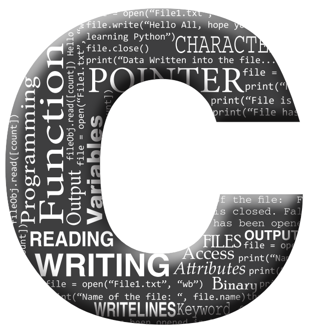
Programming in C



CS 8251 Programming in C
As per Anna University R17 syllabus

Revised First 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 trade mark of
Oxford University Press in the UK and in certain other countries.

Published in India by
Oxford University Press
22 Workspace, 2nd Floor, 1/22 Asaf Ali Road, New Delhi 110002

First published in 2018
Revised first edition published in 2020

ISBN-13: 978-0-19-012655-1
ISBN-10: 0-19-012655-8

Typeset in Times New Roman
by Pee-Gee Graphics, New Delhi
Printed in India by

Cover illustration: Alok Rawat

For product information and current price, please visit www.india.oup.com

Features of the Book

Introduction to Programming	**Arrays**
Introduction to C	**Strings**
Decision Control and Looping Statements	**Pointers**
Functions	**Files**

Comprehensive Coverage

The book provides comprehensive coverage of C programming constructs.

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 250 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 soft are 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 soft are is wri en by computer programmers 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.

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 a er '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.

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

Programming in C is designed to serve as a textbook for the first year engineering students of Anna University. This book is developed as per the latest syllabus (2017) of affiliated colleges.

This 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 theory, this book focuses on its applicability angle by providing numerous programming examples and a rich set of programming exercises at the end of each chapter.

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

CONTENT AND COVERAGE

The book is organized into 11 chapters.

Chapter 1 provides an introduction to computer software. It also provides an insight into different programming paradigms, 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 linked list which 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 linked lists.

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

The book also provides two appendices. Appendix A includes additional C programs, and Appendix B provides answers to objective questions.

ACKNOWLEDGMENTS

The mammoth task of writing this book required the help and support of many individuals. Fortunately, I have had the wholehearted support of my family and friends. I would like to specially thank my father Mr Janak Raj Thareja and mother Mrs Usha Thareja, my brother Pallav, and sisters Kimi and Rashi who are a source of abiding inspiration and divine blessings for me. I am especially thankful to my son Goransh who has been very patient and cooperative in letting me realize my dreams.

Finally, I would like to thank the editorial team at Oxford University Press India for their help and support.

ReemaThareja

Brief Contents

Detailed Contents

1 Introduction to Programming

Takeaways

- Hardware
- Application so are
- Assembly language
- Procedural programming
- System soft are
- Generation of p ogramming languages
- Monolithic programming
- Object-oriented programming
- Compiler, interpreter, linker, loader
- Machine language
- Structured programming

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.

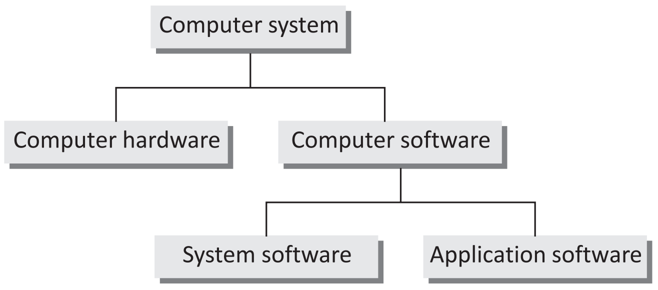


Figure 1.1 Parts of a computer system

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.

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.

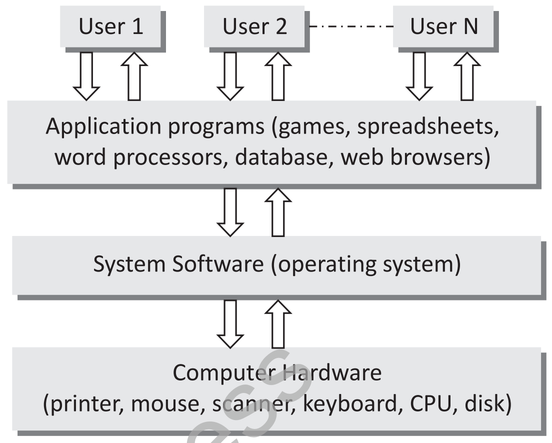

User 1
User 2
User N
Application programs (games, spreadsheets, word processors, database, web browsers)
System Software (operating system)
Computer Hardware
(printer, mouse, scanner, keyboard, CPU, disk)

