each chapter

Note

The `printf` and `return` statements have been indented or moved away from the left side. This is done to make the code more readable.

1. Find out the output of the following program.

```
#include <stdio.h>
int main()
{
   int a, b;
   printf("\n Enter two four digit numbers : ");
   scanf("%2d %4d", &a, &b);
   printf("\n The two numbers are : %d and
     %d", a, b);
   return 0;
}
```

Output

```
Enter two four digit numbers : 1234 5678
The two numbers are : 12 and 34
```

Programming Examples

As many as 260 C programs are included, which demonstrate the applicability of the concepts learned

Points to Remember

A list of key topics at the end of each chapter helps readers to revise all the important concepts explained in the chapter

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.
- Application software is designed to solve a particular problem for users.

Glossary

All chapters provide a list of key terms along with their definitions for a quick recapitulation of important terms learned

GLOSSARY

ANSI C American National Standards Institute's definition of the C programming language. It is the same as the ISO definition.

Constant A value that cannot be changed.

Data type Defines the type of values that a data can take. For example, int, char, float.

Escape sequence Control codes that comprise of combinations of a backslash followed by letters or digits which represent non-printing characters.

Expression A sequence of operators and operands that may yield a single value as the result of its computation.

C

Case Study 1: Chapters 2 and 3

We have learnt the basics of programming in C language and concepts to write decision-making programs, let us now apply our learning to write some useful programs.

ROMAN NUMERALS

Roman numerals are written as combinations of the seven letters. These letters include:

I = 1	C = 100
V = 5	D = 500
X = 10	M = 1000
L = 50	

Roman Numeral Table

1	I	14	XIV	27	XXVII	150	CL
2	II	15	XV	28	XXVIII	200	CC
3	III	16	XVI	29	XXIX	300	CCC
4	IV	17	XVII	30	XXX	400	CD
5	V	18	XVIII	31	XXXI	500	D
6	VI	19	XIX	40	XL	600	DC
7	VII	20	XX	50	L	700	DCC
8	VIII	21	XXI	60	LX	800	DCCC
9	IX	22	XXII	70	LXX	900	CM
10	X	23	XXIII	80	LXXX	1000	M
11	XI	24	XXIV	90	XC	1600	MDC

Case Studies

Select chapters are followed by case studies that show how C can be used to create programs demonstrating real-life applications

Programming Tips

These elements educate readers about common programming errors and how to resolve them

Programming Tip:
If you do not place a parenthesis after 'main', a compiler error will be generated.

Programming Exercises

1. Write a program which deletes all duplicate elements from the array.
2. Write a program that tests the equality of two one-dimensional arrays.
3. Write a program that reads an array of 100 integers. Display all pairs of elements whose sum is 50.

Programming Exercises

Numerous exercises at the end of every chapter test the readers' understanding of the concepts learned.

Companion Online Resources for Instructors and Students

Visit www.oupinheonline.com to access both teaching and learning solutions online.

The following resources are available to support the faculty and students using this book:

For Faculty

- Solutions manual
- PowerPoint slides
- Projects

For Students

- Multiple Choice Questions
- Projects
- Model Question Papers
- Supplementary Reading Material on Graphics/Mouse Programming

Steps to Register

Step 1: Getting Started

- Go to www.oupinheonline.com

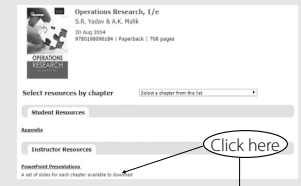
Step 2: Browse quickly by:

- BASIC SEARCH
  - → Author
  - → Title
  - → ISBN
- ADVANCED SEARCH
  - ▷ Subjects
  - ▷ Recent titles

Step 3: Select Title

Select title for which you are looking for resources.

