R. S. Salaria

PROGRAMMING FOR PROBLEM SOLVING

WITH LAB MANUAL

KHANNA BOOK PUBLISHING CO. (P) LTD.
PUBLISHER OF ENGINEERING AND COMPUTER BOOKS
4C/4344, Ansari Road, Darya Ganj, New Delhi-110002
Phone: 011-23244447-48 **Mobile:** +91-99109 09320
E-mail: contact@khannabooks.com
Website: www.khannabooks.com

Dear Readers,

To prevent the piracy, this book is secured with HIGH SECURITY HOLOGRAM on the front title cover. In case you don't find the hologram on the front cover title, please write us to at contact@khannabooks.com or whatsapp us at +91-99109 09320 and avail special gift voucher for yourself.

Specimen of Hologram on front Cover title:

Moreover, there is a SPECIAL DISCOUNT COUPON for you with EVERY HOLOGRAM.

How to avail this SPECIAL DISCOUNT:

Step 1: Scratch the hologram
Step 2: Under the scratch area, your "coupon code" is available
Step 3: Logon to www.khannabooks.com
Step 4: Use your "coupon code" in the shopping cart and get your copy at a special discount
Step 5: Enjoy your reading!

ISBN: 978-93-91505-21-9
Book Code: UG003EN

Programming for Problem Solving
by R. S. Salaria
[English Edition]

First Edition: 2021

Published by:

Khanna Book Publishing Co. (P) Ltd.
Visit us at: **www.khannabooks.com**
Write us at: **contact@khannabooks.com**
CIN: U22110DL1998PTC095547

To view complete list of books,
Please scan the QR Code:

KPH

Printed in India.

प्रो. अनिल डी. सहस्रबुद्धे
अध्यक्ष
Prof. Anil D. Sahasrabudhe
Chairman

सत्यमेव जयते

अखिल भारतीय तकनीकी शिक्षा परिषद्
(भारत सरकार का एक सांविधिक निकाय)
(शिक्षा मंत्रालय, भारत सरकार)
नेल्सन मंडेला मार्ग, वसंत कुंज, नई दिल्ली–110070
दूरभाष : 011–26131498
ई–मेल : chairman@aicte-india.org

ALL INDIA COUNCIL FOR TECHNICAL EDUCATION
(A STATUTORY BODY OF THE GOVT. OF INDIA)
(Ministry of Education, Govt. of India)
Nelson Mandela Marg, Vasant Kunj, New Delhi-110070
Phone : 011-26131498
E-mail : chairman@aicte-india.org

FOREWORD

Engineering has played a very significant role in the progress and expansion of mankind and society for centuries. Engineering ideas that originated in the Indian subcontinent have had a thoughtful impact on the world.

All India Council for Technical Education (AICTE) had always been at the forefront of assisting Technical students in every possible manner since its inception in 1987. The goal of AICTE has been to promote quality Technical Education and thereby take the industry to a greater heights and ultimately turn our dear motherland India into a Modern Developed Nation. It will not be inept to mention here that Engineers are the backbone of the modern society - better the engineers, better the industry, and better the industry, better the country.

NEP 2020 envisages education in regional languages to all, thereby ensuring that each and every student becomes capable and competent enough and is in a position to contribute towards the national growth and development.

One of the spheres where AICTE had been relentlessly working from last few years was to provide high-quality moderately priced books of International standard prepared in various regional languages to all it's Engineering students. These books are not only prepared keeping in mind it's easy language, real life examples, rich contents and but also the industry needs in this everyday changing world. These books are as per AICTE Model Curriculum of Engineering & Technology – 2018.

Eminent Professors from all over India with great knowledge and experience have written these books for the benefit of academic fraternity. AICTE is confident that these books with their rich contents will help technical students master the subjects with greater ease and quality.

AICTE appreciates the hard work of the original authors, coordinators and the translators for their endeavour in making these Engineering subjects more lucid.

(**Anil D. Sahasrabudhe**)

Acknowledgement

The author is grateful to AICTE for their meticulous planning and execution to publish the technical book for Engineering and Technology students.

This book is an outcome of various suggestions of AICTE members, experts and authors who shared their opinion and thoughts to further develop the engineering education in our country.

It is also with great honour that I state that this book is aligned to the AICTE Model Curriculum and in line with the guidelines of National Education Policy (NEP) -2020. Towards promoting education in regional languages, this book is being translated in scheduled Indian regional languages.

Acknowledgements are due to the contributors and different workers in this field whose published books, review articles, papers, photographs, footnotes, references and other valuable information enriched us at the time of writing the book.

Finally, I like to express my sincere thanks to the publishing house, M/s. Khanna Book Publishing Company Private Limited, New Delhi, whose entire team was always ready to cooperate on all the aspects of publishing to make it a wonderful experience.

R. S. Salaria

Preface

Problem solving is unquestionably one of the most important skills; other skills such as writing efficient code, effective communication, working with a team, and many others, are also very important. It's impossible to say any one skill is the MOST important.

The subject of **Programming for Problem Solving** aims at developing problem solving skills and the skills to create programs in C language for their implementation.

The textbook in your hand is designed as per the model curriculum of AICTE for the first year students of all branches of under graduate programme in Engineering & Technology (BE/BTech).

Programming books do teach problem solving. Unfortunately there are limitations on what can actually be taught. It is mostly learned by practice.

The point that I wish to make is to get students to see the problem solving process in action. For example, designing a sorting algorithm is a basic example of a "problem" that needs to be "solved". Understanding how to implement different algorithms and select the best strategy for sorting helps you learn how to solve problems, in a very rudimentary way. Careful examination of what is covered in this type of analysis will improve your problem solving process. Unfortunately, most students just learn the algorithms and complete the exercises, and don't dig deeper than that.

If you just read the book or notes taken in the class and implement the solution, you aren't learning to solve the problem. A more effective method is to read the problem, then close the book/notes and try to come up with a solution. After creating a solution on your own, go back and compare your results with what is written in the book/notes. Then you learn how to solve problems.

Problem solving is really a self-directed process. It requires curiosity, flexibility, careful observation and analysis, and a conceptual framework that grows slowly over time. Of those things, the only one that can be taught is the conceptual framework, so that is what books and classes provide. The rest is up to you.

The book in your hand will provide you everything you need to become a good problem solver and good programmer as well.

Enjoy reading this book, and feel free to send your free and frank comments, suggestions, and positive criticism. Your valuable inputs will help me to improve and fine tune the book to meet your expectations.

Last but not the least, while writing the book, due care has been taken to avoid errors (misprints and/or mistakes), yet it is difficult to claim perfection. I will be grateful to the readers if any errors are pointed out.

R. S. Salaria

OUTCOME BASED EDUCATION

Outcome-Based Education (OBE) is a student-centric teaching and learning methodology in which the course delivery, assessment are planned to achieve stated objectives and outcomes. It focuses on measuring student performance, i.e., outcomes at different levels.

Some important aspects of the Outcome Based Education:

1. Course is defined as a theory, practical or theory cum practical subject studied in a semester.
2. Course Outcomes (COs) are statements that describe significant and essential learning that learners have achieved, and can reliably demonstrate at the end of a course. Generally three or more course outcomes may be specified for each course based on its weightage.
3. Programme is defined as the specialization or discipline of a Degree. It is the interconnected arrangement of courses, co-curricular and extracurricular activities to accomplish predetermined objectives leading to the awarding of a degree. For example, BE/BTech – Computer Science & Engineering
4. Programme Outcomes (POs) are narrower statements that describe what students are expected to be able to do by the time of graduation. POs are expected to be aligned closely with Graduate Attributes.
5. Programme Educational Objectives (PEOs) of a program are the statements that describe the expected achievements of graduates in their career, and also in particular, what the graduates are expected to perform and achieve during the first few years after graduation.
6. Programme Specific Outcomes (PSOs) are what the students should be able to do at the time of graduation with reference to a specific discipline. Usually there are two to four PSOs for a programme.

Knowledge levels for assessment of outcomes based education on Blooms Taxonomy:

Level	Parameter	Description
K1	Remember	It is the ability to remember the previously learned material/information
K2	Understand	It is the ability to grasp the meaning of material.
K3	Apply	It is the ability to use learned material in new and concrete situations
K4	Analyze	It is the ability to break down material/concept into its component parts/ subsections so that its organizational structure may be understood.
K5	Evaluate	It is the ability to judge the value of material/concept/statement/creative material /research report for a given purpose.
K6	Create	It is the ability to put parts/subsections together to form a new whole material/ idea/concept/information.

PROGRAMME OUTCOME (POs)

For the implementation of an outcome based education the first requirement is to develop an outcome based curriculum and incorporate an outcome based assessment in the education system. By going through outcome based assessments evaluators will be able to evaluate whether the students have achieved the outlined standard, specific and measurable outcomes. With the proper incorporation of outcome based education there will be a definite commitment to achieve a minimum standard for all learners without giving up at any level. At the end of the programme running with the aid of outcome based education, a students will be able to arrive at the following outcomes:

PO-1: **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO-2: **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO-3: **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO-4: **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO-5: **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO-6: **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO-7: **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO-8: **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO-9: **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO-10: **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO-11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO-12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

COURSE OUTCOMES

The course will enable the students

CO-1: To formulate simple algorithms for arithmetic and logical problems

CO-2: To translate the algorithms to programs (in C language)

CO-3: To test and execute the programs and correct syntax and logical errors

CO-4: To implement conditional branching, iteration and recursion

CO-5: To decompose a problem into functions and synthesize a complete program using divide and conquer approach

CO-6: To use arrays, pointers and structures to formulate algorithms and programs

CO-7: To apply programming to solve matrix addition and multiplication problems and searching and sorting problems

CO-8: To apply programming to solve simple numerical method problems, namely root finding of function, differentiation of function and simple integration

Course Outcomes	**Expected Mapping with Programme Outcomes** *(1- Weak Correlation; 2- Medium correlation; 3- Strong Correlation)*											
	PO-1	**PO-2**	**PO-3**	**PO-4**	**PO-5**	**PO-6**	**PO-7**	**PO-8**	**PO-9**	**PO-10**	**PO-11**	**PO-12**
CO-1	3	3	3	-	-	-	-	-	-	-	-	-
CO-2	3	-	1	-	-	-	-	-	-	-	-	-
CO-3	3	-	1	-	3	-	-	-	-	-	-	-
CO-4	3	-	-	-	-	-	-	-	-	-	-	-
CO-5	3	3	1	2	-	-	-	-	-	-	-	-
CO-6	3	-	1	-	-	-	-	-	-	-	-	-
CO-7	3	2	1	1	-	-	-	-	-	-	-	-
CO-8	3	2	1	1	-	-	-	-	-	-	-	-

ABBREVIATIONS

Abbreviations	Full form
ALU	Arithmetic and Logic Unit
ANSI	American National Standards Institute
ASCII	American Standard Code For Information Interchange
BIOS	Basic Input Output System
BIOS	Basic Input Output System
BMI	Body Mass Index
BODMAS	Bracket, Of, Division, Multiplication, Addition, And Subtraction
BOSS	Bharat Operating System Solutions
CPU	Central Processing Unit
EOF	End-Of-File
GCD	Greatest Common Divisor
GUI	Graphical User Interfaces
HCF	Highest Common Factor
HLL	High-Level Language
IDE	Integrated Developments Environment
IPO	Input-Process-Output
OS	Operating System
PDL	Program Design Language
POST	Power On Self Test
RAM	Random Access Memory
ROM	Read Only Memory
VDU	Visual Display Unit
VLSI	Very Large Scale Integration

LIST OF FIGURES

Unit-1: Introduction to Programming

Unit-3: Conditional Branching and Loops

Unit-4: Arrays

Unit-5: Basic Algorithms

Unit-6: Functions

Unit-7: Recursion

Unit-8: Structures

Unit-9: Pointers

Unit-10: File Handling

LIST OF TABLES

Unit-7: Recursion

Unit-9: Pointers

Unit-10: File Handling

GUIDELINES FOR TEACHERS

To implement Outcome Based Education (OBE) knowledge level and skill set of the students should be enhanced. Teachers should take a major responsibility for the proper implementation of OBE.

Some of the responsibilities (not limited to) for the teachers in OBE system may be as follows:

- Within reasonable constraint, they should manipulate time to the best advantage of all students.
- They should assess the students only upon certain defined criterion without considering any other potential ineligibility to discriminate them.
- They should try to grow the learning abilities of the students to a certain level before they leave the institute.
- They should try to ensure that all the students are equipped with the quality knowledge as well as competence after they finish their education.
- They should always encourage the students to develop their ultimate performance capabilities.
- They should facilitate and encourage group work and team work to consolidate newer approach.
- They should follow Blooms taxonomy in every part of the assessment.

Bloom's Taxonomy

Level	Teacher should Check	Student should be able to	Possible Mode of Assessment
Creating	Students ability to create	Design or Create	Mini project
Evaluating	Students ability to Justify	Argue or Defend	Assignment
Analysing	Students ability to distinguish	Differentiate or Distinguish	Project/Lab Methodology
Applying	Students ability to use information	Operate or Demonstrate	Technical Presentation/ Demonstration
Understanding	Students ability to explain the ideas	Explain or Classify	Presentation/Seminar
Remembering	Students ability to recall (or remember)	Define or Recall	Quiz

GUIDELINES FOR STUDENTS

Students should take equal responsibility for implementing the outcome based education (OBE).

Some of the responsibilities (not limited to) for the students in OBE system are as follows:

- Students should be well aware of each unit outcomes (UOs) before the start of a unit in each and every course.
- Students should be well aware of each course outcomes (COs) before the start of the course.
- Students should be well aware of each programme outcomes (POs) before the start of the programme.
- Students should think critically and reasonably with proper reflection and action.
- Learning of the students should be connected and integrated with practical and real life consequences.
- Students should be well aware of their competency at every level of OBE

CONTENTS

1 Introduction to Programming

UNIT SPECIFICS

This unit discusses the topics related to introduction to computer and its components, software and its types, important software tools related to the course, development of algorithms, translating algorithms to programs, and the various steps related to creating, editing, compiling and executing C programs, and introduces basic fundamental elements of C languages.

RATIONALE

We all are living in a *digital era*. Therefore, there is a need to empower everyone with the knowledge and ability to use computers and technology efficiently. Engineers theorize, design, develop, and apply hardware and software solutions to real-life problems that we confront day in day out.

This fundamental unit helps students to understand components of a computer system and associated devices, and how they communicate with each other to accomplish a task; development of algorithms for solving logical and numerical problems; and various stages in converting an algorithm to executable program using C language.

PRE-REQUISITES

- Working of computers
- Basic mathematics
- Logical reasoning
- Communication skills

UNIT OUTCOMES

Upon completion of the unit, students will be able to

U1-O1: formulate algorithms for simple arithmetic and logical problems

U1-O2: translate algorithms to C programs

U1-O3: create, compile, and execute programs on a given platform/develop environment

U1-O4: demonstrate testing & debugging of a program

Unit 1 Outcomes	Expected Mapping With Course Outcomes (1 – Weak Correlation; 2 – Medium Correlation; 3 – Strong Correlation)							
	CO-1	CO-2	CO-3	CO-4	CO-5	CO-6	CO-7	CO-8
U1-O1	3	-	-	-	-	-	-	-
U1-O2	-	3	-	-	-	-	-	-
U1-O3	-	-	3	-	-	-	-	-
U1-O4	-	-	3	-	-	-	-	-

1.1 INTRODUCTION TO COMPUTERS

Computer is an electronic machine that performs tasks or computations according to a set of instructions called *programs*.

The first fully electronic computers, introduced in the 1940s, were huge machines that required teams of people to operate. Compared to those early machines, today's computers are amazing - they are thousand times faster and can fit on your desk, in your lap, or even in your pocket.

A computer, in general, is capable of

- Receiving the data and instructions in various forms,
- Storing the data and instructions in its memory,
- Retrieving the data already stored on secondary storage,
- Performing arithmetic and logical operations, according to the specified instructions, with very high speed and greater accuracy,
- Producing the results in many forms, and
- Communicating with other computers.

1.1.1 Components of a Computer System

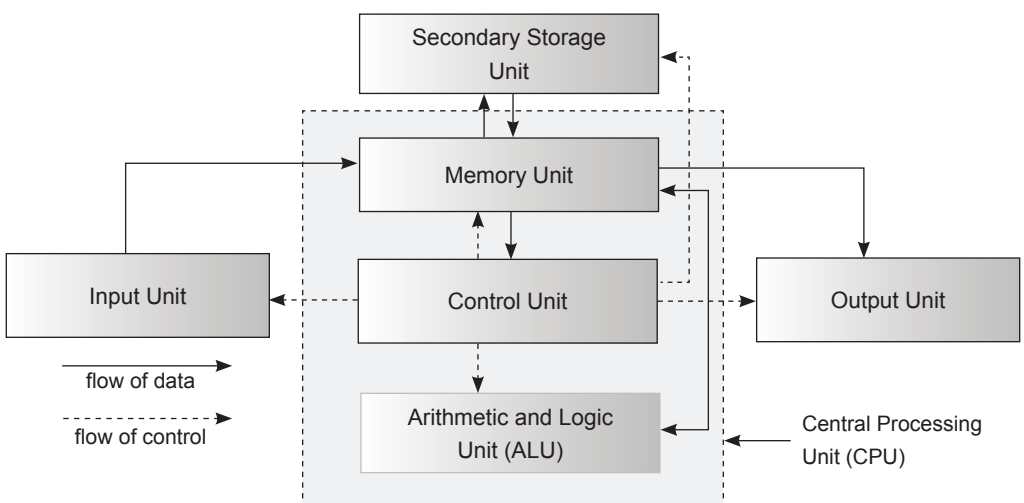


Fig. 1.1: Functional components of a computer system

A computer is a combination of hardware and software. Hardware represents physical components of a computer like motherboard, memory devices, monitor, keyboard, etc., while software represents a set of programs or instructions. Both hardware and software together make the computer system function.

Every task given to a computer follows an Input-Process-Output Cycle (IPO cycle). It needs certain input, processes that input and produces the desired output. The input unit takes the input, the central processing unit does the processing of data and the output unit produces the output. The memory unit holds the data and instructions during the processing.

Let us have a look at the functional components of a computer one-by-one.

Input Unit

The *input unit* consists of various input devices that can be connected to a computer, such as *keyboard*, *mouse*, *light pen*, *microphone*, etc. The standard input device is *keyboard*, which will come with every new computer. At any time, more than one input device can be connected to a computer.

The purpose of the input unit is to receive information from the outside world, convert it into binary form, and then transfer it to the main memory. It may consist of entering data to be processed by some program, recording of speech, capturing of a scene or photograph, or selecting some objects on the display.

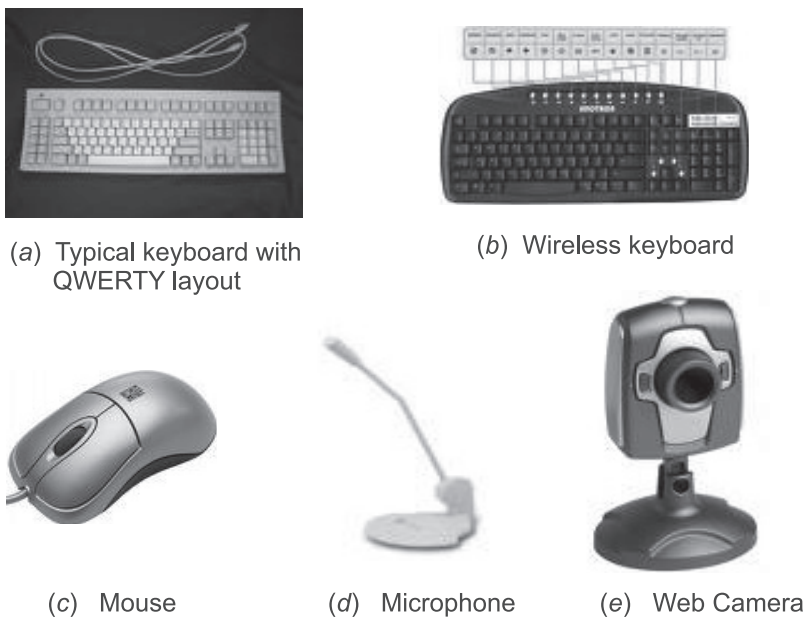


Fig. 1.2: Common input devices

Output Unit

The *output unit* consists of various output devices that can be connected to a computer, such as *visual display unit* (VDU) or simply known as *monitor, printers, plotters, audio systems* etc. The standard output device is a *monitor*, which will come with every new computer. At any time, like input devices, more than one output device can be connected to a computer.

The purpose of the output unit is to receive the information, in binary form, from the main memory unit, convert it into human understandable form, and then output it. It may consist of displaying, printing, or plotting the results of some computer program or even playing some music or movie.

(*a*) Monitor (*b*) Printer (*c*) Plotter (*d*) Speakers

Fig. 1.3: Common output devices

Memory Unit

The *memory unit* acts as the warehouse of a computer, where executing programs, input data, intermediate and final results of computations are stored. Information entered into a computer is first stored in the memory unit. Similarly, the data or results to be output are taken from the memory unit. The memory unit represents a *volatile* memory, which means that the stored information will be lost when power is switched off.

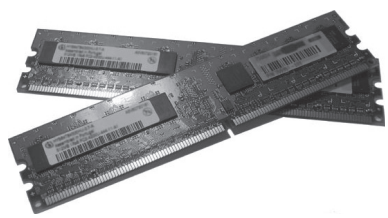


Fig. 1.4: Memory chips

Arithmetic and Logic Unit

The arithmetic and logic unit (ALU), as its name implies, performs arithmetic and logical operations. Arithmetic operations include addition, subtraction, multiplication, and division. Logical operations are used to compare two data items, and include operations such as less than (<), less than or equal to (≤), greater than (>), greater than or equal to (≥), equal to (=), and not equal to (≠).

Control Unit

The control unit co-ordinates and controls all the other parts of computer system. Under the direction of a program, the control unit performs four basic operations:

- *Fetch* - getting the next program instruction from the computer's memory.
- *Decode* - figuring out what the program is telling the computer to do?
- *Execute* - performing the requested action, such as adding two numbers or deciding whether one of them is larger.
- *Write-Back* - writing the results back to memory.

This four-step process is called *a machine cycle,* or *a processing cycle*, and consists of two phases: *the instruction cycle* (*fetch* and *decode*) and *the execution cycle* (*execute* and *write-back*).

Secondary Storage Unit

The *secondary storage unit* provides long-term storage for important data for future use. The data is transferred from the memory unit to the secondary storage unit. In order to process the data residing in secondary storage, it is transferred to the memory. Please remember that data stored in secondary storage can not be processed directly in secondary storage itself. The data can only be processed if it is in the memory.

(a) Hard disk (b) CD/DVD (c) Pen drive

Fig. 1.5: Common **s**torage devices

- *Control unit*, *Arithmetic and logic unit*, and *Main memory unit*, collectively, are known as *Central Processing Unit* (CPU).
- All the external devices, such as *input device* and *output devices*, connected to the computer are commonly known as *peripherals*.

Advancement in the field of materials and microelectronics, has made it became possible to embed *control unit*, *arithmetic and logical unit*, and limited high speed memory in the form of few *registers*, inside the single *Very Large Scale Integration* (VLSI) chip, known as a *microprocessor*.

Formally, a *microprocessor*, commonly referred as *processor,* is a general-purpose programmable device, which can receive information in binary form, process it and transmit the results. It can be programmed to solve a wide variety of problems using instructions that it understands. Set of all the instructions that a processor can understand is known as the *instruction set*.

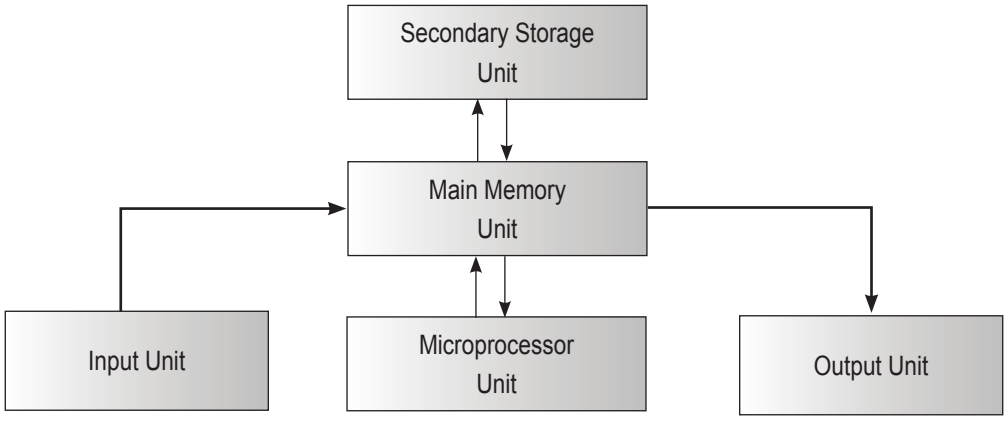


Fig. 1.6: Functional diagram of a Personal Computer

The role of other units remains the same as in case of a general digital computer. Majority of the present PCs including laptops are built around *Intel* or *AMD* based microprocessors.

1.1.2 Hardware

Hardware refers to the parts of a computer that you can see and touch, including the case and everything inside it. Various components of the computer system that constitutes its hardware include system unit (which houses major hardware components such as *motherboard*, *power supply, hard disk*, etc.), input/output devices as well as storage devices.

1.1.3 Software

Software, in its most general sense, is a set of instructions or programs that tell the hardware what to do.

Software can be difficult to describe because it is *virtual* and not physical like computer hardware. Instead, software consists of lines of code (instructions) written by computer programmers that have been compiled into a computer program. Software programs are stored as binary data that is copied to a computer's hard drive, when it is installed. Since software is virtual and does not take up any physical space, it is much easier (and often cheaper) to upgrade than computer hardware.

The most important example of software is an operating system (OS) that manages the resources of a computer system, and provides an interface using which the user can interact with computer to perform various tasks.

It is important to remember that hardware and software are integral parts of a computer system — *hardware is of no use without software and software cannot be used without hardware*. It is the software that drives hardware, i.e., it is the software that gives hardware its capability.

Software can be broadly categorized as

- **System software:** It is a type of software that is designed to run a computer's hardware and application programs. If we think of a computer system as a layered model, the system software is the interface between the hardware and user applications. The *operating system* and *language translators* (*assembler*, *compiler* and *interpreter*), *linker* and *loader* are examples of system software.
- **Application software:** It is a type of software that is designed for a specific purpose and is used by end users. Word processor, database software, spreadsheet software, entertainment software, reservation software, inventory management software, etc., are examples of application software.
- **Utility software:** It is a type of software that is designed to help analyze, configure, optimize or maintain a computer. It is used to support the computer infrastructure. Antivirus software, compression tools, disk management tools, etc., are examples of utility software.

1.1.3.1 Operating System

The most important program on any computer is the *Operating System* or simply OS. It is a large program made up of many smaller programs. It acts as a resource manager of the computer system and provides an interface between the users and the hardware.

The resources of a computer system include processor, memories and I/O devices, users, etc. As a manager it coordinates the activities of various resources so as to provide the users with the required information.

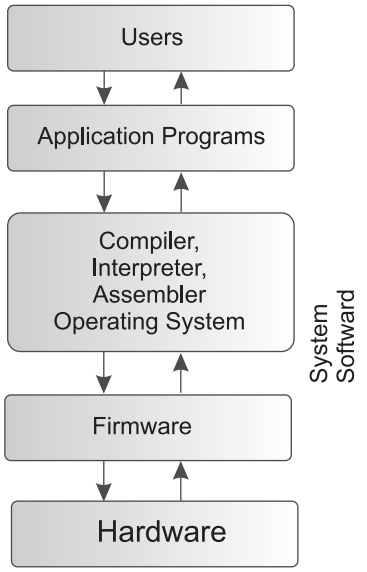


Fig. 1.7: Layered architecture of a Computer System

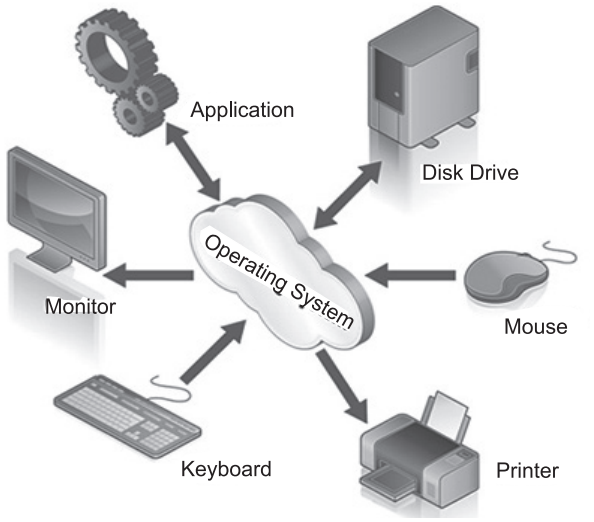


Fig. 1.8: Operating System managing various resources of a Computer System

Commonly used Operating Systems for PCs/Laptops include

- Unix
- Linux
- Windows
- Solaris
- BOSS (Bharat Operating System Solutions)

Commonly used Operating Systems for Mobile/Hand Held Devices include

- Android
- iOS
- Symbian

1.1.3.2 Language Translators

A software component that translates program written in one language to another language is known as *language translator* (also known as *language processor*).

It is pertinent to mention here that while machine language requires no translation, but the programs written in high-level languages, such as C, requires a translator to translate instructions written in C language to machine language before they can be executed.

A program written using C language is known as *source code.* The translated version of the source code is a program in machine language, known as *object code.*

The language translator for assembly language is known as an *assembler,* and the language translators for high-level languages are known as *compilers* and *interpreters*.

Assembler

An assembler is a program that translates a program in assembly language (*source code*) to machine language (*object code*).

As there is one-to-one correspondence between instructions in machine language and assembly language, the assembler translates each instruction in assembly language to corresponding instruction in machine language.

Compiler

A compiler is a program that translates (*compiles*) a program written in a high-level language (*source code*) to machine language (*object code*).

This process of translation is known as *compilation.* During compilation, compiler reads the source code, line by line, and checks for syntax errors. Note that any violation of the rules of the programming language (also known as the grammar of the language), is known as *syntax error.*

If there are syntax errors, then it displays error messages identifying the error locations and prints a program listing that highlights the locations and the likely causes of the errors, and no translation takes place. However, if there are no syntax errors in the source code, the compiler re-reads the source code and translate it into an object code which then it writes onto another file, usually with extension *obj* (for object).

Once translation is done, the compiler has no further role in the subsequent steps. However, if the source code is modified, we need to recompile it to incorporate the modification in the object code.

Interpreter

An interpreter is another type of translator used for translating high-level languages. However, interpreter doesn't produce object code. Instead, it translates one line of source code at a time and executes the translated instruction. It continues till all the instructions are translated and executed. Note that if any instruction is to be executed repeatedly, it will be translated each time.

Once the source code is translated into the object code by the compiler, which is a onetime task, the object code, after further processing (linking), is executed directly. This results in faster execution of the program. However, the interpreter first understands the code and then executes the instruction, which lead to slow execution of the program. In addition, every time the program is executed using interpreter, the translation will be repeated.

Although interpreters lead to slower translation of program, they are helpful tools for beginners for learning language and also for debugging the programs. As the program executes line by line the programmer can see exactly what each line does.

Here it is important to mention that in order to execute a program using interpreter, first we need to run the interpreter, only then we can instruct the interpreter to execute the user program. Therefore, every time you run a program you need to run an interpreter as well.

1.1.3.3 Programming Environment

Programming environment is an environment in which programs are created and tested, and it has less influence on the design of the language than the operating environment on which the programs are expected to be run.

A programming environment consists of support tools and a command language (set of commands) to invoke them. Each support tool is another program that may be used by the programmer as an aid during one or more stages of creation of a program.

Typical tools in a programming environment include *editors*, *language translators*, *linkers*, *loaders*, etc. Language translators are explained in the earlier section, the remaining tools are described here.

- **Editor –** *Editor* is a program that helps in creating and editing files. Further, there are editors that allow working with text files only, and are called *text editors.* Examples of popular text editors are *notepad* and *edit* of *Windows* operating system, *vi* of *Unix* and *Linux* operating systems. Even popular word processing programs such as MS Word, allows creation of text files in addition to word documents.
- **Linker –** Note that the object code produced by the compiler is not in a form that can be executed by the computer. Some more information is needed to be pumped into the object code to make it in a form that can be executed by the computer.

  When you write a program, you may be referring to various library functions for performing input/output operations and for mathematical computation (*sqrt*, *sin*, *cos*, *abs*, *exp* to name a few frequently used mathematical library functions). The instructions defining the task to be performed by these library function are in the machine language (binary) form and are contained in system libraries (files with extension LIB) supplied with the compiler. Plus, you may be referring to some user-defined functions which are written in a separate program file, and whose object code is lying separately than the object code under consideration. Therefore, there is a need to combine the object code of the library as well as user-defined functions with the object code under consideration, and this task is accomplished by the program called *linker*.

  Linker is a program that combines the object code of the program with the object codes of the library and user-defined functions, in the desired manner, and writes the same in a separate program file with extension, usually EXE under Windows. This file produced by the linker is known as *executable file* or simply as *run file,* and is the file that is used to execute the program.
- **Loader –** *Loader* is a program that is responsible for transferring the executable file from the secondary storage to memory. On execution, the loader, with assistance of the operating system, first finds free memory of the requisite amount and then transfers the executable file into that part of memory, takes appropriates steps such as address translation, and then initiates its execution. From that point onwards, the role of the loader is over, and the control is taken over

by the program. Instructions of the program are executed to perform the desired task, but under the overall supervision of operating system. As soon as the program finishes its execution, it is removed from the memory.

Popular Integrated Developments Environments (IDEs), such as Turbo C/C++ and Borland C++, Visual Studio, Dev C++, CodeBlocks, Netbeans, Eclipse, *etc.*, integrate various tools (editor, compiler, linker, debugger, etc.) in a unified manner, so that a programmer can work with these tools with much ease and enhanced productivity.

1.1.4 Concept of Booting

In simple terms, *booting* is a process in which your computer gets initialized. This process includes initializing all your hardware components in your computer and getting them to work together and to load your default operating system which will make your computer operational.

The technical details of booting process can be summarized as follows:

1. When the computer is switched on, a copy of boot program is brought from ROM into the main memory.
2. Then CPU runs a jump instruction that transfers to BIOS (Basic Input output System) and it starts executing.
3. BIOS conduct a series of self diagnostic tests called POST (Power On Self Test). These tests include memory test, configuring and starting video circuitry, configuring system's hardware and checking other devices that help computer to function properly.
4. Thereafter, the BIOS locate a bootable drive to load the boot sector. The Boot Strap Loader program loads and executes the operating system.

The booting process is of two types:

- **Cold booting:** When the system starts from initial state, *i.e.*, it is switched on. It is also called *hard booting*. The system goes through a number of stages as explained above
- **Warm booting:** When the Reset button or *Ctrl+Alt+Del* key combination is used to restart the system. It is also called *soft booting*. Here the system does not start from the initial state and therefore the diagnostic tests are not carried out in this case.

1.2 IDEA OF ALGORITHMS

An *algorithm* is a finite sequence of instructions defining the solution of a particular problem.

Characteristics of a good algorithm:

There are five important characteristics of an algorithm that should be considered while designing any algorithm for problem.

- **Input:** An algorithm must have zero or more but finite number of inputs, which are externally supplied.

  Example of zero input algorithms can be to find the sum of first 100 natural numbers. Here, the user doesn't need to supply any external input since it is already specified to find the sum of first 100 natural numbers.

However, if the above problem is re-stated as finding the sum of first *n* natural numbers, the user is required to provide single input denoting the value for *n*.

- **Output:** An algorithm must have at-least one desirable outcome, i.e., output.
- **Definiteness (No ambiguity):** Each step must be clear and unambiguous, *i.e.*, having one and only one meaning.
- **Finiteness:** If we trace the steps of an algorithm, then for all cases, the algorithm must terminate after a finite number of steps.
- **Effectiveness:** Each step must be sufficiently basic that it can in principle be carried out by a person using only paper and pencil. In addition, not only should each step be definite, it must also be feasible.

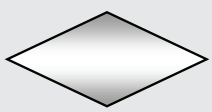
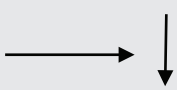
An algorithm can be represented using a flowchart or a pseudocode.

1.2.1 Flowchart

A flowchart is a pictorial representation of an algorithm. It uses different shapes to denote different types of instructions. The actual instructions are written within the shapes using clear and concise statements. These shapes are connected by directed lines to indicate the sequence in which instructions are to be executed.

Table 1.1 shows various symbols used in flowcharts along with their name and brief description.

Table 1.1: Various flowchart symbols and their brief description

Symbol	Name	Purpose
	Oval	*Terminal* - to mark the beginning and end of the program logic flow.
	Parallelogram	*Input/Output* - to denote input to the program or output from the program.
	Rectangle	*Processing* - to denote arithmetic operations and movement of data.
	Diamond	*Decision* - to denote a point where decision has to be made to branch to one of the alternatives.
	Small circle	*Connector* - To provide a logical link between segments of a flowchart.
	Directed lines	*Flow Lines* - To indicate the sequence in which instructions are to be executed.

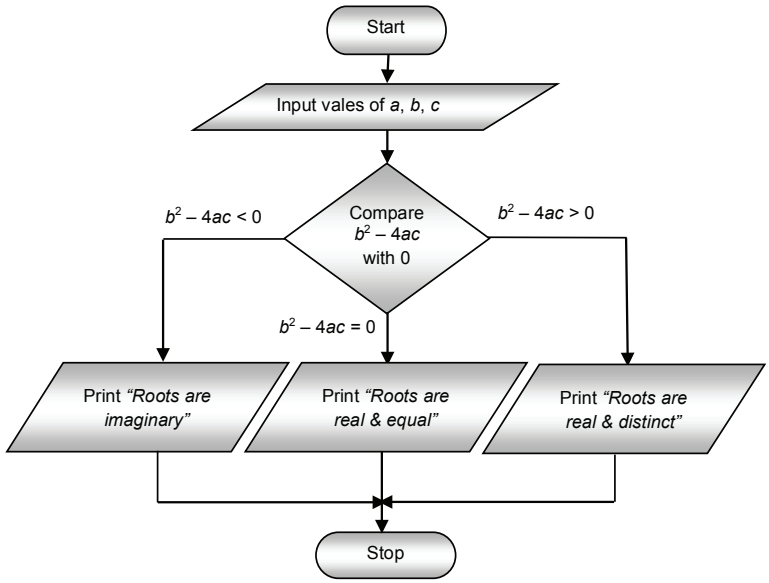


Fig. 1.9: Flowchart to find the nature of roots of a quadratic equation

1.2.2 Pseudocode

The word "*pseudo*" means imitation or false and the word "*code*" refers to the instruction written in a programming language. Pseudocode, therefore, is an imitation of actual computer instruction. Pseudo instructions are phrases written in English like statements. Instead of using symbols to describe the logic of the program, as in flowcharts, pseudocode uses a structure that resembles computer instructions. Because, it emphasizes the design of the program, pseudocode is also called *Program Design Language* (PDL).

Pseudocode is made up of the following basic logic structures that have proved to be sufficient for writing any computer program:

1. Sequence
2. Selection (*If... Endif*, *If... Else... Endif, If... Else If... Endif*)
3. Iteration (*While... Endwhile*, *Do... While*)

Sequence logic is used for performing instructions one after another in sequence. Thus, for sequence logic, pseudocode instructions are written in the sequence in which they are to be executed.

The flow of logic is from top to bottom.

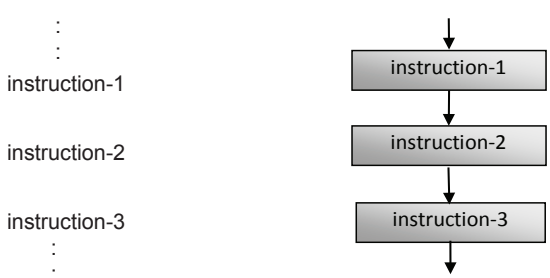


Fig. 1.10: Pseudocode and flowchart for sequence structure

Selection logic, also known as decision logic, is used for making decision. It is used for selecting a proper path out of the alternative paths in the program logic.

Selection logic is depicted as either *If... Endif* or *If... Else... Endif* or *If... Else If... Endif* structure.

The *If... Endif* construct says that if the expression is true, then execute statement else (if the expression is false) skip over the statement.

```
            :
            :
If ( expression ) then
    statement
Endif
            :
            :
```

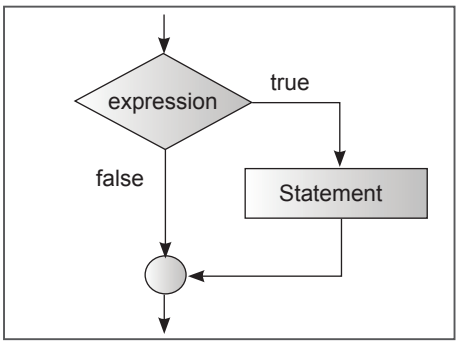


Fig. 1.11: Pseudocode and flowchart for *If... Endif* selection structure

The *If... Else... Endif* construct says that if the expression is true then execute statement-1, else (if the expression is false) execute statement-2.

```
       :
       :
If ( expression ) then
   statement-1
Else
   statement-2
Endif
       :
       :
```

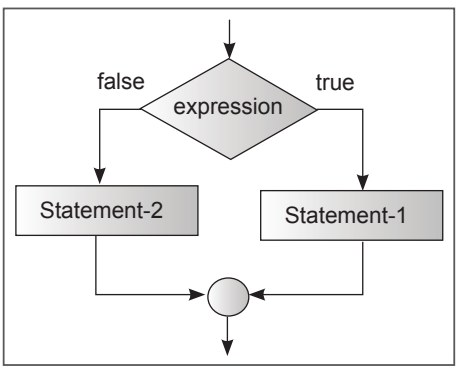


Fig. 1.12: Pseudocode and flowchart for *If... Else... Endif* selection structure

Depending on the outcome of the expression being tested, if there are multiple alternatives (execution paths), the *Else If* ladder is a very handy construct.