Figure 1.2 Relationship b tween hardware, system soft are, and application so are

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

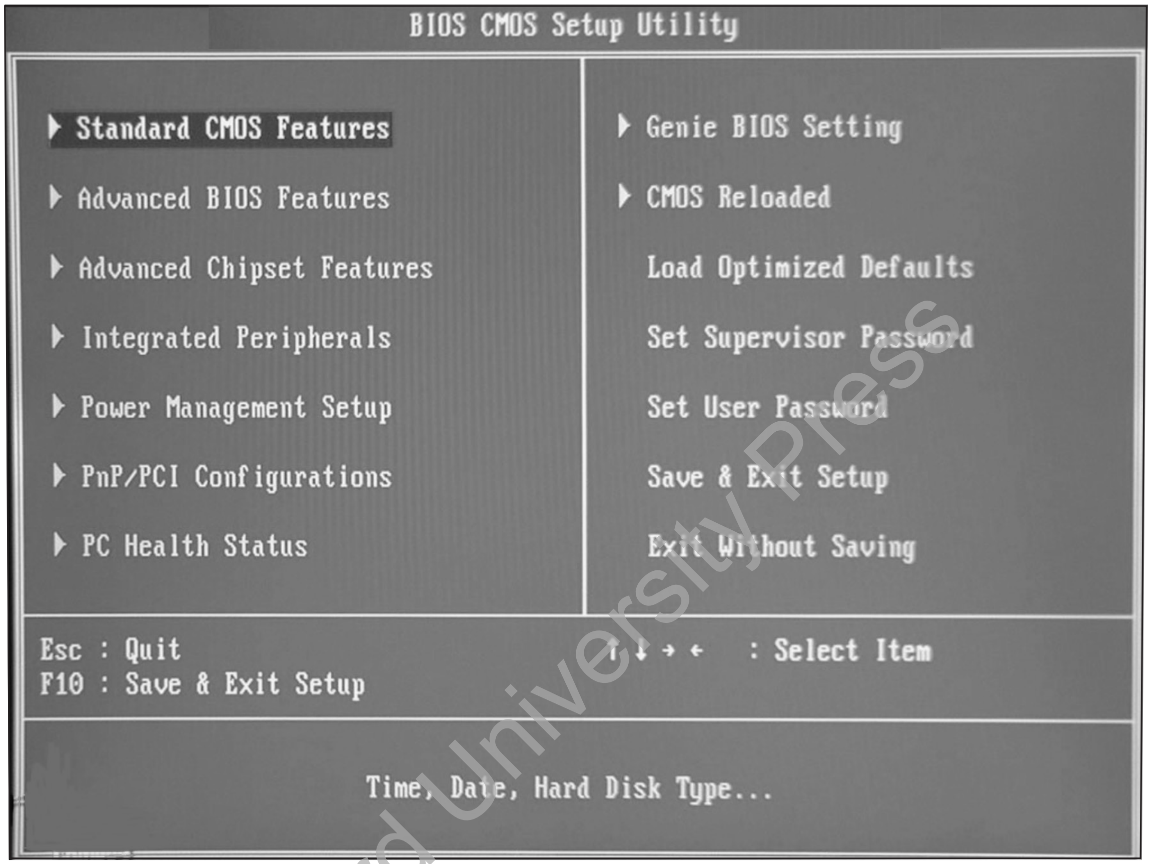
The computer BIOS and device drivers provide basic functionality to operate and control the hardware connected to or built into the computer.

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.


BIOS CMOS Setup Utility

▶ Standard CMOS Features
▶ Advanced BIOS Features
▶ Advanced Chipset Features
▶ Integrated Peripherals
▶ Power Management Setup
▶ PnP/PCI Configurations
▶ PC Health Status

▶ Genie BIOS Setting
▶ CMOS Reloaded
Load Optimized Defaults
Set Supervisor Password
Set User Password
Save & Exit Setup
Exit Without Saving

Esc : Quit
F10 : Save & Exit Setup
↑ ↓ → ← : Select Item

Time, Date, Hard Disk Type...

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.