Step 4: Search Results with Resources available

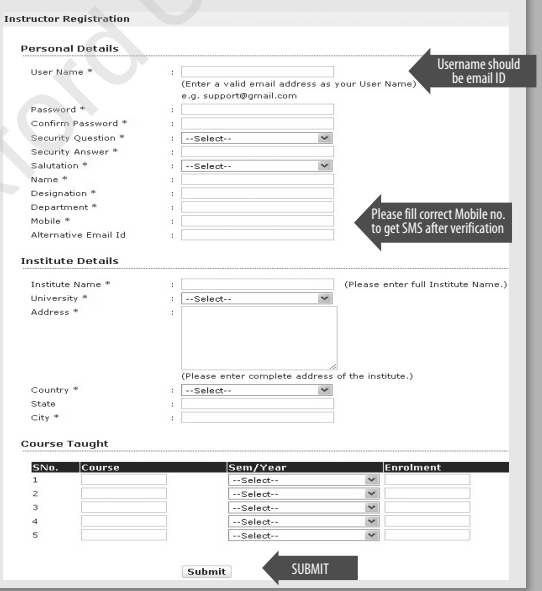


Step 5: To download Results

Step 6: Login to download

Step 7: Registration Form

Please fill correct details and *marked fields are mandatory



Step 8: Message after completing the registration form

Thank you for registering with us. We shall revert to you within 48 hours after verifying the details provided by you. Once validated please login using your username and the password and access the resources.

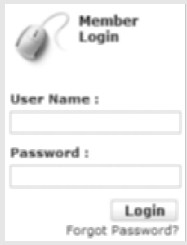
Step 9: Verification

You will receive a confirmation on your mobile & email ID.

For any further query please write to us at **HEMarketing.in@oup.com** with your mobile number

Step 10: Visit us again after validation

- Go to www.oupinheonline.com
- Login from Member Login



Step 11: My Subscriptions

You can view Subscriptions in your account

- Click on the title
- Select Chapter or "Select All"
- Click on "Download All"
- Click on "I Accept"'
- A zip file will be downloaded on your system. You may use this along with the textbook.

Preface to the Second Edition

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. Efforts have 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
- *Numerous programs* that have been tested and executed
- *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
- *Case studies* containing programs that harness the concepts learnt in chapters
- *Appendices* at the end of book provide additional information to enthusiastic readers

NEW TO THIS EDITION

- A chapter on *Developing Efficient Programs*, which details steps for developing correct, efficient, and maintainable programs
- An annexure on how to write and execute C programs on platforms such as Unix / Linux and Ubuntu
- Sections on C tokens, structures inside unions, array representation of sparse matrices, and applications of arrays, queues, and trees

CONTENT AND COVERAGE

The book is organized into 15 chapters.

Chapter 1 provides an introduction to computer software. It also provides an insight into different programming languages and the generations through which these languages have evolved.

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. Annexure 1 shows the steps to write, compile, and execute a C program in Unix, Linux, and Ubuntu environments.

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

Case Study 1 includes two programs which harness the concepts learnt in Chapters 2 and 3.

Chapter 4 deals with declaring, defining, and calling functions. The chapter also discusses the storage classes as well as variable scope in C. The chapter ends with the important concept of recursive functions and a discussion on the Tower of Hanoi problem. Annexure 2 discusses how to create user-defined header files.

Chapter 5 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.

Case Study 2 provides an introduction to sorting and various sorting techniques such as bubble sort, insertion sort, and selection sort.

Chapter 6 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 7 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. Annexure 3 explains the process of deciphering pointer declarations.

Case Study 3 includes a program which demonstrates how pointers can be used to access and manipulate strings.

Chapter 8 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. Annexure 4 provides an explanation about bit fields and slack bytes.

Chapter 9 explains how data can be stored in files. The chapter deals with opening, processing, and closing of files though a C program. These files are handled in text mode as well as binary mode for better clarity of the concepts.

Chapter 10 discusses the concept of pre-processor directives. The chapter includes small program codes that illustrate the use of different directives in a C program. Annexure 5 provides an introduction to data structures.

Chapter 11 discusses different types of linked lists such as singly linked lists, doubly linked lists, circular linked lists, circular doubly linked lists, and header linked lists. Asa 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.

Case Study 4 shows how a telephone directory can be implemented using C.

Chapter 12 introduces data structures—stacks and queues. In this chapter, these data structures are implemented using arrays. The chapter also provides the operations and applications of stacks and queues in the world of programming.