The expressions are evaluated in order, and if any expression is true then the statement block associated with it is executed, and this terminates the whole chain. The last *else* part handles *none of the above* or default case where none of the specified expressions are satisfied.

```
If ( expression-1 ) then
   statement-1
Else If ( expression-2 ) then
   statement-2
Else If ( expression-3 ) then
   statement-3
       ⋮
Else If ( expression-n ) then
   statement-n
Else
   statement-s
Endif
```

Fig. 1.13: Syntax of *else if* ladder

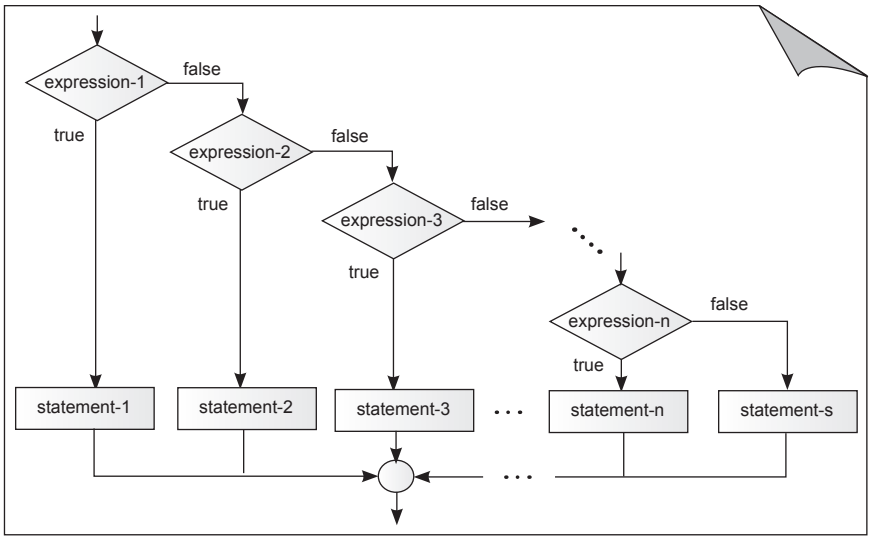


Fig. 1.14: Logic flow of *else if* ladder

Iterative logic is used to produce loops when one or more instructions are to be executed several times depending on some expressions. It uses two structures called *While... Endwhile* and *Do... While*.

```
        :
        :
While ( expression )
   statement
Endwhile
        :
        :
```

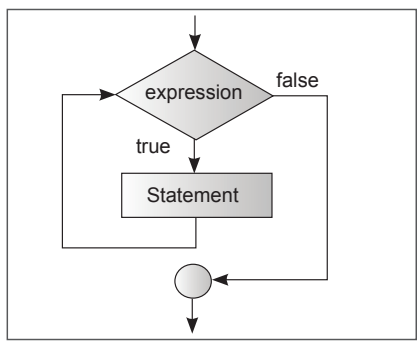


Fig. 1.15: Pseudocode and flowchart for *While... Endwhile* iterative structure

```
        :
        :
Do
   statement
While ( expression )
        :
        :
```

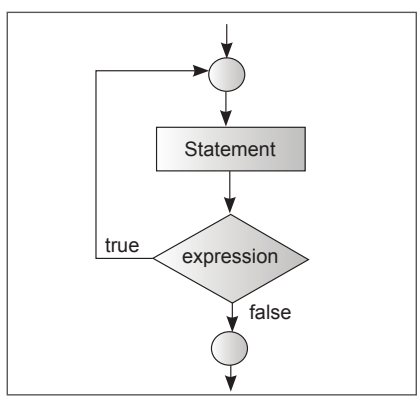


Fig. 1.16: Pseudocode and flowchart for *Do... While* iterative structure

The only difference between them is that *Do... While* loop is always executed at least once as the expression is tested at the end of the loop, whereas the *While... Endwhile* loop may not execute even once since the expression is tested in the beginning and the expression may fail for the first time itself.

Pseudocode Description

Comments

Each instruction may be followed by a comment. The comments begin with a double slash, and the explain the purpose of the instruction, such as

Read: *n* // Enter the value of variable *n*

Appropriate use of comments enhances the readability of the pseudocode, which in turn helps in maintaining the pseudocode.

Variable Names

For variable names, we will use italicized lowercase letters such as *max*, *loc*, etc., whereas for defined constants, if any, we will use uppercase letters.

Assignment Statement

The assignment statement will use the notation as

Set $max = a$

to assign the value of *a* to *max*. The right hand side of the assignment statement can have a *value*, *a variable* or *an expression*.

However, if several assignment statements appear in the same line, such as

Set $k = 1$, $loc = 1$, $max = a_i$

then they are executed from left to right.

Input/Output

Data may be input and assigned to variables by means of a *read* statement with the following format

Read: *Variable list*

where the *Variable list* consists of one or more variables separated by comma.

Similarly, the data held by the variables and the messages, if any, enclosed in double quotes can be output by means of a *print* statement with the following format

Print: *message* and/or *Variable list*

where the *message* and the variables in the *Variable list* are separated by comma.

Execution of Instructions

The instructions are usually executed one after the other as they appear in the pseudocode. However, there may be instances when some instructions are skipped or some instructions may be repeated as a result of certain expressions.

Completion of the Algorithm

A pseudocode is completed with execution of the last instruction. However, it can be terminated at any intermediate state using the *exit* instruction.

Pseudocode to display the nature of roots of a quadratic equation of the type

$ax^2 + bx + c = 0$ provided $a \neq 0$.

Pseudocode 1.1

```
Begin
    Read: a, b, c
    Set disc = b² – 4ac
    If ( disc = 0 ) then
          Print: "Roots are real and equal"
    Else If  ( disc > 0 ) then
          Print: "Roots are real and distinct"
    Else
          Print: "Roots are imaginary"
    Endif
End.
```

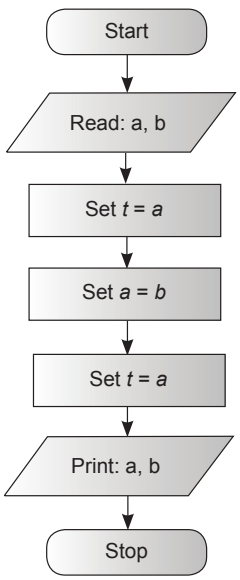
ILLUSTRATIVE EXAMPLES

Example 1.1: *Draw a flowchart and write a pseudocode to swap (interchange) two variables say a and b.*

Solution:

Think of the scenario – we have water in one glass and juice in another glass. We want to have water in a glass in which we have juice, and juice in a glass in which we have water. *How can this task be accomplished*?

In a similar way, we have to use a third variable say *t*, to facilitate the swapping of values of two variables as demonstrated in the following flowchart and pseudocode.



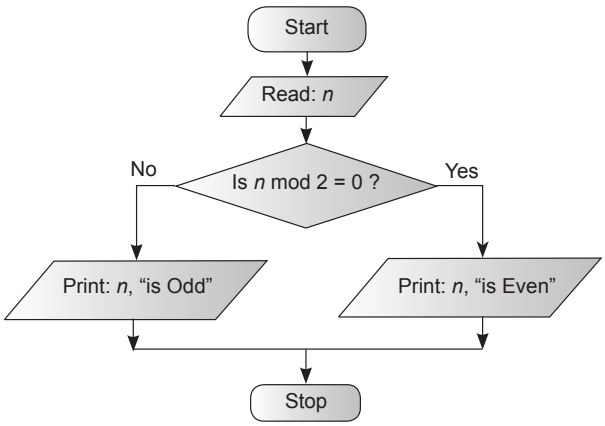
Pseudocode 1.2

```
Begin
    Read: a, b
    Set t = a
    Set a = b
    Set b = t
    Print: a, b
End.
```

Fig. 1.17: Flowchart and pseudocode to swap two variables

Example 1.2: *Draw a flowchart and write a pseudocode to test whether a given natural number 'n' is even or odd.*

Solution: You all may know that any natural number is even if it is exactly divisible by 2, *i.e.*, division by 2 gives 0 as remainder. The operation of obtaining remainder is called modulo (mod in short) operation. The following flowchart and psuedocode demonstrates the logic.



Pseudocode 1.3

```
Begin
   Read: n
   If ( n mod 2 = 0 ) then
        Print: n, "is Even"
   Else
        Print: n, "is Odd"
   Endif
End.
```

Fig. 1.18: Flowchart and pseudocode to test whether given number is Even or Odd

Example 1.3: *Draw a flowchart and write a pseudocode to find the largest of three numbers, say a, b, c.*

Solution: We first compare *a* with *b*. If *a* is greater than *b* then we compare *a* with *c*. If *a* is greater than *c*, then *a* is taken as the largest number otherwise we take c as the largest number.

However, if *a* is not greater than *b*, we compare *b* with *c*. If *b* is greater than *c* then *b* is taken as the largest number otherwise we take *c* as the largest number.

The following flowchart and pseudocode demonstrates the logic.

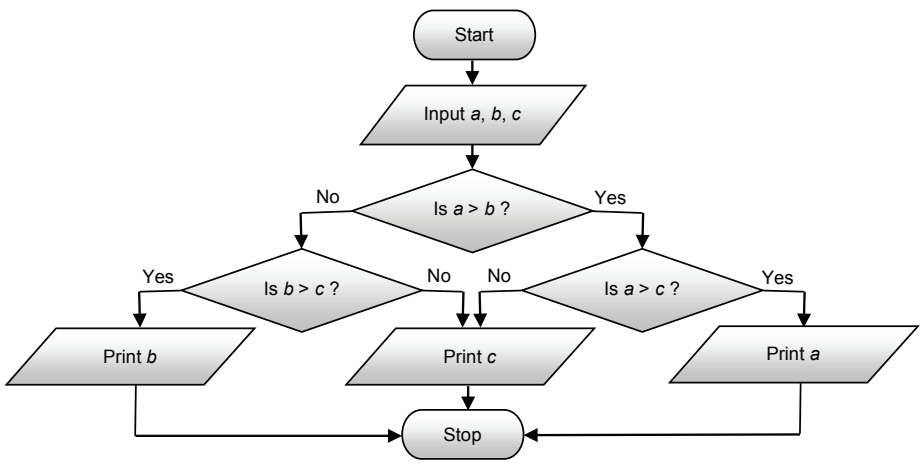


Fig. 1.19: Flowchart and pseudocode to find largest of three numbers

Pseudocode 1.4

```
Begin
  Read: a, b, c
  If ( a  > b )
      If ( a  > c ) then
         Print: a
      Else
         Print: c
      Endif
  Else
      If ( b  > c ) then
         Print: b
      Else
         Print: c
      Endif
  Endif
End.
```

Example 1.4: *Based on the percentage of marks in a subject, letter grade is assigned to a student as per the following examination policy:*

Percentage of Marks	Grade
percentage ≥ 90	A+
90 > percentage ≥ 80	A
80 > percentage ≥ 70	B
70 > percentage ≥ 60	C
60 > percentage ≥ 50	D
percentage < 50	F

Write a pseudocode to assign a letter grade to a student whose percentage of marks in a subject is given.

Pseudocode 1.5

```
Begin
   Read: percentage
   If ( percentage >= 90 )
      Print: "Grade = A+"
   Else If ( percentage >= 80 )
      Print: "Grade = A"
   Else If ( percentage >= 70 )
      Print: "Grade = B"
   Else If ( percentage >= 60 )
      Print: "Grade = C"
   Else If ( percentage >= 50 )
      Print: "Grade = D"
   Else
      Print: "Grade = F"
   Endif
End.
```

Example 1.5: *Commission on sales by a salesman is calculated as per following policy:*

Amount of Sale (in ₹)	Commission Rate
0 – 5000	Nil
5001 – 10000	5 % excess of 5000
10001 – 15000	7.5 % excess of 10000
> 15000	10 % excess of 15000

Draw a flowchart and write a pseudocode that accepts sales made by a salesman and displays the commission due.

Solution:

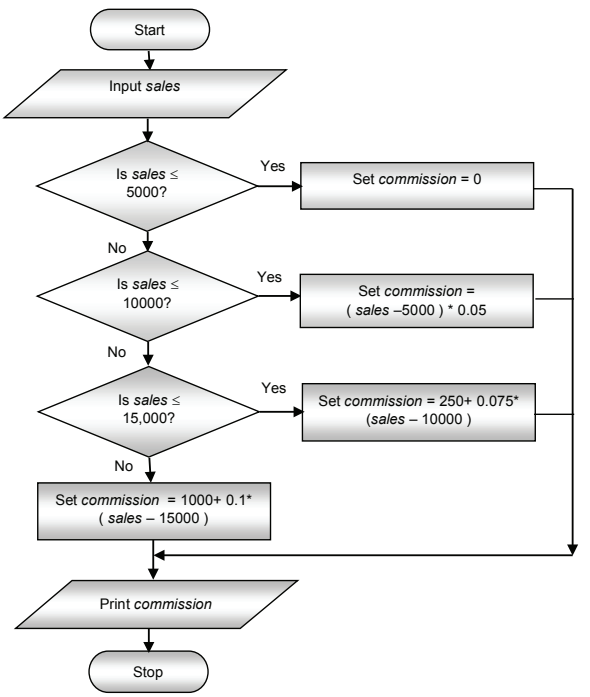


Fig. 1.20: Flowchart to compute the commission

Pseudocode 1.6

```
Begin
    Read: sales
    If ( sales <= 5000 )
        Set commission = 0
    Else If (sales <= 10000  )
        Set commission = ( sales - 5000 ) * 0.05;
    Else If ( sales <= 15000 )
        Set commission = 250 + ( sales - 10000 ) * 0.075;
    Else
```

```
        Set commission = 1000 + ( sales – 15000 ) * 0.1;
    Endif
    Print: "Computed commission = ", commission
End.
```

Example 1.6: *Draw a flowchart and write a pseudocode to find the sum of digits of a number n.*
Solution:

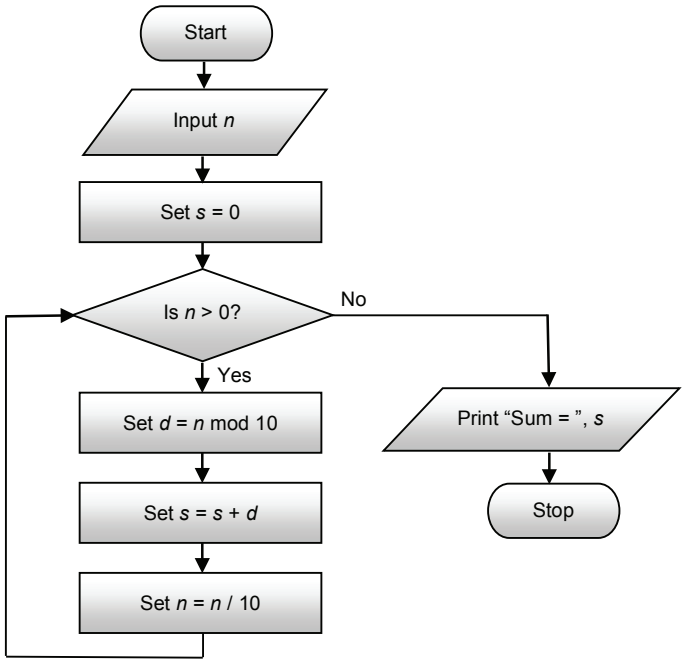


Fig. 1.21: Flowchart and pseudocode to find the sum of digits of a number

Pseudocode 1.7

```
Begin
    Read: n
    Set s = 0
    While  ( n > 0 ) do
        Set d = n mod 10
        Set s = s + d
        Set n = n / 10
    Endwhile
    Print: "Sum = ", s
End.
```

Example 1.7: *Draw a flowchart and write pseudocode to check whether the given number n is palindrome or not.*

Solution:

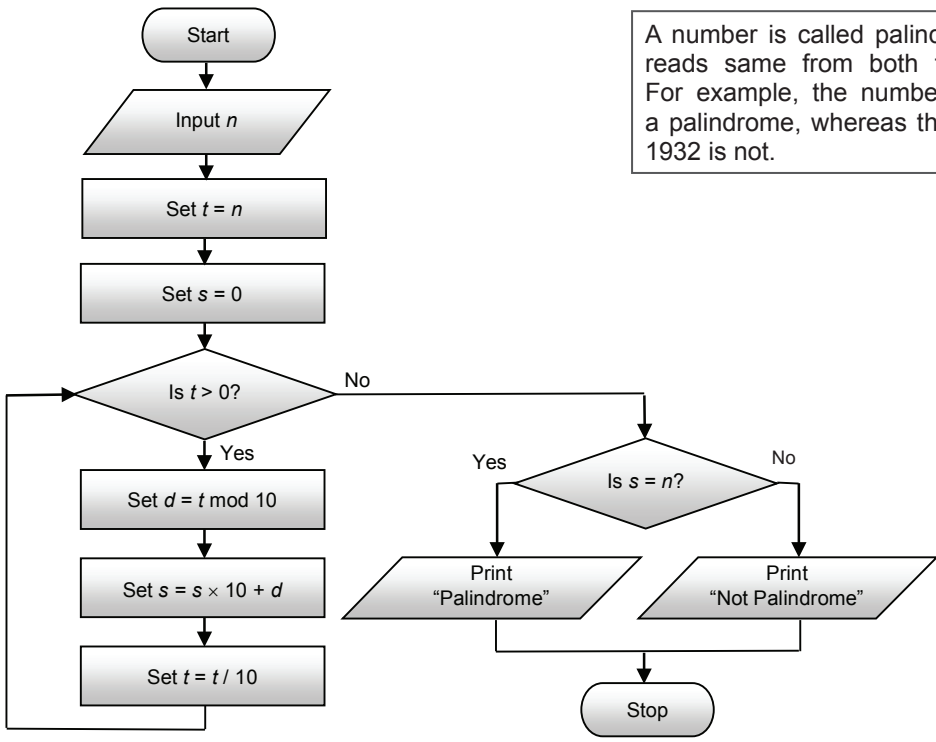


Fig. 1.22: Flowchart and pseudocode to check whether the given number *n* is palindrome or not

Pseudocode 1.8

```
Begin
     Read: n
     Set t = n,  s = 0
     While ( t > 0 ) do
          Set d = t mod 10
          Set s = s × 10 + d
          Set t = t / 10
     Endwhile
     If ( s = n ) then
          Print: "Palindrome"
     Else
          Print: "Not a Palindrome"
     Endif
End.
```

Example 1.8: *Draw a flowchart and write a pseudocode to check whether the given number n is an Armstrong number or not.*

Solution:

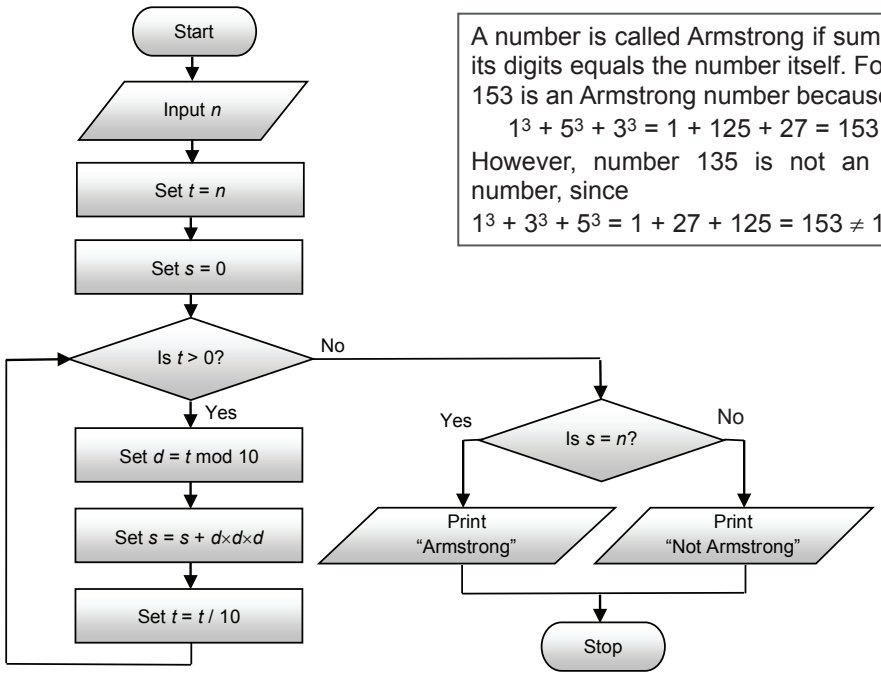


Fig. 1.23: Flowchart and pseudocode to check whether the given number *n* is an Armstrong number or not

Pseudocode 1.9

```
Begin
     Read: n
     Set t = n,  s = 0
     While  ( t > 0 ) do
          Set d = t mod 10
          Set s = s  + d × d × d
          Set t = t / 10
     Endwhile
     If  ( s = n ) then
          Print: "Armstrong"
     Else
          Print: "Not Armstrong"
     Endif
End.
```

Example 1.9: *Draw a flowchart to find whether the given natural number n is a prime number or not.*
Solution: A natural number is said to be prime if it is divisible by 1 and itself only, *i.e.*, it cannot be factorized. In addition, to this definition, an even number except 2 is not a prime number.

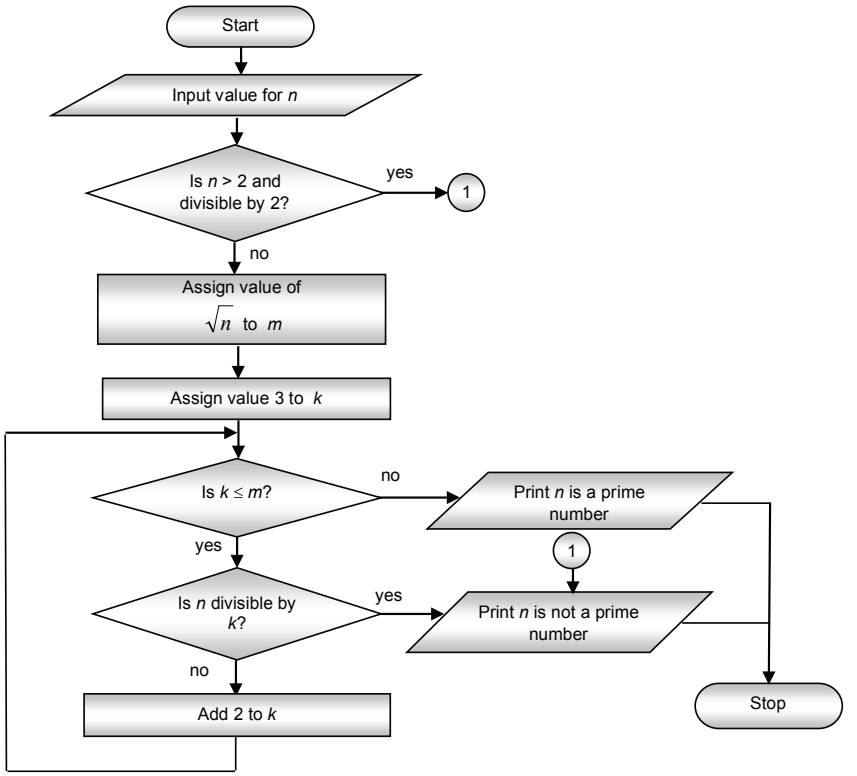


Fig. 1.24: Flowchart to check whether number *n* is prime or not

Therefore, our test criteria becomes

1. If *n* is greater than 2 and is even then *n* is not a prime number.
2. If test at step 1 fails, then we try to divide number *n* by factors $k = 3, 5, 7, \ldots \sqrt{n}$. Therefore, if *n* is divisible by any value of *k*, number *n* is not a prime number.
3. If test at step 2 also fails, then *n* is a prime number.

Following is the pseudocode to find whether the given positive number *n* is a prime number or not.

Pseudocode 1.10

```
Begin
   Read: n
   If ( n > 2 and n mod 2 == 0 ) then
       Print: n, " is not a prime number"
       Exit
   Else
       Set m = √n
```

```
        For k = 3 to m by 2 do
            If ( n mod k == 0 ) then
                Print: n, " is not a prime number"
                Exit
            Endif
        Endfor
        Print: n, " is a prime number"
    Endif
End.
```

Example 1.10: *To find highest common factor (HCF), also known as greatest common divisor (GCD), of two natural numbers m and n.*

Solution:

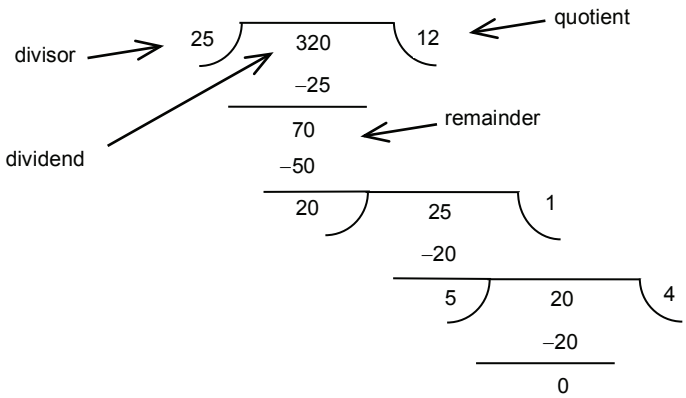


Fig. 1.25: Illustration of computational procedure for HCF/GCD

Fig. 1.25 demonstrates the long/continued division method to find the HCF/GCD of two natural numbers. You must have observed that in successive divisions, the divisor of the previous division becomes dividend, remainder of becomes divisor, and division is again carried out. This process is continued till the reminder becomes zero, and the current divisor is taken as HCF/GCD of the given natural numbers.

This process can be implemented by using the following steps

1. Perform division.
2. If remainder is zero, then stop and take the divisor as HCF/GCD.
3. Replace dividend by divisor.
4. Replace divisor by remainder.
5. Repeat from step 1.

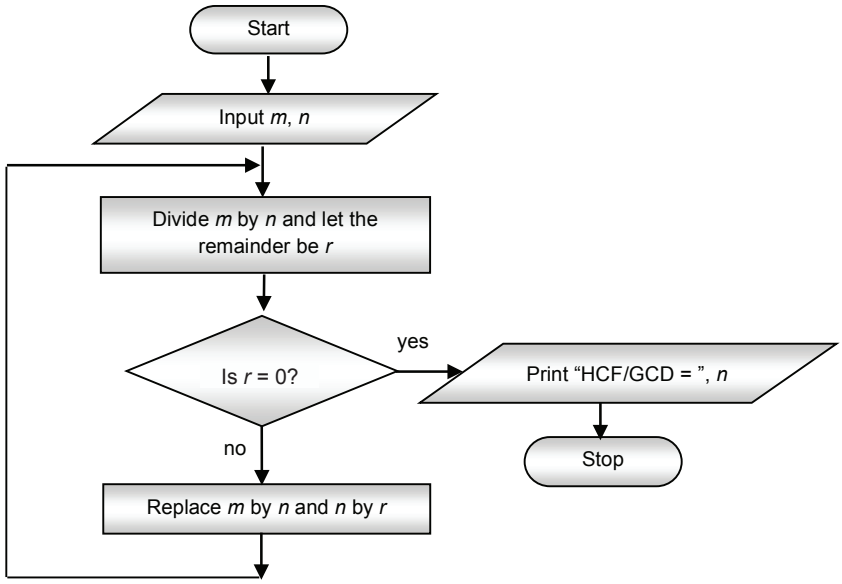


Fig. 1.26: Flowchart and pseudocode to compute HCF of two numbers

Pseudocode 1.11

```
Begin
     Read: m, n
     Set r = m mod n
     While ( r ≠ 0 ) do
          Set m = n
          Set n = r
          Set r = m mod n
     Endwhile
     Print: "HCF/GCD = ", n
End.
```

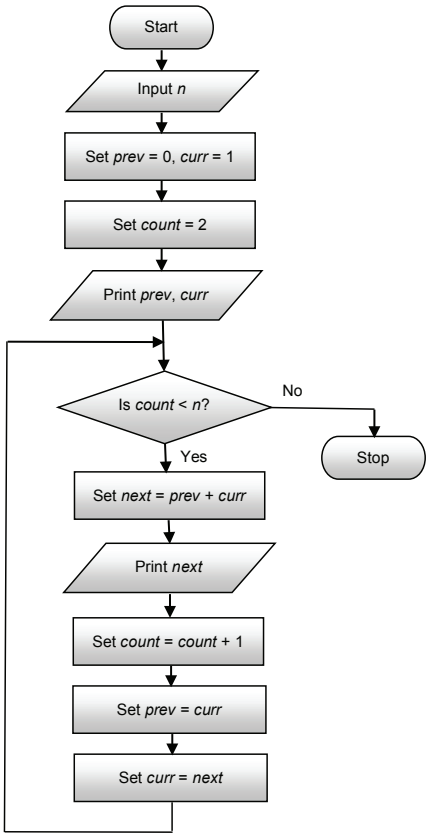
Example 1.11: *Draw a flowchart and write a pseudocode to print first n terms of the Fibonacci sequence. For example, if input value for n is 8, the output should be*

0 1 1 2 3 5 8 13

Solution: Observe that, leaving first two terms, each term is obtained as the sum of the immediately preceding two terms.

If we use variable *prev* for previous term, *curr* for current term, *next* for next term, and setting *prev* and *curr* to values 0 and 1, respectively, *i.e.*, first two terms of the sequence, then the entire sequence can be generated by using the recurrence relation

next = *prev* + *curr*
replace *prev* by *curr*
replace *curr* by *next*



Pseudocode 1.12

```
Begin
      Read: n
      Set prev = 0, curr = 1
      Set count = 2
      Print: prev, curr
      While ( count < n ) do
            Set next = prev + curr
            Print: next
            Set count = count + 1
            Set prev = curr
            Set curr = next
      Endwhile
End.
```

Fig. 1.27: Flowchart and pseudocode to print first *n* terms of the Fibonacci sequence

1.3 FROM ALGORITHMS TO PROGRAM

By now, you have learned that algorithms are a way to solve given problems using computer. Algorithm contains the logic to solve a given problem. This logic need to be converted to a program using a programming language. C language is our programming language for this course.

In this section, we will learn the basic aspects of C language that will enable us to convert simple algorithms to a C program.

1.3.1 Structure of a C Program

The usual order of instructions/statements written in a C program is as depicted below.

A C program is written in a free format. By free format, we mean that an instruction can start anywhere in a line and can end anywhere. Even an instruction can span more than one line. Spacing is of no consequence in C language, it is just used to enhance the readability of the program.

Further, C language is case sensitive, *i.e.*, it differentiates between lowercase and uppercase letters. Every C program is written in lowercase letters only.

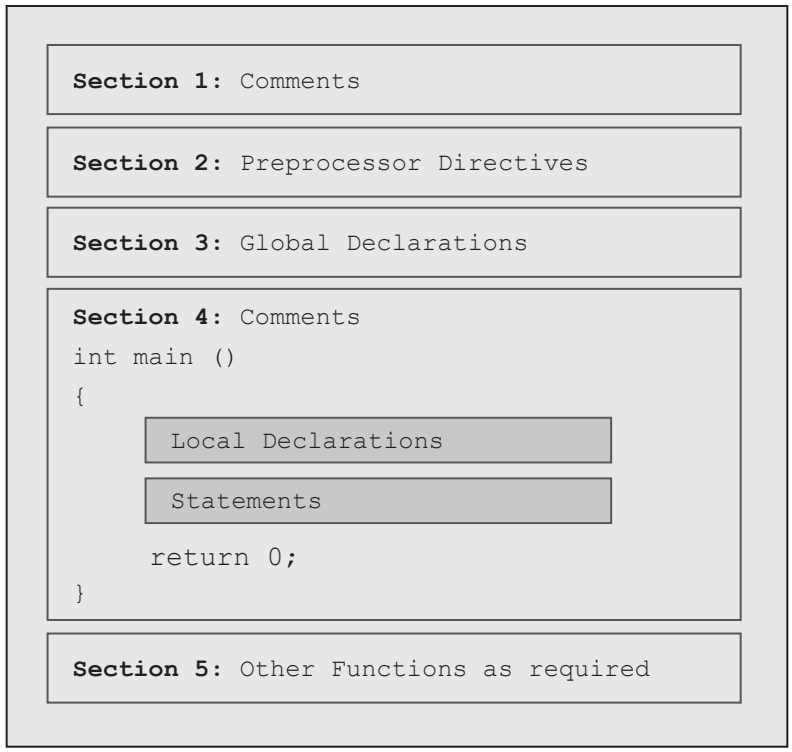


Fig. 1.28: General Structure of a C Program

Now, let us understand the role of each section in a C program one-by-one:

- **Section 1** is optional. If present it contains the description about the program. This description usually contains the information about the task being accomplished by the program.
  For example,

```
/*
*    Program to compute the factorial of a given
*    natural number (positive integer number).
*
*    Filename : prime.c
*/
```

  Note that any text enclosed in pair of "/*" and "*/", no space in between, is ignored by the compiler, *i.e.*, it is not translated to machine code.

- **Section 2** contains the preprocessor directives. The frequently used preprocessor directives are *#include* and *#define*. These directives tell the preprocessor how to prepare the program for compilation.
  The *#include* directive tells which header files are to be included in the program and the *#define* directive is usually used to associate an identifier with a literal (constant) that is to be used at many places in the program.

For example,

```
#include<stdio.h>
```

tells the preprocessor to include the contents of the header file named *stdio.h* to at this point in the program file.

Likewise,

```
#define N 200
```

tells the preprocessor to replace every occurrence of identifier N by value 200.

- **Section 3** is optional. If present, it contains the global declarations. These declarations usually include the declaration of data items (variables) which are to be shared between different functions in the program. In addition, these declarations can also include declaration of other functions (prototypes) to be used in the program.
- **Section 4** contains the *main()* function. The execution of the program always begins with the execution of the *main()* function. It can call any number of other functions, and those called function can further call other functions.

  The first section in the *main()* function, as well as other functions, contains local declarations. These declarations are local in the sense that they pertain to the requirements of that function only. The second section contains the instructions (also known as statements) that define the actions to be performed by the function.
- **Section 5** is also optional. If present, it contains the definitions of other functions.

  If the problem to be solved is simple and small in size, then only the *main()* function is sufficient to accomplish the task.

  However, if the problem is complex and the size of the problem is large, it is divided into small and independent subproblems, and then we write separate functions for each subproblem.

  The *main()* function coordinates the execution of these functions by appropriate calls to these functions, and synthesizes the solutions of the subproblems obtained from these functions.

1.3.2 Example C Programs

In order to have a feel of the structure of a C program, let us consider few simple example C programs.

The following program is the standard first C program that is given in every textbook of C. When executed, it simply displays the message *Hello, World*.

Listing 1.1

```
 1: /*
 2: *  Program to greet the user with the message "Hello World".
 3: *  File name: hello.c
 4: */
 5: #include<stdio.h>
 6: int main()
 7: {
 8:    printf("Hello, World\n");
 9:    return 0;
10: }
```

Test Run

```
Hello, World
```

Dissection of the Program

Here the line numbers are added for ready reference.

Lines 1-4 contains the comments.

Line 5 contains preprocessor *#include* directive.

Line 6 specifies a function named *main*. This is a special name that is recognized by the system. It points to the precise place in the program where execution begins. Every C program must have a *main* function.

Every C function has a return type associated with it. Where return type is one of the data types that determine the type of the value returned by the function. If a function is not supposed to return any value, its return type is specified as *void*.

Since main function is the function from where execution begins, it is usually declared with type *int*. A pair of parentheses follows the word *main*.

Line 7 contains character '{', called left brace or left curly bracket. There is also matching right brace '}' in line 10, which appears at the end of the main function. This pair of matching braces encloses the body of the function.

Line 8 uses library function *printf()* that displays "*Hello, World*" on the computer screen.

Line 9 uses *return* statement, which on execution terminates the execution of the program and return the control back to the operating system. In addition, it also returns value 0 to the operating system indicating that the program terminated successfully.

Line 10 marks the end of the *main* function as well as end of the program as this is the only function in the program.

Let us take another example program, where some input is provided by the user, computations is performed, and the results of the computations are displayed.

The program in this example, takes temperature in Celsius scale as input, computes its equivalent temperature in Fahrenheit scale, and displays the same on computer screen.

The relation between temperature in Celsius and Fahrenheit is

$$\frac{C}{100} = \frac{F - 32}{180}$$

which on simplification gives

$$F = 1.8 \times C + 32$$

This mathematical relation is used to compute F for given value of C.

Listing 1.2

```
1: /*
2: * Program to convert temperature from Centigrade scale
3: * to Fahrenheit scale
4: */
```

```
 5: #include<stdio.h>
 6: int main()
 7: {
 8:    float f, c;
 9:    printf("\nEnter temperature in Celsius scale: ");
10:    scanf("%f", &c);
11:    f = 1.8 * centigrade + 32;
12:    printf(„\nEquivalent temperature in Fahrenheit scale");
13:    printf( „ = %.2f\n",f);
14:    return 0;
15: }
```

Test Run

```
Enter temperature in Celsius scale: 30
Equivalent temperature in Fahrenheit scale = 86.00
```

Dissection of the Program

Lines 1-4 represent comments.

Line 5 represent preprocessor directive.

Line 6-15 represent the definition of the main function enclosed in pair of braces {}.

Line 8 declares two variables, *f* and *c* of type *float*, which can represent and store two real numbers (float) in computer memory. Variable *c* is used to hold the value, representing temperature in Celsius scale, entered by the user during program execution. Variable *f* is to used to hold the computed value, representing temperature in Fahrenheit scale, equivalent to given temperature in Celsius scale.

Line 9 uses library function *printf()* that displays "*Enter temperature in Celsius scale:* " on the computer screen. We call these kinds of messages as *user prompts* as they guide the user to enter the desired input. In this case, the value to be entered is temperature in Celsius scale.

You must have observed that the entire sequence of characters enclosed in double quotes is not printed. Then, *what is the role of characters* '\n' *in the beginning*? These two characters together represent the *new line* character. Although *new line* character is a combination of two characters, *i.e.*, '\' and 'n', they are translated by the C compiler into a single character. The *new line* character instructs the compiler to advance to the next line before printing subsequent information.

Line 10 uses another library function *scanf()* that accepts the user input and stores in variable *c* (in fact, in a memory location that is reserved for *c*). The first argument is a *format string* that is enclosed in double quotes and tells the system how the entered value is to be interpreted. Following the format string, there is a list of one or more variables; each variable prefixed with character '&', called *addressof* operator.

Line 11 is an assignment statement that computes the temperature in Fahrenheit scale equivalent to given temperature in Celsius scale, and assigns to variable *f*, *i.e.*, stores in variable *f*.

Lines 12-13 uses library function *printf()* that outputs the stored value in variable *f* along with the message *"Equivalent Temperature in fahrenheit = "*. The use of these kinds of messages is not mandatory, but is very useful as they make the output easy to interpret.

Line 14 uses *return* statement, which terminates the program and returns value 0 to the operating system.

1.3.3 Creating, Compiling and Executing a Program

Once the algorithm is ready, the next step is to convert the algorithm into a computer program. During this conversion each step of the algorithm is coded as one or more C language instructions. It is recommended that student should write the program first on a piece of paper before typing in the computer.

To demonstrate the various steps, we will consider Turbo C/C++ Compiler, which is easy to use for the beginners.

Creating and Editing Programs

Once the program is ready on paper, we type in computer memory using a *text editor*. A text editor helps us to enter the character data into computer memory, allows editing (changing) the data in computer memory, and save the data from memory in a disk file to secondary memory with extension "*.c*".

This stored file is known as *source file*, and its contents are known as *source code*. This source file will be the input for the compiler.

The programmer must carefully follow the C language rules. Violation of language rules results in grammatical errors, more precisely known as *syntax errors*. The *Compiler* will check for syntax errors. These errors must be eliminated before moving further.

Compiling Programs

The source code in the source file, stored on the disk, must to be translated into machine language. This job is done by the *Compiler*. The C compiler actually is a combination of two separate programs – the *preprocessor* and the *translator*.

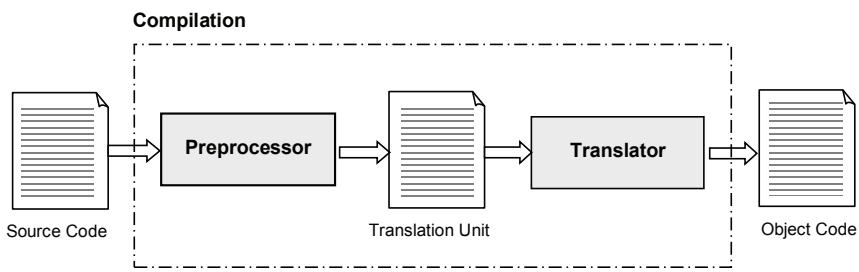


Fig. 1.29: Compilation process

The *preprocessor* reads the source code and prepares it for translation. While reading the source code, it scans the code for preprocessor directives and processes them accordingly.

For example, when it encounters the *#include* directive, it substitutes that directive with the contents of the specified header file (such as *stdio.h*), and when it encounters the *#define* directive, it substitutes

the identifier with the specified literal (constant). The output from the preprocessor is an intermediate file, known as *translation unit.*

The translator reads the translation unit instruction-by-instruction and checks them for their grammatical accuracy. If there is any syntax error, it flags an error message – called *diagnostic message* on the screen. These diagnostics messages help the programmer to identify the cause of these errors and the places where they are present.

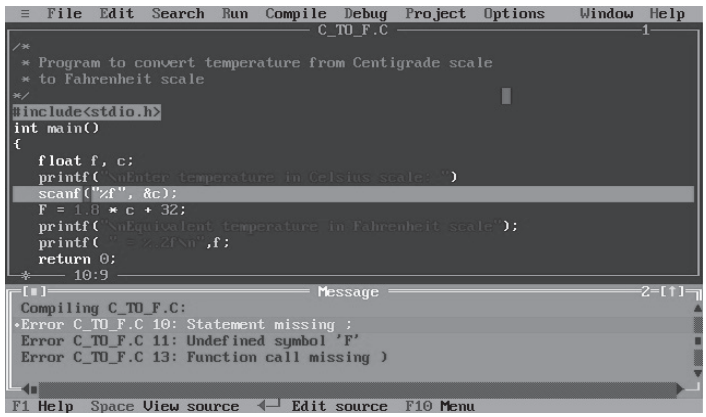


Fig. 1.30: Turbo C/C++ Compiler's screen shot indicating syntax errors

Therefore, if there is even a single syntax error, the translation process, known as *compilation,* is terminated. In this case, open the source file using the text editor, and make the necessary corrections and repeat the compilation.

However, if there are no syntax errors in the translation unit, the translator rereads the instruction from the beginning, translates them into machine language, and writes them onto a disk file. The translated version of the source code is known as *object code,* and is stored in the disk file with extension "**.obj*".