1.5 PROGRAMMING PARADIGMS

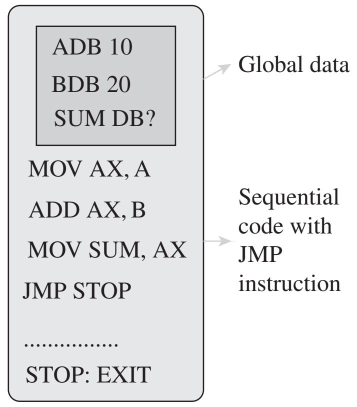
A programming paradigm is a fundamental style of programming that defines how the structure and basic elements of a computer program will be built. The style of writing programs and the set of capabilities and limitations that a particular programming language has depends on the programming paradigm it supports. While some programming languages strictly follow a single paradigm, others may draw concepts from more than one. The sweeping trend in the evolution of high-level programming languages has resulted in a shift in programming paradigm. These paradigms, in sequence of their application, can be classified as follows:

- Monolithic programming—emphasizes on finding a solution
- Procedural programming—lays stress on algorithms
- Structured programming—focuses on modules
- Object-oriented programming—emphasizes on classes and objects
- Logic-oriented programming—focuses on goals usually expressed in predicate calculus
- Rule-oriented programming—makes use of 'if-then-else' rules for computation
- Constraint-oriented programming—utilizes invariant relationships to solve a problem

Each of these paradigms has its own strengths and weaknesses and no single paradigm can suit all applications. For example, for designing computation-intensive problems, procedure-oriented programming is preferred; for designing a knowledge base, rule-based programming would be the best option; and for hypothesis derivation, logic-oriented programming is used. In this book, we will discuss only the first four paradigms.

1.5.1 Monolithic Programming

Programs written using monolithic programming languages such as assembly language and BASIC consist of global data and sequential code. The global data can be accessed and modified (knowingly or mistakenly) from any part of the program, thereby posing a serious threat to its integrity. A sequential code is one in which all instructions are executed in the specified sequence. In order to change the sequence of instructions, jump statements or 'goto' statements are used. Figure 1.5 shows the structure of a monolithic program. As the name suggests, monolithic programs have just one program module as such programming languages do not support the concept of subroutines. Therefore, all the actions required to complete a particular task are embedded within the same application itself. This not only makes the size of the program large but also makes it difficult to debug and maintain.



```
ADB 10
BDB 20
SUM DB?
MOV AX, A
ADD AX, B
MOV SUM, AX
JMP STOP
...............
STOP: EXIT
```

Figure 1.5 Structure of a monolithic program

For all these reasons, monolithic programming language is used only for very small and simple applications where reusability is not a concern.

1.5.2 Procedural Programming

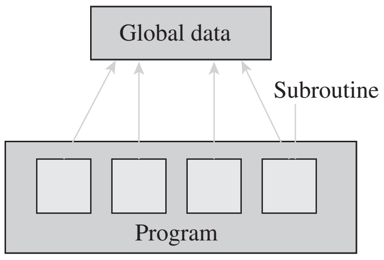


Figure 1.6 Structure of a procedural program

In procedural languages, a program is divided into subroutines that can access global data. To avoid repetition of code, each subroutine performs a well-defined task. A subroutine that needs the service provided by another subroutine can call that subroutine. Therefore, with 'jump', 'goto', and 'call' instructions, the sequence of execution of instructions can be altered. Figure 1.6 shows the structure of a procedural language. FORTRAN and COBOL are two popular procedural programming languages.

Advantages

- The only goal is to write correct programs.
- Programs are easier to write as compared to monolithic programming.

Disadvantages

- No concept of reusability
- Requires more time and effort to write programs
- Programs are difficult to maintain
- Global data is shared and therefore may get altered (mistakenly)

1.5.3 Structured Programming

Structured programming, also referred to as modular programming, was first suggested by mathematicians, Corrado Bohm and Guiseppe Jacopini in 1966. It was specifically designed to enforce a logical structure on the program to make it more efficient and easier to understand and modify. Structured programming was basically defined to be used in large programs that require large development team to develop different parts of the same program. Structured programming employs a top-down approach in which the overall program structure is broken down into separate modules. This allows the code to be loaded into memory more efficiently and also be reused in other programs. Modules are coded separately and once a module is written and tested individually, it is then integrated with other modules to form the overall program structure (refer to Fig. 1.7). Structured programming is, therefore, based on modularization which groups related statements together into modules. Modularization makes it easier to write, debug, and understand the program. Ideally, modules should not be longer than a page. It is always easy to understand a series of 10 single-page modules than a single 10-page program. For large and complex programs, the overall program structure may further require the need to break the modules into subsidiary pieces. This process continues until an individual piece of code can be written easily. Almost every modern programming language similar to C, Pascal, etc., supports the concepts of structured programming. In addition to the techniques of structured programming for writing modules, it also focuses on structuring its data. In structured programming, the program flow follows a simple sequence and usually avoids the use of 'goto' statements. Besides sequential flow, structured programming also supports selection and repetition as mentioned here.