Case Study 5 with the help of programs shows how linked lists can be used to implement stacks and queues.

Chapter 13 focuses on binary trees, their traversal schemes and representation in memory.

The chapter also discusses expression trees which is a variant of simple binary trees.

Case Study 6 discusses Huffman coding, a loss less data compression technique, which works by creating a binary tree of nodes that are stored in an array.

Chapter 14 provides an introduction to graphs. The chapter discusses the memory representation, traversal schemes, and applications of graphs in the real world.

Case Study 7 deals with topological sorting of directed acyclic graphs which is mainly used for scheduling of tasks.

Chapter 15 details the different steps in software development process which are performed for creating efficient and correct programs. It also explains the different tools which are used to obtain solution(s) of a given problem at hand as well as different types of errors and testing and debugging approaches.

The book also provides a set of six appendices. *Appendix A* discusses the different variants of C language. *Appendix B* shows the ASCII codes of characters. *Appendix C* lists some of the ANSI C library functions and their descriptions. *Appendix D* introduces some advanced type qualifiers as well as in line functions. *Appendix E* discusses bit-level programming and some of the bitwise operations. *Appendix F* provides answers to objective questions.

ONLINE RESOURCES

The following resources are available at Oxford University Press India's Higher Education Companion Site (http://oupinheonline.com) to support the faculty and students using this text:

For Faculty

- Solutions Manual
- PowerPoint Slides
- Projects

For Students

- Multiple Choice Questions
- Projects
- Model Question Papers
- Supplementary Reading Material on Graphics/ Mouse Programming

ACKNOWLEDGMENTS

I would like to gratefully acknowledge the feedback and suggestions given by various faculty members for the improvement of the book. I am obliged to the editorial team of Oxford University Press India for all their support towards revising this book. Suggestions for improving the presentation and contents can be sent to the publishers through their website www.oup.com or to the author at reemathareja@gmail.com.

ReemaThareja

Preface to the First Edition

C is the most popular and widely used language for developing computer programs. Programmers all over the world embrace the C language as it provides them maximum control and efficiency. One of the major benefits of learning C is its user friendliness that allows the user to read and write codes for a wide range of platforms, ranging from embedded micro controllers to advanced scientific systems. The versatility of C is apparent from the fact that many modern operating systems have used C to develop their core.

Developed by Dennis Ritchie in 1972, C has derived numerous features from its precursors such as ALGOL, BCPL, and B. Its tremendous growth resulted in the development of different versions of the language that were similar but incompatible with each other, for example, the traditional C and K&R C. Therefore, the American National Standards Institute (ANSI) in December 1989 defined a standard for the language and renamed it as ANSI C. In 1990, the International Standards Organization (ISO) adopted the ANSI standard. This version of C was known as C89. In 1995, some minor changes were made to C89 and the new modified version was known as C95. During 1990s, C++ and Java became popular among the users and hence the standardization committee felt the need to integrate a few features of C++/Java in C to enhance its usefulness. Some significant changes were made in 1999 and the modified version became C99.

Since C is such a widely accepted and used language, jumping to other modern-day programming languages such as C++ and Java becomes much easier for a person who is well acquainted with C. The programming constructs in C, such as 'if' statements, 'for' and 'while' loops and the types of variables used can be found in many modern languages, therefore the ideas expressed in C are well understood by program developers. Moreover, unlike most of these modern-day languages that apply object-oriented design, C applies a procedural style of design, i.e., it uses procedures such as functions, methods, or routines to call itself or other procedures. However, many applications are still better suited to the procedural style of design, which often goes untaught to many programmers, who focus exclusively on object-oriented design. Learning C provides a strong procedural background, which is a worthy skill set.

Some other features of C that give it the tag of the most widely used professional language include its *portability*, that is, a C program written on one computer can be made to run on any other computer with a little or no modification and its *extensibility*, that is, any number of functions can be added to the C library and called any number of times in the program, thus making the code easier to write and understand. Therefore, it is not only desirable but also necessary to have a strong fundamental knowledge of the C language.