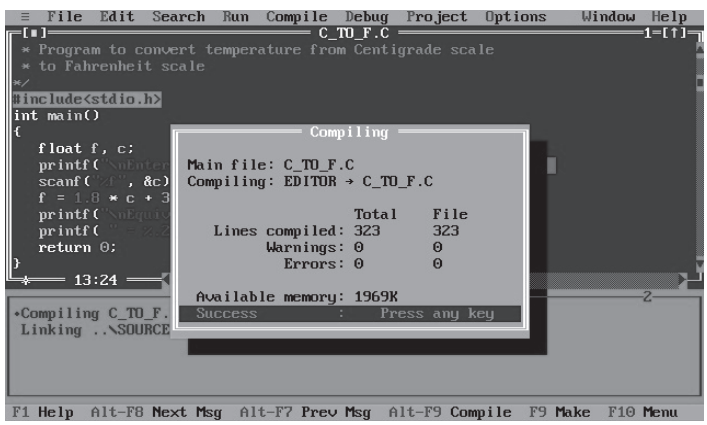


Fig. 1.31: Turbo C/C++ Compiler's screen shot indicating success of compilation process

Linking Programs

Once the source code is translated into object code, though it is in machine language, still it is not in executable form. The reason being is that it may be referring to library functions (pre-written functions supplied with the compiler in the form of libraries). All these functions also need to be included in the object code to get a final machine code, which is in the executable form, known as *executable code,* and that is stored in disk file with extension "**.exe*". This executable code is the final form of the program that is ready for execution.

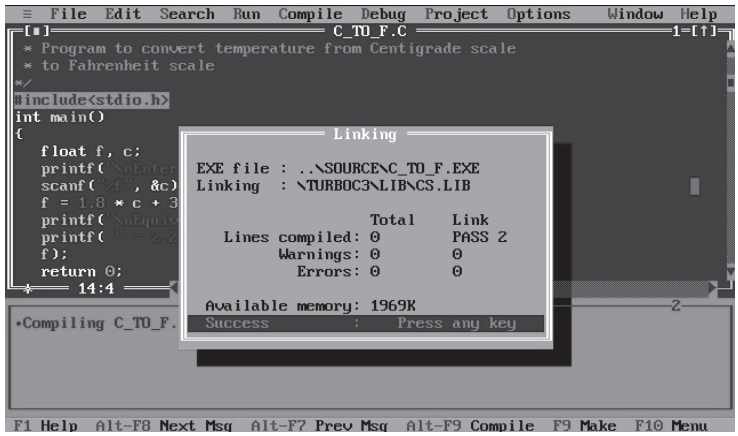


Fig. 1.32: Turbo C/C++ Compiler's screen shot indicating success of linking process

Executing Programs

Once the program is linked, it is ready for execution. To execute a program we give an operating system command, such as *run*, to load the program into computer memory and execute it. Getting the program into memory is the function of an operating system program known as *loader.* The loader locates the executable program in the secondary storage, reads it and brings it into the computer memory. Once the program is loaded, the operating system transfers the control to the program and the program begins its execution.

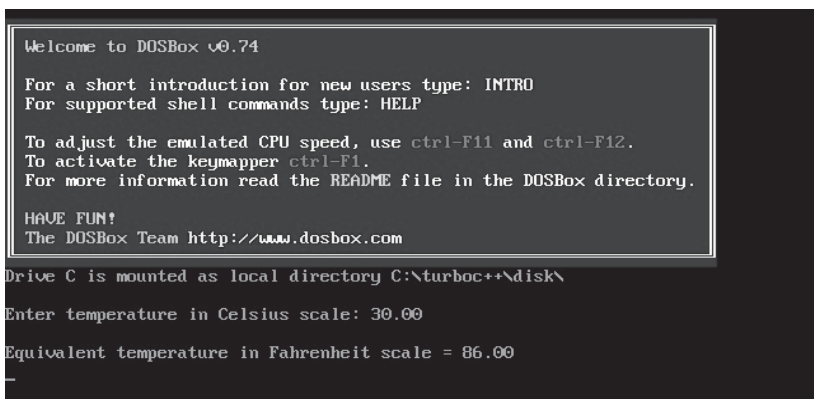


Fig. 1.33: Turbo C/C++ Compiler's user screen showing program output

Testing the Program

Even when the program is executing, the output of the program may not be correct.

This could be because of logical errors in the program. A *logical error* is a mistake that the programmer made while designing the solution to the problem. For example, a programmer tells the computer to calculate the net pay by adding deductions to the gross salary instead of subtracting. A compiler cannot detect these errors.

Therefore, the programmer must find and correct logical errors by carefully examining the program output for a set of data for which results are already known. Such type of data is known as *test data*.

Syntax errors and *logical errors* collectively are known as *bugs*. The process of identifying and eliminating these errors is known as *debugging*.

1.3.4 Various C Compilers

The following are two predominantly used C Compilers on Window based systems.

- Turbo C/C++ Complier
- Dev C++ Complier

The following are few links to online C Compliers:

- https://www.codechef.com/ide
- https://ide.geeksforgeeks.org
- https://www.onlinegdb.com/online_c_compiler
- https://www.programiz.com/c-programming/online-compiler/
- https://www.tutorialspoint.com/compile_c_online.php
- https://www.jdoodle.com/c-online-compiler/
- https://techiedelight.com/compiler/

1.4 GETTING STARTED WITH C LANGUAGE

C programming language was developed by Dennis Ritchie in AT & T Bell laboratory in 1972, and still it is one of the most popular languages.

Programming language can be divided into two broad categories, depending on their level of interaction with the underlying hardware of the computer:

- **Low-level Languages** - under this category, we have machine language and assembly language. These languages permit efficient use of the computer.

  But the problems with these languages are:

  - These are hardware dependent, *i.e.*, programs written using these languages are not portable, *i.e.*, cannot be used on other computers.
  - Programming using these languages is not an easy job. One must have thorough knowledge of the architecture of the computer.
- **High-level Languages** - under this category, we have vast collection of languages starting with FORTRAN, COBOL, PASCAL, etc., and at present, the major high level languages in demand are C, C++, Java, and Python. These languages are designed for better programming efficiency, *i.e.*, faster program development.

These languages have following advantages:

- The syntax for writing program instructions is very much like English statements. This enables readers to learn high-level languages (HLLs) quickly. In addition, the programs written in HLLs can be easily understood, which facilitates its maintenance.
- The programs written in HLLs are not hardware dependent. This means that program written for one machine can be transferred to another machine with minimal changes or none at all, *i.e.*, these languages are portable.

The C language stands between these two categories. That is why it is often called *middle-level* language, since it was designed to have best of both the worlds, *i.e.*, good programming efficiency as well as good machine efficiency.

- To achieve programming efficiency, the C language has all the elements of any other modern high-level language.
- To achieve machine efficiency, the C language has requisite features to access any hardware component of the system, to operate at register level, and to interface with high-speed assembly language routines.

1.4.1 Characteristics of C Language

C language was and will remain as one of the most popular programming languages.
Some of the key characteristics that add to its popularity are:

- It is a general purpose programming language, and therefore, can be used to solve wide variety of problems.
- It supports structuring programming, and therefore, programs developed in it can easily be understood.
- It is free form, *i.e.*, no rigid format for writing programs. Any instruction can start from anywhere and end anywhere. In addition, a single instruction can span (can be written in) many lines.
- It is case sensitive. It distinguishes between lowercase (small) and uppercase (capital) alphabets.
- It was the first language to provide rich set of operators.
- It allows you, through pointers, to access any storage location and the devices connected to your computer through your program.
- It allows you to manipulate internal processor registers, and thus it is very useful for low-level programming.
- It is portable, that means any program written in it can be used on any computer without or with little modification(s).
- It allows you to develop your own library of functions that can be linked to any program like standard library functions.

1.4.2 Application Areas of C Language

The strength of C language makes it a natural choice for

- System programming that include writing software for language translators, device drivers, editors, linkers, loaders, etc.

- Network software to implement different communication protocols.
- Graphics programming that include writing software for graphical user interfaces (GUIs), scientific visualization, and presentation graphics, etc.
- Embedded systems where C routines are interfaced with high speed assembly language routines and the resulting code is stored in ROM chip which is a part of the embedded system.

1.4.3 BASIC BUILDING BLOCKS OF C LANGUAGE

One of the difficult aspects of learning a programming language is that almost each and everything is interrelated. Therefore, it seems almost impossible to understand anything before you know little about everything.

In this section, we will learn about various functional elements, also known as *building blocks,* of the C language.

1.4.3.1 Character Set

In a most basic sense, a C program is a sequence of characters. When these characters are submitted to the compiler, they are interpreted in various contexts as *characters*, *identifiers*, *constants*, and *statements*. The characters used in a C source program belong to the American Standard Code for Information Interchange (ASCII pronounced as *ass-kee*) set.

The characters in C language are grouped into the following categories:

- Letters (A-Z, a-z)
- Digits (0-9)
- Special characters (, : ; ~ > < # % ‘ ^ + - * . = ! | ( ) { } [ ] / &)
- White space characters (*Blank space, horizontal tab, etc.)*

All C compilers ignore white space characters. These characters are basically used to enhance the readability and understandability of a C program.

1.4.3.2 Tokens

A *token* is a smallest entity that has a meaning in itself. A C program is a sequence of tokens. Tokens in C language can be classified into following categories:

- Keywords
- Identifiers
- Literals
- Punctuators
- Operators

Keywords

Every word in C is either classified as a *keyword* or an *identifier*.

Keywords are basically those words that have predefined and fixed meaning and these meanings cannot be changed. These keywords serve as basic building blocks for forming program instructions (statements). All keywords are written in lowercase. Their incorrect usage results in a syntax error.

Table 1.2: Keywords in C

asm	default	for	short	union
auto	do	goto	signed	unsigned
break	double	if	sizeof	void
case	else	int	static	volatile
char	enum	long	struct	while
Const	extern	register	switch	
continue	float	return	typedef	

Identifiers

An *identifier* is basically a name in a program. Identifiers can be used to denote *variables*, *arrays* and *functions*. These user-defined names consist of sequence of characters, where each character can be a letter, a digit or an underscore '_', and cannot start with a digit. Both uppercase and lowercase letters are permitted, however, lowercase are commonly used. The underscore is usually used to link the two or more words to create meaning full names.

Since C language is case sensitive (it distinguishes between lowercase and uppercase), therefore, the identifiers *price* and *Price* will be treated as two different identifiers.

RULES FOR IDENTIFIERS

1. The first character must be an alphabet or an underscore.
2. Must consist of letters, digits or underscore only.
3. Cannot use a keyword.
4. Cannot contain a white space.

Examples of valid identifiers:

```
myFile      roll_no      date_of_birth      _chk
file10      N1TO10       area               AbCdE
```

Literals

A *literal* (often referred to as constants) refer to fixed value that do not change during the execution of a program.

- **Integer Literals** – numbers without a decimal point. An integer number consists of a sequence of digits preceded by an optional sign, plus (+) or minus (–).
- **Real Literals** – numbers having fractional parts.
- **Single Character Literals** – consists of a single character enclosed in single quotes (') such as 'A'. Internally, each character is represented by an integral value that represents the ASCII code of the character. For example, character 'A' will be represented by integral value 65, character 'B' by 66, etc.

  The C language allows certain characters that cannot be entered/typed directly from the keyboard such as *backspace*, *tab*, *carriage return*, *newline,* etc. These characters are represented by a sequence that begin with character \ (back slash), and are known as *escape sequences*.

Table 1.3: Common Escape Sequences

Escape Sequence	Represents	Effect
\n	Newline	Subsequent output starts from new line.
\t	Horizontal Tab	Moves over to the next eight-space-wide field.
\r	carriage return	Carriage return.
\0	Null character	Terminates a string.

- **String (multi-character) Literals** – consists of a sequence of characters enclosed in double quotes. The characters can be from any character in the character set.

  Note that single character literal such as 'A' is not equivalent to single character string literal as "A".

Punctuators

Punctuators are also called separators. Various punctuators used in C++ are described in following table.

Table 1.4: List of some punctuators and their description

Punctuators	Description
Brackets [ ]	Used to enclose array subscripts.
Parentheses ( )	Used to enclose arguments in function declaration as well as function definition, parameters in function calls and expressions.
Braces { }	Used to enclose a block of statements. Also used to enclose list of elements while initializing arrays.
Comma,	Used to separate arguments in function definition, and parameters in function call.
Semicolon ;	Used to terminate a statement.

Operators

Operators are the verbs of C language that let the user perform operations on values. The C language's rich set of operators is one of its distinguished features.

These operators can be enumerated in following categories:

- Arithmetic Operators (+, -, *, /, %)
- Relational Operators (<, <=, >, >=, ==, !=)
- Logical Operators (!, &&, ||)
- Bit-wise Operators (~, >>, <<, &, |)
- Special Operators
  - ☞ Increment & Decrement Operators (++, - -)
  - ☞ The *sizeof* Operator
  - ☞ The *addressof* Operator (&)
  - ☞ Indirection/de-reference Operator (*)
  - ☞ Ternary/Conditional Operator (? :)

1.4.3.3 Concept of Data Type

A *data type* is an interpretation applied to a string of bits. Formally, *data type* is defined as a finite set of values along with well-defined set of rules for operations that can be performed on these values.

Data types in C are of two types:

- **Built-in Data Types**
  - ☞ Built-in data types are the most basic data types in C.
  - ☞ Built-in means that they are pre-defined in C and can be used directly in a program.
  - ☞ Examples are *char*, *int*, *float*, and *double*.
  - ☞ Apart from these, we also have *void* data type.
- **Derived Data Types**
  - ☞ These are derived from existing data types (built-in or user-defined).
  - ☞ Examples are *arrays* and *pointers*.
- **User-defined Data Types**
  - ☞ These are created by the user to meet their requirements.
  - ☞ Examples are *structure*, *union*, and *enumeration*.

In this section, we will learn about built-in data types only. The remaining data types, as per the requirements of the syllabus will be discussed at the appropriate place.

The various built-in data types, also known as fundamental data types or basic data types, supported by C are summarized in Table 1.5.

Table 1.5: Built-in data types

Name	Description
Integer	
int	Integer numbers
Real/Floating-point	
float	Single precision floating-point numbers *(precision is 6 decimal places)*
double	Double precision floating-point numbers *(precision is 12 decimal places)*
Character	
char	A Single Character

The *int* data type is used for integers, and consists of a subset of integers. Table 1.6 shows the different types of integer numbers, their memory requirements, and the range of values for each type.

Table 1.6: Type of Integer Numbers

Type	Memory Requirements (in bytes)		Range of Values
	16-bit Compiler **(Turbo C/C++)**	***32-bit Compiler*** **(Dev C/C++)**	
short	2	2	for 2-byte integer –32768 to +32767 for 4-byte integer –2147483648 to +2147483647
int	2	4	
long	4	4	

The *float* and *double* data types are used for numbers that have a decimal point. The only difference between *float* and *double* data types is that the range and precision, the *double* data type has greater precision than that of *float* data type.

A real value is an approximation of the desired real number correct to certain decimal positions. A real value of type *float* is accurate to six decimal positions; where as a real value of type of *double* is accurate to twelve decimal positions.

Table 1.7 shows the different types of real numbers, their memory requirements, and the range of values for each type.

Table 1.7: Type of Real Numbers

Type	Bytes	Range
float	4	3.4×10^{-38} to $3.4 \times 10^{+38}$
double	8	1.7×10^{-308} to $1.7 \times 10^{+308}$

The *char* data type is used for characters, and it requires only 1-byte of memory. Here is a special thing about *char* data type - characters are stored in memory using their ASCII codes which are numeric, *i.e.*, integers.

The word *void* means empty, therefore, we can say *void* data type has no value.

It is useful in many situations. One of such situation — it is used as return type for functions that do not return a value.

1.4.3.4 Constants

A constant refers to fixed values that do not change during the execution of a program. Constants can be handled using following ways:

- **Using literal** – directly encoding the value in an arithmetic expression.
- **Using symbolic constants** – A symbolic constant is a name that substitutes a literal.

  For example,

  ```
  #define PI 3.142
  ```

  Here, the *#define* pre-processor directive associates the name PI with the literal 3.142.

  During the compiling, the preprocessor will replace each occurrence of name PI with literal 3.142.
- **Using a variable declared as constant**

  A variable, as discussed next, is something whose value may change during the program execution. However, a variable can be marked as constant using keyword *const*.

  For example,

  ```
  const float pie = 3.142;
  ```

  Here, the variable named *pie*, which belong to data type *float,* is initialized with value 3.141, and is marked as constant. Thereafter, we can use the value of variable in our program, but we cannot change its value, i.e, now variable *pie* will behave as a constant.

1.4.3.5 Variables

A variable provides us with named storage that we can write to, retrieve, and manipulate throughout the course of the program.

In other words, variables are memory locations in the computer's memory that holds data. Contents of a variable may vary – hence the name *variable*. Variables hold different kind of data and the same variable might hold different values during the execution of a program.

Each variable in C language is associated with a specific data type, which determines the size and layout of its associated memory, the range of values that can be stored within that memory, and the set of operations that can be applied to it.

Declaring Variables

In C language, each variable in a program must be declared. The syntax for declaring and defining a variable is

```
type v1, v2,..., vn;
```

where *v1*, *v2*,..., *vn* are the names of the variables separated by comma.

Following are some examples of variable declarations and definition:

```
int count, m, n;
float value, sum;
char ch;
double deviation;
```

First statement declares variables *count*, *m* and *n* of type *int*. Compiler reserves 2 or 4 bytes (depending on the compiler) for each of these variables.

Second statement declares variable *value* and *sum* of type *float*. Compiler reserves 4-bytes for each of these variables.

Third statement declares variable *ch* of type *char*. Compiler reserves 1-bytes for it.

Fourth statement declares variable *deviation* of type *double*. Compiler reserves 8-bytes for it.

Initializing Variables

In addition to declaring variable, an initial value can also be provided to the variable.

To do that, a variable is followed by character '=' and then the value to be given to the variable. Consider the following statement

```
int sum = 0;
```

It initializes variable *sum* to a value 0.

1.4.3.6 Expressions

An *expression* is a formula for computing a value. It consists of a sequence of operands and operators. The operands may contain function references, variables, and constants. The operators specify the action to be performed on the operands.

In the following expression

```
a + b
```

plus (+) is an operator and *a, b* are operands.

The C language supports following types of expressions:

- Arithmetic expressions
- Relational expressions
- Logical expressions

Each type of expression takes certain types of operands and uses a specific set of operators. Evaluation of every expression produces a value of specific type. Remember that expressions are not statements, but may be components of statements.

For example, consider the following line of text

```
x = 2.0/3.0 + a * b;
```

The entire line is a statement, but the portion after the equal sign is an expression. In particular, this line of text represents an assignment statement that assigns the value of the expression to variable *x*.

1.4.3.7 Statements

Statements perform a number of tasks, such as computing, storing the results of computations, altering the flow of control, reading and writing from or onto devices/files, and providing the information for the compiler.

1.4.3.8 Handling Input/Output

In almost every program, the users have to input some data that will be stored in memory, and then subsequently processed. The intermediate and final results of the computations are also stored in memory. And finally, there is need to output the results of the computations so that user can use them in their day-to-day work.

The common input device is the keyboard and the output device is a computer screen, sometimes called the *monitor*. This subsystem comprising keyboard and monitor is referred to as a *console*.

To use entire range of I/O functions, we need to include header file named *stdio.h* in all our programs.

Table 1.8: I/O Functions

Function Category	Function Name	Description
Unformatted	getchar()	Returns a character that has been recently typed. The typed character is echoed to the computer screen. After typing the appropriate character, the user is required to press *Enter* key.
	getche()	Returns a character that has been recently typed. The typed character is also echoed to the computer screen. But the user is not required to press *Enter*.
	getch()	Returns a character that has been recently typed. But, neither the user is required to press *Enter* key nor the typed character is echoed to the computer screen.
	putchar()	Display a character on the screen.
	gets()	Accepts a string from the keyboard.
	puts()	Display a string on the screen.
Formatted	scanf()	Accepts formatted data from the keyboard.
	printf()	Displays formatted data on the screen.

The only difference between these functions is that the formatted functions permit the input from the keyboard or output sent to a screen to be formatted as per the requirements.

For example, if different values are to be displayed, how many columns on screen to be used, and how much space between two values is to be given. If a value to be displayed is of real type, then how many decimal places to output.

Listing 1.3

```
/*
   Program to demonstrate working of character I/O functions
*/
#include <stdio.h>
int main()
{
   char ch;
   printf( "\nEnter any character: " );
   ch = getchar();
   printf( "\nCharacter you typed is " );
   putchar(ch);
   printf( "\nEnter any character: " );
   ch = getche();
   printf( "\nCharacter you typed is " );
   putchar(ch);
   printf( "\nEnter any character: " );
   ch = getch();
   printf( "\nCharacter you typed is " );
   putchar(ch);
   return 0;
}
```

Test Run

```
Enter any character: A↵
Character you typed is A
Enter any other character: x
Character you typed is x
Enter any other character:
Character you typed is H
```

Listing 1.4

```
/*
   Program to illustrate working of string I/O functions
*/
#include <stdio.h>
int main()
{
   char str[31];
   printf( "\nEnter string of length <= 30: " );
   gets( str );
   printf( "\nString you typed is " );
   puts( str );
   return 0;
}
```

Test Run

```
Enter string of length <= 30: Wel Come
String you typed is Wel Come
```

The *scanf()* function allows us to input data in a specified format. Its syntax is

```
scanf( "format string ", list of addresses of variables );
```

where the *format string* contains the format specifiers that begin with character '%' and are separated by space or comma.

The list of addresses of variables are used so that *scanf()* function can place the data received from the keyword. The address of a variable is obtained by using address operator (character &), and pronounced as *addressof* operator.

Table 1.9: List of commonly used format specifiers

Format Specifier	Used For
%d	signed decimal integer
%f	single precision floating real number
%lf	double precision floating real number
%c	single character
%s	single-word string

On execution, the input data must be entered strictly according to the specified format string otherwise results can be very strange.

The *printf()* function allows to output the data in a specified format. Its syntax

```
printf( "format string ", list of variables );
```

where the *format string* contains the format specifiers, the escape sequences, and the text to be output along with the output data.

All the format specifiers used with *scanf()* function are valid for *printf()* function.

Listing 1.5

```
/*
   Program to demonstrate working of formatted I/O functions
*/
#include <stdio.h>
int main(void)
{
   int a;
   printf( "\nEnter values for a : " );
   scanf ( "%d", &a );
   printf( "\nValue of a = %d\n", a );
   return 0;
}
```

Test Run

```
Enter values for a : 150
Value of a = 150
```

UNIT SUMMARY

In this chapter, we have learned that

- Computer is an electronic machine that performs tasks or computations according to a set of instructions called *programs.*
- A computer can receives input in variety of forms, process these input as per the instruction, and present the results in variety of forms.
- Components of a computer system include input unit, output unit, memory unit, control unit, arithmetic and logic unit, and secondary storage unit.
- The control unit, the arithmetic and logic unit, and the main memory unit, collectively, are known as Central Processing Unit (CPU).
- All the external devices, such as input device and output devices, connected to the computer are known by the common name as peripherals.
- Hardware refers to the parts of a computer that you can see and touch, including the case and everything inside it.
- Software, in its most general sense, is a set of instructions or programs that tell the hardware what to do.
- Operating system (OS) manages the resources of the computer system, and provides an interface using which the user can interact with the computer to perform various tasks.
- Software can be categorized as system software, application software, and utility software.
- Booting is a process that refers the system getting initialized.

- An algorithm is a finite sequence of instructions defining the solution of a particular problem.
- An algorithm can be represented using a flowchart and psuedocode.
- A flowchart is a diagrammatic representation of an algorithm, where different geometrical shapes are used to represent different type of operations.
- Pseudocode is a plain language description of the steps in an algorithm.
- Syntax errors are the result of incorrect use of language rules.
- Logical errors are the result of lack of understanding the problem or its solution.
- C language is developed by Dennis Ritchie.
- C language is a middle-level language and portable.
- C language is case sensitive.
- General structure of a C program consists of five sections are – comments, preprocessor directives, global declarations, *main()* function, and other functions as required.
- Key application areas of C language include systems programming, network programming, GUI, scientific visualization, embedded systems, etc.
- Different steps in building a program are – creation, compilation, linking, and executing.
- Text editor is a program that helps us in entering and changing the program in computer memory, and finally saving on a disk file.
- Source code is program written in a high-level language such as C and is stored in a disk file known as source file.
- A compiler is a program that translates the C program to the machine language. It has two components – *preprocessor* and *translator.* The preprocessor processes the program and prepares it for translation. The translator does the final translation.
- Process of translating the source program into object program is known as *compilation.*
- Object code is program in machine language that produced by the compiler and is stored in a file known as object file.
- Linker is a program that adds the machine language of library functions from system libraries and user-defined function, and produces final executable program.
- Executable code is a program in machine language that produced by the linker and is stored in a file known as run file, which is ready for execution.
- Loader is a program that transfers the executable program from a disk file on a secondary storage into computer memory and prepares it for execution.
- Testing is the process to ensure that program works correctly for all possible inputs to the program.
- Debugging is the process of identifying, locating and fixing the bugs. It is not an independent activity in the program development process; it is always associated with testing.
- A value coded directly or represented through a identifier that does not change during the course of program execution is called a constant, where as those values that may change is called a variable.
- Data type is a set of values along with set of permissible operations.

- Various data types supported by C language are named as built-in data types, derived data types, and user-defined data types.
- Every data item to be processed by the program needs to be declared.
- Input/output in a C program can be formatted or unformatted.

EXERCISE

Subjective Questions

1. What is a computer?
2. Explain the working of functional components of a computer with the help of a block diagram.
3. When we say memory is volatile, what does it mean?
4. Differentiate between hardware and software
5. What an operating system?
6. What do you understand by term booting?
7. What are types of software?
8. What is an algorithm? What are the desirable characteristics of a good algorithm?
9. What is a flowchart? Describe various flowcharts symbols.
10. What is a pseudocode?
11. Given a choice, would you like to represent an algorithm using flowchart or pseudocode?
12. Describe the general structure of a C program.
13. Describe the various steps in the development of a C program.
14. Why C language is called middle-level language?
15. What is a token? Describe various tokens in C.
16. What is a data type? Name various data types supported by C language.
17. Differentiate between constants and variables.
18. Describe the syntax for declaring and initializing variables.
19. Name various functions for handling input/output in C.

Multiple Choice Questions

1. If a program written in any language on a given platform can be transported to another platform without any change or with minimal changes, then the language is said to _____.
   (a) Understandable (b) Readable
   (c) Portable (d) Maintainable
2. Who developed C language?
   (a) Von Neuman (b) Dennis Ritchie
   (c) Peter Norton (d) Ken Thompson

3. Which of the following character is used to terminate an instruction in a C program?
(a) , (comma) (b) : (colon)
(c) ; (semicolon) (d) . (period)

4. Which of the following cannot be a first letter of an identifier?
(a) digit (b) underscore
(c) alphabet (d) uppercase alphabet

5. Which of the following are used to enclose the body of a function?
(a) [ ] (b) { }
(c) ( ) (d) < >

6. The C language can be used on which of the following platforms?
(a) Unix (b) Windows
(c) Linux (d) All of the above

7. The C language is a _____ language.
(a) High (b) Low
(c) Middle (d) Symbolic

8. The acronym ANSI stands for _____.
(a) American National Standards International
(b) American National Software Incorporation
(c) American National Standards Institute
(d) American National Standards Instructions

9. Find the odd term
(a) Source code (b) Object code
(c) Executable code (d) ASCII code

10. Testing of a program is done to ensure that
(a) it should not contain any syntax error
(b) it performs its intended task
(c) it compiles successfully
(d) it should not run infinitely

11. The term *bug* refers to _____.
(a) Syntax error (b) Logical error
(c) Runtime error (d) All of the above

12. The acronym ASCII stands for _____.
(a) American Standard Code for International Information
(b) American System for Compiler Information International
(c) American System for Code Information International
(d) American Standard Code for Information Interchange

13. Which of the following statements is not true about C language?
(a) Every instruction is terminated by semicolon
(b) It is case insensitive
(c) Comments can be placed anywhere in the program code
(d) It has features of both high-level and low-level languages

14. The term *debugging* refers to the process of _____.
(a) translating the source code to object code
(b) linking the object code with the system libraries
(c) identifying the bugs and fixing them
(d) None of above

15. The term *portability* means that _____.
(a) Program code is understandable
(b) Program code is maintainable
(c) Program code is bug free
(d) Program code can be transported without/with minimal changes to another computer

16. The C programs are converted into machine language using _____.
(a) Assembler
(b) Interpreter
(c) Compiler
(d) Operating system

17. Extension of file containing machine code produced by the Turbo C/C++ Compiler is _____.
(a) *.com (b) *.exe
(c) *.c (d) *.obj

18. Which of the following is not a keyword?
(a) for (b) main
(c) break (d) else

19. Which of the following is not a valid data type in C language?
(a) Char (b) float
(c) long (d) double

20. Which of the following doesn't denote a primitive data value in C?
(a) "a" (b) 'k'
(c) 35.25 (d) 14

ANSWERS													
1.	(c)	2.	(b)	3.	(c)	4.	(a)	5.	(b)	6.	(d)	7.	(c)
8.	(c)	9.	(d)	10.	(b)	11.	(d)	12.	(d)	13.	(b)	14.	(c)
15.	(d)	16.	(c)	17.	(d)	18.	(b)	19.	(a)	20.	(a)		

Computational Problems

Develop algorithm and represent it using flowchart and/or psuedocode:

1. To test whether the given year is leap year or not.
2. To test whether the given date is valid or not.
3. To find the largest digit in a natural number 'n'.
4. To find the smallest digit in a natural number 'n'.
5. To find the nth prime number.
6. To test whether a triangle can be formed with given three line segments with lengths 'a', ' b' and 'c' in cms.
7. To test whether a triangle can be formed with given three angles with measures 'a', ' b' and 'c' in degrees.
8. Two test whether two lines intersect each other or not. Given two points A(xa,ya) & B(xb,yb) on one line and C(xc,yx) & D(xd, yd) on the second line.
9. The Body Mass Index (BMI) of a person is calculated as BMI = $\frac{W}{H^2}$, where *W* is weight in kilograms and *H* is height in meters.

BMI	Classification
< 18.5	Under weight
18.5–24.9	Normal weight
25.0–29.9	Overweight
30.0–34.9	Class I obesity
35.0–39.9	Class II obesity

Classify obesity of a person whose weight and height is given.

10. The monthly telephone bill is to be computed as follows:

    Minimum ₹ 200 for upto 100 calls

    plus ₹ 0.60 per call for next 50 calls

    plus ₹ 0.50 per call for next 50 calls

    plus ₹ 0.40 per call for any call beyond 200 calls.

Compute monthly bill for given number of calls.

PRACTICALS

1. Familiarization with the operations of the PC/Laptop: concept of booting, configuring desktop, working with files and folders, installing software, and executing various applications.
2. Familiarization with programming environment: creating & editing, compiling, and executing a C program.

KNOW MORE

Problem solving is a skill, and a skill can't be learned overnight. Therefore, to inculcate the required skills among the students, teacher should demonstrate the problem solving approach by taking suitable examples, and involving the students in developing the solution.

Finally, teacher should give ample related problems so that students can apply the concepts learned to solve problems they have not seen or solved earlier.

REFERENCES & SUGGESTED READINGS

1. R.G. Dromey, How to solve it by Computer, Pearson Education.
2. R. S. Salaria, Problem Solving & Programming in C, Khanna Book Publishing Co(P) Ltd., New Delhi.
3. E. Balagurusamy, Programming in ANSI C, Tata McGraw Hill, New Delhi.
4. V. Anton Spraul, Think Like a Programmer, No Starch Press.
5. https://onlinecourses.nptel.ac.in/noc21_cs01/preview
6. https://ocw.mit.edu/courses/intro-programming/
7. https://www.programiz.com/c-programming
8. https://www.javatpoint.com/c-programming-language-tutorial

2 Arithmetic Expressions and Precedence

UNIT SPECIFICS

This unit discusses the topics related to various types of operators, their precedence and associativity, various types of expressions, rules governing the evaluation of expressions, and library functions.

RATIONALE

In problems related to science & engineering, the programmer has to deal with variety of algebraic expressions that may involve various arithmetic operations and standard mathematical & trigonometric functions. To convert those algebraic expressions to their equivalent expressions in C, and to control their order of evaluation, the programmer should have knowledge of various operators supported by C, how expressions are evaluated, and other related issues.

This unit helps the students to understand various aspects related to operators, expressions, and their evaluation.

PRE-REQUISITES

- Linear algebra
- Standard mathematical functions

UNIT OUTCOMES

Upon completion of the unit, students will be able to

U2-O1: Explain various operators and their usage

U2-O2: Convert algebraic expressions to C expressions

U2-O3: Explain the way expressions are evaluated

U2-O4: Explain the concept of precedence and associativity among operators

U2-O5: Explain type conversion in assignment and expressions

Unit 2 Outcomes	EXPECTED MAPPING WITH COURSE OUTCOMES (1- Weak Correlation; 2- Medium correlation; 3- Strong Correlation)							
	CO-1	CO-2	CO-3	CO-4	CO-5	CO-6	CO-7	CO-8
U2-O1	1	-	-	-	-	-	-	-
U2-O2	-	1	-	-	-	-	-	-
U2-O3	-	-	2	-	-	-	-	-
U2-O4	-	-	2	-	-	-	-	-
U2-O5	-	-	1	-	-	-	-	-

2.1 INTRODUCTION

Operators are the verbs of a language that let a user perform computations on values. You can think of operators as the verbs and operands as the subjects and object of those verbs. C language's rich set of operators is one of its distinguished features.

An *expression* is a formula consisting of operands and operators linked together to compute a value. The computed values are to be stored in a variable for future use. This is done using an assignment statement.

In a computer program, variety of computations are to be performed as per requirements of the problem in hand. Yet, there are some routine type computations that may be part of almost every program. Therefore as a convenience, every modern programming language provides a support for these routine type computations in the form of library functions.

This unit attempts to describe these semantic units of the C language.

2.2 OPERATORS

Operators are the foundation of any programming language. The functionality of any programming language is incomplete without the use of operators.

Operators, generally, are of two types:

- *Unary Operators* – operators that operate on a single operand. The unary operators are prefixed with their operands.

  For example, *–x*, here unary minus '–' operator precedes by its operand *x* (supposed to hold a numeric value), and negates its value, *i.e.*, changes its sign.
- *Binary Operators* – operators that operate with two operands. The binary operators are embedded between their operands.

  For example, *a+b*, here binary plus '+' operator appears between its operands *a* and *b* (both supposed to hold numeric values), and adds the value of variable *b* to that of *a*.

The C language's rich set of operators is one of its distinguished features. These operators can be enumerated in following categories:

- Arithmetic Operators (+, –, *, /, %)
- Relational Operators (<, <=, >, >=, ==, !=)
- Logical Operators (!, &&, ||)
- Bit-wise Operators (~, >>, <<, &, | , ^)
- Special Operators
  - Increment & Decrement Operators (++, - -)
  - The *sizeof* Operator
  - The *addressof* Operator (&)
  - Indirection/de-reference Operator (*)
  - Ternary/Conditional Operator (? :)

In this unit, we will focus mainly on operators used in arithmetic computations, the other type of operators will be introduced for the completeness of the topic.

2.2.1 Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

Table 2.1: Binary Arithmetic Operators

Operator	Symbol	Form	Operation
Addition	+	x + y	Adds value of y to value of x.
Subtraction	–	x – y	Subtracts value of y from value of x.
Multiplication	*	x * y	Multiplies value of x by value of y.
Division	/	x / y	Divides value of x by value of y.
Modulus	%	x % y	Divides value of x by value of y and gives remainder.

Integer Arithmetic

When both the operands are integers, the operations will be called *integer arithmetic* and will always yield an integer value.

Consider the following statements

```
int x = 13, y = 5;
```

then we have the following results:

```
x – y =  8
x + y =  18
x * y =  65
x / y =  2 (decimal part truncated)
x % y =  3 (remainder of division)
```

Real Arithmetic

When both the operands are real numbers, the operations will be called *real arithmetic* and will always yield a real value. Since real numbers (floating-point values) are rounded to permissible number of significant digits, result of arithmetic operations is an approximate value of correct result.

That is

```
 6.0 /  7.0  =  0.857143
–1.0 /  3.0  = –0.333333
 3.0 / –2.0  = –1.500000
```

The modulus operation cannot be used with real operands.

Mixed-mode Arithmetic

When one of the operands is real and the other is integer, the operations will be called *mixed-mode arithmetic* and will always yield a real value. The integer operand is converted into real operand and then real arithmetic is performed, resulting in a real value.

That is

```
 6 / 7.0  →  6.0 / 7.0  =  0.857143
–1.0 / 3  →  –1.0 / 3.0 =  – 0.333333
```

Note that

15 / 10.0 = 1.5

15.0 / 10 = 1.5

where as

15 / 10 = 1

2.2.2 Relational (Comparison) Operators

These operators are used to compare values of the operands that must be compatible, in order to facilitate decision-making. If the comparison succeeds, they return a value 1 (equivalent to Boolean value *true*) and if comparison fails then they return a value 0 (equivalent to Boolean value *false*).

Table 2.2: Relational (Comparison) Operators

Operator	Symbol	Form	Meaning
Greater than	>	x > y	Return True if x is greater than y.
Greater than or equal to	>=	x >= y	Return True if x is greater than or equal to y.
Less than	<	x < y	Return True if x is less than y.
Less than or equal to	<=	x <= y	Return True if x is less than or equal to y.
Equal to	==	x == y	Return True if x and y are equal.
Not equal to	!=	x != y	Return True if x and y are not equal.

Note that in C language, the non-zero value is treated as equivalent to Boolean True and zero value as equivalent to Boolean False. Further, there is no support for Boolean data as such in C language in old versions of C language. The new version of C language, known as C99 standard, supports the Boolean type.

2.2.3 Logical Operators

These operators are used to form compound conditions by joining two or more simple conditions formed using relational operators.

Table 2.3: Logical Operators

Operator	Symbol	Form	Meaning
Logical AND	&&	x && y	Return True if both the operands are True.
Logical OR	||	x || y	Return True if either of the operands is True.
Logical NOT	!	! x	Return True if operand is false, and False if operand True (complements the operand).

2.2.4 Bitwise Operators

Bitwise operators act on operands as if they were string of binary digits. It operates bit by bit, hence the name. Note that these operators are used with integral operands only.

To demonstrate the working of bitwise operators, let $x = 10$ (0000 1010 in binary) and $y = 4$ (0000 0100 in binary), assuming that 8 bits are used to represent an integer number.

Table 2.4: Bitwise Operators

Operator	Symbol	Form	Meaning	Example
Bitwise AND	&	x & y	Return 1 if corresponding bits are 1 else 0.	x = 0000 1010 (10) y = 0000 0100 (4) x & y = 0000 0000 (0)
Bitwise OR	|	x | y	Return 1 if either or both of corresponding bits are 1 else 0.	x = 0000 1011 (11) y = 0000 0101 (5) x | y = 0000 1111 (15)
Bitwise XOR	^	x ^ y	Return 1 if corresponding bits are not equal.	x = 0000 1011 (11) y = 0000 0101 (5) x ^ y = 0000 1110 (14)
Bitwise NOT	~	~ x	Inverts 1 to 0 and 0 to 1	x = 0000 1011 (11) ~x = 1111 0100 (-12)
Bitwise right shift	>>	x >> y	Shifts the bits of left operand to right by number of bits specified by the right operand. The right most bit is shifted out after each shift, while the left most bit is restored. Note: Each right shift by 1-bit is equivalent to integral division by 2.	x = 0000 1011 (11) x >> 1 = 0000 0101 (5)
Bitwise left shift	<<	x << y	Shifts the bits of left operand to left by number of bits specified by the right operand. The left most bit is shifted out after each shift, while the right most bit is filled with 0. Note: Each left shift by 1-bit is equivalent to multiplication by 2.	x = 0000 1011 (11) x << 1 = 0001 0110 (22)

2.2.5 Special Operators

Let us discuss each of them briefly, and their utility you will be able to appreciate in the subsequent sections and chapters.

2.2.5.1 Increment & Decrement Operators

The C language supports two very useful operators – increment operator (++) and decrement operator (--). These operators can be used with all basic data types. The increment operator adds 1 to the operand, while decrement operator subtracts 1 from the operand.

Both operators are unary operators and can be used in the prefix as well as postfix notation as shown below:

```
++k;        or      k++;
--k;        or      k--;
```

Here, `++k` (as well as `k++`) is equivalent to `k = k + 1` or `k += 1`, and `--k` (as well as `k--`) is equivalent to `k = k - 1` or `k - = 1`.

These operators are most frequently used in *while* and *for* loops as control variables.

While `++k` or `k++` means the same thing when they form statements independently, they behave differently when they are used as part of other expressions.

Consider the following statements

```
int y, k = 5;
y = ++k;
```

In this case, first the value of *k* is incremented and then assigned to *y*, and hence y and k will have same value as 6. Thus, the above statements are equivalent to following statements

```
int y, k = 5;
++k;
y = k;
```

However, if we write the above statements as

```
int y, k = 5;
y = k++;
```

then first the current value of *k* (value 5) is assigned to y and then the value of *k* is incremented, and hence *y* will have value 5 while *k* will have value as 6.

Thus, the above statements are equivalent to following statements

```
int y, k = 5;
y = k;
++k;
```

or

```
int y, k = 5;
y = k;
k++;
```

2.2.5.2 The *sizeof* Operator

The *size of* operator returns the size, in bytes, of the given operand. The syntax of *sizeof* operator is

```
sizeof( exp )
```

where *exp* represents a data type (built-in or user-defined), a literal/constant or a variable. For example

```
sizeof( float )
```

returns value 4.

Table 2.5: Use of *sizeof* Operator

sizeof Operator used as	Returns	Justification
`sizeof(12)`	2	Integer constant by default belong to data type *int*, and size for *int* is 2 on Turbo C/C++ and 4 on Dev C++.
`sizeof('a')`	1	Size of Character constant is 1.
`sizeof(int)`	2	Size of data type *int* is 2 on Turbo C/C++ and 4 on Dev C++.
`sizeof(125.25)`	8	Real constant by default belong to data type *double*.
`sizeof(125.25F)`	4	Real constant is of data type *float*, and size for *float* is 4.
`double x;` `sizeof(x);`	8	Variable *x* is of type *double*, and size for *double* is 8.

2.2.5.3 The *addressof* Operator

Character '&' when prefixed with a variable returns the address of the memory locations, hence its name *addressof* operator. It is used in the *scanf()* function and to initialize the pointers. The use of *addressof* (&) operator is demonstrated in Table 5.11.