- Selection allows for choosing any one of a number of statements to execute, based on the current status of the program. Selection statements contain keywords such as 'if', 'then', 'end if', or 'switch' that help to identify the order as a logical executable.
- In repetition, a selected statement remains active until the program reaches a point where there is a need for some other action to take place. It includes keywords such as 'repeat', 'for', or 'do… until'. Essentially, repetition instructs the program as to how long it needs to continue the function before requesting further instructions.

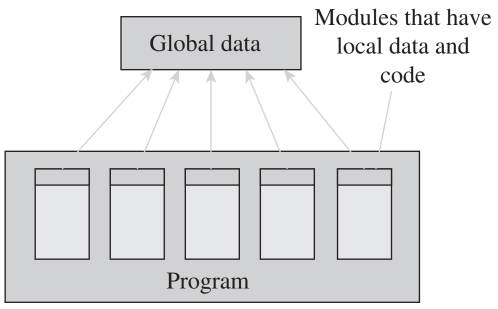


Figure 1.7 Structured program

Advantages

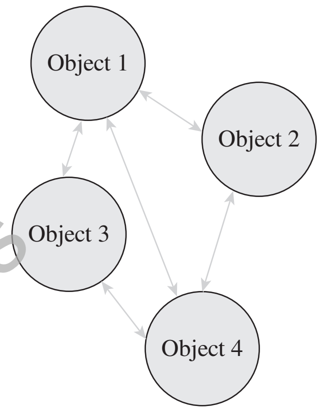
- The goal of structured programming is to write correct programs that are easy to understand and change.
- Modules enhance programmers' productivity by allowing them to look at the big picture first and focus on details later.
- With modules, many programmers can work on a single, large program, with each working on a different module.
- A structured program takes less time to be written than other programs. Modules or procedures written for one program can be reused in other programs as well.
- Each module performs a specific task.
- Each module has its own local data.
- A structured program is easy to debug because each procedure is specialized to perform just one task and every procedure can be checked individually for the presence of any error. In striking contrast, unstructured programs consist of a sequence of instructions that are not grouped for specific tasks. Their logic is cluttered with details and, therefore, difficult to follow.
- Individual procedures are easy to change as well as understand. In a structured program, every procedure has meaningful names and has clear documentation to identify the task performed by it. Moreover, a correctly written structured program is self-documenting and can be easily understood by another programmer.
- More emphasis is given on the code and the least importance is given to the data.

Disadvantages

- Not data-centred
- Global data is shared and therefore may get inadvertently modified
- Main focus is on functions

1.5.4 Object-oriented Programming (OOP)

With the increase in size and complexity of programs, there was a need for a new programming paradigm that could help to develop maintainable programs. To implement this, the flaws in previous paradigms had to be corrected. Consequently, OOP was developed. It treats data as a critical element in the program development and restricts its flow freely around the system. We have seen that monolithic, procedural, and structured programming paradigms are task-based as they focus on the actions the program should accomplish. However, the object-oriented paradigm is task-based and data-based. In this paradigm, all the relevant data and tasks are grouped together in entities known as objects (refer to Fig. 1.8). For example, consider a list of numbers. The procedural or structured programming paradigm considers this list as merely a collection of data. Any program that accesses this list must have some procedures or functions to process this list. For example, to find the largest number or to sort the numbers in the list, we need specific procedures or functions to do the task. Therefore, the list is a passive entity as it is maintained by a controlling program rather than having the responsibility of maintaining itself. However, in the object-oriented paradigm, the list and the associated operations are treated as one entity known as an object. In this approach, the list is considered an object consisting of the list, along with a collection of routines for manipulating the list. In the list object, there may be routines for adding a number to the list, deleting a number from the list, sorting the list, etc. The major difference between OOP and traditional approaches is that the program accessing this list need not contain procedures for performing tasks; rather, it uses the routines provided in the object. In other words, instead of sorting the list as in the procedural paradigm, the program asks the list to sort itself. Therefore, we can conclude that the object-oriented paradigm is task-based (as it considers operations) as well as data-based (as these operations are grouped with the relevant data).