ABOUT THE BOOK

Programming in C is designed to serve as a textbook for undergraduate-level courses in computer science and engineering and postgraduate-level courses of computer applications. The book explains the fundamental concepts

of the C programming language and shows a step-by-step approach of how to apply these concepts for solving real-world problems.

Unlike existing textbooks on C which concentrate more on the theory, this book focuses on its applicability angle in addition to the theory by providing numerous programming examples and a rich set of programming exercises at the end of each chapter. The book is also useful as a reference and resource to computer professionals operating in the domain of C and other C-like languages.

Every chapter in this book contains a number of programming examples to impart practical knowledge of the concepts. To reinforce the concepts, there are numerous objective, subjective, and programming exercises at the end of each chapter.

ACKNOWLEDGEMENTS

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 go to my uncle Mr B. L. Thareja 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

1 Introduction to Programming

Takeaways

- Hardware
- Application software
- Assembly language
- System software
- Generation of programming languages
- Procedural and non-procedural languages
- Compiler, interpreter, linker, loader
- Machine language

1.1 INTRODUCTION TO COMPUTER SOFTWARE

When we talk about a computer, we actually mean two things (Figure 1.1):

- First is the computer hardware that does all the physical work computers are known for.
- Second is the computer software that commands the hardware what to do and how to do it.

If we think of computer as a living being, then the hardware would be the body that does things like seeing with eyes and lifting objects with hands, whereas the software would be the intelligence which helps in interpreting the images that are seen by the eyes and instructing the arms how to lift objects.

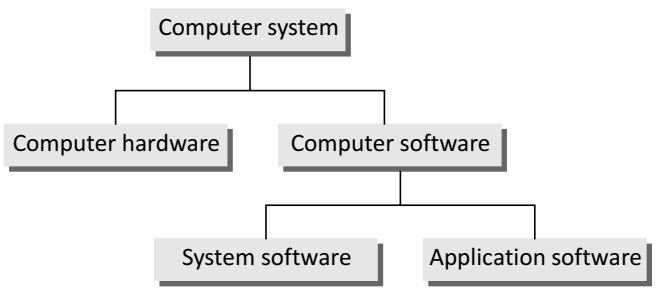


Figure 1.1 Parts of a computer system

Since computer hardware is a digital machine, it can only understand two basic states: on and off. Computer software was developed to make efficient use of this binary system which is used internally by all computers to instruct the hardware to perform meaningful tasks.

The computer hardware cannot think and make decisions on its own. So, it cannot be used to analyse a given set of data and find a solution on its own. The hardware needs a software (a set of programs) to instruct what has to be done. 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. The programmer writes a set of instructions (program) using a specific programming language. Such instructions are known as the source code. Another computer program called a compiler is used which transforms the source code into a language that the computer can understand. The result is an executable computer program, which is another software.

Examples of computer software include:

- Computer games which are widely used as a form of entertainment.

- Driver software which allows a computer to interact with hardware devices such as printers, scanners, and video cards.
- Educational software which includes programs and games that help in teaching and providing drills to help memorize facts. Educational software can be diversely used—from teaching computer-related activities like typing to higher education subjects like Chemistry.
- Media players and media development software that are specifically designed to play and/or edit digital media files such as music and videos.
- Productivity software is an older term used to denote any program that allows users to be more productive in a business environment. Examples of such software include word processors, database management utilities, and presentation software.
- Operating system software which helps in coordinating system resources and allows execution of other programs. Some popular operating systems are Windows Vista, Mac OS X, and Linux.

1.2 CLASSIFICATION OF 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 1.2 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.

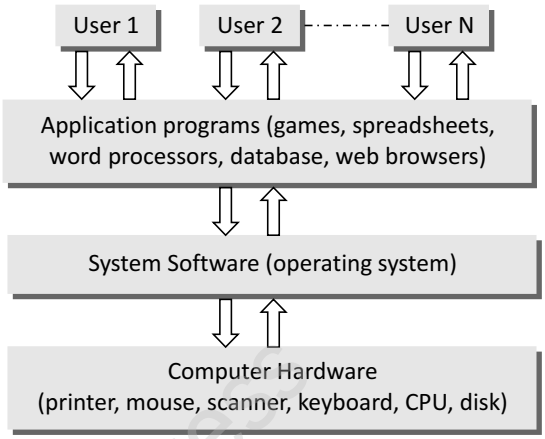


Figure 1.2 Relationship between hardware, system software, and application software

1.2.1 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 1.3) 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.