2.2.5.4 Indirection Operator

Character '*' when prefixed with a pointer variable returns the value stored in a memory location whose address is held in pointer variable. That is, value is accessed through pointer variable indirectly, hence its name indirection operator.

Consider following statements

```
int y = 10, x;
int *py;
py = &y;
x = *py;
```

The expression "**py*" returns the values at address which held in pointer variable *py*, *i.e.*, value of variable y which is 10, gets assigned to variable *x*.

Table 2.6: Use of *addressof* (&) Operator

Addressof (&) Operator used as	Task Performed

`int y = 10;`
`int *py;`
`py = &y;`
	First statement declares and defines variable *y* of type *int*. It also initializes it with value 10.
	Second statement declares and defines a pointer variable *py* that can hold an address of a memory location reserved for storing value of type *int*.
	Third statement assigns the address of variable y to pointer variable *py*.

`int x, y;` `scanf("%d %d", &x, &y);`	First statement declares and defines two variable *x* and *y* of type *int*. Second statement takes two integer values as input from the keyboard and stores those values at addresses of variable *x* and *y*, respectively.

2.2.5.5 Conditional/Ternary Operator

Consider the following segment, written in pseudocode

```
if ( a > b ) then
   set big = a
else
   set big = b
endif
```

This segments assigns the maximum of the values of *a* and *b* to *big*. It is clear that the value assigned to variable *big* will depend on the outcome of the test condition "*a* > *b*".

Such expressions are known as *conditional expressions*, and can be written using conditional operator. The syntax of using ternary operator is

```
exp1 ? exp2 : exp3;
```

where *exp1*, *exp2*, *exp3* are expressions.

The expression *exp1* is evaluated first. If it is non-zero (true), then the expression *exp2* is evaluated, and that is the value of the conditional expression; otherwise expression *exp3* is evaluated, and that is the value of the conditional expression. Note that only one of the expression *exp2* and *exp3* is evaluated.

Thus, the above pseudocode segment using conditional operator can be written as

```
big = ( a > b ) ? a : b;
```

Parentheses are not necessary around the first expression of the conditional expression since the precedence of '?:' is very low, just above assignment operator. However, the use of parentheses is recommended as they make the condition part easier to see.

2.2.5.6 Assignment operators

Assignment operators are used to assign values to variables. For example,

```
a = 5
```

Here '=' is a simple assignment operator that assigns the value 5 to the variable *a*. It can also be used to assign value of another variable or an expression.

There are various compound operators in C like

```
a += 5
```

that adds to the variable and later assigns the same. It is equivalent to

```
a = a + 5
```

Operators of type '+=' are also called *shorthand* assignment operators.

Table 2.7: Assignment Operators

Operator	Example	Equivalent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5

Operator	Example	Equivalent to
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

2.3 EXPRESSIONS

An *expression* is a formula consisting of one or more operands and zero or more operators linked together to compute a value. An operand may be a function reference, a variable, an array element or a constant.

For example, in the expression

```
a + b
```

plus character '+' is an operator and *a* and *b* are operands.

There are four types of expressions in C. These are

1. Arithmetic expressions
2. Relational expressions
3. Logical expressions
4. Conditional expressions

Each type of expression takes certain types of operands and uses a specific set of operators. Evaluation of every expression produces a value of specific type. Expressions are not statements, but may be components of statements.

For example, consider the line

```
x = 2.0/3.0 + a * b;
```

The entire line is a statement, but the portion after the assignment operator is an expression.

In this section, our discussion will be limited to arithmetic expression.

2.3.1 Arithmetic Expressions

An arithmetic expression is made up of operands and arithmetic operators. It produces a value that is of type *int*, *float* or *double*. When an expression involves only integral operands, it is known as a *pure integer expression*, when it involves only real operands it is known as a *pure real expression*, and when it involves both integral and real operands it is known as a *mixed mode expression*.

Table 2.8: Some sample arithmetic expressions

Mathematical/Algebraic Expressions	The C Arithmetic Expressions
$\frac{a+b}{2}$	`(a+b)/2`
$a+\frac{b}{c}+d$	`a+b/c+d`
$\frac{ab}{c-d^2}+d$	`(a*b)/(c-d*d)+d`
$1+\frac{a}{b+\frac{1}{c}}$	`1+a/(b+1/c)`
$\frac{a}{\frac{c+b}{d}}-e$	`a/((c+b)/d)-e`
ax^2+bx+c	`a*x*x+b*x+c`
$ut+\frac{1}{2}at^2$	`u*t+0.5*a*t*t`
$m\left(a\times h+\frac{v^2}{2}\right)$	`m*(a*h+(v*v)/2)`

While converting mathematical/algebraic expressions into C arithmetic expressions, parentheses can be used to control associativity and the order in which operators are evaluated. If parentheses are

present in an expression, then the expression within the parentheses is evaluated first and within the parentheses the implicit precedence is observed. If there is nesting of parentheses (parentheses inside parentheses), then the inner most parentheses are evaluated first.

2.3.2 Evaluation of Arithmetic Expressions

As you know expressions are evaluated by performing one operation at a time, and same is the case when expressions are evaluated by a computer. The order of evaluation of individual operations is governed by the precedence and associativity of operators.

However, when individual operations are performed, the following cases can happen:

- When both of the operands are of type integer, the integer arithmetic will be performed and the results of the operation will be an integer value.

  For example, 5/2 will yield 2 not 2.5 as the fractional part is ignored.
- When both of the operands are of type real, the real arithmetic will be performed and the results of the operation will a real value.

  For example, 5.0/2.0 will yield 2.5.
- However, when one operand is of type integer and the second one is of type real, the mixed mode arithmetic will be performed. In this case, first integer operand will be converted into real operand, and then real arithmetic will be performed and the results of the operation will be a real value. For example, 5/2.0 or 5.0/2 will yield 2.5, because in the first case, value 5 will be converted to 5.0 and then division will be performed while in the second case value 2 will be converted to 2.0 and then the division will be performed.

To illustrate the way arithmetic expression are evaluated, consider the following expression

```
5 * 4 / ( 1 + 5 * 2 / 3 + 6 ) + 8 * ( 7 / 4 )
```

This expression is evaluated as demonstrated in Table depicted below:

Table 2.9: Illustration of evaluation of arithmetic expression

Operation by Operation Evaluation of Expression	Description of Each Operation
`5 * 4 / ( 1 + 5 * 2 / 3 + 6 ) + 8 * ( 7 / 4 )`	Given expression
`5 * 4 / ( 1 + 10 / 3 + 6 ) + 8 * ( 7 / 4 )`	5 is multiplied by 2, giving10
`5 * 4 / ( 1 + 3 + 6 ) + 8 * ( 7 / 4 )`	10 is divided by 3 using integral division, giving 3
`5 * 4 / ( 4 + 6 ) + 8 * ( 7 / 4 )`	3 is added to 1, giving 4
`5 * 4 / 10 + 8 * ( 7 / 4 )`	6 is added to 4, giving 10
`5 * 4 / 10 + 8 * 1`	7 is divided by 4 using integral division, giving 1
`20 / 10 + 8 * 1`	5 is multiplied by 4, giving 20
`2 + 8 * 1`	20 is divided by 10 using integral division, giving 2
`2 + 8`	8 is multiplied by 1, giving 8
`10`	8 is added to 2, giving 10, which is the final value of the expression.

2.3.3 Type Conversion

When values of different types are mixed in an expression, then they are converted to the same type before performing any operation. This process of converting values from one type to another type is known as *type conversion*.

The C language facilitates type conversion in following two forms:

- Implicit type conversion
- Explicit type conversion

2.3.3.1 Implicit Type Conversion

An implicit type conversion is automatically performed by the compiler without programmer's intervention. This is done when expression is in mixed mode, so as not to lose the information.

Conversion in Evaluation of Expressions

When an operation is to be performed on operands of different types, then operand of lower size is converted to a type that matches the operand of higher size. Once this conversion of ranks is done, the operation is performed and the result of the operation will be of type having higher size.

For example, suppose variables *x* and *y* are of type *float* and variables `i` and `j` are of type *int*, in expression:

```
x / i + y * j
```

While evaluating expression "*x*/*i*", integer operand *i* will be converted to *float* type and then operand *x* will be divided by *i*, and the result of the expression will be a value of type *float*.

Conversions in Assignment Statements

An assignment statement involves an assignment operator (=) and two operands. The operand on the left hand side of assignment operator is a variable while the operand on the right side can be a literal/constant, variable or an expression.

Depending on the difference of the size of the operands, the C system tries to either *promote* or *demote* the right operand so as to make its size match the left operand (variable). *Promotion* occurs if the right operand has lower size; *demotion* occurs if the right operand has higher size.

2.3.3.2 Explicit Type Conversion

An explicit type conversion is user-defined that forces an operand or expression to be of specific type. The process of explicit type conversion is also known as *type casting*.

The syntax for explicit type conversion or type casting is

```
(type)operand
```

Here *type* in the parentheses is referred to as *cast operator*, which has a precedence of 2. As shown in the syntax, to cast data from one type to another type, we specify the new type in parentheses before the value that we want to convert.

For example, to convert variable *x* of type *int* to *float* type, we write the expression as

```
(float)x
```

It is important to note here that in this operation like any other unary operation, the value stored in *x* is still of type *int*, but the value of the expression is promoted to type *float*.

One use of the cast operator is to ensure that the result of a divide operation on integer operands results in a real number.

For example, if we want to calculate the average, which may contain fraction as well, without a cast operator, the result will be an integer number because of integer division.

To force the real result, we can write the statement to compute average as

```
average = (float)total / n;
```

In this statement, there is explicit conversion of *total* to *float*, and then implicit conversion of *n* to *float*. Now real division will take place and the result will be a real number, as desired, which will be assigned to *average*.

2.4 PRECEDENCE AND ASSOCIATIVITY

Precedence is used to determine the order in which different operators are evaluated in an expression.

Associativity is used to determine the order in which different operators with same precedence are evaluated in an expression.

Precedence is applied before associativity to determine the order in which expressions are evaluated. Associativity is applied later, if necessary.

Precedence

The concept of precedence is well defined in the subject of mathematics. You must be aware of rule BODMAS – Brackets, Of, Division, Multiplication, Addition, and Subtraction. In algebra, division and multiplication is performed before addition and subtraction.

The following is a simple example of precedence:

```
5 + 3 * 4
```

This expression consists of one addition and one multiplication operator. As you know from the knowledge of algebra, multiplication has higher precedence than addition, multiplication is performed before addition.

Therefore, the expression will be evaluated as

```
( 5 + ( 3 * 4 ) ) → ( 5 + 12 ) → 17
```

giving 17 as the value of the entire expression.

Associativity

Associativity is applied when more than one operator of same precedence is used in an expression. Associativity can be *left-to-right* or *right-to-left*. Left-to-right associativity evaluates the expression by starting on the left and moving to the right whereas the right-to-left associativity evaluates the expression by starting on the right and moving to the left.

The following is a simple example of associativity:

```
2 + 5 + 7
```

This expression consists of two addition operators. Here the associativity determines how the sub expressions are grouped together. Since the addition operator has left-to-right associativity, the expression will be grouped as

```
( ( 2 + 5 ) + 7 ) → ( 7 + 7 ) → 14
```

giving 14 as the value of the entire expression.

Table 2.10: Precedence and associativity of operators

Operator	Description	Precedence Level (Rank)	Associativity
() [] –> .	Function call Array element reference Structure operator used with pointer to a structure Structure operator used with structure variable	1	Left-to-right
! ~ ++ –– + – * & (type) sizeof	Logical NOT operator 1's complement Increment Decrement Unary plus Unary minus Pointer reference (indirection) Address of Type cast Size of an operand	2	Right-to-left
* / %	Multiplication Division Modulus (remainder)	3	Left-to-right
+ –	Addition Subtraction	4	Left-to-right
<< >>	Left shift Right shift	5	Left-to-right
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	6	Left-to-right
== !=	Equal to Not equal to	7	Left-to-right
&	Bitwise AND	8	Left-to-right
^	Bitwise exclusive OR	9	Left-to-right
|	Bitwise inclusive OR	10	Left-to-right
&&	Logical AND	11	Left-to-right
||	Logical OR	12	Left-to-right
?:	Condition (ternary)	13	Right-to-left
= += –= *= /= %= &= ^= != <<= >>=	Assignment operators	14	Right-to-left

2.5 LIBRARY FUNCTIONS

Quite often it is necessary to evaluate several mathematical functions that are used frequently, such as square root, logarithms, exponentials, trigonometric functions, etc. Rather than each user writes their own code to compute these functions, C compiler provides them as built-in library functions as a convenience.

The user has to include *math.h* header file in the program unit. In addition to these functions, there are many other library functions that perform specialized functions, and are defined in different header files. For example, library functions that manipulate strings are defined in *string.h*.

Arguments to certain library functions are limited by the definition of the function. For example, logarithm of a negative number is mathematically undefined. Similarly, square root of a negative number is mathematically undefined, and therefore, not permitted.

Table 2.11: Some commonly used mathematical functions

Function	Description
`pow(x,y)`	Returns *x* raised to power *y, i.e.*, .
`pow10(p)`	Returns the value .
`exp(x)`	Returns *e* to the *xth* power, *i.e.*, .
`log(x)`	Returns the value of *ln*(*x*).
`log10(x)`	Returns the value of *log10*(*x*).
`sqrt(x)`	Returns the value of .
`sin(x)`	Returns the value sine of *x*, where the angle *x* is in radians.
`ceil(x)`	Returns the smallest integer greater than or equal to *x*.
`floor(x)`	Returns largest integer less than or equal to *x*.

ILLUSTRATIVE EXAMPLES

In this section, let us consider few situations and/or problems that will give us an opportunity to use operators and expressions. Before that let us learn about the concept of *dry run*.

A *dry run* basically means we will imitate a computer and trace the execution of the instructions using paper pencil. During the process, we record values of the variables as they change during the execution including the initial values. Note that if a variable is not initialized, its value will be shown with character '?'.

In the illustrative examples, you will see the dry run of the program and/or section of the program code, wherever it is deemed necessary to clarify the key concept.

Example 2.1: *Swap values of two variables 'a' & 'b' using third variable 't'.*

Solution:

```
int a=10,b=20,t;          /* 1 */
t = a;                    /* 2 */
a = b;                    /* 3 */
b = t;                    /* 4 */
```

The last three lines of the code can also be written in C as

```
t = a, a = b, b = t;
```

Dry Run

Statement No.	*a*	*b*	*t*
1	10	20	?
2	10	20	10
3	20	20	10
4	10	20	10

Example 2.2: *Swap value of two variables 'a' & 'b' without using third variable and using arithmetic addition (+) and subtraction (–) operations only.*

Solution:

```
int a=10,b=20;              /* 1 */
a = a + b;                  /* 2 */
b = a –b;                   /* 3 */
a = a –b;                   /* 4 */
```

This task can also be implemented using following single statement:

```
a = ( a + b ) -  ( b = a );
```

Dry Run

Statement No.	*a*	*b*
1	10	20
2	30	20
3	30	10
4	20	10

Example 2.3: *Swap value of two variables 'a' & 'b' without using third variable and using arithmetic multiplication (*) and division (/) operations only.*

Solution:

```
int a=10,b=20;       /* 1 */
a = a * b;           /* 2 */
b = a / b;           /* 3 */
a = a / b;           /* 4 */
```

This task can also be implemented using following single statement:

```
 a = ( a * b ) /  ( b = a );
```

Dry Run

Statement No.	*a*	*b*
1	10	20
2	200	20
3	200	10
4	20	10

Example 2.4: *To find the largest of three numbers among a, b, and c.*

Solution:

```
int a=10,b=30,c=25, big;
big = (a>b)?((a>c)?a:c)):((b>c)?b:c));
```

Here, if *a* is greater than *b*, expression ((*a*>*c*)?*a*:*c*) is evaluated; otherwise expression ((*b*>*c*)?*a*:*c*).

Expression ((*a*>*c*)?*a*:*c*) will return *a* if *a* is greater than *c* else returns *c*.

Expression ((*b*>*c*)?*b*:*c*) will return *b* if *b* is greater than *c* else returns *c*.

Example 2.5: *Write appropriate statements to find the sum of the right-most digit and left-most digit of a 4-digit number stored in n.*

Solution:

```
left_most_digit = n / 1000;
right_most_digit = n % 1000;
sum = left_most_digit + right_most_digit;
```

Example 2.6: *Write appropriate statements to convert temperature in Celsius scale into Fahrenheit scale.*

Solution:

The relationship between temperatures in Celsius scale and Fahrenheit scale is

$$\frac{C}{100} = \frac{F-32}{180} \Rightarrow F-32 = \frac{180}{100}C \Rightarrow F = 1.8C + 32$$

```
float c, f;
f = 1.8 * C + 32;
```

UNIT SUMMARY

In this chapter, we have learned that

- Operators are the verbs of a language that let the user perform computations on values.
- Operator can be unary or binary
- C language was the first language that is rich in operators.
- Operators are primarily divided into arithmetic operators, relational operators, logical operators, bit-wise operators, and special operators.
- Arithmetic operations can be classified as integer arithmetic, real arithmetic, and mixed mode arithmetic.
- Assignment operators are used to assign values to variables.
- An expression is a formula consisting of operands and operators linked together to compute a value.
- Expressions are primarily divided into arithmetic expressions, relational expressions, logical expressions, and conditional expressions.
- The process of converting values for one type to another type is known as type conversion.
- An implicit type conversion is automatically performed by the compiler without programmer's intervention.
- An explicit type conversion is user-defined that forces an operand or expression to be of specific type.
- The process of explicit type conversion is also known as type casting.
- Precedence is used to determine the order in which different operators are evaluated in an expression.
- Associativity is used to determine the order in which different operators with same precedence are evaluated in an expression.
- Library functions are pre-written functions to implement various routine type operations, and are provided as part of the compiler. To use them , you need to include appropriate header files in your program.

EXERCISE

Subjective Questions

1. What is an operator? Describe the usage of various arithmetic operators with suitable examples.
2. Among arithmetic operators, what are the broad categories of operators?
3. Is there any restriction on modulus operator '%'?
4. Describe the working of conditional operator with suitable example.
5. Explain the function of sizeof operator.
6. Describe the rules that govern the evaluation of algebraic expressions.
7. What do you mean by precedence of operators? What is the precedence among various arithmetic operators?
8. What do you mean by associativity of operators?

9. Summarize the rules that apply to expressions whose operands are of different type.
10. When should parentheses be used in arithmetic expressions? Give at least one example.
11. Are the library functions actually a part of the 'C' language? Explain.
12. Describe the working of conditional operator with suitable example.
13. What integer value is equivalent to Boolean false?
14. What is an expression?
15. Explain the working of increment and decrement operators.
16. What is type conversion?
17. Enumerate the situations where implicit type conversion takes place.
18. How explicit type casting is done? Give example.
19. Name the header file that you need to include in order to use mathematical library functions.
20. Write C expressions equivalent to the following algebraic expressions:

(*a*) $a - \frac{a}{a^{-x}} + b$ (*b*) $1 + \frac{1}{a + \frac{1}{a}} + b$ (*c*) $x + \frac{y^{-z}}{a + \frac{1}{a}} - \sqrt{x}$

(*d*) $x^3 - \frac{y^7}{-z} + e^x$ (*e*) $\left(\frac{1-x}{1+x}\right) / \left(\frac{1+y}{1-y}\right)$ (*f*) $\frac{1}{r} = \frac{1}{r_1} + \frac{1}{r_2}$

(*g*) $\sqrt{(x^2 + y^2 + z^2)^3}$ (*h*) $\frac{-b - \sqrt{b^2 - 4ac}}{2a}$

Multiple Choice Questions

1. Which of the following is not an arithmetic operator?
   (*a*) + (*b*) &
   (*c*) % (*d*) *
2. Precedence of operators determines which operator
   (*a*) is used first (*b*) is important
   (*c*) operates on largest numbers (*d*) executes fast
3. The operator % can be applied to
   (*a*) Float values (*b*) Double values
   (*c*) Integral values (*d*) All of the above
4. Expression *x* % *y* is equivalent to
   (*a*) (*x*–(*x*/*y*)) (*b*) (*x*–(*x*/*y*)**y*)
   (*c*) (*y*–(*y*/*x*)) (*d*) (*y*–(*x*/*y*)**x*)
5. Which of the following is not a bit-wise operator?
   (*a*) > (*b*) >>
   (*c*) ^ (*d*) &

6. What will be the value of expression 5 / 6 / 3 + 8 / 3?
   (*a*) 4 (*b*) 2
   (*c*) 2.333 (approx.) (*d*) None of above
7. In expression *x* / *y*, the value of *y* should be
   (*a*) integer (*b*) non-zero
   (*c*) non-negative (*d*) positive
8. If *x* is a variable of type *int*, which of the following is odd?
   (*a*) *x* << 1 (*b*) *x* = *x* * 2
   (*c*) *x* *= 2 (*d*) *x* <<= 1
9. Consider the following statement in C language

```
x = (a > b) ? ((a > c) ? a : c) : (b > c) ? b : c;
```

   What will be the value of *x* if *a* = 3, *b* = –5, *c* = 2?
   (*a*) 2 (*b*) 3
   (*c*) –5 (*d*) None of the above
10. When a relational expression is false, it has the value _____.
    (*a*) 1 (*b*) 0
    (*c*) –ve (*d*) +ve
11. If *a* = 15 then the statement *b* = *a* << 2; will result in
    (*a*) 30 (*b*) 60
    (*c*) 7 (*d*) None of the above
12. Consider the following code segment

```
int i, j = 10;
i = j++;
```

    The value of *i* and *j* at the end of this segment will be
    (*a*) 10, 10 (*b*) 10, 11
    (*c*) 11, 10 (*d*) 11, 11
13. Consider the following code segment

```
int x = 5.234, y;
y = sizeof(x);
```

    The value of *y* will be
    (*a*) 4 (*b*) 2
    (*c*) 8 (*d*) Syntax error
14. Which of the following operator can be used to perform division by 2 on an integral operand?
    (a) / (b) >>
    (c) << (d) Both (*a*) & (*b*)

15. Which of the following statement is not correct about the ++ operator?

(a) It is a unary operator
(b) Operand can come before or after operator
(c) It can be applied to an expression
(d) Its associates from right-to-left.

ANSWERS															
1.	(*b*)	2.	(*a*)	3.	(*c*)	4.	(*b*)	5.	(*a*)	6.	(*b*)	7.	(*b*)	8.	(*a*)
9.	(*b*)	10.	(*b*)	11.	(*b*)	12.	(*b*)	13.	(*b*)	14.	(*b*)	15.	(*c*)		

PRACTICALS

1. Write a program to find simple and compound interest.

Sol. The formulae to compute simple and the compound interest are

$$\text{Simple interest} = \frac{p \times r \times t}{100}, \text{ and Compound interest} = p\left[\left(1+\frac{r}{100}\right)^{t} - 1\right]$$

where p is the principle amount, r is the rate of interest per annum, and t is the time of deposit in years.

Listing 2.1

```
/* program to compute simple and compound interest */
#include <stdio.h>
#include <math.h>
int main()
{
   float p, r, si, ci;
   int t;
   printf("\nEnter principle amount : ");
   scanf("%f", &p);
   printf("Enter rate of interest : ");
   scanf("%f", &r);
   printf("Enter principle amount : ");
   scanf("%d", &t);
   si = (p*r*t)/100;
   ci = p*(pow(1+r/100,t)-1);
   printf("\nsimple interest = Rs. %.2f", si);
   printf("\ncompound interest = Rs. %.2f", ci);
   return 0;
}
```

Test Run

```
Enter principle amount : 1000
Enter rate of interest : 5
Enter principle amount : 4

Simple interest = Rs. 200.00
Compound interest = Rs. 215.51
```

2. Write a program to find the area of a triangle whose measure of three sides is given as *a*, *b*, and *c*, respectively. The values of a, b, and c must satisfy the condition that

$a + b > c$ and $b + c > a$ and $c + a > b$

Sol. The formulae to compute area of triangle is

$$s = \frac{a+b+c}{2}, \quad \text{Area} = \sqrt{s(s-a)(s-b)(s-c)}$$

Listing 2.2

```
/* program to compute area of a triangle whose sides are given */
#include <stdio.h>
#include <math.h>
int main()
{
   float a, b, c, s, area;
   printf("\nEnter value for a : ");
   scanf("%f", &a);
   printf("Enter value for b : ");
   scanf("%f", &b);
   printf("Enter value for c : ");
   scanf("%f", &c);
   s = (a+b+c)/2;
   area = sqrt(s*(s-a)*(s-b)*(s-c));
   printf("\nArea of triangle = %.2f Sq. Units", area);
   return 0;
}
```

Test Run

```
Enter value for a : 5
Enter value for b : 7
Enter value for c : 6
Area of triangle = 14.70 Sq. Units
```

3. A building has 10 floors with a floor height of 3 meters each. A ball is dropped from the top of the building. Write a program to find the time taken by the ball to reach each floor.

Sol. Since height of each floor is 3 meters, therefore, height (h) of building with 10 floors is 30 meters.

Here we use equation of motion,

$$s = ut + \frac{1}{2}at^2$$

Here s is replaced by h, and a by $g(9.8\ m/s^2)$.

Since ball is dropped from the top (10^{th}) floor, $u = 0$. Therefore, formula reduces to

$$h = \frac{1}{2}gt^2 \qquad \Rightarrow \qquad t = \sqrt{\frac{2h}{g}}$$

Listing 2.3

```
/* program to compute the time taken by ball to reach
  the ground floor, when it is released from the top floor
*/
#include <stdio.h>
#include <math.h>
int main()
{
   float h = 30, g = 9.8, t;
   t = sqrt(2*h/g);
   printf("\nTime taken by ball = %.2f seconds", t);
   return 0;
}
```

Test Run

```
Time taken by ball = 2.47 seconds
```

KNOW MORE

Arithmetic expressions are backbone of programs for Science & Engineering related problems. Such problems involve complex mathematical and algebraic expressions. Therefore, for converting such expressions into C expressions, one should know the precedence and associativity of operators, types of arithmetic, and the kind of type conversion that takes place during the evaluation of expressions.

The teacher is expected to develop an understanding about formations and evaluation of expression and other related issues.

The teacher should also demonstrate the formation and evaluation of expression of various types with active participation of the students.

REFERENCES & SUGGESTED READINGS

1. R. S. Salaria, Problem Solving & Programming in C, Khanna Book Publishing Co(P) Ltd., New Delhi.
2. E. Balagurusamy, Programming in ANSI C, Tata McGraw Hill, New Delhi.
3. Yashavant Kanetkar, Let Us C, BPB Publications, New Delhi.
4. Byron Gottfried, Programming with C, Schaum's Outlines.
5. https://onlinecourses.nptel.ac.in/noc21_cs01/preview
6. https://ocw.mit.edu/courses/intro-programming/
7. https://www.programiz.com/c-programming
8. https://www.javatpoint.com/c-programming-language-tutorial

3 Conditional Branching and Loops

UNIT SPECIFICS

This unit discusses the topics related to various control statements. These control statements allow the programmer to alter the sequence of statements or control the repetition of statements in a program. The various control statements explained in this unit include if, if-else, if-else if ladder, switch, for, while, do-while, break, and continue statements. There use is demonstrated with suitable examples.

RATIONALE

A computer program is a set of instructions for a computer. These instructions are executed sequentially, i.e., one after the other as they appear in a program. This order of execution of statement works well for simple problems where neither any decision making process nor any repetition of instructions is involved.

However, in practice, it is often required to change the order of execution of instructions or repeat a group of instructions for known number of times or until certain conditions are satisfied. In such cases, the order in which these statements will be executed needs to be controlled.

This unit helps the students to understand syntax and working of various statements that can be used for implementing conditional branching and looping to solve real-life problems.

PRE-REQUISITES

- Relational operators
- Logical operators
- Increment/decrement operators
- Evaluation of relational/logical expressions

UNIT OUTCOMES

Upon completion of the unit, students will be able to

U3-O1: explain the need for conditional branching and looping

U3-O2: select the correct type of conditional branching statement based on a given problem

U3-O3: select the correct type of loop based on a given problem

U3-O4: use break and continue keywords

U3-O5: create programs to solve real-life problems using conditional branching and loops

Unit 3 Outcomes	EXPECTED MAPPING WITH COURSE OUTCOMES (1- Weak Correlation; 2- Medium correlation; 3- Strong Correlation)							
	CO-1	CO-2	CO-3	CO-4	CO-5	CO-6	CO-7	CO-8
U3-O1	1	-	-	-	-	-	-	-
U3-O2	-	-	-	2	-	-	-	-
U3-O3	-	-	-	2	-	-	-	-
U3-O4	-	1	-	2	-	-	-	-
U3-O5	-	-	-	3	-	-	-	-

3.1 INTRODUCTION

Changing the order of execution of the statements, *i.e.,* changing the flow of control, might involve one of the following situations:

- *Branching* – executing one group of instructions depending on the outcome of a decision.
- *Looping* – executing a group of instructions repetitively either for a given number of times or until the required condition is met.
- *Jumping* – transferring control from one point to another point in the same program unit.

The C Language provides facilities for controlling the order of execution of the statements, which are referred to as *flow control statements* or simply as *control statements*.

The various flow control statements are clubbed in the following categories:

1. *Conditional Branching* – In this category, C language provides the following statements
   - *if* statement
   - *if – else* statement
   - *else if* ladder
   - *switch* statement
2. *Looping* – In this category, C language provides the following statements
   - *for* statement
   - *while* statement
   - *do – while* statement
3. *Jumping* – In this category, C language provides the following statements
   - *break* statement
   - *continue* statement

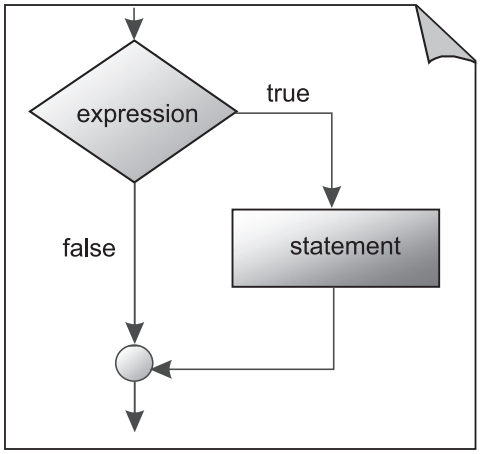
In this unit, we discuss about these statements, and the discussion is supported by plenty of solved programs.

3.2 CONDITIONAL BRANCHING

The various statements under this category allow the execution of selective statements based on certain decision criteria. We will see their working supported by well-designed illustrative examples.

3.2.1 The *if* Statement

The syntax of *if* statement is



(*a*) Logic flow control of *if* statement

```
if (expression)
{
    statement
}
```

(*b*) Code of *if* statement

Fig. 3.1: Logic flow control and code of *if* statement

The *expression* may represent a relational expression, a logical expression, a numeric variable or a numeric constant. The specified expression may be a simple expression or compound expression.

Expressions in C evaluate to 0 or 1. If expression evaluates to value 1, then the statements in the statement-block are executed; otherwise they are bypassed.

Example 3.1: *Program to check whether given integer number is negative or positive.*

Listing 3.1

```
#include<stdio.h>
int main()
{
   int n;
   printf( "\nEnter integer number : " );
   scanf( "%d", &n );

   if ( n < 0 ) {
      printf( "\nNumber is negative.\n" );
      return 0;
   }
   printf( "\nNumber is positive.\n" );
   return 0;
}
```

Test Runs

First Run

```
Enter integer number : -25
Number is negative.
```

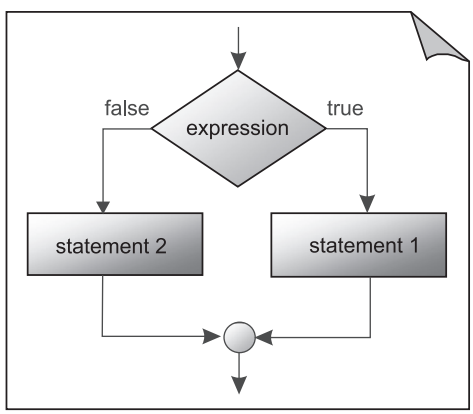
Second Run

```
Enter integer number : 30
Number is positive.
```

3.2.2 The *if – else* Statement

The *if* statement executes a single statement, (simple or compound), when the specified expression evaluates to a *non-zero* value. It does nothing when it evaluates to a *zero* value. *Is there any way whereby one statement is executed if the expression evaluates to a non-zero value and another statement if the expression evaluates to a zero value*? The answer is yes.

This objective is achieved by using *if - else* statement whose syntax is



(a) Logic flow control of *if - else* statement

```
if (expression)
{
    statement-1
}
else
{
    statement-2
}
```

(b) Code of *if - else* statement

Fig. 3.2: Logic flow control and code of *if-else* statement

If *expression* evaluates to 1, *statement-1* is executed and *statement-2* is bypassed. However, if the *expression* evaluates to 0, *statement-1* is bypassed and *statement-2* is executed.

Example 3.2: *Program to check whether the natural number is even or odd.*

Listing 3.2

```
#include<stdio.h>
int main()
{
   int n;
   printf( "\nEnter any natural number : " );
   scanf( "%d", &n );
```

```
    if ( n % 2 == 0 )
       printf( "\nNumber entered is EVEN\n" );
    else
       printf( "\nNumber entered is ODD\n" );
    return 0;
}
```

Test Runs

First Run

```
Enter any whole number: 22
Number entered is EVEN
```

Second Run

```
Enter any number: 15
Number entered is ODD
```

Example 3.3: *Program to find the larger of two numbers.*

Listing 3.3

```
include<stdio.h>
int main()
{
    int a, b;
    printf( "\nEnter value for a : " );
    scanf( "%d", &a );
    printf( "\nEnter value for b : " );
    scanf( "%d", &b );
    if ( a > b )
       printf( "\n%d is the larger.\n", a );
    else
       printf( "\n%d is the larger.\n", b );
    return 0;
}
```

Test Runs

First Run

```
Enter value for a : 20
Enter value for b : 10
20 is the larger.
```

Second Run

```
Enter value for a : 15
Enter value for b : 25
25 is the larger.
```

3.2.3 Nested *if* and *if – else* Statements

If statements can be nested, *i.e.*, an *if* statement can be contained within another *if* statement. The inner *if* statement will be executed if expression of outer *if* statement evaluates to *non-zero* value.

Likewise, *if-else* statements can also be nested. Here, *if* statement can be nested in *if* part *or else* part. Likewise, *if - else* statement can be nested in *if* part *or else* part.

Example 3.4: Write a program to determine whether the given year is a leap year. A given year is a *leap year* if any one of the following conditions is satisfied:

1. Year is evenly divisible by 4 and not divisible by 100.
2. Year is evenly divisible by 4, evenly divisible by 100 as well as evenly divisible by 400.

In all other cases, given year is not a leap year.

For example,

1. The years 1988, 1992, and 1996 are leap years, as they are divisible by 4 but not by 100.
2. The years 1700, 2100, and 2500 are not leap years as they are divisible by 4, by 100 but not by 400.
3. The years 1600, 2000, and 2400 are leap years, as they are divisible by 100 as well as by 400.

Listing 3.4

```
#include <stdio.h>
int main()
{
   int year;
   printf("Enter given year : ");
   scanf("%d", &year);
   if ( year % 4 == 0 ) {
      if ( year % 100 == 0 ) {
          if ( year % 400 == 0 )
             printf("\nYear %d is a leap year.\n", year);
          else
             printf("\nYear %d is not a leap year.\n", year);
      }
      else
      {
         printf("\nYear %d is a leap year.\n", year);
      }
   }
   else
   {
      printf("\nYear %d is not a leap year.\n", year);
   }
   return 0;
}
```

Test Runs

First Run

```
Enter any year : 2000
Year 2000 is a leap year.
```

Second Run

```
Enter any year : 2008
Year 2008 is a leap year.
```

Third Run

```
Enter any year : 2006
Year 2006 is not a leap year.
```

3.2.4 The *if-else if* Ladder

The *if-else-if* ladder is an extension to the *if-else* statement. It is used in a scenario where there are multiple cases to be performed for different conditions.

The syntax of *if-else-if* ladder is shown in Fig. 3.3.

The expressions are evaluated in order, and if any expression is true then the statement block associated with it is executed, and this terminates the whole chain.

The last *else* part handles *none of the above* or default case where none of the specified expressions are satisfied.

```
if ( expression-1 )
   statement-1
else if ( expression-2 )
   statement-2
else if ( expression-3 )
   statement-3
        :
else if ( expression-n )
   statement-n
else
   statement-s
```

Fig. 3.3: Code of *if - else if* ladder

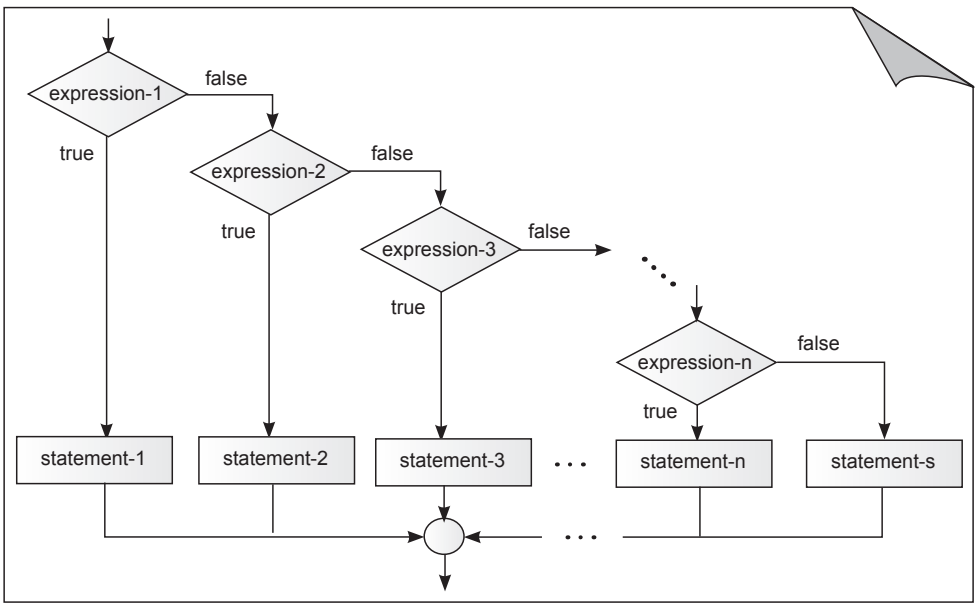


Fig. 3.4: Logic flow control of *if - else if* ladder

Example 3.5: *Given percentage of marks. The grade of student is computed as per following policy.*

Percentage of Marks	Grade
percentage ≥ 90	A
90 > percentage ≥ 75	B
75 > percentage ≥ 60	C
60 > percentage ≥ 50	D
percentage < 50	F

Write a program to print the grade of students when their percentage of marks is given.

Listing 3.5

```
#include<stdio.h>
int main()
{
   float percentage;
   printf("\nEnter percentage of marks : ");
   scanf("%f", &percentage);
   if ( percentage >= 90 )
      printf("\nGrade is A\n");
   else if ( percentage >= 75 )
      printf("\nGrade is B\n");
   else if ( percentage >= 60 )
      printf("\nGrade is C\n");
   else if ( percentage >= 50 )
      printf("\nGrade is D\n");
   else
      printf("\nGrade is E\n");
   return 0;
}
```

Test Run

```
Enter percentage of marks : 93
Grade is A
```

3.2.5 The *switch* Statement

The *switch* statement provides an alternative to multiple cases to be performed for different values of a variable/expression.

The syntax of *switch* statement is shown in Fig. 3.5.

If *expression* takes any value from *val-1*, *val-2*, *val-3*, ..., *val-n*, the control is transferred to that appropriate case. In each case, the statements are executed and then the *break* statement transfers the control out of *switch* statement.

If no *break* statement is used following a case, except the last one in the absence of *default* keyword, the control will fall through to the next case. If the value of the *expression* does not match any of the case values, control goes to the *default* keyword, which is usually at the end of the *switch* statement. The use of the *default* keyword can be of a great convenience. If there is no *default* keyword, the whole *switch* statement simply skipped when there is no match.

```
switch ( expression )
{
   case val-1 :
                statement-1
                break;
   case val-2 :
                statement-2
                break;
       :
   case val-n :
                statement-n
                break;
   default    :
                statement-d
}
```

Fig. 3.5: Code of *switch* statement

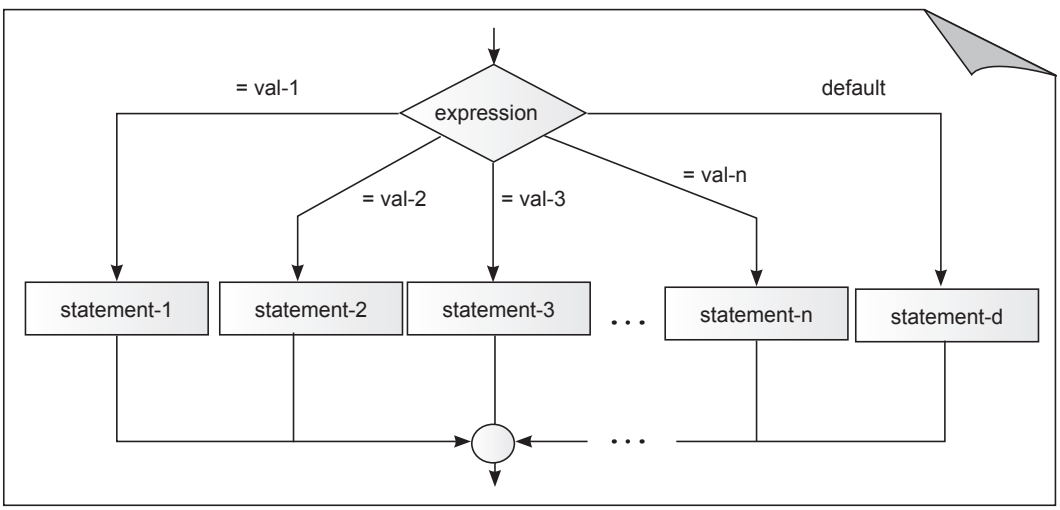


Fig. 3.6: Logic flow control of *switch* statement

Example 3.6: *Write a program to implement four function calculator.*

Listing 3.6

```
#include<stdio.h>
int main()
{
   int a, b;
   char op;
```

```
    printf("\nEnter expression such as 5 + 3 : ");
    scanf("%d %c %d", &a, &op, &b );
    switch ( op )
    {
       case '+' : printf("\n%d %c %d = %d\n", a, op, b, a+b);
                  break;
       case '-' : printf("\n%d %c %d = %d\n", a, op, b, a-b);
                  break;
       case '*' : printf("\n%d %c %d = %d\n", a, op, b, a*b);
                  break;
       case '/' : printf("\n%d %c %d = %d\n", a, op, b, a/b);
                  break;
       default : printf("\nWrong input\n");
    }
    return 0;
}
```

Test Run

```
Enter expression such as 5 + 3 : 20 / 2
20 / 2 = 10
```

There may be situations, when we want that same statement should be executed for more than one value of the *expression*. To handle such situation, we code these *cases* one after the other, and followed by the statement, as shown below:

```
switch ( expression )
{
      :
   case val-4 :
   case val-5 :
   case val-6 : statement;
                break;
      :
}
```

The *statement-block* will be executed whenever the value of the *expression* is *val-4*, *val-5*, or *val-6*.