Objects of a program interact by sending messages to each other

Figure 1.8 Object-oriented paradigm

The striking features of OOP include the following:

- Programs are data centred.
- Programs are divided in terms of objects and not procedures.
- Functions that operate on data are tied together with the data.
- Data is hidden and not accessible by external functions.

- New data and functions can be easily added as and when required.
- Follows a bottom-up approach for problem solving.

In the forthcoming chapters, we are going to study C programming language which supports both procedural as well as structured programming.

POINTS TO REMEMBER

- A computer has two parts—computer hardware which does all the physical work and computer soft are 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 soft are is wri en by computer programmers using a programming language.
- Application soft are is designed to solve a particular problem for users.
- System soft are represents programs that allow the hardware to run properly. It acts as an interface between the hardware of the computer and the application soft are 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 soft are is used to analyse, configu e, optimi e, and maintain the computer system.
- A compiler is a special type of program that transforms source code wri en 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 wri en 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.
- Programs wri en using monolithic programming languages such as assembly language and BASIC consist of global data and sequential ode.
- In procedural languages, a program is divided into subroutines th t can access global data.
- Structured programming employs a top-down approach in which the overall program is broken down into separate modules.
- Object-oriented programming treats data as a criti al element in the program development and restricts its fl w freely around the system.

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 p ogram is called _______.
4. ________ is used to write computer soft are.
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 pla orm for running applicatio soft are.
9. ________ can be used to encrypt and decrypt fil s.
10. An assembly language statement consists of a ________, an ________, and ________.
11. ________ and ________ statements are used to change the sequence of execution of in tructions
12. ________ paradigm supports bo om-up approach of problem-solving.
13. FORTRAN and COBOL are two popular ________ programming languages.

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 so are?
   (a) Word processor
   (b) Antiviru
   (c) Desktop publishing tool
   (d) Compiler
8. POST is performed by
   (a) Operating ystem (b) Assembler
   (c) BIOS (d) Linker
9. Printer, monitor, keyboard, and mouse are examples of
   (a) Operating ystem (b) Computer hardware
   (c) Firmware (d) Device drivers
10. Windows VISTA, Linux, Unix are examples of
   (a) Operating ystem (b) Computer hardware
   (c) Firmware (d) Device drivers
11. Which programming paradigm utili es invariant relationshi s to solve a problem?
   (a) Rule-based (b) Constraint-based
   (c) Structured (d) Object-oriented
12. Which is the preferred paradigm for designing a knowledge base?
   (a) Rule-based (b) Constraint-based
   (c) Structured (d) Object-oriented
13. Which type of programming does not support subroutines
   (a) Monolithic (b) Structured
   (c) Rule-based (d) Object-oriented
14. C and Pascal belong to which type of programming language?
   (a) Monolithic (b) Structured
   (c) Logic-oriented (d) Object-oriented
15. Which paradigm holds data as a priority?
   (a) Monolithic (b) Structured
   (c) Logic-oriented (d) Object-oriented

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 soft are 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 so are.
5. Desktop publishing system is a system soft are.
6. BIOS defines fi ware 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 identi y and reference instructions in the p ogram.
10. 3GLs are procedural languages.
11. In monolithic paradigm, global data can be accessed and modified f om any part of the program.
12. Monolithic programs have two modules.
13. Monolithic programs are easy to debug and maintain.
14. Structured programming is based on modularization
15. Object-oriented programming supports modularization
16. Structured programming heavily uses goto statements.
17. Modules enhance programmers' productivit .
18. A structured program takes more time to be wri en than other programs.

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. Di erenti te between computer hardware and soft are.
3. Define p ogramming.
4. Define sou ce code.
5. What is booting
6. What criteria are used to select the language in which the program will be wri en?
7. Explain the role of operating ystem.
8. Give some examples of computer soft are.
9. Di erenti te between the source code and the object code.
10. Why are compilers and interpreters used?
11. Is there any di erence between a compiler and an interpreter?
12. What is application so are? Give examples.
13. What is BIOS?
14. What do you understand by utility soft are? Is it a must to have it?
15. Di erenti te between syntax errors and logical errors.
16. Can a program wri en 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 do you understand by the term 'programming paradigm'?
19. Discuss any three programming paradigms in detail.
20. How is structured programming be er than monolithic programming?
21. Describe the special characteristics of monolithic programming.
22. Explain how functional abstraction is achieved in structured programming.
23. Which programming paradigm is data-based and why?
24. What are the advantages of modularization