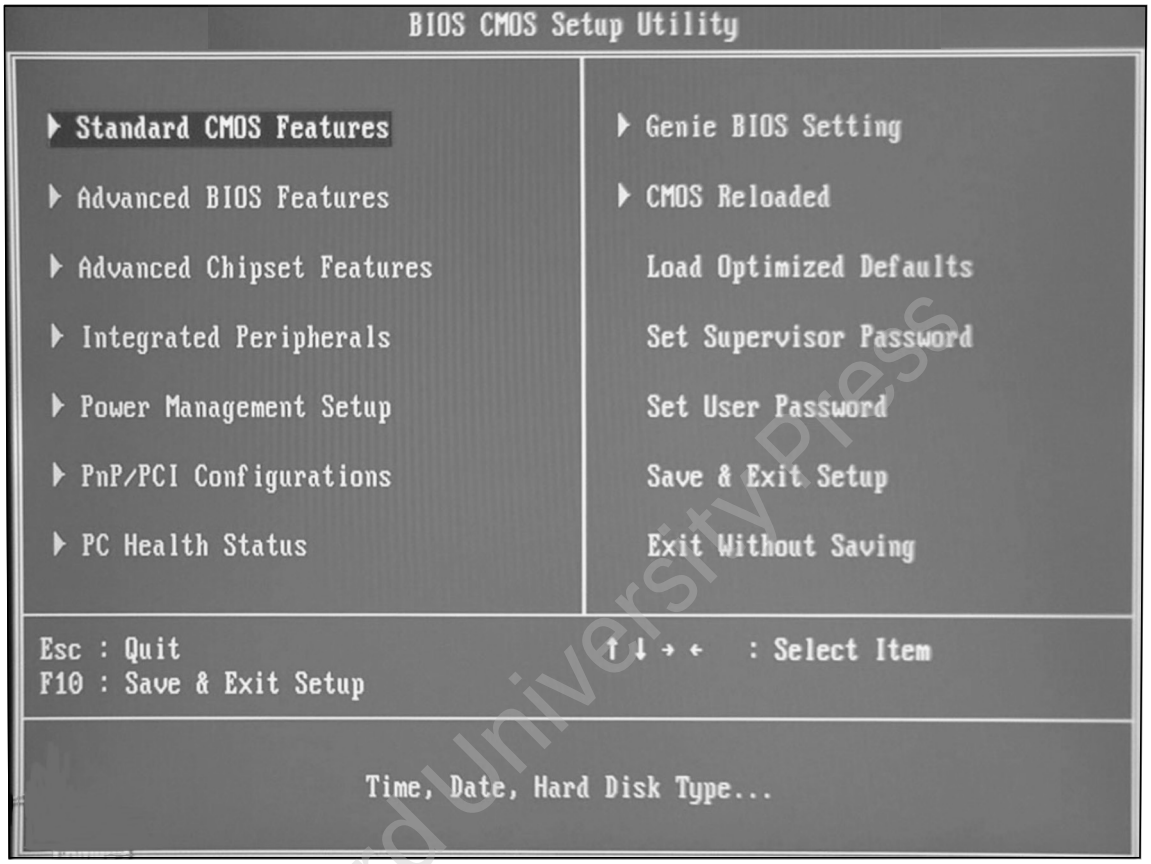


Figure 1.3 BIOS menu

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) c*onvenient* 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 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.

1.2.2 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 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.

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.

1.4 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)

1.4.1 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 1.4).

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 1.4 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.

1.4.2 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.

1.4.3 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 *translato*r 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.

1.4.4 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.

1.4.5 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.

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.
- 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.

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 _____.

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

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.

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.