Example 3.7: *Program to test whether an alphabet is vowel or consonant.*

Listing 3.7

```
#include<stdio.h>
int main()
{
```

```
    char ch;
    printf("\nEnter an alphabet : ");
    ch = getche();
    switch ( ch )
    {
       case 'a' :
       case 'A' :
       case 'e' :
       case 'E' :
       case 'i' :
       case 'I' :
       case 'o' :
       case 'O' :
       case 'u' :
       case 'U' : printf("\nAlphabet is vowel.\n");
                  break;
       default  :
                  printf("\nAlphabet is consonant.\n");
    }
    return 0;
}
```

Test Run

```
Enter an alphabet : A
Alphabet is vowel.
```

ILLUSTRATIVE EXAMPLES FOR CONDITIONAL BRANCHING

Example 3.8: Program to find roots of a quadratic equation of type

$$ax^2 + bx + c = 0 \text{ provided } a \neq 0.$$

Solution: The roots of the quadratic equation are given by the formula:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where expression $b^2 - 4ac$ is known as the discriminant.

Depending on the sign of the discriminant, there are three possibilities for the roots:

1. If $b^2 - 4ac < 0$, then the root are imaginary, and we can compute real part and imaginary part separately as

   real part $= -\frac{b}{2a}$ imaginary part $= \frac{\sqrt{-(b^2 - 4ac)}}{2a}$

2. If $b^2 - 4ac = 0$, then the roots are real and equal, and root is simply computed as

$$\text{root} = -\frac{b}{2a}$$

3. If $b^2 - 4ac > 0$, then the roots are real and distinct and are computed as

$$\text{root one} = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \qquad \text{root two} = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Listing 3.8

```
#include<stdio.h>
#include<math.h>
int main()
{
   float a, b, c, disc, root1, root2;
   float realPart, imagPart;
   printf( "\nEnter values of a,b,c: " );
   scanf( "%f,%f,%f", &a, &b, &c);
   if ( a == 0 ) {
      printf( "\nIt is not an quadratic equation\n" );
      return 0;
   }
   disc = b * b - 4.0 * a * c;
   if ( disc < 0 )
   {
      realPart = - b / (2*a);
      imagPart = sqrt((double)-disc) / (2*a);
      printf( "\nRoots are imaginary" );
      printf( "\nReal part = %.3f", realPart );
      printf( "\nImaginary part = %.3f\n", imagPart );
   }
   else if ( disc == 0 )
   {
      root1 = - b / (2*a);
      printf( "\nRoots are real and equal" );
      printf( "\nEach root = %.3f\n", root1 );
   }
   else
   {
      root1 = (-b-sqrt((double)disc )) / (2*a);
      root2 = (-b+sqrt((double)disc )) / (2*a);
      printf( "\nRoots are real and distinct" );
      printf( "\nRoot 1 = %.3f", root1 );
      printf( "\nRoot 2 = %.3f\n", root2 );
   }
   return 0;
}
```

Test Run

```
Enter values of a,b,c: 2,3,5
Roots are imaginary
Real part = -0.750
Imaginary part = 1.392
```

Example 3.9: *Suppose income tax for individuals is computed on slab rates as follows:*

Income	Tax Payable
Upto ₹ 1,00,000/-	Nil
From ₹ 1,00,001/- to ₹ 2,00,000/-	10% of the excess over ₹ 1,00,000/-
From ₹ 2,00,001/- to ₹ 3,00,000/-	20% of the excess over ₹ 2,00,000/-
Above ₹ 3,00,000/-	30% of the excess over ₹ 3,00,000/-

Write a program that reads the income and prints the income tax due.

Solution: Note the following:

- If the income is upto ₹ 1,00,000/- only, then the tax is *nil*.
- If the income is in the range of ₹ 1,00,001/- to ₹ 2,00,000/-, then the tax is 10% of the amount in excess of ₹ 1,00,000/-.
- If the income is in the range of ₹ 2,00,001/- to ₹ 3,00,000/-, then the tax is 10% of ₹ 1,00,000/- (₹ 10,000/-) for a slab of ₹ 1,00,001 to ₹ 2,00,000/- plus 20% of the amount in excess of ₹ 2,00,000/-.
- If the income is more than ₹ 3,00,000/-, then the tax is 10% of ₹ 1,00,000/- (₹ 10,000/-) for a slab of ₹1,00,001 to ₹ 2,00,000/- plus 20% of ₹ 1,00,000/- (₹ 20,000/-) for a slab of ₹ 2,00,001 to ₹ 3,00,000/- plus 30% of the amount in excess of ₹ 3,00,000/-.

Let us have hands on by computing the tax if the given income is

(*a*) ₹ 2,50,000/- (*b*) ₹ 1,75,000/- (*c*) ₹ 3,20,000/-

(*a*) *Tax Computations for income of ₹ 2,50,000/-*

For first ₹ 1,00,000/-	Tax is Nil
For next ₹ 1,00,000/-	Tax is 10,000/- (@ 10%)
For next ₹ 50,000/-	Tax is 10,000/- (@ 20%)
Total Tax due	**₹ 20,000/-**

(*b*) *Tax Computations for income of ₹ 1,75,000/-*

For first ₹ 1,00,000/-	Tax is Nil
For next ₹ 75,000/-	Tax is 7,500/- (@ 10%)
Total Tax due	**₹ 7,500/-**

(*c*) *Tax Computations for income of ₹ 3,20,000/-*

For first ₹ 1,00,000/-	Tax is Nil
For next ₹ 1,00,000/-	Tax is 10,000/- (@ 10%)
For next ₹ 1,00,000/-	Tax is 20,000/- (@ 20%)
For next ₹ 20,000/-	Tax is 6,000/- (@ 30%)
Total Tax due	**₹ 36,000/-**

Listing 3.9

```
#include<stdio.h>
void main()
{
   float income, tax;
   printf( "\nEnter gross income (in Rs.): " );
   scanf( "%f", &income );
   if ( income <= 100000.0 )
      tax = 0;
   else if ( income <= 200000.0 )
      tax = ( income - 100000.0 ) * 0.1;
   else if ( income <= 300000.0 )
      tax = 10000 + ( income - 200000.0 ) * 0.2;
   else
      tax = 30000 + ( income - 300000.0 ) * 0.3;
   printf( "\nTax due = Rs. %.2f\n", tax );
}
```

Test Run

```
Enter gross income : 250000
Tax due = Rs. 20000.00
```

Example 3.10: *Write a program that accepts day of week as number and prints name of corresponding day, i.e., 0 for Sunday, 1 for Monday, ..., 6 for Saturday.*

Listing 3.10

```
#include <stdio.h>
int main()
{
   int day;
   printf("\nEnter day of week as number ( 0 - 6 ) : ");
   scanf("%d", &day);
   switch ( day )
   {
      case 0  : printf("\nDay of week is Sunday." );
                break;
      case 1  : printf("\nDay of week is Monday." );
                break;
      case 2  : printf("\nDay of week is Tuesday." );
                break;
      case 3  : printf("\nDay of week is Wednesday." );
                break;
```

```
      case 4  : printf("\nDay of week is Thursday." );
                break;
      case 5  : printf("\nDay of week is Friday." );
                break;
      case 6  : printf("\nDay of week is Saturday." );
                break;
      default : printf("\nWrong input" );
   }
   printf("\n");
   return 0;
}
```

Test Run

```
Enter day of week as number ( 0 - 6 ) : 2
Day of week is Tuesday
```

Example 3.11: *Write a program to test whether the natural number is even or odd without using modulus by 2 operation and if statement.*

Listing 3.11

```
#include <stdio.h>
int main()
{
   int n;
   printf("\nEnter any natural number : ");
   scanf("%d", &n);
   switch ( n%10 )
   {
      case 0  :
      case 2  :
      case 4  :
      case 6  :
      case 8  : printf("\n%d is even.\n");
                break;
      default : printf("\n%d is even.\n");
   }
   return 0;
}
```

Test Runs

First Run

```
Enter any natural number : 120
120 is even.
```

Second Run

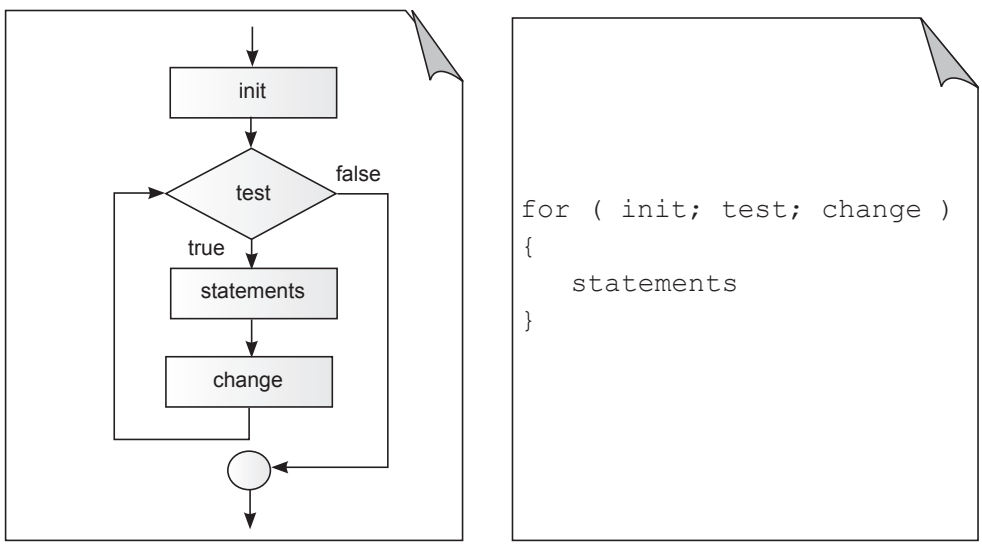
```
Enter any natural number : 75
75 is odd.
```

3.3 LOOPING

The various statements under this category allow the repetitive execution of group of statements either for a given number of times or till certain conditions are met. We will see their working supported by well-designed illustrative examples.

3.3.1 The *for* Statement

The *for* statement is suited for problems where the number of times a statement or statement-block will be executed is known in advance.



(a) Logic flow control of *for* statement

(b) Code of *for* statement

Fig. 3.7: Logic flow control and code of *for* statement

where *init* is an expression to initialize the counter, *test* is an expression to see when to stop iterating, and *change* is an expression to change the counter for each pass of the loop. The *init* and *change* parts can have more than one statement separated by a comma.

To demonstrate the use of *for* statement, let us develop some program making using of *for* statement.

Example 3.12: *Write a program to find the sum of first n natural numbers.*

Listing 3.12

```
#include<stdio.h>
int main(void)
{
   int i, n, sum = 0;
   printf("\nEnter value for n : " );
   scanf("%d", &n);
   for ( i = 1; i <= n; i++ ) {
      sum = sum + i;
   }
   printf("\nSum of first %d natural numbers = %d\n", n, sum );
   return 0;
}
```

Test Run

```
Enter value for n : 10
Sum of first 10 natural numbers = 55
```

Example 3.13: *Write a program to find the factorial of a natural number n.*

Listing 3.13

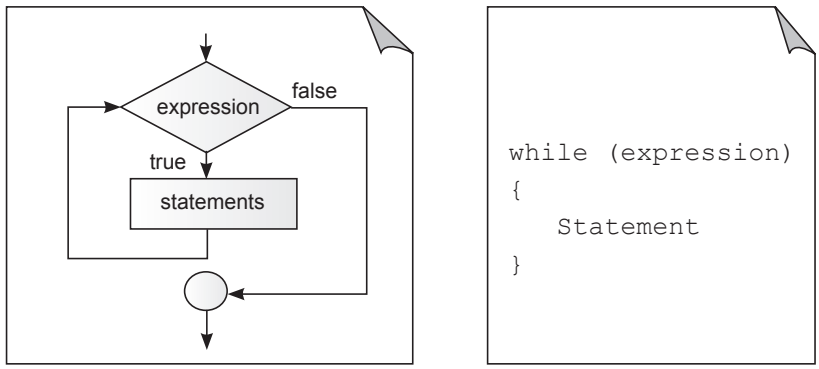
```
#include<stdio.h>
int main(void)
{
   int i, n, prod = 1;
   printf("\nEnter value for n : " );
   scanf("%d", &n);
   for ( i = 1; i <= n; i++ ) {
      prod = prod * i;
   }
   printf("\nFactorial %d = %d\n", n, prod );
   return 0;
}
```

Test Run

```
Enter value for n : 5
Factorial 5 = 120
```

3.3.2 The *while* Statement

The *while* statement is suited for problems where it is not known in advance that how many times a statement or a statement-block will be executed.



(a) Logic flow control of *while* Statement

(b) Code of *while* statement

Fig. 3.8: Logic flow control and code of *while* statement

Here *expression* can be a constant, a variable or an expression. The *statement* is executed repeatedly till the *expression* evaluates to 1. Whenever *expression* evaluates to 0, the execution of *while* statement will terminate and control will pass to a statement immediately following it.

The following points must be kept in mind while using *while* statement:

- There must be a statement prior to *while* statement that initializes the *expression*.
- In the statement-block, there must be a statement that modifies the *expression*.

To demonstrate the use of *while* statement, let us develop some program making using of *while* statement.

Example 3.14: *Write a program to find the sum of digits of a natural number n.*

Listing 3.14

```
#include <stdio.h>
int main()
{
   int i, n, sum = 0, temp, digit;
   printf("\nEnter any natural number : " );
   scanf("%d", &n);
   temp = n;
   while ( temp > 0 )
   {
      digit = temp % 10;
      sum = sum + digit;
      temp = temp / 10;
   }
   printf( "\nSum of digits of %d = %d\n", n, sum );
   return 0;
}
```

Test Run

```
Enter any natural number : 2375
Sum of digits of 2375 = 17
```

Example 3.15: *Write a program to count number of characters, vowels, and words in a paragraph.*

Listing 3.15

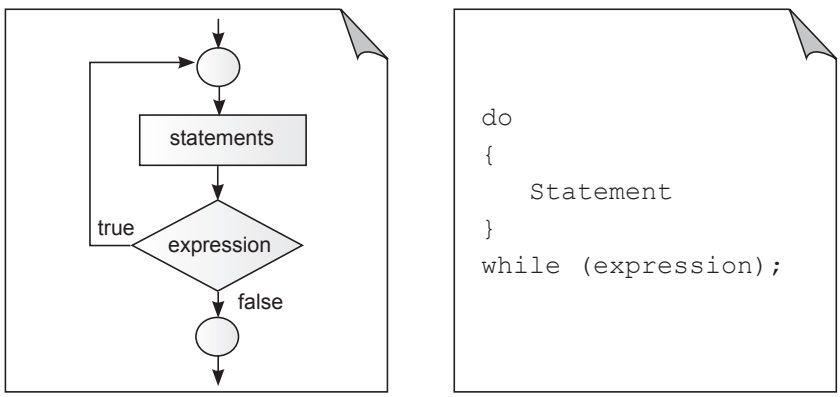
```
#include <stdio.h>
int main()
{
   int vowelCount = 0,characterCount = 0,wordCount = 0;
   char ch;
   printf( "Type in the paragraph and terminate by ENTER key\n\n" );
   while ( ( ch = getche() ) != '\r' )
   {
      characterCount++;
      switch ( ch )
      {
         case ' '  :
         case '\t' :
                     wordCount++;
                     break;
         case 'a'  :
         case 'A'  :
         case 'e'  :
         case 'E'  :
         case 'i'  :
         case 'I'  :
         case 'o'  :
         case 'O'  :
         case 'u'  :
         case 'U'  :
                     vowelCount++;
                     break;
      }
   }
   wordCount++;
   printf( "\nCharacter count = %d", characterCount);
   printf( "\nVowel count = %d", vowelCount);
   printf( "\nWord count = %d\n", wordCount);
   return 0;
}
```

Test Run

```
Type in the paragraph and terminate by ENTER key
Programming is the way to instruct computer to do a particular task.
Character count = 68
Vowel count = 20
Word count = 12
```

3.3.3 The *do* – *while* Statement

The *do-while* statement, like *while* statement, is also suited for problems where it is not known in advance that how many times a statement will be executed.



(a) Logic flow control of *do-while* statement

(b) Code of *do-while* statement

Fig. 3.9: Logic flow control and code of *do-while* statement

where *expression* is a constant, a variable or an expression. The *statement* is executed repeatedly till the *expression* evaluates to 1.

Example 3.16: *Write a program to compute the average of unknown numbers.*

Listing 3.16

```
#include<stdio.h>
int main()
{
   int n = 0;
   char yesno;
   float number, sum = 0, average;
   do {
      printf( "\nEnter number %d: ", n+1 );
      scanf( "%f", &number );
      n++;
```

```
        sum += number;
        printf( "\nAny more number [yn]?: " );
        yesno = getchar();
    }
    while ( ( yesno == 'y' ) || ( yesno == 'Y' )  );
    average = sum / n;
    printf( "\n\nAverage of given numbers = %.2f\n", average );
    return 0;
}
```

Test Run

```
Enter number 1: 2.5
Any more number [yn]?: y
Enter number 2: 12.0
Any more number [yn]?: y
Enter number 3: 10.25
Any more number [yn]?: y
Enter number 4: 8.75
Any more number [yn]?: y
Enter number 5: 11.0
Any more number [yn]?: n
Average of given numbers = 8.90
```

3.3.4 Nested *while*, *for* and *do* – *while* Statements

Just as *if* statements can be nested, these statements can also be nested. The inner loop is executed from the beginning for each iteration of the outer loop.

The following sections of code show the nesting of looping statements within their own types.

```
for(i=0;i<m;i++)
{
   :
   for(j=0;j<n;j++)
   {
      :

   }
}
```

```
i=0;
while(i<m)
{
   :
   j=0;
   while(j<n)
   {
      :
     j++;
   }
   i++;
}
```

```
i=0;
do
{
   :
   j=0;
   do
   {
      :
     j++;
   } while(j<n);
   i++;
} while(i<m);
```

In general, it is possible to nest while and do-while statements inside *for* statement, *for* statement inside the *while* and *do-while* statements, *i.e.*, all sort of nesting combinations are permitted.

To demonstrate the use of nested loops, let us develop a few programs using nested *for* loops.

Example 3.17: *Write a program to print the following pattern*

```
1
2 2
3 3 3
```

Listing 3.17

```
#include <stdio.h>
int main()
{
    int i, j;
    printf("\n");
    for ( i = i; i <= 3; i++ )
    {
        for ( j = 1; j <= i; j++ )
        {
            printf("%4d", i);
        }
        printf("\n");
    }
    return 0;
}
```

Example 3.18: *Write a program to print the following pattern*

```
3 3 3
2 2
1
```

Listing 3.18

```
#include <stdio.h>
int main()
{
    int i, j;
    printf("\n");
    for ( i = 3; i >= 1; i-- )
    {
        for ( j = 1; j <= i; j++ )
        {
            printf("%4d", i);
```

```
        }
        printf("\n");
    }
    return 0;
}
```

Example 3.19: *Write a program to print the following pattern*

```
1
1  2
1  2  3
```

Listing 3.19

```
#include <stdio.h>
int main()
{
    int i, j;
    printf("\n");
    for ( i = i; i <=3; i++ )
    {
        for ( j = 1; j <= i; j++ )
        {
            printf("%4d", j);
        }
        printf("\n");
    }
    return 0;
}
```

Example 3.20: *Write a program to print the following pattern*

```
*
* *
* * *
```

Listing 3.20

```
#include <stdio.h>
int main()
{
    int i, j;
    printf("\n");
    for ( i = i; i <= 3; i++ )
    {
        for ( j = 1; j <= i; j++ )
```

```
        {
            printf("    *");
        }
        printf("\n");
    }
    return 0;
}
```

3.4 JUMPING STATEMENTS

These statements transfer the control from one part of the program to another part. In this section, we will see their working.

3.4.1 The *break* Statement

The *break* statement is always used inside the body of the *switch* statement, and looping statements.

```
switch ( expression )
{
    case val-1 : statement-1
                 break;
    case val-2 : statement-2
                 break;
    case val-3 : statement-3
                 break;
       :
       :
    case val-n : statement-n
                 break;
    default    :
                 statement-d
}
```

Fig. 3.10: Action of *break* statement in *switch* statement

In *switch* statement, it is used as the last statement of every case except the last one. When executed, it transfers the control out of *switch* statement and the execution of the program continues from the statement following *switch* statement.

In *for*, *while* and *do-while* statements, it is always used in conjunction with *if* statement. Note that *break* never used with *if* statement if it is not a part of the body of the looping statement. When executed, it transfers the control out of looping statement and the execution of the program continues from the statement following looping statement.

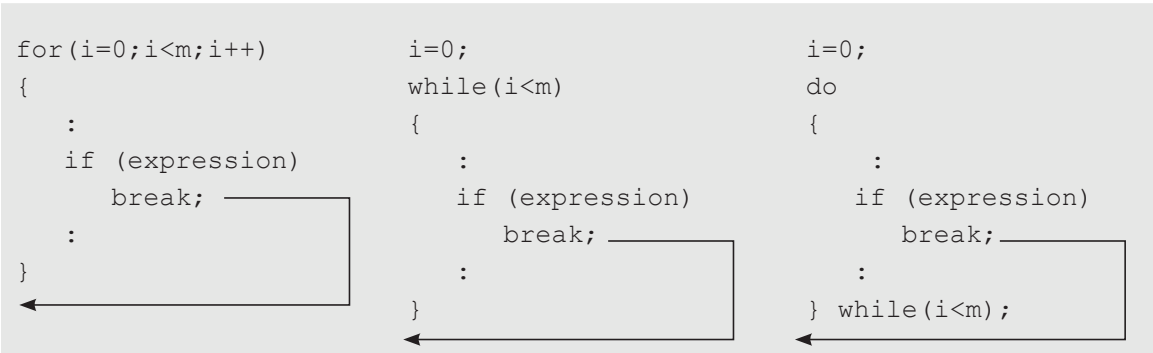


Fig. 3.11: Action of *break* statement in *for, while* and *do-while* statements

In simple terms, we can say that *break* when executed terminates the execution of the loop.

Example 3.21: *Write a program to demonstrate the use of break statement in a loop.*

Listing 3.21

```
#include <stdio.h>
int main()
{
   int i;
   for ( i = 1; i < 10; i++ )
   {
       if ( i % 5 == 0 )
           break;
       printf("%d ", i);
   }
   return 0;
}
```

Test Run

```
1 2 3 4
```

The use of *break* statement in conjunction with *if* statement terminates the *for* loop for value of *i* that is divisible by 5.

3.4.2 The *continue* Statement

The *continue* statement is always used inside the body of the looping statements. There may be a situation where we want that from a given statement onward, the rest of the statements up to the last statement of the loop should be skipped. This task is accomplished by using *continue* statement.

The *continue* statement transfers the control to the beginning of the next iteration of the loop thus bypassing the statements which are not yet executed. Note that the *continue* statement is always used in conjunction with the *if* statement.

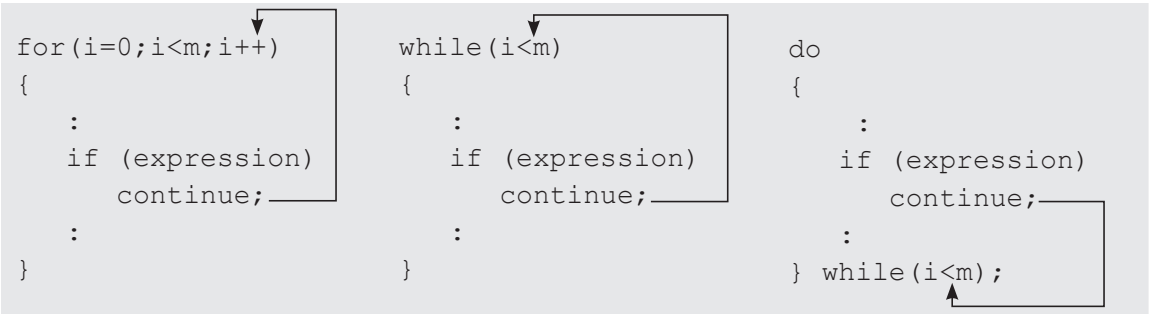


Fig. 3.12: Action of *continue* statement in *for, while* and *do-while* statements

In simple terms, we can say that the continue statement when executed terminates the current iteration of the loop.

Example 3.22: *Write a program to demonstrate the use of continue statement in a loop.*

Listing 3.22

```
#include <stdio.h>
int main()
{
   int i;
   for ( i = 1; i < 10; i++ )
   {
       if ( i % 5 == 0 )
           continue;
       printf("%d ", i);
   }
   return 0;
}
```

Test Run

```
1 2 3 4 6 7 8 9
```

The use of *continue* statement in conjunction with *if* statement skips the rest of the statements of the *for* loop for values of *i* that is divisible by 5.

ILLUSTRATIVE EXAMPLES FOR LOOPS

Example 3.23: *Write a program to print a multiplication table for a given number and the number of rows in the table.*

For example, for a number 5 and rows = 3, the output should be:

$5 \times 1 = 5$

$5 \times 2 = 10$

$5 \times 3 = 15$

Listing 3.23

```
#include <stdio.h>
int main()
{
   int num, rows, i, j;
   printf( "\nEnter number whose table to print : " );
   scanf( "%d", &num );
   printf( "\nEnter rows to print : " );
   scanf( "%d", &rows );
   for ( i = 1; i <= rows; i++ )
   {
       printf("\n%d x %d = %d", num, i, num*i) ;
   }
   return 0;
}
```

Test Run

```
Enter number whose table to print : 5
Enter rows to print : 3
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
```

Example 3.24: *Write a program to find whether the given natural number n is a prime number or not. A natural number is said to be prime if it is divisible by 1 and itself only, i.e., it cannot be factorized. In addition, to this definition, an even number except 2, is not a prime number.*

Therefore, our test criteria becomes

1. If n is greater than 2 and is even then n is not a prime number.
2. If test at step 1 fails, then we try to divide number n by factors $k = 3, 5, 7, \ldots \sqrt{n}$. Therefore, if n is divisible by any value of k, number n is not a prime number.
3. If test at step 2 also fails, then n is a prime number.

The following program implements this criterion.

Listing 3.24

```
#include<stdio.h>
#include<math.h>
int main()
{
   int n, k, m;
   printf( "\nEnter a positive integer number: " );
   scanf("%d", &n);
```

```
    if ( ( n > 2 ) && ( ( n % 2 ) == 0 ) )
    {
       printf( "\n%d is not a prime number.\n", n );
       return 0;
    }
    m = sqrt( n );
    for ( k = 3; k <= m; k += 2 )
    {
        if ( n % k == 0 )
        {
            printf( "\n%d is not a prime number.\n", n );
            return 0;
        }
    }
    printf( "\n%d is a prime number.\n", n );
    return 0;
}
```

Test Runs

First Run

```
Enter a positive integer number: 43
43 is a prime number.
```

Second Run

```
Enter a positive integer number: 92
92 is not a prime number.
```

Example 3.25: *The number 1991 is a palindrome because it is the same number when read forward or backward. Write a program to check whether the given number is palindrome or not.*

Here, we first form a new number by reversing the digits of a given number and then we compare the given number with reversed number. If they match, then the given number is palindrome otherwise it is not a palindrome.

Listing 3.25

```
#include<stdio.h>
int main()
{
    int sum = 0, digit;
    int number, temp;
    printf( "\nEnter any positive integer number: " );
    scanf( "%d", &number );
    temp = number;
    while ( temp > 0 )
    {
```

```
      digit = temp % 10;
      temp /= 10;
      sum = sum * 10 + digit;
   }
   if ( number == sum )
      printf( "\n%d is a palindrome number.\n", number );
   else
      printf( "\n%d is not a palindrome number.\n", number );
   return 0;
}
```

Test Runs

First Run

```
Enter any positive integer number: 1991
1991 is a palindrome number.
```

Second Run

```
Enter any positive integer number: 1234
1234 is not a palindrome number.
```

Example 3.26: *Write a program to check whether the given natural number is an Armstrong number. Armstrong number is a 3–digit number whose sum of cubes of its digits equals the number itself. For example, number 153 is an Armstrong number as*

$1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$

Listing 3.26

```
#include<stdio.h>
int main()
{
  int n, t, s = 0, d;
  printf( "\nEnter 3-dgit natural number : " );
  scanf("%d", &n);
  t = n;
  while ( t > 0 )
  {
     d = t % 10;
     s = s + d * d * d;
     t = t / 10;
  }
  if ( s == n )
     printf("\n%d is an Armstrong number.\n", n);
  else
     printf( "\n%d is not an Armstrong number,\n", n);
  return 0;
}
```

Test Runs

First Run

```
Enter 3-dgit natural number : 153
153 is an Armstrong number.
```

Second Run

```
Enter 3-dgit natural number : 135
135 is not an Armstrong number.
```

Example 3.27: *A Fibonacci sequence is defined as follows: the first and second terms in the sequence are 0 and 1. Subsequent terms are found by adding the preceding two terms in the sequence. For example, first 10 terms of the Fibonacci sequence are*

0 1 1 2 3 5 8 13 21 34

Write a program to generate the first n terms of the sequence.

Listing 3.27

```
#include<stdio.h>
int main()
{
   int prev = 0, curr = 1, next, n, count;
   printf( "\nEnter value for n(>2) : " );
   scanf( "%d", &n );
   printf( "%d  %d", prev, curr );
   count = 2;
   while ( count < n )
   {
       next = prev + curr;
       printf( "%d  ", next );
       count++;
       previ = curr;
       curr = next;
   }
   printf( "\n" );
   return 0;
}
```

Test Run

```
Enter value for n(>2) : 10
0  1  1  2  3  5  8  13  21  34
```

Example 3.28: *The following figure demonstrates the way to compute greatest common divisor (GCD) of two positive integer 25 and 320 using long division.*

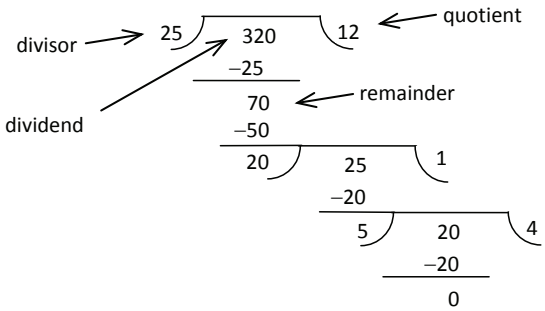


Fig. 3.13: Illustration of computational procedure for GCD using long division

From above figure, you must have observed that in successive divisions, the divisor of the previous division becomes dividend, remainder becomes divisor, and the division is again carried out. This process is carried out till the reminder becomes zero, and the current divisor is taken as GCD of the given integer numbers.

Write a program to find greatest common divisor (GCD), of two positive integer *m* and *n*.

Listing 3.28

```
#include<stdio.h>
void main()
{
   int m, n, r;
   printf( "\nEnter value for m : " );
   scanf( "%d", &m );
   printf( "Enter value for n : " );
   scanf( "%d", &n );
   while ( 1 )
   {
      r = n % m;
      if ( r == 0 )
      {
         printf( "\nGCD = %d\n", n );
         return;
      }
      n = m;
      m = r;
   }
}
```

Test Run

```
Enter value for m : 25
Enter value for n : 320
GCD = 5
```

UNIT SUMMARY

In this chapter, we have learned that

- The control statements allow controlling the order of execution of statements.
- This controlling of statements may involve selecting a statement from alternate statements or executing some selected statements repeatedly.
- The *if – else* and *switch* statements are known as decision making statement as they allow to select a statement from the alternatives depending upon the outcome of the given condition.
- The *switch* statement works with only those situations where a condition can take integral values.
- The *for*, *while* and *do–while* statements are known as iterative or looping statements as they allow to repeat (iterate) the selected statements.
- The *for* statement is preferred in situations where we know in advance the number of times the statements are to be executed.
- The *while* and *do–while* statements are preferred when the number of times the statements are to be executed depends on the satisfaction of certain conditions.
- The only difference between *while* and *do–while* statements that is *do-while* is always executed at least once whereas *while* loop may or may not be executed at all.
- The *break* statement is used with *switch* and all the looping statements. On its execution, it transfers the control outside the block.
- When used with the *switch* statement, the *break* statement takes the control outside the scope of the *switch* statement.
- When used with looping statement, the *break* statement takes the control outside the loop, *i.e.*, it terminates the execution of the loop.
- The *continue* statement is used only with looping statements. Its execution skips the statements following it, *i.e.*, it terminates the current iteration and the next iteration starts afresh.

EXERCISE

Subjective Questions

1. When the *if* statement does not have an associated *else*, what happens when the condition evaluates to zero value?
2. What is the advantage of *switch* statement over *else-if* construct?
3. What does nesting mean?
4. Is anything wrong with the following program?

```
void main()
{
    char ch;
    if ( ( ch = getche() ) == 'a' )
        printf("\nYou typed character a\n");
}
```

5. Where is the break statement used, and what is its function?

6. Rewrite the following code fragment using the *switch* statement:

```
char code;
code = getchar();
if ( code == 'A' )
   printf("\nAccountant\n");
else if ( code == 'C' || code == 'G' )
   printf("\nGrade IV\n");
else if ( code == 'F' )
   printf("\nFinancial Advisor\n");
else
   printf("\nIncorrect code\n");
```

7. Rewrite the following code fragment using the *if-else* statements:

```
int month;
scanf("%d", &month);
switch ( month )
{
  case 1 :
  case 3 :
  case 5 :
  case 7 :
  case 8 :
  case 10:
  case 12: printf("\nIt is a month having 31 days\n");
           break;
  case 4 :
  case 6 :
  case 9 :
  case 11: printf("\nIt is a month having 30 days\n");
           break;
  case 2:  printf("\nIt is a month having 28/29 days\n");
  default: printf("\nIncorrect month\n");
}
```

8. Rewrite the following code segments using the *switch* statement:

```
if ( ch == 'N' )
   north++;
if ( ch == 'S' )
   south++;
if ( ch == 'E' )
   east++;
```

```
if (ch == 'O')
   Outstanding++;
else if (ch == 'E')
   Excellent++;
else if (ch == 'G')
   Good++ ;
```

```
if ( ch == 'W' )
   west++;
else
   unknown++;
```

```
else if (ch == 'P')
   Poor++;
else
   Unknown++ ;
```

9. Write any two point of difference in operation of *switch* and *if- else*.
10. What is the effect of the absence of *break* statement in a *switch* statement?
11. What happens if the expression of a *while* statement is initially false?
12. What is the difference between *while* and *do - while* statements?
13. When is the *do - while* statement preferred over *while* statement?
14. Can there be multiple increment expressions in increment part of the *for* statement?
15. When is the *while* statement preferred over a *for* statement?
16. Where the *break* statement is used, and what is its function?
17. What is the action of *continue* statement?
18. Differentiate between the working of *break* and *continue* statement.

Multiple Choice Questions

1. Consider the following code:

```
switch(ch)
{
   case 'a': printf("a");
   case 'b': printf("b");
   default: printf("error");
}
```

If value 'a' is given to character variable *ch*, then the output will be

(*a*) a (*b*) ab (*c*) aberror (*d*) Error

2. Consider the following code:

```
int a = 6, b = 6;
if ( b == 6 || --a )
{
   statement1;
   statement2;
}
```

Suppose *statement1* and *statement2* does not modify the value of *a* and *b*, then what is the value of *a* after executing the above code?

(*a*) 6 (*b*) 5 (*c*) 7 (*d*) Error

3. Which of the following statement is true about *switch* statement?

(*a*) It may contain zero or more cases.

(*b*) Constant expressions are valid case values.

(*c*) Statement block of every case must have *break* statement as its last statement to avoid the control to fall through the later cases.

(*d*) All of the above.

4. If default statement is omitted in *switch* statement and there is no match with case values, then

(*a*) No statement in the switch block is executed.

(*b*) Execute all statements in the switch block.

(*c*) Execute the statements in the last case block only.

(*d*) A run time error occurs.

5. What will be the result of attempting to compile and run the following program?

```
#include <stdio.h>
void main()
{
   int c;
   printf( "\nEnter value of c: " );
   scanf( "%d", &c);
   switch(c);
   {
        case 1 : printf( "\nHello 1\n" );
                 break;
        case 2 : printf( "\nHello 2\n" );
                 break;
       default : printf( "\nInvaild value\n" );
                  break;
        case 3 : printf( "\nHello 3\n" );
                 break;
        case 4 : printf( "\nHello 4\n" );
   }
}
```

(*a*) Program will fail to compile because *default* can only appear at the end after all valid cases.

(*b*) Program will compile and execute.

(*c*) Program will fail to compile and compiler will report the syntax error as *"Case outside of switch statement in function main"* followed *"Misplaced break in function main"* for all cases and the *break* statements.

(*d*) None of above.

6. Consider the following program

```
#include <stdio.h>
void main()
{
```

```
    int a, b = 10;
    scanf( "%d", &a );
    if ( a = 0 )
       b *= 2;
    else
       b /= 2;
    printf( "%d", b );
}
```

What will be result of executing the above program with user input of number 6?

(*a*) 20 (*b*) 10 (*c*) 5 (*d*) None

7. Which of the following is not a valid type of expression in the *switch* statement?

(*a*) character (*b*) integer (*c*) float (*d*) enum

8. What will be the output of following program?

```
void main() {
   char val=1;
   if(val--==0)
     printf("TRUE");
   else
     printf("FALSE");
}
```

(*a*) TRUE (*b*) FALSE (*c*) Error (*d*) one of above

9. What will be the output of following program?

```
#define TRUE 1
void main()
{
    if(TRUE)
        printf("1");
        printf("2");
    else
        printf("3");
        printf("4");
}
```

(*a*) Error (*b*) 1 (*c*) 12 (*d*) 34

10. What will be the output of following program?

```
void main()
{
   int a=10;
   switch(a) {
      case 5+5:
         printf("Hello\n");
```

```
        default:
            printf("OK\n");
    }
}
```

(*a*) Hello (*b*) OK (*c*) Hello, OK (*d*) Error

11. What will be the output of following program?

```
void main()
{
    int a=2;
    switch(a)
    {
        printf("Message\n");
        default:
            printf("Default\n");
        case 2:
            printf("Case-2\n");
        case 3:
            printf("Case-3\n");
    }
    printf("Exit from switch\n");
}
```

(*a*) Case–2 (*b*) Message (*c*) Message (*d*) ase–2
Case–2 Case–3
Exit from switch

12. What will be the output of following program?

```
void main()
{
    short day=2;
    switch(day)
    {
        case 2: || case 22:
            printf("%dnd",day);
        break;
        default:
            printf("%dth",day);
        break;
    }
}
```

(*a*) 2nd (*b*) 22nd (*c*) Error (*d*) 2nd
22nd

13. What will be the output of following program?

```
void main()
{
    int a=2;
    switch(a)
    {
        case 1L:
            printf("One\n"); break;
        case 2L:
            printf("Two\n"); break;
        default:
            printf("Else\n"); break;
    }
}
```

(*a*) One (*b*) Two (*c*) Error (*d*) Else

14. What will be the output of following program?

```
#define TRUE 1
void main()
{
    switch(TRUE)
    {
        printf("Hello");
    }
}
```

(*a*) Hello (*b*) No output (*c*) Garbage value (*d*) Error

15. What will be the output of following program?

```
void main()
{
   int x;
   float y = 5.5;
   switch(x=y+1)
   {
      case 6: printf("It's Eight."); break;
      default: printf("Oops No choice here.");
   }
}
```

(*a*) Oops No choice here. (*b*) It's Eight.Oops No choice here!!!

(*c*) It's Eight. (*d*) Error

16. Consider the following code:

```
while ( ++i <= n );
```

What will be the value of *i* when the loop completes, initial value of *i* being 1?

(*a*) *n* (*b*) *n*-1 (*c*) *n*+1 (*d*) *n*+2

17. Consider the following code. What is its output?

```
for ( i = 1;i <= 5; i++ )
{
   printf( "%d", i += 2 );
}
```

(*a*) 1 2 3 4 5 (*b*) 3 4 5 6 7 (*c*) 3 5 7 9 11 (*d*) 3 6

18. Which of the following statement is not true about *continue* statement?

(*a*) It can be used in conjunction with *switch* statement.

(*b*) It terminates the loop.

(*c*) It terminates the current iteration.

(*d*) It can be used with all looping statements as well as *switch* statement.

19. Which of the following statement is not true about *switch* statement?

(*a*) It may contain zero or more cases.

(*b*) Constant expressions are valid case values.

(*c*) The expression in *switch* statement can be of *float* type.

(*d*) Statement block of every case must have *break* statement as its last statement to avoid the control to fall through the later cases.

20. The *continue* statement is used to

(*a*) continue the next iteration of the loop statement.

(*b*) exit the block of loop statement .

(*c*) exit from the outermost block even it is used in the innermost block.

(*d*) continue execution of the program even error occurs.

21. Consider the following program

```
void main()
{
    int i, j;
    for ( i = 0; j = 10; i < j;  i++, j-- );
        printf( "x" );
}
```

How many times letter 'x' will be printed?

(*a*) 5 (*b*) 1 (*c*) 10 (*d*) 4

22. Consider the following program

```
void main()
{
    unsigned char ch;
    for ( ch = 0; ch <= 256; ch++ )
        printf( "%d = %c", ch, ch );
}
```

How many times *for* loop will be executed?

(*a*) 256 (*b*) 255 (*c*) 257 (*d*) Infinitely

23. How many times the while loop will be executed in the following program?

```
void main()
{
   int j = 1;
   while ( j <= 100 );
   {
      printf("\nj = %d, j*j = %d", j, j*j );
      j++;
   }
}
```

(*a*) Infinite times (*b*) 100 times (*c*) 99 times (*d*) 101 times

24. What will be the output of following program?

```
void main()
{
   int i;
   for( i = 0; i < 5; i++ ) {
     int j = 3;
     printf("%d  ",i*j);
   }
   printf("%d",j);
}
```

(*a*) Program will compile and execute successfully. (*b*) Program will fail to compile.

(*c*) Program will give output as 0 3 6 9 12 3 (*d*) None of the above

25. What will be the output of following program?

```
void main()
{
   int i = 0;
   for(;;) {
     if ( i++ == 4 ) break;
     continue;
   }
   printf("i = %d",i);
}
```

(*a*) i = 4 (*b*) i = 5 (*c*) i = 0 (*d*) Error

ANSWERS									
1.	(*c*)	2.	(*a*)	3.	(*d*)	4.	(*a*)	5.	(*c*)
6.	(*c*)	7.	(*c*)	8.	(*a*)	9.	(*c*)	10.	(*c*)
11.	(*d*)	12.	(*c*)	13.	(*b*)	14.	(*b*)	15.	(*c*)
16.	(*c*)	17.	(*d*)	18.	(*d*)	19.	(*c*)	20.	(*a*)
21.	(*b*)	22.	(*d*)	23.	(*a*)	24.	(*b*)	25.	(*b*)

Programming Problems

1. Write a program to check whether the given natural number is EVEN or ODD without using any arithmetic operator.
2. Given measure of three line segments as *a*, *b*, and *c*, write a program to check whether these line segments can be used to form a triangle or not.
3. Given measure of three angles of a triangle ABC as *a*, *b*, and *c* degrees, respectively, write a program to check whether a triangle can be formed with these angles or not.
4. Write a program, using switch statement, to check whether the given alphabet is a vowel or a consonant.
5. Given a measure of an angle in degrees and in anti-clockwise direction, write a program to find the type of the angle.
6. Given three points $A(x_1, y_1)$, $B(x_2, y_2)$ and $C(x_3, y_3)$, determine whether they are collinear, i.e., lie on the same line.
7. Given points (x_1, y_1) & (x_2, y_2) on line *AB*, and points (x_3, y_3) & (x_4, y_4) on line *CD*, write a program to determine whether lines AB & CD intersect each other.
8. Write a program to find the day of the week on a given date.
9. Write a program to find your age when your date-of-birth and today's date is given, both in the format *dd*/*mm*/*yyyy*.
10. Write a program to convert time from 12 hours system to 24 hours system.
11. Write a program to convert time from 24 hours system to 12 hours system.
12. Write a program to find difference in time when the start time and ending time is given, both in the format *hh*:*mm*:*ss*.
13. Write a program to find the *n*th prime number.
14. Write a program to print first *n* prime numbers.
15. Write a program to find the *n*th term of the Fibonacci sequence.
16. You know that an even number is a number which is divisible by 2. However there is another approach that can be used to check whether the given number is even or not. The approach is by inspecting the unit digit - *if the unit digit is 0, 2, 4, 6, or 8, the number is even*. So you are required to write a program to check whether the given number is even or not without performing the division by 2.

PRACTICALS

1. Write a program to find the roots of a quadratic equation.
   Refer to Example 3.8.
2. Write a program to test whether given natural number is prime number or not.
   Refer to Example 3.24.

3. Write a program to test whether given natural number is palindrome number or not. Refer to Example 3.25.
4. Write a program to check whether a given date in format *dd/mm/yyyy* is valid or not.

Listing 3.29

```
/*
    Program to check whether date given in format dd/mm/yyyy
    is valid or not.
*/
#include<stdio.h>
int main()
{
   int mm, dd, yyyy;
   char ch;
   printf( "\nEnter date in format dd/mm/yyyy : ");
   scanf( "%2d%c%2d%c%4d", &dd, &ch, &mm, &ch, &yyyy );
   if ( mm < 1 || mm > 12 ) {
       printf( "\nDate is Invald.\n");
       return 0;
   }
   if ( mm == 2 ) {
       /* condition to check for leap year  */
       if ( (yyyy%400==0)||( (yyyy%4==0)&&(yyyy%100!=0) ) ) {
           if ( dd > 1 && dd <= 29 )
               printf( "\nDate is valid.\n");
           else
               printf( "\nDate is invald.\n");
       } else {
           if ( dd > 1 && dd <= 28 )
               printf( "\nDate is valid.\n");
           else
               printf( "\nDate is invald.\n");
       }
   } else if ( mm == 4 || mm == 6 || mm == 9 || mm == 11 ) {
       if ( dd > 1 && dd <= 30 )
           printf( "\nDate is valid.\n");
       else
           printf( "\nDate is invald.\n");
   } else {
       if ( dd > 1 && dd <= 31 )
           printf( "\nDate is valid.\n");
       else
           printf( "\nDate is invald.\n");
   }
   return 0;
}
```

Test Runs

First Run

```
Enter date in format dd/mm/yyyy : 29/2/2020
Date is valid.
```

Second Run

```
Enter date in format dd/mm/yyyy : 29/2/2021
Date is invalid.
```

5. Write a program to print all Armstrong numbers.

Note that Armstrong numbers 3–digit numbers. Therefore, we start from 100 (first 3-digit number) and continue upto 999 (last 3-digit number), and on the way test each number to see whether it is an Armstrong number.

Listing 3.30

```
/*
    Program to print all Armstrong numbers
*/

#include<stdio.h>
int main()
{
   int i, t, s, d;
   printf( "\nFollowing is list of all Armstrong numbers.\n\n");
   for ( i = 100; i <= 999; i++ )
   {
      t = i;
      s = 0;
      while ( t > 0 )
      {
         d = t % 10;
         s = s + d * d * d;
         t = t / 10;
      }
      if ( s == i )
          printf("%d  ", i);
   }
   printf( "\n");
   return 0;
}
```

Test Run

```
Following is list of all Armstrong numbers.
153  370  371  407
```

6. Write a program to print all prime number in the range *m..n*. The values for *m* and *n* will be supplied by the user at execution time.

Listing 3.31

```
/*
    Program to find prime numbers in the range m..n. Value
    of m and n will be supplied by user at execution time
*/
#include<stdio.h>
#include<math.h>
int main()
{
   int i, m, n, k, mm;
   printf("\nProgram to find prime numbers in range m..n\n\n");
   printf( "\nEnter value of m : ");
   scanf( "%d", &m );
   printf( "\nEnter value of n : ");
   scanf( "%d", &n );
   printf( "\n\nPrime numbers in range %d..%d are\n\n", m, n);
   for ( i = m; i <= n; i++ )
   {
       if ( ( i > 2 ) && ( ( i % 2 ) == 0 ) )
           continue;
       mm = sqrt(i);
       for ( k = 3; k <= mm; k += 2 )
       {
           if ( i % k == 0 )
               break;
       }
       if ( k > mm )
           printf( "%d  ", i );
   }
   printf( "\n");
   return 0;
}
```

Test Run

```
Program to find prime numbers in range m..n
Enter value of m : 10
Enter value of n : 50
Prime numbers in range 10..50 are
11  13  17  19  23  29  31  37  41  43  47
```

KNOW MORE

The topic of conditional branching and looping constitutes the core part of programming. Therefore, students are expected to have sound grip over the topic.

The teacher is expected to discuss the problem undertaken, to develop the logic with the active participations of the students.

The teacher should not dictate the solution of the problem, hence program, rather should facilitate that students develop programs on their own using programs listed in the books as a reference.

The teacher should also demonstrate the process of testing & debugging to ensure the correctness of the program.

REFERENCES & SUGGESTED READINGS

1. R. S. Salaria, Problem Solving & Programming in C, Khanna Book Publishing Co(P) Ltd., New Delhi.
2. E. Balagurusamy, Programming in ANSI C, Tata McGraw Hill, New Delhi..
3. Yashavant Kanetkar, Let Us C, BPB Publications, New Delhi.
4. Byron Gottfried, Programming with C, Schaum's Outlines.
5. https://onlinecourses.nptel.ac.in/noc21_cs01/preview
6. https://ocw.mit.edu/courses/intro-programming/
7. https://www.programiz.com/c-programming
8. https://www.javatpoint.com/c-programming-language-tutorial

4 Arrays

UNIT SPECIFICS

This unit discusses the topics related to arrays. These arrays can be of numeric data or character data. The array of characters is used to handle strings in C language. This unit explains various aspects of the arrays & strings, and demonstrates their use with suitable examples.

RATIONALE

In real-life problems, we come across real-life situations where we have to deal with the collection of data that may be homogenous or heterogeneous. Further that collection of data may be numeric in nature or non-numeric.

Therefore, to solve real-life problems dealing with collecting data, homogenous in particular, we need appropriate data structure to store those collections in memory, and facilitate their processing to arrive at the solution of the problem in-hand.

An array is a data structure rich enough to store the collection of homogeneous data as well as very easy to process.

This unit will help the student understand the various aspects related to arrays and develop programs for real-life problems dealing with homogeneous data collection.

PRE-REQUISITES

- Basic data types
- Condition branching
- Loops and nested loops

UNIT OUTCOMES

Upon completion of the unit, students will be able to

U4-O1: explain the concept of arrays

U4-O2: declare, initialize, and perform input/output of arrays

U4-O3: Use arrays to formulate algorithms and programs

U4-O4: Use arrays to solve matrix related problems

U4-O5: explain strings as an array of characters

U4-O6: demonstrate the use of standard string handling functions

Unit 4 Outcomes	EXPECTED MAPPING WITH COURSE OUTCOMES (1- Weak Correlation; 2- Medium correlation; 3- Strong Correlation)							
	CO-1	CO-2	CO-3	CO-4	CO-5	CO-6	CO-7	CO-8
U4-O1	1	-	-	-	-	-	-	-
U4-O2	-	-	1	-	-	-	-	-
U4-O3	-	-	-	-	-	3	-	-
U4-O4	-	-	-	-	-	3	-	-
U4-O5	-	-	1	-	-	-	-	-
U4-O6	-	-	-	-	-	2	-	-

4.1 INTRODUCTION

An *array* is a collection of homogeneous data elements (i.e., of the same data type) described by a single name, and each individual element of an array is referenced by a *subscripted* variable, formed by affixing to the array name a *subscript* or *index* enclosed in brackets.

The term subscript has the same meaning as in the mathematical notation.

Type of Arrays

- **One-dimensional (1D) array** – It is a type of array where we need only single subscript to access its elements. A one-dimensional array is also known as a linear array. It is the most frequently used type of array.
- **Two-dimensional (2D) array** – It is a type of array where we need two subscripts to access its elements. This type of array is used to store type of collection of data where its elements are arranged into rectangular fashion, *i.e.*, in rows and columns. In business terminology we call it a table, and in mathematics terminology, we call it a matrix.
- **Multi-dimensional arrays** – It is a type of array where we need more than two subscripts to access its elements. These types of arrays are used to store data related to complex engineering problems.

In this unit, our discussion will be restricted to 1D and 2D. In addition, we will also discuss strings that are a special case of character arrays.

4.2 ONE-DIMENSIONAL ARRAYS

Fig. 4.1 depicts various aspects of a one-dimensional array, named *marks*, that stores marks of 10 students.

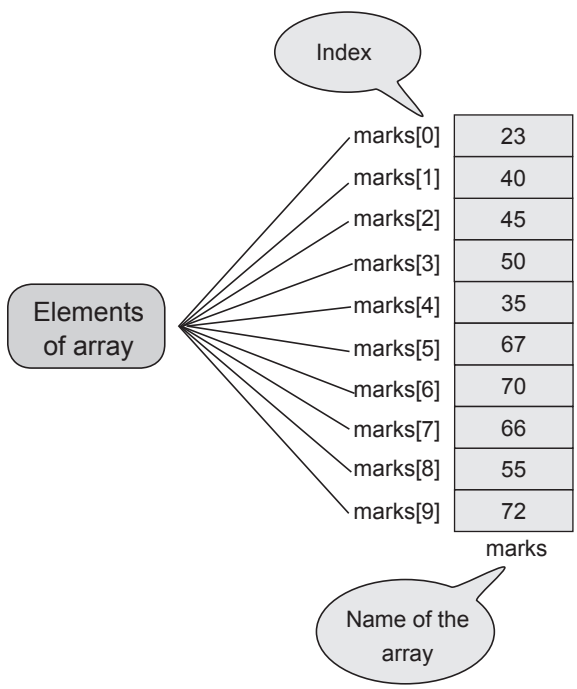


Fig. 4.1: The *marks* Array

4.2.1 Declaration

An array must be declared before it can be used. Array declaration tell the compiler *the name of the array*, *the type of its elements*, and *the size or number of elements in the array*. The size of the array is a constant and must have a value at compilation time.

The syntax for declaration of a one-dimensional array is

```
type arrayName[arraySize];
```

Fig. 4.2 shows three different array declarations, one for integers, one for floating-point numbers, and one for characters. Elements of the arrays are shown just for completeness at this moment.

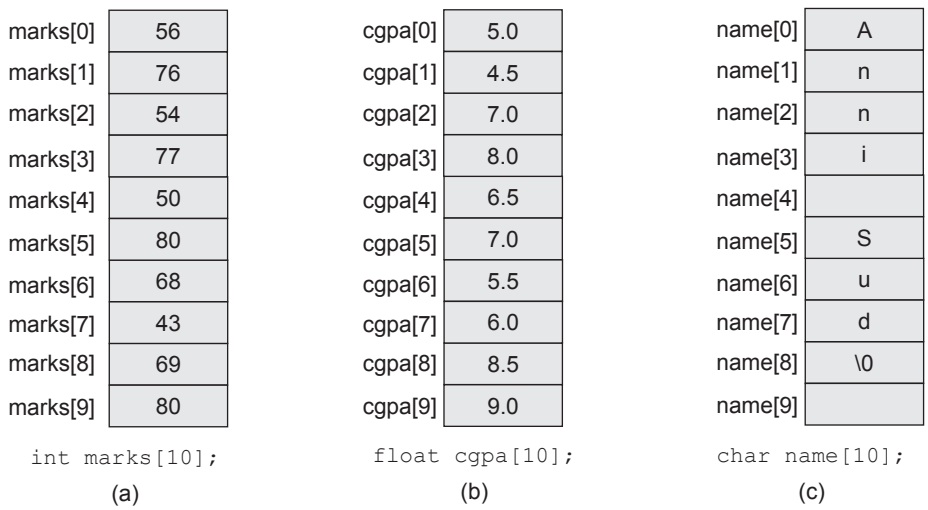


Fig. 4.2: Declaration of 1D arrays

The declaration statement

```
int marks[10];
```

allocates a contiguous memory block of 20 bytes (in Turbo C/C++ Compiler, which is a 16-bit compiler), 2-bytes for each element of type *int*. The first 2-bytes are used for the first element of the array (*marks*[0]), the next 2-bytes for the second element of the array (*marks*[1]), and so on. The last 2-bytes are used for element *marks*[9].

The declaration statement

```
float cgpa[10];
```

allocates a contiguous memory block of 40 bytes, 4-bytes for each element of type *float*. The first 4-bytes are used for the first element of the array (*cgpa*[0]), the next 4-bytes for the second element of the array (*cgpa*[1]), and so on. The last 4-bytes are used for element *cgpa*[9].

The declaration statement

```
char name[10];
```

allocates a contiguous memory block of 10 bytes, 1-byte for each element of type *char*. The first byte is used for first element of the array (*name*[0]), the next byte for the second element of the array (*name*[1]), and so on. The last byte is used for element *name*[9].

4.2.2 Initialization

Just as ordinary variables can be initialized along with declaration, the elements of the arrays can also be initialized. For each element in the array, we provide a value. The only difference is that the values must be enclosed in braces, and if there is more than one, separated by a comma.

The following examples provide all possible ways of initializing one-dimensional array.

The following statement

```
int b[10]={12, 0, 14, –4, 7, 8, 10, 11, 9, 15};
```

initializes element *b*[0] to 12, *b*[1] to 0, *b*[2] to 14, *b*[3] to –4, *b*[4] to 7, *b*[5] to 8, *b*[6] to 10, *b*[7] to 11, *b*[8] to 9, *b*[9] to 15.

If the array is initialized like this

```
int b[] = {12, 0, 14, –4, 7};
```

Since the size of the array is not specified, the compiler counts the number of elements in the initialization list and fixes that as the array size.

What can you expect from the following declaration statement?

```
int b[10] = {1, 2, 3};
```

It initializes element *b*[0] to 1, *b*[1] to 2, *b*[2] to 3, and all remaining elements are initialized to 0. We can say that in partially initialized arrays, all remaining elements are initialized to 0.

What can you expect from the following declaration statement?

```
int a[6]={12, 0, 14, –4, 7, 15, –20, 22, 25};
```

The compiler will flag an error message as the initialization list contains more elements than the declared size.

4.2.3 Accessing Elements

A single *index* is used to access individual elements in a one-dimensional array. The index must be an integral value or an expression that evaluates to an integral value.

For example, given the *marks* array in Fig. 4.2 (*a*), we would access the first element as

```
marks[0]
```

To process all the elements in *marks*, a loop similar to the following code is used:

```
for (i = 0; i < 10; i++ )
    process ( marks[i] );
```

4.2.4 Input

Consider the declaration

```
int a[50];
```

Suppose, the actual number of elements in a 1D array is *n* (<=50). The user will supply the value of *n* at execution time.

The following segment shows the way data is entered into one-dimensional array.

```
for ( i = 0; i < n; i++ )
{
   scanf( "%d", &a[i] );
}
```

Observe the following things:

1. We start the index, *i*, at 0 and goes upto (*n*-1).
2. Even though we are dealing with array elements, the *address of* operator (&) is still necessary in the *scanf()* function call.

The elements of array can be entered on same line by separating with space/horizontal-tab or one element per line.

4.2.5 Output

To output values of elements of an array, we use the loop as follows:

```
for ( i = 0; i < n; i++ ) {
   printf( "%d  ", a[i] );
}
printf( "\n" );
```

All the values are displayed on one line, separated by two spaces.

However, if all the values cannot be displayed in one line either because of their magnitude of values or number of values, they are displayed in subsequent lines. After, the *for* loop completes, a call to *printf()* function advances the cursor to the next line.

The following *for* loop display one value per line:

```
for ( i = 0; i < n; i++ ) {
   printf( "\n%d", a[i] );
}
printf( "\n" );
```

ILLUSTRATIVE EXAMPLES OF 1D ARRAYS

Having understood the various aspects of the one-dimensional arrays, let us write some example programs to demonstrate the power of one-dimensional arrays to handle complex problems.

Example 4.1: *Given an array, named a, of size n(≤ 50) whose elements are of type int. Write a program that output those elements of the array that are even.*

Listing 4.1

```
/*
   Program that outputs those elements of a 1D array that are EVEN
*/
#include<stdio.h>
int main()
{
```

```
    int a[50], i, n;
    printf("\nEnter size of array n(<=50) : ");
    scanf("%d", &n);
    printf("\nEnter %d natural numbers as elements of array\n", n);
    for ( i = 0; i < n; i++ )
       scanf("%d", &a[i]);
    printf("\nEVEN elements of array are\n");
    for ( i = 0; i < n; i++ )
    {
       if ( a[i] % 2 == 0 )
           printf("%d  ", a[i]);
    }
    printf("\n");
    return 0;
}
```

Test Run

```
Enter size of array n(<=50) : 10
Enter 10 natural numbers as elements of array
10 7 15 14 24 23 77 35 70 12
EVEN elements of the array are
10 14 24 70 12
```

Example 4.2: *Given an array, named a, of size n(≤ 50) whose elements are of type int. Write a program to find the largest element, smallest element, and the average of elements of 1D array.*

Listing 4.2

```
/*
   Program to find the largest element, smallest element,
   and the average of elements of 1D array
*/
#include <stdio.h>
int main()
{
   int a[20];
   int i, j, n, largest, smallest, sum;
   float average;
   printf("\nEnter size of array n(<=20) : ");
   scanf("%d", &n);
   printf("\nEnter %d element of 1D array\n", n);
```

```
    for ( i = 0; i < n; i++ )
        scanf("%d", &a[i]);
    /*  code segment to find largest element  */
    largest = a[0];
    for ( i = 1; i < n; i++ ) {
        if ( a[i] > largest )
            largest = a[i];
    }
    /*  code segment to find smallest element  */
    smallest = a[0];
    for ( i = 1; i < n; i++ ) {
        if ( a[i] < smallest )
            smallest = a[i];
    }
    /*  code segment to find average of elements  */
    sum = 0;
    for ( i = 0; i < n; i++ )
         sum = sum + a[i];
    average = (float) sum/n;
    /*  code segment to output the results  */
    printf("\nLargest element    = %d", largest );
    printf("\nSmallest element   = %d", smallest );
    printf("\nAverage of elements = %.2f\n", average );
    return 0;
}
```

Test Run

```
Enter size of array n(<=20) : 10
Enter 10 element of 1D array
10 24 33 15 19 21 35 20 40 16
Largest element    = 40
Smallest element   = 10
Average of elements = 23.30
```

Example 4.3: *Given a decimal number n. Write a program to convert n into its equivalent binary number.*

Listing 4.3

```
/*
   Program to convert a decimal number 'n' into its equivalent
   binary number. The program uses 1D array to store remainders
   during the process of conversion and then prints this array in
   reverse order.
*/
```

```
#include <stdio.h>
int main()
{
   int a[20];
   int i, j, t, n;

   printf("\nEnter decimal number n : ");
   scanf("%d", &n);
   t = n;
   i = 0;
   while ( t > 0 )
   {
       a[i] = t % 2;
       i++;
       t = t / 2;
   }
   printf("\nDecimal number %d is equivalent to ", n);
   for ( j = i-1; j >= 0; j-- )
      printf("%d", a[j]);
   printf(" binary.\n");
   return 0;
}
```

Test Run

```
Enter decimal number n : 10
Decimal number 10 is equivalent to 1010 binary.
```

Example 4.4: *Given an array, named a, of size n(≤ 50) whose elements are of type int. Write a program to swap the adjacent elements of a 1D array.*

Listing 4.4

```
/*
   Program to swap adjacent elements of a 1D array.
   Size of array is even number.
*/
#include <stdio.h>

int main()
{
   int a[20];
   int i, n, t;
```

```
    printf("\nEnter size of array n(<=20) : ");
    scanf("%d", &n);
    printf("\nEnter %d element of 1D array\n", n);
    for ( i = 0; i < n; i++ )
        scanf("%d", &a[i]);
    /*  code segment to swap adjacent elements  */
    for ( i = 0; i < n-1; i += 2 ) {
        t = a[i];
        a[i] = a[i+1];
        a[i+1] = t;
    }
    printf("\nElements of array after swapping adjacent elements\n");
    for ( i = 0; i < n; i++ )
       printf("%d  ", a[i]);
    printf("\n\n");
    return 0;
}
```

Test Run

```
Enter size of array n(<=20) : 10

Enter 10 element of 1D array
10 24 33 15 19 21 35 20 40 16

Elements of an array after swapping adjacent elements
14 10 15 33 21 19 20 35 16 40
```

4.3 TWO-DIMENSIONAL ARRAYS

The array we have discussed so for are known as one-dimensional arrays because the data are organized linearly only in one direction (dimension). Many applications require that data be organized in more than one dimension. One common example is a matrix (a table), which is an array that consists of rows and columns.

Fig. 4.3 shows a matrix (table) with 4 rows and 4 columns which is commonly used as a two-dimensional array.

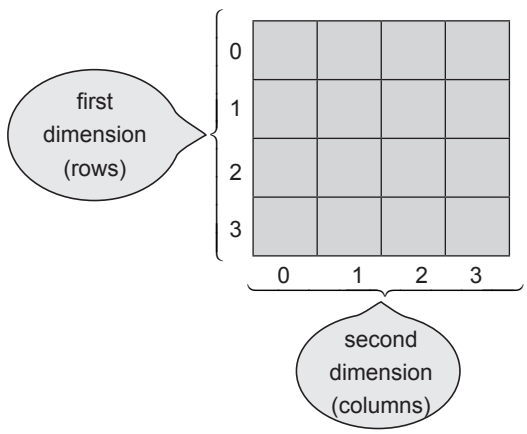


Fig. 4.3: Two-dimensional array

4.3.1 Declaration

Two-dimensional arrays, like one-dimensional arrays, must be declared before being used. Declaration tells the compiler *the name of the array*, *the type of its elements*, and *the size of each dimension*.

The syntax for declaration of two-dimensional array is

```
type arrayName[RowSize][ColSize];
```

By convention, the first dimension specifies the number of *rows* in the array while the second dimension specifies the number of *columns* in each row.

Consider the declaration

```
int a[3][3];
```

This declarations statement allocates a contiguous memory block of 18 bytes, 6-bytes for each row. The first set of 6-bytes is used to store elements of first row, the second the set of 6-bytes are used to store elements of the second row, and the third set of 6-bytes are used to store elements of third (last) row. Within in each row, the first 2-bytes are used to store element of the first column, the next 2-bytes are used to store element of second column, and final 2-bytes are used to store element of the third (last) column.

Elements of a two-dimensional array are stored row-wise, *i.e.*, in the allocated contiguous block of memory, the first elements of first row are stored, then elements of the second row, and finally elements of the third row, and so on.

4.3.2 Initialization

One way to initialize two-dimensional arrays is as shown below:

```
int a[3][3] = { 0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3 };
```

It will initialize all elements of the first row with value 0, elements of the second row with value 1, the elements of third row with 2 and elements of the fourth row with value 3.

Though the above initialization has no problems, it highly recommended that the nest braces be used to show the exact nature of the two-dimensional array.

The array *a* is better initialized as shown below:

```
int a[3][3] = { { 0, 0, 0 },
                { 1, 1, 1 },
                { 2, 2, 2 },
                { 3, 3, 3 },
              };
```

In the above initialization process, we initialize each row as a one-dimensional array of three elements enclosed in braces. The array of three rows also has its set of braces. Note that we use commas between the elements in the columns and also commas between the rows.

4.3.3 Accessing Elements

To access *j*th element of the *i*th row, we write

```
a[i][j]
```

To process all the elements row-wise of *a*, nested loops similar to the following code will be used:

```
for (i = 0; i < 3; i++ )
{
    for (j = 0; j < 3; j++ )
    {
        process ( a[i][j] );
    }
}
```

4.4.4 Input

Following segment shows the way elements can be read row-wise using nested for loops:

```
for (i = 0; i < 3; i++ )
{
   for (j = 0; j < 3; j++ )
   {
      scanf( "%d", &a[i][j] );
   }
}
```

In general, if the two-dimensional array is of size $m \times n$, the first loop varies from 0 to $(m - 1)$ and the second loop varies from 0 to $(n - 1)$.

4.3.5 Output

To output values of elements of two-dimensional array, we use nested loops as follows:

```
for ( i = 0; i < 3; i++ )
{
    for ( j = 0; j < 3; j++ )
    {
       printf( "%d  ", a[i][j] );
    }
    printf( "\n" );
}
```

The elements of the two-dimensional array will be output row-wise by the above segment. Thus, the first loop controls the rows while the second loop controls the columns.

ILLUSTRATIVE EXAMPLES OF 2D ARRAYS

Example 4.5: *Program to compute the transpose of a matrix A of order m × n.*

We know that if we interchange the rows with columns, we get transpose of a given matrix, say *B* of order $n \times m$, *i.e.*,

$$b_{ji} = a_{ij}$$

Listing 4.5

```
/*
   Program to compute transpose of matrix A(mxn)
*/
#include<stdio.h>
int main()
{
    int a[10][10], b[10][10];
    int n, m, i, j;
    printf( "Enter the size of matrix A as m,n: " );
    scanf( "%d,%d", &m, &n);
    printf("\nEnter %d elements of matrix A row-wise\n",m*n);
    for ( i = 0 ; i < m; i++ ) {
        for ( j = 0; j < n; j++ ) {
           scanf( "%d", &a[i][j] );
        }
    }
    for ( i = 0 ; i < m ; i++ ) {
        for ( j = 0 ; j < n ; j++ ) {
           b[j][i] = a[i][j];
        }
    }
    printf( "\nTranspose is\n\n" );
    for ( i = 0 ; i < n; i++ ) {
        for ( j = 0; j < m; j++ ) {
           printf( "%5d", b[i][j] );
        }
        printf ( "\n" );
    }
    return 0;
}
```

Test Run

```
Enter the size of matrix A as m,n: 3,3
Enter 9 elements of matrix A row-wise
1 2 3
4 5 6
7 8 9
Transpose is
  1   4   7
  2   5   8
  3   6   9
```

Example 4.6: *Program to add two matrices A and B, each of order m × n.*

We know that to add two matrices, we add corresponding elements to get the element of a resultant matrix, say *C* of order *m* × *n*, *i.e.*,

$$c_{ij} = a_{ij} + b_{ij}$$

Listing 4.6

```
/*
   Program to add matrix A and B, each of order mxn
*/
#include<stdio.h>
int main()
{
    int a[10][10], b[10][10], c[10][10];
    int n, m, i, j;
    printf( "Enter the size of matrices as m,n: " );
    scanf( "%d,%d", &m, &n);
    printf( "Enter %d elements of matrix A row-wise\n", m*n );
    for ( i = 0 ; i < m; i++ ) {
        for ( j = 0; j < n; j++ ) {
            scanf( "%d", &a[i][j] );
        }
    }
    printf( "Enter %d elements of matrix B row-wise\n", m*n );
    for ( i = 0 ; i < m; i++ ) {
        for ( j = 0; j < n; j++ ) {
            scanf( "%d", &b[i][j] );
        }
    }
    for ( i = 0 ; i < m ; i++ ) {
       for ( j = 0 ; j < n ; j++ ) {
          c[i][j] = a[i][j] + b[i][j];
       }
    }
    printf( "\nSum A+B is\n\n" );
    for ( i = 0 ; i < m; i++ ) {
        for ( j = 0; j < n; j++ ) {
          printf( "%4df", c[i][j] );
        }
        printf ( "\n" );
    }
    return 0;
}
```

Test Run

```
Enter size of matrices as m,n: 3,3
Enter 9 elements of matrix A row-wise
1 1 1
1 1 1
1 1 1
Enter 9 elements of matrix B row-wise
2 2 2
2 2 2
2 2 2
Sum A+B is
 3  3  3
 3  3  3
 3  3  3
```

Example 4.7: *Program to multiply two matrices A and B. The matrix A is of order m × n matrix and the matrix B is of order p × q matrix, provided n = p.*

As we want to multiply two matrices *A* and *B*, where the matrix *A* is of order $m \times n$ and the matrix *B* is of order $p \times q$, the product matrix, say matrix *C*, will be of order $m \times p$ with entry c_{ij}, which appears in the *i*th row and *j*th column is given as:

c_{ij} = sum of the products of the entries in the *i*th row of matrix *A* with the corresponding entries in the *j*th column of matrix *B*

$$= a_{i0} \times b_{0j} + a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + \ldots + a_{i(n-1)} \times b_{(n-1)j}$$

$$= \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

Listing 4.7

```
/*
   Program to multiply matrix A of size mxn by matrix B of
  size pxq. The program also checks for the feasibility of product.
*/
#include<stdio.h>
int main()
{
    int a[10][10], b[10][10], c[10][10];
    int n, m, p, q, i, j, k;
    printf( "Enter the size of matrix A as m,n: " );
    scanf( "%d,%d", &m, &n);
    printf( "Enter the size of matrix B as p,q: " );
    scanf( "%d,%d", &p, &q);
```

```
    if ( n != p ) {
        printf( "\nMatrix Product AxB is not feasible\n" );
        return 1;
    }
    printf("\nEnter %d elements of matrix A row-wise\n", m*n);
    for ( i = 0 ; i < m; i++ ) {
       for ( j = 0; j < n; j++ ) {
          scanf( "%a", &a[i][j] );
       }
    }
    printf("\nEnter %d elements of matrix B row-wise\n", p*q);
    for ( i = 0 ; i < p; i++ )
    {
       for ( j = 0; j < q; j++ )
       {
          scanf( "%d", &b[i][j] );
       }
    }
    for ( i = 0 ; i < m ; i++ ) {
       for ( j = 0 ; j < q ; j++ ) {
           c[i][j] = 0;
           for ( k = 0; k < n; k++ ) {
               c[i][j] += a[i][k] * b[k][j];
           }
       }
    }
    printf( "\nProduct AxB is\n\n" );
    for ( i = 0 ; i < m; i++ ) {
      for ( j = 0; j < q; j++ ) {
           printf( "%4d", c[i][j] );
       }
       printf ( "\n" );
    }
    return 0;
}
```

Test Run

```
Enter the size of matrix A as m,n: 3,3
Enter the size of matrix B as p,q: 3,3
```

```
Enter 9 elements of matrix A row-wise
1 1 1
1 1 1
1 1 1
Enter 9 elements of matrix B row-wise
2 2 2
2 2 2
2 2 2
Product AxB is
  6   6   6
  6   6   6
  6   6   6
```

4.4 CHARACTER ARRAYS AND STRINGS

Each of the arrays illustrated so far contained numeric elements. However, we can also have an array of characters. Using an array of characters, we can store string data. But do remember that strings are not directly supported in C language, though sometimes loosely we refer to array of characters as strings.

A string in C language is a variable-length array of characters that is delimited by a null character ('\0'). Generally, characters comprising a string are selected only from printable characters; however, C language does permit the use of any character except the null character. In fact, it is common to use formatting characters, such as tabs, format specifiers, etc., in strings.

In this section, we will discuss the various aspects related to the handling of strings in the C language.

The C language uses variable length, delimited strings. The null character '\0' is used as a delimiting character. Though the null character '\0' looks like a sequence of two characters, '\' and '0', but in C language, it is treated as a single character.

4.4.1 Storing Strings

In C language, a string is stored in an array of characters. It is terminated by the null ('\0') character. Fig. 4.4 shows how a string "Hello" is stored in memory. Because a string is stored in an array, the name of the array, and hence string, is a pointer to the beginning of the string.

Fig. 4.5 shows the difference between the storage of a character and a one-character string. The character requires single storage location of 1-byte whereas one-character string requires two storage locations of 1-byte each; one for the character and one for the delimiter. The figure also shows how an empty string is stored. The empty string consists of null character only.

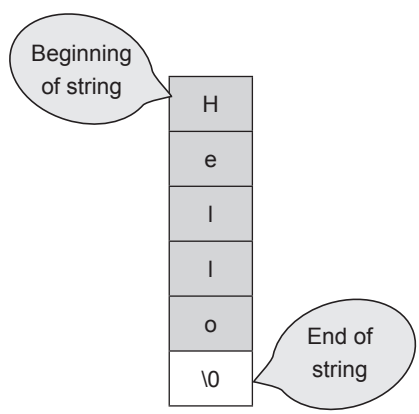


Fig. 4.4: Storing a string in memory

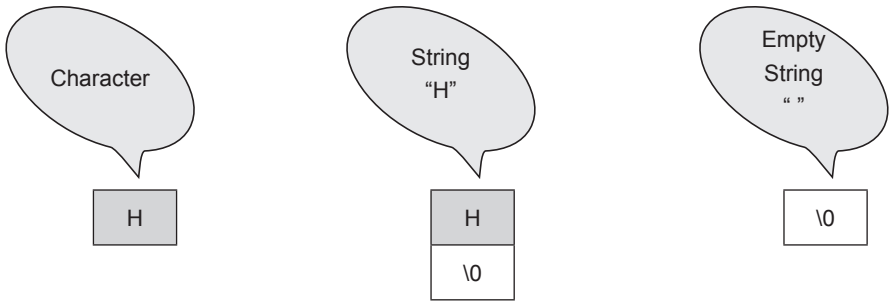


Fig. 4.5: Difference in the storage of characters and strings

4.4.2 Need of String Delimiter

The question – *Why actually we need a null character to terminate a string*? - may be coming to your mind. The answer to this question is that a string is not a data type; it is data structure – *an array*. Therefore, the implementation of a string is logical, not physical. The physical structure is the array in which a string is stored. Since, strings are of varying length, there is a need to identify the logical end of data within a physical structure.

Let us look at the need of a string delimiter from another point of view. If the data is of fixed length, then we don't need string data structure to store them rather it can easily be stored in an array. When data is stored in an array, the end of the data is always indicated by the last element of the array. But, if the data is not of fixed length (*i.e.*, varying length), then we need some other way to determine the end of the data. The null character is used to mark the end of the string data.

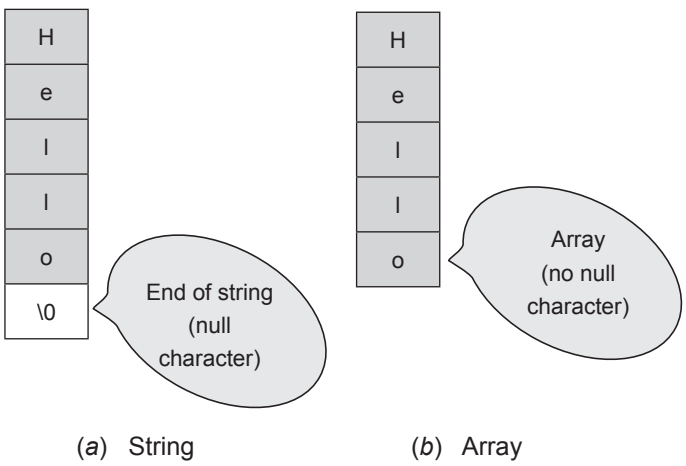


Fig. 4.6: Difference between a string and an array of characters

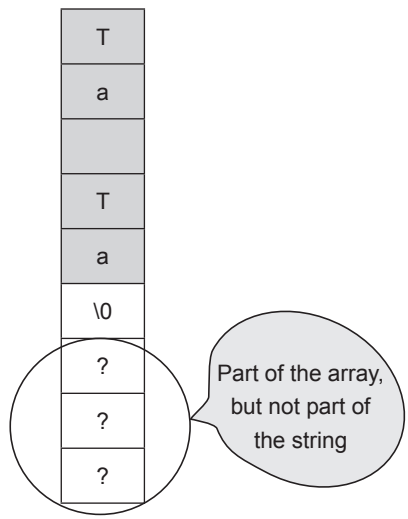


Fig. 4.7: String stored in part of array

Because strings are variable-length structures, enough space for the maximum-length string that we have to store plus one for the null character (delimiter) be reserved.

In many situations it may be possible that the entire array is not filled and we may have null character in the middle of the array. In that case, the array from the beginning to the null character is treated as string and remaining elements of the array are ignored. This means that a part of an array of characters can be treated as a string provided it ends with a null character, as shown in Fig. 4.7.

4.4.3 String Literals

A *string literal*, also known as *string constant*, is a sequence of characters enclosed in double-quotes. The following are examples of some string literals:

```
"Hello! you are wel come"
"Hari\'s Book"
""
"A"
```

Note that embedded spaces are significant. And if quotes, single (‘) or double (“), are part of the string, they must be used with the escape sequence.

4.4.4 String Variables

A string variable is an array of characters.

4.4.4.1 Declaring String Variable

A string variable is declared as shown in the following statement

```
char str[20];
```

This statement declares string variable *str* of length 20. In fact, the maximum characters that can be stored are 19 as the last character is always the null character. In fact, the above declarations reserve a memory block of 20 bytes, with the array name *str* representing the address of the first element of the array.

4.4.4.2 Initializing String Variables

The string variable can also be initialized during declaration. The following declaration illustrates one such approach.

```
char mess[] = {'Y', 'o', 'u', ' ', 'a', 'r', 'e', ' ',
               'w', 'e', 'l', ' ', 'c', 'o', 'm', 'e', '\0'};
```

Here string variable *mess* is initialized as sequence of character constants. In this style it is programmer’s job to specify the null character as the last character. In addition to this, as you can see, the size of the array is not specified. *Can you guess what will be the size of mess*? In fact it is the number of character constants specified, including the null character.

Since strings will often be used, the C language provides a short and efficient approach for initializing strings.

For example, the task of above statement can also be accomplished as

```
char mess[] = "You are wel come";
```

Note that in this approach, the compiler automatically appends the null character.

4.4.5 Input/Output of Strings

A string can be read and written. The C language provides library function *gets()* and *puts()* for input and output of string data.

Listing 4.8

```
/*
   Program to perform I/O of string data with string I/O functions
*/
#include<stdio.h>
int main()
{
   char str[30];
   printf( "\nEnter string of length <= 29" );
   printf( "\nand terminate with ENTER key\n\n" );
   gets( str );
   printf( "\nYou have entered\n\n" );
   puts( str );
   return 0;
}
```

Test Run

```
Enter string of length <= 29
and terminate with ENTER key
There is no substitute of hard work.↵
You have entered
There is no substitute of hard work.
```

4.4.6 String Manipulation Functions

Almost every C Compiler provides a large number of library functions for manipulating strings. Table 4.1 lists some frequently used string functions.

Table 4.1: Frequently used string functions

String Function	Operation Performed
`strlen(str)`	Return length of string *str*.
`strcat(str1, str2)`	Concats or joins string *str2* with string *str1*. The result is stored in string *str1*.
`strcpy(str1, str2)`	Copies the contents of string *str2* to string *str1*.
`strcmp(str1, str2)`	Compares the first string *str1* with second string *str2*, and returns the value • < 0, if string *str1* comes earlier in dictionary order than *str2* • = 0, if both strings *str1* and *str2* are same • > 0, if string *str1* comes later in dictionary order than *str2*

String Function	Operation Performed
`strrev(str)`	Reverses the string *str*.
`strlwr(str)`	Converts alphabets in string *str* to lowercase.
`strupr(str)`	Converts alphabets in string *str* to uppercase.

Let us discuss some of these string functions one by one.

4.4.6.1 The *strlen()* Function

This function takes one argument that can be a string constant or a variable. It counts the number of characters present in the string. Do remember that null character '\0' is not a part of the character; it is merely used to mark the end of the string, so it is not counted.

Listing 4.9

```
/*
    Program to illustrate the use of strlen()function
*/
#include <stdio.h>
#include <string.h>
int main()
{
   char str[30];
   printf( "\nEnter string : " );
   gets (str);
   printf( "Length of string = %d\n", strlen(str) );
   return 0;
}
```

Test Run

```
Enter string : Programming
Length of string = 11
```

4.4.6.2 The *strcpy()* Function

This function takes two arguments - first is a string variable and the second can be a string constant or a variable. It copies the character(s) of the second arguments to the first argument.

Listing 4.10

```
/*
   Program to illustrate the use of strcpy()function
*/
#include <stdio.h>
#include <string.h>
int main()
```

```
{
   char str1[30], str2[30];
   printf( "\nEnter string str1 : );
   gets(str1);
   strcpy( str2, str1 );
   printf( "String str2 = %s\n", str2 );
   return 0;
}
```

Test Run

```
Enter string str1 : Hey, how are you?
String str2 = Hey, how are you?
```

4.4.6.3 The *strcat()* Function

This function takes two arguments - first is a string variable and the second can be a string constant or a variable. It appends the character(s) of the second arguments at the end of the first argument.

Listing 4.11

```
/*
    Program to illustrate the use of strcat()function
*/
#include <stdio.h>
#include <string.h>
int main()
{
   char str1[30], str2[30];
   printf( "\nEnter string str1 : " );
   gets(str1);
   printf( "\nEnter string str2 : " );
   gets(str2);
   strcat( str1, str2 );
   printf("String str1 after concatenation with str2 = %s\n", str1);
   return 0;
}
```

Test Run

```
Enter string str1 : naughty
Enter string str2 : boy
String str1 after concatenation with str2 = naughtyboy
```

4.4.6.4 The *strcmp()* Function

This function takes two arguments - both can be string variables or constants. It compares the characters of each but one at a time and returns a value indicating the result of the comparison. For example, in the statement

```
strcmp ( str1, str2 );
```

value returned will have the meanings as explained in Table 4.2.

Table 4.2: Interpretation of value returned *by strcmp()* function

Value Returned	Meaning
< 0	String *str1* comes earlier in dictionary order than *str2*.
= 0	Strings *str1* and *str2* are same (identical).
> 0	String *str1* comes later in dictionary order than *str2*.

Listing 4.12

```
/*
    Program to illustrate the use of strcmp()function
*/
#include <stdio.h>
#include <string.h>
int main()
{
   char str1[30], str2[30];
   int value;
   printf( "\nEnter string str1 : " );
   gets(str1);
   printf( "Enter string str2 : " );
   gets(str2);
   value = strcmp( str1, str2 );
   if ( value > 0 ) {
      printf("\n%s comes after %s", str1, str2);
      printf(" in dictionary order\n" );
   } else if ( value < 0 ) {
      printf("\n%s comes before %s", str1, str2);
      printf(" in dictionary order\n" );
   } else
      printf("\nBoth strings are same\n"  );
   return 0;
}
```

Test Run

```
Enter string str1 : software
Enter string str2 : hardware
hardware comes before software in dictionary order
```

4.4.6.5 The *strrev()* Function

This function takes string variable as its argument. It reverses the order of characters in the string.

Listing 4.13

```
/*
    Program to illustrate the use of strrev()function
*/
#include <stdio.h>
#include <string.h>
int main()
{
   char str[50];
   printf( "\nEnter string : " );
   gets(str);
   strrev( str );
   printf( "\nReverse string = %s\n", str );
   return 0;
}
```

Test Run

```
Enter string : programming
Reverse string = gnimmargorp
```

4.4.6.6 The *strupr()* Function

This function takes string variable as its argument. It converts the alphabets in a string to uppercase.

Listing 4.14

```
/*  Program to illustrate the use of strupr)function  */
#include <stdio.h>
#include <string.h>
int main()
{
   char str[50];
   printf( "\nEnter string in lowercase : " );
   gets(str);
```

```
    strupr( str );
    printf( "\nGiven string in UPPERCASE = %s\n", str );
    return 0;
}
```

Test Run

```
Enter string in lowercase : programming
Given string after conversion to UPPERCASE = PROGRAMMING
```

4.4.6.7 The *strlwr()* Function

This function takes string variable as its argument. It converts the alphabets in a string to lowercase.

Listing 4.15

```
/*
    Program to illustrate the use of strupr)function
*/
#include <stdio.h>
#include <string.h>
int main()
{
   char str[50];
   printf( "\nEnter string in UPPERCASE : " );
   gets(str);
   strlwr( str );
   printf( "\nGiven string in Lowercase = %s\n", str );
   return 0;
}
```

Test Run

```
Enter string in UPPERCASE : PROGRAMMING
Given string in Lowercase = programming
```

4.4.7 Array of Strings

In the last section, we discussed string variables that can store one string value at a time. Suppose we want to handle a list (array) of such strings, then *what is the solution*? The answer to this question can be found by using a two-dimensional array of characters. In two-dimensional arrays, the first row stores the first string; the second row stores the second string, and so on.

Suppose, we want to handle a list of names of 4 persons, where the name of each person can be maximum of 30 characters long, the following will be the required declaration

```
char names[4][31];
```

The following statement shows the way a list of strings can be initialized.

```
char names[4][31] = { "Ram Parkash",
                      "Inder Mohan Singh Sidhu",
                      "Amanpreet",
                      "Rajan"
                    };
```

Listing 4.16

```
/*
   Program to illustrates input/output of array of strings
   represented as two-dimensional array of characters
*/
#include <stdio.h>
int main()
{
    char names[50][31];
    int i, n;
    printf( "Enter number of strings : " );
    scanf( "%d", &n );
    fflush(stdin); /* clear keyboard buffer */
    for ( i = 0; i < n; ++i )
    {
       printf( "Enter string %d: ", i+1 );
       gets( names[i] );
    }
    printf( "\nList of given strings\n\n" );
    for ( i = 0; i < n; ++i )
       puts ( names[i] );
    return 0;
}
```

Test Run

```
Enter number of strings : 5
Enter string 1 : Hari
Enter string 2 : Santosh
Enter string 3 : Balwinder
Enter string 4 : Ram
Enter string 5 : Sham
List of given strings
Hari
Santosh
```

```
Balwinder
Ram
Sham
```

ILLUSTRATIVE EXAMPLES

Example 4.8: *Write a program to find the frequency of a given character in a string.*

Listing 4.17

```
/*
  Program to find the frequency of a given character in a string
*/
#include <stdio.h>
int main()
{
   char str[81], ch;
   int i, count = 0;
   printf("\nEnter a string : ");
   gets(str);
   printf("\nEnter a character to find its frequency : ");
   ch = getchar();
   for (i = 0; str[i] != '\0'; i++)
   {
       if (ch == str[i])
           count++;
   }
   printf("\nFrequency of %c = %d", ch, count);
   return 0;
}
```

Test Run

```
Enter a string : I love programming in C language.
Enter a character to find its frequency : a
Frequency of a = 3
```

Example 4.9: *Write a program to count the number of vowels, consonants, digits, and white-spaces in a string.*

Listing 4.18

```
/*
   Program to count the number of vowels, consonants, digits,
   and white-spaces in a string
```

```
*/
#include <stdio.h>
int main()
{
   char str[80];
   int vowels = 0, consonants = 0, digits = 0, spaces = 0;
   int i;
   printf("\nEnter a line of text : ");
   gets(str);
   for (i = 0; str[i] != '\0'; i++)
   {
       if ( str[i] == 'a' || str[i] == 'A' || str[i] == 'e' ||
            str[i] == 'E' || str[i] == 'i' || str[i] == 'I' ||
            str[i] == 'o' || str[i] == 'O' || str[i] == 'O' ||
            str[i] == 'U' ) {
            vowels++;
       } else if ( (str[i] >='a' && str[i]<='z')
                 ||(str[i]>= 'A' && str[i] <= 'Z') ) {
           consonants++;
       } else if ( str[i] >= '0' && str[i] <= '9' ) {
            digits++;
       } else if ( str[i] == ' ' || str[i] == '\t') {
            spaces++;
       }
   }
   printf("\nVowels       = %d", vowels);
   printf("\nConsonants   = %d", consonants);
   printf("\nDigits       = %d", digits);
   printf("\nWhite spaces = %d", spaces);
   return 0;
}
```

Test Run

```
Enter a line of text : I love programming in C language.
Vowels       = 10
Consonants   = 17
Digits       = 0
White spaces = 5
```

Example 4.10: *Write a program that displays a string starting with a given character.*

Listing 4.19

```
/*
   Program to display strings which start with a given character
*/
#include <stdio.h>
int main()
{
    char str[20][31], ch;
    int i, n;
    printf( "\nEnter number of strings : " );
    scanf( "%d", &n );
    fflush(stdin);
    for ( i = 0; i < n; ++i )
        gets( str[i] );
    printf("\nEnter a character : ");
    ch = getchar();
    printf( "\nStrings that start with character : %c\n", ch );
    for ( i = 0; i < n; ++i ) {
         if ( ch == str[i][0] )
             puts(str[i]);
    }
    return 0;
}
```

Test Run

```
Enter number of strings : 10
Delhi
Dehradun
Hyderabad
Bangaluru
Dalhousie
Nagpur
Darjiling
Jaipur
Agra
Pune
Enter a character : D
```

```
Strings that start with character : D
Delhi
Dehradun
Dalhousie
Darjiling
```

UNIT SUMMARY

In this chapter, we have learned that

- An array is a data structure to store collections of similar values under a given name.
- Elements of an array stored in contiguous memory locations.
- Elements of an array are accessed by the name of the array followed by an index within brackets. One pair of brackets is used for each dimension.
- An array must be declared before use. Declaration tells the compiler about the name of the array, type of each element and the size for each dimension.
- Arrays can be initialized at the time of declaration.
- When an array is partially initialized, the rest of the elements are set to zero.
- A two-dimensional array is a representation of a table (matrix) with rows & columns.
- A multi-dimensional array is an extension of a two-dimensional array to three, four, or more dimensions.
- As such, C language does not support string data type however strings are handled as an array of characters.
- The C language uses null-terminated variable length strings.
- A string constant in C language is a sequence of characters enclosed in double quotes.
- To store a string, we need an array of characters with size one plus than the maximum length string.
- The list of strings is handled by using two-dimensional array of characters.
- There is a rich library of functions for manipulating strings.

EXERCISE

Subjective Questions

1. What is an array?
2. Is it possible to declare and initialize an array in C simultaneously? If yes, how?
3. What are the rules for naming an array?
4. What is the subscript of the first element of the array?
5. How a real array named *data* with 100 elements will be declared? What will be the subscript of the last element?
6. Is it possible to declare more than one array in the same declaration statement?

7. Is the following array declaration is correct? If not, why?

```
int a(50);
```

8. If the declaration segment is

```
#define ROWS  100
float height[ROWS];
```

Is the following segment to enter data in array is correct?

```
for ( i = 0; i <= ROWS; i++ )
    scanf ( "%f", height[i] );
```

9. When an array is passed as an argument to a function, what is actually passed?
10. When an entire array is passed as an argument to a function, the function can alter the values of the elements of the array. We want to prevent the function from doing so, how can this task be accomplished?
11. Is anything wrong with the following declaration? Explain.

```
int a[3][] = { { 1, 2, 3 }, { 3, 2, 1 }, { 4, 5, 2 } };
```

12. How a string is stored in C language?
13. What will be the declaration for array named *str* in which string "*C is just like sea*" is to be stored?
14. What is the difference between "A" and 'A'?
15. Consider the following declaration

```
char name[20];
```

16. What is the maximum length of string which can be stored correctly in name?
17. Consider the following declaration

```
char name[20];
```

Is following the correct way of assigning the string "*Rajan Mehta*" to *name*?

```
name = "Rajan Mehta";
```

Multiple Choice Questions

1. Elements of two-dimensional array are stored in
   (*a*) column major order (*b*) row major order
   (*c*) random order (*d*) None
2. The value within [] in an array declaration specifies
   (*a*) size of the array (*b*) largest permitted subscript value
   (*c*) both (*a*) & (*b*) (*d*) None of the above
3. Given the declaration "int a[10];" Identify the which of the following is wrong?
   (*a*) *a*[–1] (*b*) *a*[0] (*c*) *a*[10] (*d*) ++a
4. When we should use an array?
   (*a*) When we need to hold constants.
   (*b*) When we need to hold data of same type.
   (*c*) When we need to hold data of different type.
   (*d*) When we need to obtain automatic memory cleanup functionality.

5. The use of arrays makes a program _____ .
   (*a*) general and capable of handling class of problems.
   (*b*) difficult to understand.
   (*c*) compact and efficient.
   (*d*) both *a* & *c*.
6. What is the difference between the 5's in the below expressions?

```
int num[5];
num[5]=10;
```

   (*a*) First specifies the array size, the second one specifies an individual element
   (*b*) First specifies the individual element, the second one specifies the size
   (*c*) Both specify the array size
   (*d*) None of the above
7. What would be the output of the program, if the array begins at address 1200?

```
void main()
{
   int a[5]={ 2,4,6,8,10 };
    printf("%u, %d ", a, sizeof(a) );
}
```

   (*a*) 1202, 10 (*b*) 10, 1202
   (*c*) 10, 1200 (*d*) 1200, 10
8. A two dimensional array can be called as a _____ .
   (*a*) Matrix (*b*) Vector
   (*c*) Stack (*d*) queue
9. What will be the output of the following program?

```
void main()
{
   void fun(int x[]);
   int a[] = {1, 2, 3, 4, 5 };
   fun(a);
   printf("\n%d,%d", a[0], a[1]);
   printf("%d,%d,%d\n", a[2], a[3], a[4]);
}
void fun(int x[])
{
   x[2] = x[2] + 2;
   x[4] = x[4] + 4;
}
```

   (*a*) 5,6,7,8,9 (*b*) 2,4,6,8,10
   (*c*) 3,4,5,6,7 (*d*) 1, 2, 5, 4, 9

10. Which would be the output of the following program?

```
void main()
{
   int a[5] = { 1, 2 };
   printf("\n%d,%d,%d\n", a[2], a[3], a[4]);
}
```

(*a*) 0, 0, 0 (*b*) 2, 2, 2 (*c*) 1, 1, 1 (*d*) Garbage

11. What will happen if you assign a value to an array element in a C program whose subscript exceeds the size of the array?

(*a*) Nothing will happen, and the element will get a given value.

(*b*) Compiler will flag a syntax error.

(*c*) Memory location immediately after the last memory location of the array will be overwritten, and this may cause the system to crash.

(*d*) The array size will automatically be increased to accommodate the new value.

12. If the declaration is "`char sample[80];`", what is the length of string that can be correctly represented by *sample*?

(*a*) 80 (*b*) 79 (*c*) 81 (*d*) None

13. Which of the following statement is true about strings in C?

(*a*) Every string must be terminated by a null character ('\0').

(*b*) String is a primitive data type in C.

(*c*) String can be assigned to another string using assignment operator '='.

(*d*) All of the above.

14. What will be the output of the following program?

```
void main()
{
   printf(5+"Fascimile");
}
```

(*a*) Error (*b*) Fascimile (c) mile (*d*) Fasci

15. Which of the following is not correct for creating and initializing a *char* array named *vowels* with string value "aeiou"?

(*a*) `char vowels[] = { 'a', 'e', 'i', 'o', 'u', '\0' };`

(*b*) `char vowels[] = { 'a', 'e', 'i', 'o', 'u', '\n' };`

(*c*) `char vowels[] = "aeiou";`

(*d*) None of the above

16. If two strings *str1* and *str2* are identical, then the value *y* in the following code segment will be _____.

```
int y;
y = strcmp(str1, str2);
```

(*a*) 1 (*b*) −1 (*c*) 0 (*d*) None of the above

17. Which of the following is a correct output of the following program?

```
#include <string.h>
void main() {
   char str[] = "Spider\0man\0";
   printf("%d", strlen(str));
}
```

(*a*) 6 (*b*) 10 (*c*) 11 (*d*) 13

18. Which of the following is a correct output of the following program?

```
void main()
{
   char str[] = "Sales\0man";
   puts(str);
}
```

(*a*) Sales (*b*) man (*c*) Salesman (*d*) Error

19. What will be the output of the following program?

```
int main()
{
     printf("%d", strcmp("ABC","abc"));
     return 0;
}
```

(*a*) 0 (b) 1 (c) -1 (d) Error

20. Which of the following functions is more appropriate to read a multiword string?

(*a*) scanf() (*b*) gets() (*c*) getchar() (*d*) getstring()

ANSWERS															
1.	(*b*)	2.	(*a*)	3.	(*d*)	4.	(*b*)	5.	(*d*)	6.	(*a*)	7.	(*d*)	8.	(*a*)
9.	(*d*)	10.	(*a*)	11.	(*c*)	12.	(*b*)	13.	(*a*)	14.	(*c*)	15.	(*b*)	16.	(*c*)
17.	(*a*)	18.	(*a*)	19.	(*c*)	20.	(*b*)								

Programming Problems

1. Write a program that accepts an array, interchanges the first element with the last element, the second element with the second last element, and so on, and finally prints the new array.
2. Write a program to read a set of n integers between 1 and 10 and count the number of times each integer appears in the set. Print the integers in order of frequency, the number occurring the highest number of times first.
3. A department store chain has m ($\leq$10) stores, and each store has the same n ($\leq$15) departments. The weekly sales of the chain are stored in $m \times n$ array *SALES* (say). Write down the program which
   - Prints the total weekly sales of each store.
   - Prints the total weekly sales of each department.
   - Prints the total weekly sales of the chain.
4. Write a program that inserts an element d in kth position of an array a of size n.

5. Write a program that removes an element from the *k*th position of an array *a* of size *n*.
6. Write a program to remove duplicate elements from an array.
7. Given a matrix *A* of order $m \times n$, write a program to find the row having the maximum number of elements that are even.
8. Write a program that asks the user to enter a sentence and then splits out the words in the sentence and puts them in a table.
9. Write a program that asks the user for a list of words and prints the list in reverse order.
10. Suppose you translate a number into a string of digits spelled out, one word for each digit, followed by a single space. For example, the number 407 become the digit string: "*Four Zero Seven*". Write a program to accept a positive number and print out its digit string.
11. Write a program that accepts an amount in figures and prints that in words. For example, for an amount of 12500 it should output the string *Twelve Thousand Five Hundred only*.
12. Write a program to find third largest element in an unsorted array.

PRACTICALS

1. You are given a 1D arrays, named *x*, that stores data of some survey with $n(\leq 50)$ observations. Write a program to compute the mean, variance, and standard deviation of the data.

Sol. The formulae to compute mean, variance, and standard deviation are

$$\text{mean } (\bar{x}) = \frac{\sum_{i=1}^{n} x_i}{n}, \quad \text{variance} = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n}, \quad \text{standrad deviation} = \sqrt{\text{variance}}$$

Listing 4.20

```
 /*
    Program to find mean, variance, and standard deviation
*/
#include <stdio.h>
#include <math.h>
int main()
{
    float x[50];
    int  i, n;
    float mean, variance, std_deviation, sum, d;
    printf("\nEnter value for n : ");
    scanf("%d", &n);
    printf("\nEnter %d values\n", n);
    for ( i = 0; i < n; i++ ) {
        scanf("%f", &x[i]);
    }
    /*  Compute the sum of all elements */
    sum = 0;
    for ( i = 0; i < n; i++ ) {
        sum = sum + x[i];
```

```
    }
    /*  Compute mean (average) */
    mean = sum / n;
    /*  Compute  variance  */
    sum = 0;
    for ( i = 0; i < n; i++ ) {
        d = x[i]-mean;
        sum = sum + d * d;
    }
    variance = sum / n;
    /*  Compute  standard deviation  */
    std_deviation = sqrt(variance);
    /*  print results of computations, rounded to 2 decimals  */
    printf("Mean = %.2f\n", mean);
    printf("Variance = %.2f\n", variance);
    printf("Standard deviation = %.2f\n", std_deviation);
}
```

Test Run

```
Enter value for n : 10

Enter 10 values
12.5
10
15
25.75
30
22
16
19
24
11

Mean = 18.52
Variance = 41.06
Standard deviation = 6.41
```

2. Write a program to add matrices $A(m \times n)$ and $B(m \times n)$.

Refer to Example 4.6.

3. Write a program to multiply matrix $A(m \times n)$ by $B(p \times q)$.

Refer to Example 4.7.

4. Write a program to test whether the given word is palindrome or not.

A word is a palindrome if reads same from both ends. Examples are words like "*nitin*", "*radar*", "*madam*", "*refer*", etc.

To test a word to see whether it is palindrome, one approach is to use the *strrev()* library function to reverse a string, and then compare the original string with the reversed string. If both are same, then the word is palindrome otherwise the word is not a palindrome.

Listing 4.21

```
/*
    Program to check whether given word is palindrome or not,
    using library function strrev()
*/
#include<stdio.h>
#include<string.h>
int main()
{
    char str[30], temp[30];
    printf("\nEnter any word : ");
    gets(str);
    strcpy(temp, str);
    strrev(temp);
    printf("\nGiven word = %s\n", str);
    printf("\n%s after reversing = %s\n", str, temp);
    if ( strcmp(str, temp) == 0 )
        printf("\n%s is a palindrome word.\n", str);
    else
        printf("\n%s is not a palindrome word.\n", str);

    return 0;
}
```

Test Runs

First Run

```
Enter any word : nitin
Given word = nitin
nitin after reversing = nitin
nitin is a palindrome word.
```

Second Run

```
Enter any word : never
Given word = never
never after reversing = reven
never is not a palindrome word.
```

The second approach is that we take two index variables, say *i* and *j*, *i* is initialized with the index of the first character and *j* is initialized with the index of the last character of the word.

We iterate till *i* < *j*, and in each iteration, we compare the character at index *i* and at index *j* to find the first mismatch. If a mismatch is found, then the word is not a palindrome.

Further, in each iteration, the index *i* is incremented and index *j* is decremented.

If no mismatch is found and the condition *i* = *j* is reached, it will indicate that the given word is a palindrome.

Listing 4.22

```
/*
    Program to check whether given word is palindrome or not,
    without using any string manipulation function
*/
#include<stdio.h>
#include<string.h>
int main()
{
    char str[30];
    int i = 0, j, n = 0;
    printf("\nEnter any word : ");
    gets(str);
    /*  to get length of word in variable 'n'  */
    while ( str[n] != '\0' )
        n++;
    i = 0;      /* index of first character  */
    j = n-1;    /* index of last character   */
    while ( i < j )
    {
        if ( str[i] != str[j] ) {
            printf("\n%s is not a palindrome word.\n", str);
            return 0;
        }
        i++;
        j--;
    }
    printf("\n%s is a palindrome word.\n", str);
    return 0;
}
```

Test Runs

First Run

```
Enter any word : madam
madam is a palindrome word.
```

Second Run

```
Enter any word : india
india is not a palindrome word.
```

KNOW MORE

The topic of arrays is one of the important aspects of a programming language. There are numerous real-life problems that require handling large collection of data, and arrays are the ideal data structures to store such data in a computer program.

The teacher is expected to understand the concepts of arrays, need of arrays, and the various aspects related to handling of arrays in C language.

The teacher should demonstrate the use of arrays by taking examples from real-life situations and creating C programs to solve them.

REFERENCES & SUGGESTED READINGS

1. R. S. Salaria, Problem Solving & Programming in C, Khanna Book Publishing Co(P) Ltd., New Delhi.
2. E. Balagurusamy, Programming in ANSI C, Tata McGraw Hill, New Delhi..
3. Yashavant Kanetkar, Let Us C, BPB Publications, New Delhi.
4. Byron Gottfried, Programming with C, Schaum's Outlines.
5. https://onlinecourses.nptel.ac.in/noc21_cs01/preview
6. https://ocw.mit.edu/courses/intro-programming/
7. https://www.programiz.com/c-programming
8. https://www.javatpoint.com/c-programming-language-tutorial

5 Basic Algorithms

UNIT SPECIFICS

This unit discusses the topics related to searching, sorting, and finding solution of an equation. The working of various searching and sorting are explained with suitable examples in this unit. In addition, the process of finding the solution of an equation using Bisection method is also explained.

RATIONALE

There are many situations in real-life where we have to search for a particular piece of information. If the information is not organized the only way to check each and every item of information till we find that one or we may end up that item is not found. However, if the information is organized in a particular order, then we can search any item very quickly using efficient algorithms.

This unit helps students to understand the various techniques to arrange (sort) information in desired order, the various techniques to search a given item of information.

In addition, this unit also explains how we can find the roots of an equation.

PRE-REQUISITES

- Conditional branching
- Looping/Iteration
- Arrays

UNIT OUTCOMES

Upon completion of the unit, students will be able to

U5-O1: explain and implement various searching algorithms

U5-O2: explain and implement various sorting algorithms

U5-O3: explain and implement the method of finding roots of an equation

Unit 5 Outcomes	EXPECTED MAPPING WITH COURSE OUTCOMES (1- Weak Correlation; 2- Medium correlation; 3- Strong Correlation)							
	CO-1	CO-2	CO-3	CO-4	CO-5	CO-6	CO-7	CO-8
U5-O1	3	3	-	-	-	3	-	-
U5-O2	3	3	-	-	-	3	-	-
U5-O3	3	3	-	-	-	3	-	-

5.1 SEARCHING ALGORITHMS

Searching is the process of finding the location of given element in an array. The search is said to be successful if the given element is found, *i.e.*, the element does exists in the array; otherwise unsuccessful.

There are two approaches to search operation:

- Linear search
- Binary search

The algorithm that one chooses generally depends on organization of the array elements. If the elements are in random order, then one have to use linear search technique, and if the array elements are sorted, then it is preferable to use binary search technique. These two search techniques are described in the next sections.

5.1.1 Linear Search

Given no information about the array *a*, the only way to search for given element *item* is to compare *item* with each element of *a* one by one. This method, which traverses *a* sequentially to locate *item* is called *linear search* or *sequential search*.

Listing 5.1

```
/*
*   Program to demonstrate the use of linear search technique
*   to search a given element in an un-sorted array
*/
#include <stdio.h>
int main()
{
   int a[50], i, n, item, flag;
   printf("\nEnter size of array n(<=50) : " );
   scanf("%d", &n );
   printf("\nEnter %d elements of array\n", n);
   for ( i = 0; i < n; i++ )
      scanf( "%d", &a[i] );
   printf( "\nEnter element to search : " );
   scanf( "%d", &item );
   flag = 0;
   for ( i = 0; i < n; i++ )
   {
      if( item == a[i] )      /* match found  */
      {
         flag = 1;
         break;
      }
   }
```

```
    if ( flag == 1 )
       printf( "\nElement found at index: %d\n", i );
    else
       printf( "\nElement not found...\n" );
    return 0;
}
```

Test Run - 1

```
Enter size of array n(<=50) : 10
Enter 10 elements of array
12 5 18 9 11 10 25 36 22 100
Enter element to search : 11
Element found at index: 4
```

Test Run - 2

```
Enter size of array n(<=50) : 10
Enter 10 elements of array
12 5 18 9 11 10 25 36 22 100
Enter element to search : 55
Element not found...
```

Complexity Analysis

In the best possible case, the *item* may occur at first position. In this case, the search operation terminates in success with just one comparison. However, the worst case occurs when either the item is present at last position or missing from the array. In the former case, the search terminates in success with n comparisons. In the later case, the search terminates in a failure with n comparisons. Thus, we find that in worst case the linear search needs O(n) operation.

5.1.2 Binary Search

Suppose the elements of the array A are sorted in ascending order (if the elements are numbers) or dictionary order (if the elements are strings). The best searching algorithm, called *binary search,* is used to find the location of the given element.

We do use this approach in our daily life. For example, suppose we want to find the meaning of the term in a computer dictionary. Obviously, we don't search page by page. We open the dictionary in the middle (roughly) to determine which half contains the term being sought. Then for the subsequent search one half is discarded, and we search in the other half. This process is continued till either we have located the required term or that term is missing from the dictionary, which will indicated by the fact that at the end we will be left with only one page.

Example 5.1: *To illustrate the working of the binary search technique, consider the following sorted array A with 7 elements*

3, 10, 15, 20, 35, 40, 60

and we want to search element 15.

Solution:

	a[0]	*a*[1]	*a*[2]	*a*[3]	*a*[4]	*a*[5]	*a*[6]
Given Array *a*	3	10	15	20	35	40	60

To start with, we take *beg* = 0, *end* = 6, and compute location of middle element as

mid = (*beg* + *end*) / 2 = (0 + 6) / 2 = 3

Since *a*[*mid*], *i.e.*, *a*[3] ≠ 15, and *beg* <= *end*. We start next iteration.

As *a*[*mid*] = 20 > 15, therefore, we take *end* = *mid* − 1 = 3 − 1 = 2, whereas *beg* remains unchanged. Compute location of middle element

mid = (*beg* + *end*) / 2 = (0 + 2) / 2 = 1

Since *a*[*mid*], *i.e.*, *a*[1] ≠ 15, and *beg* <= *end*. We start next iteration.

As *a*[*mid*] = 10 < 15, therefore, we take *beg* = *mid* + 1 = 1 + 1 = 2, whereas *end* remains unchanged.

Since *beg* = *end,* compute location of the middle element

mid = (*beg* + *end*) / 2 = (2 + 2) / 2 = 2

Since *a*[*mid*], *i.e.*, *a*[2] = 15, the search terminates on success.

The element is found at index 2.

Listing 5.2

```
/*
*   Program to demonstrate the use of binary search technique
*   to search a given element in a sorted array
*/
#include <stdio.h>
int main()
{
   int a[50], i, n, item, beg, end, mid, flag;
   printf("\nEnter size of array n(<=50) : " );
   scanf("%d", &n );
   printf("\nEnter %d elements of array in ascending order\n", n);
   for ( i = 0; i < n; i++ )
      scanf( "%d", &a[i] );
   printf( "\nEnter element to search : " );
   scanf( "%d", &item );
   flag = 0;
   beg = 0;
   end = n - 1;
   while ( beg < end )
   {
      mid = ( beg + end ) / 2;
      if( item == a[mid] ) {     /* match found  */
         flag = 1;
         break;
      }
```

```
        if( item < a[mid] )
           end = mid - 1;
        else
           beg = mid + 1;
     }

     if ( flag == 1 )
        printf( "\nElement found at index: %d\n", mid );
     else
        printf( "\nElement not found...\n" );
     return 0;
}
```

Test Run - 1

```
Enter size of array n(<=50) : 10↵
Enter 10 elements of array in ascending order
5 9 10 11 12 18 25 36 44 100
Enter element to search : 44
Element found at index: 8
```

Test Run - 2

```
Enter size of array n(<=50) : 10
Enter 10 elements of array in ascending order
5 9 10 11 12 18 25 36 44 100
Enter element to search : 35
Element not found...
```

Complexity Analysis

In each iteration, the search is reduced to one half of the array. Therefore, for n elements in the array, there will be $\log_2 n$ iterations. Thus, the complexity of binary search is $O(\log_2 n)$. This complexity will be same irrespective of the position of the element, even if it is not present in the array.

5.2 SORTING ALGORITHMS

Sorting is the process of arranging the elements in some logical order. This logical order may be ascending or descending in case of numeric values or dictionary order in case of alphanumeric values.

We do, almost, use this process daily in our routine activities. For example, we do arrange our class notes in increasing order of date in order to refer them quickly later on. Like this there are many situations, which you can enumerate.

In this section, we sill discuss following sorting algorithms:

- Bubble sort
- Insertion sort
- Selection sort

We will consider an array named *a* of size *n* whose elements are of type *int* for the discussion of all these algorithms.

5.2.1 Bubble Sort

The Bubble sort method requires $(n-1)$ pass to sort an array. In each pass, every element $a[i]$ is compared with $a[i+1]$, for $i = 0$ to $(n-k)$, where k is the pass number, and if they are out of order, *i.e.*, if $a[i] > a[i+1]$, they are swapped. This will cause the largest element move or bubble up.

After the end of the first pass, the largest element in the array will be placed in $(n-1)$th position, and on each successive pass, the next largest element is placed at position $(n-2)$, $(n-3)$, ..., 1, respectively.

For more clarity, carefully examine the following steps.

Pass 1:

Step 1. If $a[0] > a[1]$ then swap $a[0]$ and $a[1]$.
Step 2. If $a[1] > a[2]$ then swap $a[1]$ and $a[2]$.
⋮
Step n-1. If $a[n-2] > a[n-1]$ then swap $a[n-2]$ and $a[n-1]$.

Pass 2:

Step 1. If $a[0] > a[1]$ then swap $a[0]$ and $a[1]$.
Step 2. If $a[1] > a[2]$ then swap $a[1]$ and $a[2]$.

⋮
Step n-2. If a$[n-3] > a[n-2]$ then swap $a[n-3]$ and $a[n-2]$.
⋮

Pass *k*:

Step 1. If $a[0] > a[1]$ then swap $a[0]$ and $a[1]$.
Step 2. If $a[1] > a[2]$ then swap $a[1]$ and $a[2]$.
⋮
Step n-k. If $a[n-k+1] > a[n-k]$ then swap $a[n-k+1]$ and $a[n-k]$.
⋮

Pass *n*-1:

Step 1. If $a[0] > a[1]$ then swap $a[0]$ and $a[1]$.

After $(n-1)$ passes, the array will be sorted in ascending order.

Example 5.2: *To illustrate the working of bubble sort method, consider the sorting of the following array in ascending order.*

12 40 3 2 15

Solution: Note that, the output of a given pass becomes the input for the next pass. The sorting process is self explanatory. In each pass, one element, shown as shaded, reaches to its final position.

Given array can be shown as

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$
12	40	3	2	15

Pass-1:

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$
(12)	(40)	3	2	15

(a) Compare $a[0]$ and $a[1]$. Since $a[0] < a[1]$ $(12 < 40)$, no action is taken.

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$
12	(40)	(3)	2	15

(*b*) Compare $a[1]$ and $a[2]$. Since $a[1] > a[2]$ (40 > 3), swap these elements.

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$
12	3	(40)	(2)	15

(*c*) Compare $a[2]$ and $a[3]$. Since $a[2] > a[3]$ (40 > 2), swap these elements.

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$
12	3	2	(40)	(15)

(*d*) Compare $a[3]$ and $a[4]$. Since $a[3] > a[4]$ (40 > 15), swap these elements.

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$
12	3	2	15	40

Thus, after first pass, the largest element, 40, has moved to its final position. However, the rest of the numbers, which may have changed their positions, are yet to be sorted. The shaded part shows the elements that have been sorted.

For the remaining passes, only comparisons and swaps, if required, are shown.

Pass-2:

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Action
(12)	(3)	2	15	40	Swap
3	(12)	(2)	15	40	Swap
3	2	(12)	(15)	40	No swap
3	2	12	15	40	

After second pass, the second largest element, 15, has moved to its final position.

Pass-3:

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Action
(3)	(2)	12	15	40	Swap
2	(3)	(12)	15	40	No swap
2	3	12	15	40	

After third pass, the third largest element, 12, has moved to its final position.

Pass-4:

a[0]	a[1]	a[2]	a[3]	a[4]	Action
(2)	(3)	12	15	40	No swap
(2)	3	12	15	40	

After fourth pass, the fourth largest element, 3, has moved to its final position. Thus, the array is completely sorted in 4 passes.

The above sorting process can also be visualized as shown below:

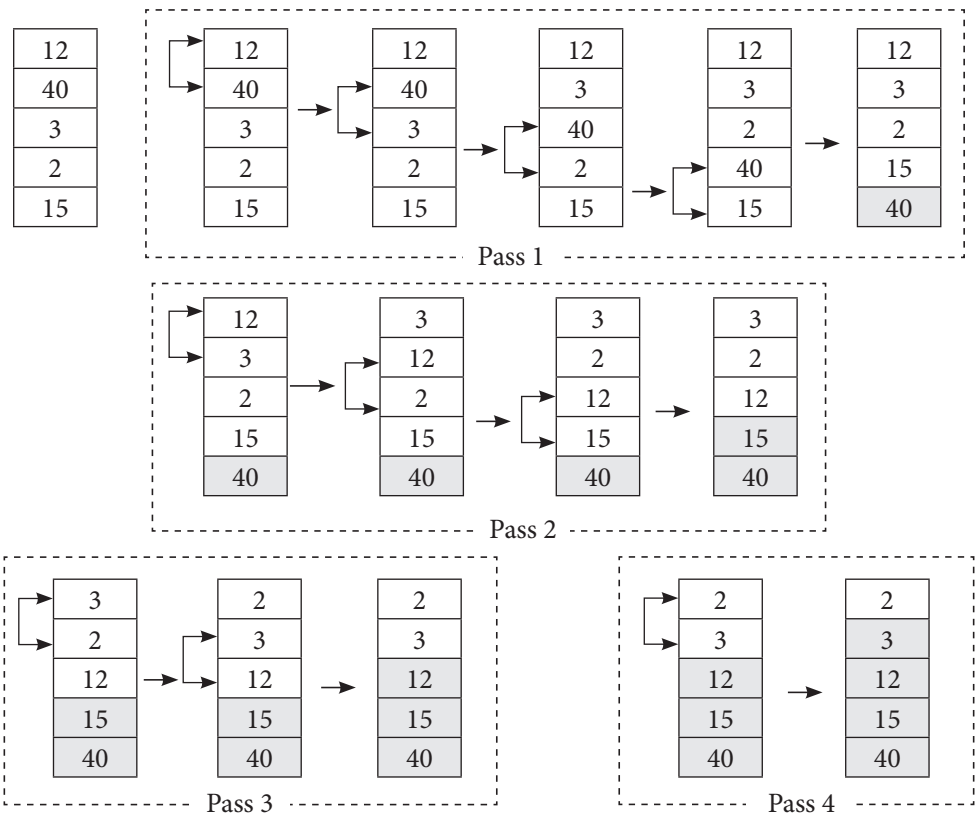


Fig. 5.1: Illustration of Bubble sort method

Complexity Analysis

After (n–1) passes, the array will be sorted in ascending order. As you must have noticed, the first pass requires (n–1) comparisons, the second pass requires (n–2),..., kth pass requires (n–k), and the last pass requires only one comparison. Therefore, total comparisons are

$$f(n) = (n–1) + (n–2) + (n–3) + \ldots + (n–k) + \ldots + 3 + 2 + 1 = n(n–1)/2 = O(n^2)$$

Listing 5.3

```
/*
*   Program to demonstrate the use of Bubble sort method
*   to sort a given array in ascending order
*/

#include <stdio.h>

int main()
{
   int a[50], i, j, n, temp;
   printf("\nEnter size of array n(<=50) : " );
   scanf("%d", &n );
   printf("\nEnter %d elements of array\n", n);
   for ( i = 0; i < n; i++ )
      scanf( "%d", &a[i] );
   for ( i = 0; i < n-1; i++ )
   {
      for ( j = 0; j < n-i-1; j++ )
      {
         if ( a[j] > a[j+1] ) {
             temp = a[j];
             a[j] = a[j+1];
             a[j+1] = temp;
         }
      }
   }
   printf( "\nArray after sorting...\n\n" );
   for ( i = 0; i < n; i++ )
       printf( "%d ", a[i] );
   printf( "\n" );
   return 0;
}
```

Test Run

```
Enter size of array n(<=50) : 10
Enter 10 elements of array
5 12 10 15 22 18 2 -10 7 16
Array after sorting...
-10 2 5 7 10 12 15 16 18 22
```

5.2.2 Selection Sort

Selection sort method also requires (n–1) pass to sort an array. In the first pass, find the smallest element from elements $a[0]$, $a[1]$, $a[2]$, . . ., $a[n-1]$ and swap with the first element, *i.e*, $a[0]$. In the second pass, find the smallest element from elements $a[1]$, $a[2]$, $a[3]$, . . ., $a[n-1]$ and swap with $a[1]$. And so on.

For more clarity, carefully examine the following steps:

Pass 1:

- Find the location *loc* of the smallest element in the entire array, *i.e.*, $a[0]$, $a[1]$, $a[2]$,..., $a[n-1]$.
- Interchange $a[0]$ & $a[loc]$. Then $a[0]$ is trivially sorted.

Pass 2:

- Find the location *loc* of the smallest element in the sub array $a[1]$, $a[2]$, $a[3]$,. . ., $a[n-1]$.
- Interchange $a[1]$ & $a[loc]$. Then elements $a[0]$ and $a[1]$ are sorted, since $a[0] \leq a[1]$.

⋮

Pass *k*:

- Find the location *loc* of the smallest element in the sub array $a[k]$, $a[k+1]$, $a[k+2]$,.. ., $a[n-1]$.
- Interchange $a[k]$ & $a[loc]$. Then elements $a[0]$, $a[1]$, $a[2]$, . . ., $a[k]$ are sorted.

⋮

Pass *n*–1:

- Find the location *loc* of the smaller of the elements $a[n-2]$, $a[n-1]$.
- Interchange $a[n-2]$ & $a[loc]$. Then elements $a[0]$, $a[1]$, $a[2]$, . . ., $a[n-1]$ are sorted.

After (n–1) passes, the array will be sorted in ascending order.

Example 5.3: To illustrate the working of the selection sort method, consider the following array a with 7 elements as

20, 35, 40, 100, 3, 10, 15

Solution: In each iteration i, we find the location *loc* of the smallest element in the unsorted part of the array. If $loc \neq i$ then element at i and *loc* are swapped.

Given Array a

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$
20	35	40	100	3	10	15

Pass 1: $i = 0$, $loc = 4$

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$
20	35	40	100	3	10	15

Interchange elements $a[0]$ & $a[4]$, *i.e.*, 20 & 3 to obtain following array.

Pass 2: $i = 1$, $loc = 5$

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$
3	35	40	100	20	10	15

Interchange elements $a[1]$ & $a[5]$, *i.e.*, 35 & 10 to obtain following array.

Pass 3: $i = 2$, $loc = 6$

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$
3	10	40	100	20	35	15

Interchange elements $a[2]$ & $a[6]$, *i.e.*, 40 & 15 to obtain following array.

Pass 4:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	10	15	100	20	35	40

$i = 3$
$loc = 4$

Interchange elements $a[3]$ & $a[4]$, *i.e.*, 100 & 20 to obtain following array.

Pass 5:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	10	15	20	100	35	40

$i = 4$
$loc = 5$

Interchange elements $a[4]$ & $a[5]$, *i.e.*, 100 & 35 to obtain following array.

Pass 6:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	10	15	20	35	100	40

$i = 5$
$loc = 6$

Interchange elements $a[5]$ & $a[6]$, *i.e.*,100 & 40 to obtain following sorted array.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	10	15	20	35	40	100

Fig. 5.2: Illustration of selection sort method

The shaded part shows the part of the array sorted so for after each iteration. In each iteration, one element is added on the right of the sorted part.

To implement the selection sort algorithm, we need a routine to find the location *loc* of the smallest element among the elements $a[k-1]$, $a[k+1]$, $a[k+2]$,. . ., $a[n-1]$, during kth pass.

Listing 5.4

```
/*
*   Program to demonstrate the use of selection sort method
*   to sort a given array in ascending order
*/
#include <stdio.h>
int main()
{
   int a[50], i, j, n, temp, min, loc;
   printf("\nEnter size of array n(<=50) : " );
   scanf("%d", &n );
   printf("\nEnter %d elements of array\n", n);
   for ( i = 0; i < n; i++ )
      scanf( "%d", &a[i] );
   for ( i = 0; i < n-1; i++ )
   {
      min = a[i];
      loc = i;
      for ( j = i+1; j < n; j++ )
```

```
        {
            if ( a[j] < min ) {
                min = a[j];
                loc = j;
            }
        }
        if ( loc != i ) {
            temp = a[i];
            a[i] = a[loc];
            a[loc] = temp;
        }
    }
    printf( "\nArray after sorting...\n\n" );
    for ( i = 0; i < n; i++ )
        printf( "%d ", a[i] );
    printf( "\n" );
    return 0;
}
```

Test Run

```
Enter size of array n(<=50) : 6
Enter 6 elements of array
12 5 10 15 22 18
Array after sorting...
5 10 12 15 18 22
```

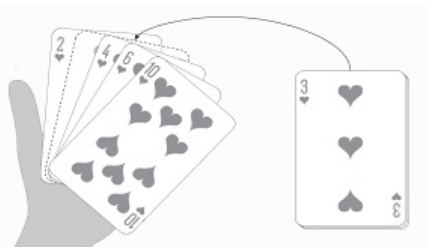
Complexity Analysis

As you must have noticed, the first pass requires (n–1) comparisons to find the location *loc* of smallest element, the second pass requires (n–2), ..., kth pass requires (n–k), and the last pass requires only one comparison.

Therefore total comparisons are

$$f(n) = (n-1) + (n-2) + (n-3) + \ldots + (n-k) + \ldots + 3 + 2 + 1$$
$$= n(n-1)/2 = O(n^2)$$

5.2.3 Insertion Sort



This algorithm is very popular with bridge players when they first sort their cards.

In this procedure, we pick up a particular value and then insert it at the appropriate place in the sorted sub list, *i.e.*, during kth iteration the element $a[k]$ is inserted in its proper place in the sorted sub array $a[1]$, $a[2]$, $a[3]$,. . ., $a[k-1]$.

This task is accomplished by comparing $a[k]$ with $a[k-1]$, $a[k-2]$, $a[k-3]$, and so on until the first element $a[j]$ such that $a[j] \leq a[k]$ is found. Then each of the elements $a[k-1]$, $a[k-2]$, . . ., $a[j+1]$ are moved one position up, and then element $a[k]$ is inserted in $(j+1)$st position in the array.

For more clarity, carefully examine the following steps.

Pass 1: $a[1]$ is inserted either before or after $a[0]$ so that $a[0]$ and $a[1]$ are sorted.

Pass 2: $a[2]$ is inserted either before $a[0]$ or between $a[0]$ and $a[1]$ or after $a[1]$ so that the elements $a[0]$, $a[1]$, $a[2]$ are sorted.

Pass 3: $a[3]$ is inserted either before $a[0]$ or between $a[0]$ and $a[1]$ or between $a[1]$ and $a[2]$ or after $a[2]$ so that the elements $a[0]$, $a[1]$, $a[2]$, $a[3]$ are sorted.

⋮

Pass *k*: $a[k]$ is inserted in proper place in the sorted sub array $a[0]$, $a[1]$, $a[2]$, . . ., $a[k-1]$ so that after insertion, the elements $a[0]$, $a[1]$, $a[2]$, . . ., $a[k-1]$, $a[k]$ are sorted.

⋮

Pass *n*–1: $a[n-1]$ is inserted in proper place in the sorted sub array $a[0]$, $a[1]$, $a[2]$, . . ., $a[n-2]$, so that after insertion, the elements $a[0]$, $a[1]$, $a[2]$, . . ., $a[n-1]$ are sorted.

Thus, after $(n-1)$ passes, the array will be sorted in ascending order.

Example 5.4: *To illustrate the working of the insertion sort method, consider the following array a with 7 elements as*

35, 20, 40, 100, 3, 10, 15

Solution: Given array *a*

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$
35	20	40	100	3	10	15

Pass 1:

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$
35	20	40	100	3	10	15

Since $a[1] < a[0]$, insert element $a[1]$ before $a[0]$ giving the following array.

Pass 2:

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$
20	35	40	100	3	10	15

Since $a[2] > a[1]$, no action is performed.

Pass 3:

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$
20	35	40	100	3	10	15

Since $a[3] > a[2]$, again no action is performed.

Pass 4:

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$
20	35	40	100	3	10	15

Since $a[4]$ is less than $a[3]$, $a[2]$, $a[1]$ as well as $a[0]$, therefore insert $a[4]$ before $a[0]$, giving the following array.

Pass 5:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	20	35	40	100	10	15

Since *a*[5] is less than *a*[4], *a*[3], *a*[2], and *a*[1], therefore insert *a*[5] before *a*[1], giving the following array.

Pass 6:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	10	20	35	40	100	15

Since *a*[6] is less than *a*[5], *a*[4], *a*[3], and *a*[2], therefore insert *a*[6] before *a*[2], giving the following sorted array.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	10	15	20	35	40	100

Fig. 5.3: Illustration of insertion sort method

Listing 5.5

```
/*  Program to demonstrate the use of insertion sort method
*   to sort a given array in ascending order
*/
#include <stdio.h>
void main()
{
   int a[50], i, j, k, n, temp;
   printf("\nEnter size of array n(<=50) : " );
   scanf("%d", &n );
   printf("\nEnter %d elements of array\n", n);
   for ( i = 0; i < n; i++ )
      scanf( "%d", &a[i] );
   for ( k = 1; k < n; k++ ) {
      temp = a[k];
      j = k - 1;
      while ( ( temp < a[j] ) && ( j >= 0 ) ) {
          a[j+1] = a[j];
          j--;
      }
      a[j+1] = temp;
   }
   printf( "\nArray after sorting...\n\n" );
   for ( i = 0; i < n; i++ )
       printf( "%d ", a[i] );
   printf( "\n" );
}
```

Test Run

```
Enter size of array n(<=50) : 7
Enter 7 elements of array
5 12 10 15 22 18 16
Array after sorting...
5 10 12 15 16 18 22
```

5.3 FINDING ROOT OF AN EQUATION

In general, there are two types of methods to find roots of an equation.

Direct methods

Direct methods give the roots of an equation in a finite number of steps. In addition, these methods are capable of giving all the roots at the same time.

For example, the roots of the quadratic equation

$$ax^2 + bx + c = 0$$

where $a \neq 0$

are given by $$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Iterative methods

Iterative methods, also known as *trial and error* methods, are based on the idea of successive approximations. They start with one or more initial approximations to the root and obtain a sequence of approximations by repeating a fixed sequence of steps till the solution with reasonable accuracy is obtained. Iterative methods, generally, give one root at a time.

Iterative methods are very cumbersome and time-consuming for solving equations manually. However, they are best suited for use on computers, due to following reasons:

1. Iterative methods can be concisely expressed as computational algorithms.
2. It is possible to formulate algorithms that can handle class of similar problems. For example, an algorithm can be developed to solve a polynomial equation of degree n.
3. Round-off errors are negligible in iterative methods as compared to direct methods.

The following are popular iterative methods to find root of an equation.

- Bisection Method
- Regula-falsi method
- Secant method
- Newton-Raphson method
- Method of Successive approximations

We will learn about Bisection method to find the root of an equation of type

$$f(x) = 0$$

in this section.

Bisection method is one of the simplest iterative methods. To start with, two initial approximations, say x_1 and x_2 such that $f(x_1) \times f(x_2) < 0$, which ensures that root lies between x_1 and x_2, are taken. Next x-value, say x_3, as mid-point of interval $[x_1, x_2]$ is computed.

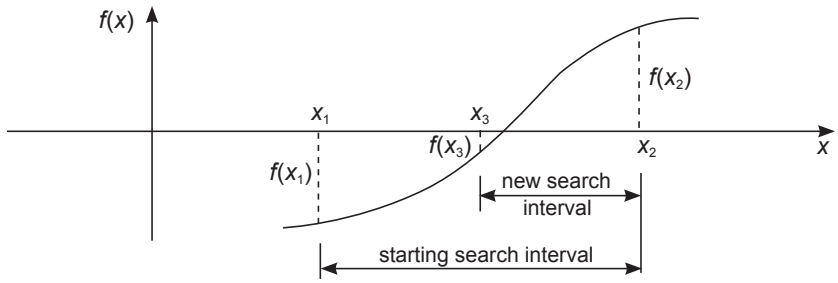


Fig. 5.4: Root approximation by Bisection method

There are three possibilities that can arise:

(*i*) If $f(x_3) = 0$, then we have a root at x_3.

(*ii*) If $f(x_1)$ and $f(x_3)$ are of opposite sign, then the root lies in the interval (x_1, x_3). Thus x_2 is replaced by x_3, and the new interval, which is half of the current interval, is again bisected.

(*iii*) If $f(x_1)$ and $f(x_3)$ are of same sign, then the root lies in the interval (x_3, x_2). Thus x_1 is replaced by x_3, and the new interval is again bisected.

Therefore by repeating this interval bisection procedure, we keep enclosing the root in a new search interval, which is halved in each iteration.

This iterative cycle is terminated when the search interval becomes smaller than the prescribed tolerance (error permitted in the root). Hence, if *epsilon* is the prescribed tolerance in the required root, then the iterative cycle terminates when the absolute error becomes less than or equal to *epsilon*, *i.e.*,

$$|x_1 - x_2| \leq epsilon$$

We take the mid-point of the last search interval as the desired approximation to the root.

Listing 5.6

```
/*
*   Program to implement Bisection method to find one of the
*   root of equation x^3 - 4x - 9 = 0
*/
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
double f( double x )
{
   return pow((double)x,(double)3.0)-4*x-9;
}
int main()
{
   float x1, x2, epsilon, x3;
   printf("Enter first point of the search interval : ");
   scanf("%f", &x1);
   printf("Enter second point of the search interval : ");
   scanf("%f", &x2);
```

```
    if ( ( f(x1) * f(x2) ) > 0 ) {
       printf("\nInitial approximations are unsuitable\n");
       return 1;
    }
    printf("Enter prescribed tolerance : ");
    scanf("%f", &epsilon);
    do {
       x3 = (x1+x2)/2;
       if ( f(x1) * f(x3) < 0 )
            x2 = x3;
       else
            x1 = x3;
    }
    while( fabs((double)(x1-x2)) > epsilon);
    printf("\nApproximate root = %8.4f\n", x3);
    return 0;
}
```

Test Runs

Test Run 1

```
Enter first point of the search interval : 2.5
Enter second point of the search interval : 2.6
Initial approximations are unsuitable
```

Test Run 2

```
Enter first point of the search interval : 2.6
Enter second point of the search interval : 2.8
Enter prescribed tolerance : 0.0001
Approximate root = 2.7065
```

UNIT SUMMARY

In this chapter, we have learned that

- Searching is the process of finding the location of given element in an array.
- Various searching techniques include linear search and binary search.
- If the array is un-sorted, only choice of searching an element is linear search. However, if the array is sorted, binary search is a better choice.
- Sorting is the process of arranging the elements in some logical order.
- Various sorting techniques include bubble sort, selection, and insertion sort.
- Among other methods, Bisection method is one of the simplest method to find of root at a time of polynomial equation of type $f(x) = 0$.

EXERCISE

Subjective Questions

1. What is searching?
2. Describe the linear search method with suitable example.
3. Describe the binary search method with suitable example.
4. What is sorting? What is the need of sorting?
5. Describe the bubble sort method with suitable example.
6. Describe the selection sort method with suitable example.
7. Describe the insertion sort method with suitable example.
8. Describe the Bisection method to find root of an equation.
9. Given the elements of an array are given as 12, 7, 13, 9, 10, 77, 2, 8. What will be the arrangement of elements after first pass of the bubble sort method?
10. Given the elements of an array are given as 11, 70, 13, 99, 5, 17, 21, 38. What will be the arrangement of elements after three passes of the selection sort method?
11. Given the elements of an array are given as 12, 7, 11, 92, 13, 71, 21, 38. What will be the arrangement of elements after four passes of the insertion sort method?
12. Given the elements of an unsorted array are given as 12, 7, 11, 92, 13, 71, 21, 38. Describe the steps in order to search elements 71 and 100.
13. Given the elements of a sorted array are given as 82, 70, 61, 52, 43, 31, 24, 18. Describe the steps in order to search elements 24 and 99.

Multiple Choice Questions

1. What is the worst-case time for linear search finding a single item in an array?
   (*a*) Constant time (*b*) Logarithmic time
   (*c*) Linear time (*d*) Quadratic time
2. What is the worst-case time for binary search finding a single item in an array?
   (*a*) Constant time (*b*) Logarithmic time
   (*c*) Linear time (*d*) Quadratic time
3. What additional requirement is placed on an array, so that binary search may be used to locate an entry?
   (*a*) The array elements must form a heap (*b*) The array must have at least 2 entries
   (*c*) The array must be sorted (*d*) The array's size must be a power of two
4. The best way to find an item in an unsorted implemented using an array list is with _____.
   (*a*) Linear search (*b*) Binary search
   (*c*) Random search (*d*) Direct search

5. Using Big O notation, the number of comparisons required by a binary search is

(*a*) $O(\log_2 n)$ (*b*) $O(n)$ (*c*) $O(n^2)$ (*d*) $O(n\log_2 n)$

6. In a selection sort of n elements, how many times, at most, the swap function is called in the complete execution of the algorithm?

(*a*) 1 (*b*) $n-1$ (*c*) $n\log_2 n$ (*d*) n^2

7. Suppose that a selection sort of 100 items has completed 42 iterations of the main loop. How many elements are now guaranteed to be in their final spot?

(*a*) 21 (*b*) 41 (*c*) 42 (*d*) 43

8. Suppose we are sorting an array of eight integers in ascending order using some sorting algorithm. After four iterations of the algorithm's main loop, the array elements are ordered as shown here:

52 54 55 57 58 51 53 56

Which statement is correct?

(*a*) Algorithm might be either selection sort or insertion sort

(*b*) Algorithm might be selection sort, but it is not insertion sort

(*c*) Algorithm is not selection sort, but it might be insertion sort

(*d*) Algorithm is neither selection sort nor insertion sort

9. Suppose we are sorting an array of ten integers in ascending order using some sorting algorithm. After four iterations of the algorithm's main loop, the array elements are ordered as shown here:

11 22 33 42 55 50 66 87 98 80

Which statement is correct?

(*a*) Algorithm might be either selection sort or insertion sort

(*b*) Algorithm might be selection sort, but it is not insertion sort

(*c*) Algorithm is not selection sort, but it might be insertion sort

(*d*) Algorithm is neither selection sort nor insertion sort

10. Which of the following algorithm can be modified to improve its performance?

(*a*) Selection sort (*b*) Bubble sort

(*c*) Insertion sort (*d*) None of the above

ANSWERS									
1.	(*c*)	2.	(*b*)	3.	(*c*)	4.	(*a*)	5.	(*a*)
6.	(*b*)	7.	(*c*)	8.	(*c*)	9.	(*b*)	10.	(*b*)

Programming Problems

1. Write a program to search an element using binary search when the given elements are sorted in descending order.

2. You are given list of names of students in your class. Write a program to sort the names in dictionary order using a sorting algorithm of your choice.

3. Write a program to sort letters of your name in reverse dictionary order.
4. Write a program to sort a list of numbers in descending order using bubble sort method. Incorporate the ability in the algorithm that if the list gets sorted before exhausting all the phases, the algorithm must stop.

PRACTICALS

In addition to the above programs for searching and sorting as part of the lab, there are programs for lab to solve numerical methods that include finding numerical differentiation and integration.

These programs are described next.

1. Numerical Differentiation

Many problems in science and engineering involve the use of differentiation of various types of functions. If a function is expressed in mathematical form, its derivative can be easily obtained by analytical methods.

But in many situations, the values of the independent variable x and the corresponding values of the dependent variable y are available in tabular form.

Consider the situation where the distance covered by an athlete as a function of time during his/her run for a 100-meter race is given in the following table:

Time (Secs.) :	0	1	2	3	4	5	6	7	8	9	10
Distance (Mts.) :	0	2.5	10	20	30.5	50	54	65.5	77.2	88.5	100

Here many questions can be asked, for example:

1. What was the speed of the athlete after 5 seconds?
2. What was the speed of the athlete as he approached the tape?
3. What was the acceleration of the athlete after 88 seconds?

Normally, speed and acceleration are calculated analytically. Suppose y represents the distance traveled by the athlete in x seconds, then

$$\text{Speed} = \frac{dy}{dx} \qquad \text{Acceleration} = \frac{d^2y}{dx^2}$$

But the mathematical relationship between y and x is not known in the above example. Therefore, it is not possible to use analytical methods. However, we can use the data given in the table to approximate the values of $\frac{dy}{dx}$ and $\frac{d^2y}{dx^2}$. Technique, which performs such calculations, is known as *numerical differentiation*.

The general formula to obtain numerical value of first derivative at one of tabulated points is

$$\left.\frac{dy}{dx}\right|_{x=x_k} = \frac{1}{h}\sum_{j=1}^{n-k}\left[(-1)^{j+1}\frac{d_{kj}}{j}\right]$$

where d_{kj} represents Δ_k^j (jth order forward difference at tabulated point $x = x_k$) and is the element of the forward difference table represented by a matrix D of order $(n-1)\times(n-1)$ for a function tabulated at n points.

Table 5.1: Forward difference table

i	x_i	y_i	Δy_i	$\Delta^2 y_i$	$\Delta^3 y_i$
1	x_1	y_1	$\Delta y_1 = y_2 - y_1$	$\Delta^2 y_1 = \Delta y_2 - \Delta y_1$	$\Delta^3 y_1 = \Delta^2 y_2 - \Delta y_1$
2	x_2	y_2	$\Delta y_2 = y_3 - y_2$	$\Delta^2 y_2 = \Delta y_3 - \Delta y_2$	
3	x_3	y_3	$\Delta y_4 = y_4 - y_3$		
4	x_4	y_4			

As we can see from the above table that the forward difference table for a function tabulated at four equally a matrix D of size 3×3 can represent spaced points. In general, the forward difference table for a function tabulated at n equally spaced points can be represented by a matrix of size $(n-1)\times(n-1)$ where the j^{th} order forward difference at the i^{th} point () is represented by the element d_{ij} of the matrix D. Note that only the elements in the column 1 to $(n-i)$, for rows $i = 1, 2, 3, \ldots, n-1$, are of interest.

Example 5.5: Given the following table

x :	0.50	0.75	1.00	1.25	1.50
$y = f(x)$:	0.13	0.42	1.00	1.95	2.35

Find $f^1(0.75)$.

Solution: The forward difference table for the given data is

i	x_i	y_i	Δy_i	$\Delta^2 y_i$	$\Delta^3 y_i$	$\Delta^4 y_i$
1	0.50	0.13	0.29	0.29	0.08	−1.00
2	0.75	0.42	0.58	0.37	−0.92	
3	1.00	1.00	0.95	−0.55		
4	1.25	1.95	0.40			
5	1.50	2.35				

Since $h = 0.25$, and the derivative is desired at the $x = 0.75$ (*i.e.*, $k = 2$). Expanding the formula up to third term, we get

$$\frac{dy}{dx}\Big|_{x=0.75} = \frac{1}{h}\left[d_{21} - \frac{d_{22}}{2} + \frac{d_{23}}{3}\right]$$

$$= \frac{1}{0.25}\left[0.58 - \frac{0.37}{2} + \frac{-0.92}{3}\right] = \frac{1}{0.25}\left[0.58 - 0.185 - 0.307\right] = 0.352$$

Listing 5.7

```
/*
*  Program to compute first derivative of the tabulated function
*/
#include<stdio.h>
int main()
{
   float x[10], y[10], d[10][10], a, sum, derivative, h, term;
   int i, j, k, n, sign;
```

```
    printf("Enter number of table points n(<=10) : ");
    scanf("%d", &n);
    printf("Enter interval size : ");
    scanf("%f", &h);
    printf("Enter %d pair of values as x,y\n", n);
    for ( i = 1; i <= n; i++ )
          scanf("%f,%f", &x[i], &y[i]);
    printf("Enter tabulated point where to find derivative : ");
    scanf("%f", &a);
    if ( (a < x[1]) || (a > x[n]) )
    {
          printf("\nValue lies outside range\n");
          return 1;
    }
    i = 1;
    while ( a != x[i] )
       i++;
    k = i;
    for ( j = 1; j <= (n-1); j++ )
    {
         for ( i = 1; i <= (n-j); i++ )
       {
            if ( j == 1 )
               d[i][j] = y[i+1] - y[i];
            else
               d[i][j] = d[i+1][j-1] - d[i][j-1];
         }
    }
    sum = 0.0;
    sign = 1;
    for ( j = 1; j <= (n-k); j++ )
    {
         term =  sign * d[k][j] / j;
         sum = sum + term;
         sign = - sign;
    }
    derivative = (1.0/h) * sum;
    printf("\nValue of derivative = %.3f\n", derivative);
    return 0;
}
```

Test Run

```
Enter number of table points n(<=10) : 5
Enter interval size : 0.25
Enter 5 pair of values as x,y
```

```
0.5,0.13
0.75,0.42
1.0,1.0
1.25,1.95
1.5,2.35
Enter tabulated point where to find derivative : 0.75
Value of derivative = 0.352
```

2. Numerical Integration

Numerical integration is the process of computing the value of a definite integral from a set of numerical values of the function. If a function is defined as a mathematical expression, then its integral is usually determined using the techniques of calculus, such solutions are called *closed form solutions*.

Often, functions that are available in the form of tables and no mathematical relationship between the variables are known.

In all such situations, values of the integral can be obtained by numerical technique whose aim is to provide effective procedures for approximate evaluation of definite integrals.

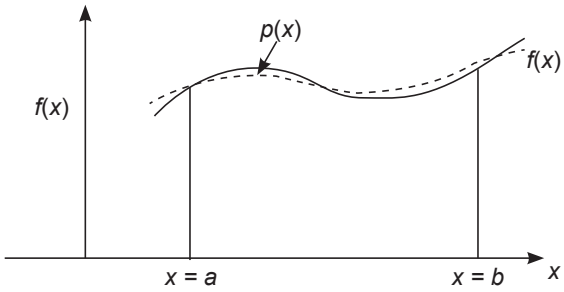


Fig. 5.5: Approximating polynomial $p(x)$ for function $f(x)$

In practice, given set of values for a function $f(x)$, the following table of data

x :	a	x_1	x_2	x_3	…	b
$f(x)$:	$f(a)$	$f(x_1)$	$f(x_2)$	$f(x_3)$	…	$f(b)$

is used to compute the value of the integral .

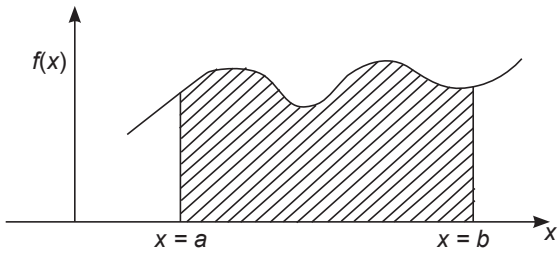


Fig. 5.6: Definite integral represented by the shaded area

The definite integral of a function is the area under the curve $y = f(x)$ enclosed between the limits $x = a$ and $x = b$, the problem of computing the integral of a function is reduced to the problem of finding the shaded area as shown in Fig. 5.6.

The popular methods used to find numerical integration include:

- Trapezoidal rule
- Simpson's 1/3rd rule
- Simpson's 3/8th Rule

We will consider Trapezoidal rule as an example to find the numerical integration of a function.

Trapezoidal Rule

The *trapezoidal rule* approximates the area under a curve by connecting successive points on the curve to form trapezoids of uniform width, and then summing the area under these trapezoids to obtain the approximate area under the curve.

Consider the function $f(x)$, whose graph between $x = a$ and $x = b$ is shown in Fig. 5.7. An approximation to the area under the curve is obtained by dividing the interval $[a, b]$ into n strips of width h each, and approximating the area of each strip by that of a trapezoid as shown by the shaded area.

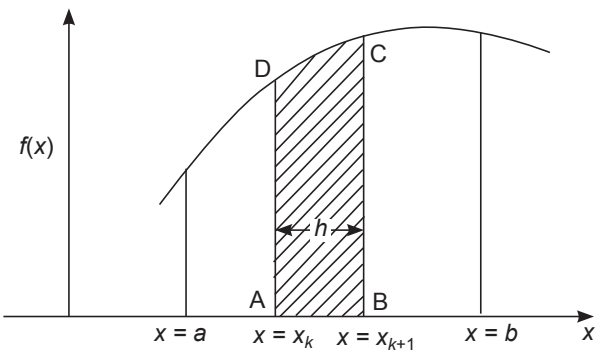


Fig. 5.7: Approximation of area by Trapezoidal rule

Formula for trapezoid rule is

$$I = \frac{h}{2}\left[y_1 + 2y_2 + 2y_3 + 2y_4 + .. + 2y_n + y_{n+1}\right]$$

assuming that function $f(x)$ is given in the following form

x	x_1	x_1+h	x_1+2h	…	x_1+nh
$y = f(x)$	y_1	y_2	y_3	…	y_{n+1}

Example 5.6: *The function f(x) is given as follows:*

x	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
y	1	1.2	1.4	1.6	1.8	2.0	2.2	2.4	2.6	2.8	3.0

Compute the integral of $f(x)$ between $x = 0$ and $x = 1.0$.

Solution: Given $h = 0.1$ and $n = 10$

The formula for Trapezoidal rule is

$$I = \frac{h}{2}\left[y_1 + 2(y_2 + y_3 + y_4 + y_5 + y_6 + y_7 + y_8 + y_9 + y_{10}) + y_{11}\right]$$

Substituting the values of y_i's and h, we get

$$I = \frac{0.1}{2}\left[1 + 2\times(1.2 + 1.4 + 1.6 + 1.8 + 2.0 + 2.2 + 2.4 + 2.6 + 2.8) + 3.0\right] = 2.00$$

Listing 5.8

```
/*  Program to implement Trapezoidal rule for tabulated function  */
#include<stdio.h>
int main()
{
   int i, n;
   float x[20], y[20], h, sum, integral;
   printf("Enter number of intervals n(<=20) : ");
   scanf("%d", &n);
   printf("Enter size of interval : ");
   scanf("%f", &h);
   printf("Enter %d pair of values as x,y \n", n+1);
   for ( i = 1; i <= (n+1); i++ )
       scanf("%f,%f", &x[i], &y[i]);
   sum = ( y[1] + y[n+1] )/2;
   for ( i = 2; i <= n; i++ )
       sum = sum + y[i];
   integral = h * sum;
   printf("\nValue of the integral = %7.2f\n", integral);
   return 0;
}
```

Test Run

```
Enter number of intervals n(<=20) : 10
Enter size of interval : 0.1
Enter 11 pair of values as x,y
0,1
0.1,1.2
0.2,1.4
0.3,1.6
0.4,2.0
0.5,2.2
0.6,2.4
0.7,2.6
```

```
0.8,2.8
0.9,3.0
1.0,1.2
Value of the integral = 2.00
```

KNOW MORE

The teacher is expected to explain these methods with the help of examples, and facilitate that should be able to solve on their own. Once they understand the various methods discussed, they should be encouraged to create program on their own.

Guide the students to take the program developed as a reference. The purpose of giving programs in the book is not to encourage spoon feeding rather demonstrate the good programming style.

REFERENCES & SUGGESTED READINGS

1. R. S. Salaria, Problem Solving & Programming in C, Khanna Book Publishing Co(P) Ltd., New Delhi.
2. E. Balagurusamy, Programming in ANSI C, Tata McGraw Hill, New Delhi..
3. Yashavant Kanetkar, Let Us C, BPB Publications, New Delhi.
4. Byron Gottfried, Programming with C, Schaum's Outlines.
5. https://onlinecourses.nptel.ac.in/noc21_cs01/preview
6. https://ocw.mit.edu/courses/intro-programming/
7. https://www.programiz.com/c-programming
8. https://www.javatpoint.com/c-programming-language-tutorial

6 Functions

UNIT SPECIFICS

This unit discusses the topics related to functions. The use of functions allows a programmer to divide complex problems into small size independent sub problems that can be solved separately and their solutions can be synthesized to obtain the solution of given complex problem. This unit explains various aspects functions and demonstrates their use with suitable examples.

RATIONALE

In many real-life situations, we may have to deal with problems whose size and complexity is higher. Solving these problems in one shot can we very tiresome and error prone. The best way to solve such problems is to divide them into sub problems that are independent of each other can be solved independently with much ease. Once each sub problem is solved, solution of these sub problems can be synthesized to obtain the solution of the entire problem.

Likewise a complex program can be divided into functions that are independent. These functions can be developed independently of each other and later integrated to make a complete program.

This unit will help the student to understand the various aspects related to functions.

PRE-REQUISITES

- Condition branching
- Loops and nested loops
- Arrays and strings

UNIT OUTCOMES

Upon completion of the unit, students will be able to

U6-O1: explain the usefulness of functions

U6-O2: explain various type of functions

U6-O3: explain the concept of local data and global data

U6-O4: explain all aspects related to declaring, defining, and calling a functions

U6-O5: explain the different ways of passing arguments to a function

U6-O6: develop modular programs for solving real-life problems

Unit 6 Outcomes	EXPECTED MAPPING WITH COURSE OUTCOMES (1- Weak Correlation; 2- Medium correlation; 3- Strong Correlation)							
	CO-1	CO-2	CO-3	CO-4	CO-5	CO-6	CO-7	CO-8
U6-O1	1	-	-	-	1	-	-	-
U6-O2	-	-	-	-	1	-	-	-
U6-O3	-	-	-	-	1	-	-	-
U6-O4	-	-	3	-	2	-	-	-
U6-O5	-	-	2	-	2	-	-	-
U6-O6	-	-	-	-	3	-	-	-

6.1 INTRODUCTION

The programs we have discussed so far were very simple and straightforward. They could solve problems that can be understood without much effort. As we move to larger and larger and more complex problems and hence programs, you will discover that it is not possible to understand all aspects of such problems and hence programs without some how reducing them to more elementary parts.

In this unit, we will learn various aspects related to functions.

6.2 WHAT IS A FUNCTION?

A function is an independent unit of execution that performs some specific task within the framework of a program.

It is independent in the sense it has its own declarations for declaring variables local to the functions and own set of instructions that define the task to be accomplished by the functions.

Every C program is made up of one or more functions. If it is made of only one function, it will be the *main* function. The execution of the program always begins from the main function. However, there is no restriction on the number of functions in a program. The determining factor regarding the number of functions required will be the size and complexity of the problem that the program is going to deal with.

When we run a C program, the operating system calls the main function, and then control is passed to the main function. The main function can call any function, and one function other than main can call another function.

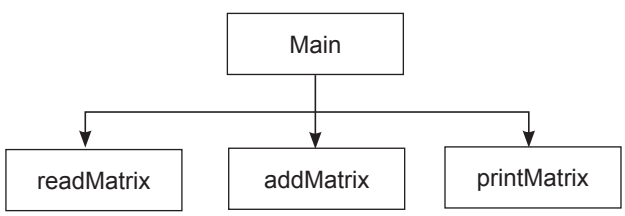


Fig. 6.1: Hierarchical organization of a multifunction program

As shown in Fig. 6.1, the program consists of four functions including *main* function, and the intended task of the program to add two matrices. The *readMatrix* function to perform the input of a given matrix, and it will be called twice by the main function to perform the input of two matrices such as $A_{m\times n}$ and $B_{m\times n}$.

Then function *addMatrices* will be called which will perform the addition of matrices $A_{m\times n}$ and $B_{m\times n}$ and as a result will return another matrix $C_{m\times n}$ to the main function.

Finally, the main function will call *printMatrix* function to perform the output of $C_{m\times n}$ matrix.

6.3 ADVANTAGES OF USING FUNCTIONS

Several advantages are associated with the use of functions. Some of the major advantages are

1. The use of functions allows managing large and complex problems by dividing them into small, simple and manageable parts.
2. The use of functions provides a way to reuse code that is required more than once a program.

   As an example, assume that a program requires computing average of series of numbers in five different parts of the program. Each time the data is different. We could write the code to compute the average five times, but this would take a lot of effort. It is much easier to write the code once as a function and then call it five times to compute the average by passing to it different sets of data.
3. The use of functions can protect data. This is done using the concept of *local data*. Local data consist of data described in a function. This data is available only to the function and that too while the function is executing.
4. Various functions comprising a program can be developed and tested in parallel, and thus reducing the total development time of the program.

6.4 TYPE OF FUNCTIONS

The following are two categories of functions:

- **Library functions** - These are the functions which are pre-written, compiled, and their machine code is available is the system library files. The machine code of the library functions referred in your program is added to your by linker during the linking process.

  Note that library functions are not part of the C language; they are provided by the vendors who developed C compiler for the convenience of the user. To use them you need to include appropriate header file.
- **User-defined functions** - These are the functions which are developed by the users to meet the requirements of the program in hand. If they are already developed, we can use their source code in our new program, and thus save lot of time and effort.

In this unit, our discussion is primarily on user-defined functions.

6.5 CONCEPT OF LOCAL DATA & GLOBAL DATA

The formal arguments and variables declared in the body of a function are unknown outside the function. The factor that determines which function recognizes a variable and which don't is called the *visibility* of variable. A local variable will be visible to the function in which it is declared, but not to others.

In contrast, a global variable is declared in the beginning, *i.e.*, before *main* function. It is visible to all the functions and exists till the program terminates.

6.6 USER-DEFINED FUNCTIONS

Though all the C Compilers available provide a rich collection of library functions, bit still as per the requirement of the program that provides a software solution to real-life problem, there is need to create functions specific to the requirements.

The C language provides all the specifications to deal with all aspects of user-defined functions that we are going to discuss next.

The user-defined functions in C fall in the following five categories:

- Functions that takes no argument(s) and return no value.
- Functions that takes argument(s) but return no value.
- Functions that takes no argument(s) but return a value.
- Functions that takes argument(s) and also return a value.
- Functions that return multiple values

6.7 DECLARING AND DEFINING FUNCTIONS

Like every variable in C, every function also needs to be declared and defined.

6.7.1 Declaring a Function

Function declaration consists only of a function header; it contains no code. Function header consists of three parts: *the return type*, *the function name*, and *the formal argument list*. A semicolon follows the function header.

```
returnType functionName(formal argument list);
```

Fig. 6.2: Syntax for function declaration

Function declaration gives the whole picture of the function that needs to be defined later.

Function declaration is also known as *function prototype*.

If the function has no arguments, then we write *void* in the parentheses. If the function has two or more arguments, each is separated by comma.

The following function declaration

```
int larget(int a, int b, int c);
```

tells the compiler that the return type of the function is *int*, name of the function is *largest*, and it has three argument, all of type *int*. The names of the arguments are not significant.

Even the following function declaration

```
int larget(int, int, int);
```

is equally good and serves the purpose as earlier declaration

6.7.2 Defining a Function

Function definition consists two parts; *function header* and *function body*.

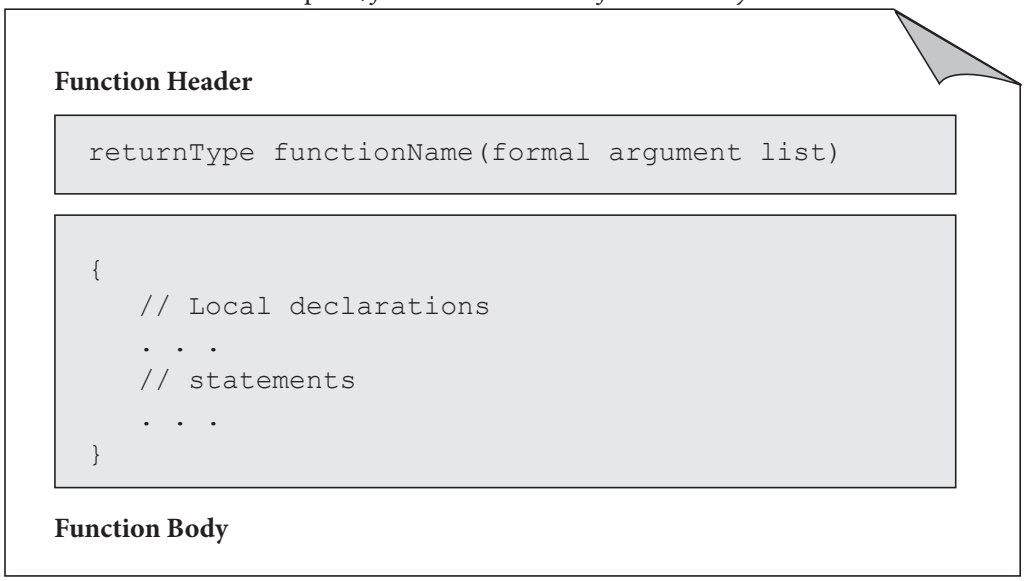


Fig. 6.3: Syntax for function definition

Function Header

Function header consists of three parts: *the return type*, *the function name*, and *the formal argument list*. A semicolon is not used at the end of function header.

When nothing is to be returned, the return type *void* is used.

The formal argument list declares the variable that will receive the data from the calling function and may also contain variable that can be used to send the data back to the calling function.

The *function prototype* and *function header* must match, except semicolon.

Function Body

The function body contains the local declarations and the instructions that define the task to be accomplished by the function. The body starts with local declarations that specify the variables needed by the function followed by the instructions.

Let us take few examples to declare and define few functions to perform simpler tasks.

Example 6.1: *Declare and define a function named maximum that takes three arguments, each of type int, and returns the largest of them.*

Declaration:

```
int maximum(int x, int y, int z);
```

Definition:

```
int maximum(int x, int y, int z)
{
    if ( ( x > y ) && ( x > z ) )
       return x;
    else if ( y > z )
       return y;
    else
       return z;
}
```

The logic of the above function can also be written as

```
int maximum(int x, int y, int z)
{
    int big;
    big = x;
    if ( y > big )
       big  = y;
    if ( z > big )
       big  = z;
    return big;
}
```

Example 6.2: *Declare and define a function named isEven that takes single argument n of type int, and returns 1 if n is even else returns 0.*

Declaration:

```
int isEven(int n);
```

Definition:

```
int isEven(int n)
{
    if ( n % 2 == 0 )
       return 1;
    else
       return 0;
}
```

The logic of the above function can also be written as

```
int isEven(int n)
{
   return ( n % 2 ? 1 : 0 );
}
```

Example 6.3: *Declare and define a function named sumOfDigits that takes single argument n of type int, and returns the sum of digits of n.*

Declaration:

```
int sumOfDigits(int n);
```

Definition:

```
int sumOfDigits(int n)
{
    int i, s = 0, d ;
    while ( n > 0 )
    {
       d = n % 10;
       s = s + d;
       n = n / 10;
    }
    return s;
}
```

6.8 CALLING A FUNCTION

Like library functions, the user-defined functions are called from a function simply by its name, including the actual arguments, if any, enclosed within parentheses.

However, parentheses must follow the function name even if there is no actual argument to be passed to the function.

The actual arguments, if any, must correspond in *number*, *type*, and *order* with formal arguments.

When the name of the function is encountered, the control is transferred to the called function. The formal arguments are replaced by the actual arguments and the execution of the function is carried out. When the *return* statement is executed or last statement has finished its execution, the control is transferred back to the calling function.

If the function is not returning a value, then the function call will appear as a standalone instruction.

If the function is returning a value, then the function call can appear in an assignment statement, in an arithmetic expression or in conditional statement or the *printf()* function as well.

The functions defined in example 6.1 to 6.3, can be called in following possible ways.

```
big = maximum(a, b, c);
```

```
if ( isEven(n) == 1 )
   printf( "\n%d is an Even number.\n" );
else
   printf( "\n%d is an Odd number.\n" );
```

```
printf( "\nSum of digits of %d = %d\n", n, sumOfDigits(n) );
```

6.9 THE *return* STATEMENT

The syntax of *return* statement is

```
return;                or          return (exp);
```

where *exp* can be a constant, variable or expression. Use of parentheses around *exp* is optional. The first form is used with functions defined with return type as *void*.

The *return* statement serves two purposes:

- Execution of *return* statement terminates the execution of the function and transfers control from the function back to the calling function.
- Whatever is following the *return* statement is returned as a value to the calling function.
- The *return* statement need not be last statement of the function. It can be used any where in the function. As soon as it is executed, the control will return to the calling function. Further, a function can contain any number of *return* statements.

There is key limitation of *return* statement ¾ it can return only one value. If you want your function to return two or more values to the calling function, you have to use other means.

6.10 PASSING ARGUMENTS TO A FUNCTION

The mechanism used to pass data to a function is via argument list, where individual arguments are called *actual arguments*. These arguments are enclosed in parentheses after the function name. The actual arguments must correspond in *number*, *type*, and *order* with formal arguments specified in the function definition. The actual arguments can be *constants*, *variables*, *array names*, or *expressions*.

There are two approaches to passing arguments to a function:

- Call by value
- Call by reference

Let us describe these one by one.

6.10.1 Call by Value

In this approach, the actual arguments are used in the function call. The actual argument can be a variable, a constant, or an expression.

When the function is called, the values of the actual arguments are substituted to the corresponding formal arguments, and then the control is transferred to the function.

The local variables of the function are created and after that the statements that define the task to be the function are executed. If the called function is supposed to return a value, it is returned via *return* statement.

Following points must be noted about passing arguments using call by value mechanism:

1. The actual arguments can be constants, variables, or expressions.
2. When the control is transferred from the calling function to the called function, the memory for local variables is allocated, and the statements in the function body are executed.
3. As soon as the called function finishes its execution, the memory allocated for its local variables is de-allocated, and finally the control is transferred back to the calling function.
4. Any change made to the formal arguments will have no effect on actual arguments, since the function will only be using the local copy of the arguments.

Following function definition illustrates the mechanism of passing arguments by value.

```
void swap( int a, int b )
{
   int temp;
   temp = a;
   a = b;
   b = temp;
}
```

The above function will be called as

```
swap(x,y);
```

where *x* and *y* are actual arguments in the calling function.

Listing 6.1

```
/*
    Program to illustrate the passing of arguments by value.
    It calls a function swap() that swaps/interchanges
    values of arguments.
*/
#include <stdio.h>
void swap(int a, int b);  /* function prototype */
int main()
{
   int x, y;
   printf(«\nEnter value for x : «);
   scanf(«%d», &x);
   printf(«\nEnter value for y : «);
   scanf(«%d», &y);
   printf("\nBefore calling swap function\n");
   printf("\nValue of x = %d, Value of y = %d\n", x, y);
   swap(x,y);  /* function call   */
   printf("\nAfter returning from swap function\n");
   printf("\nValue of x = %d, Value of y = %d\n", x, y);
   return 0;
}
void swap( int a, int b )
{
   int temp;
   printf("\nValues received from the main function\n");
   printf("\nValue of a = %d, Value of b = %d\n", a, b);
   temp  = a;
   a = b;
   b = temp;
```

```
    printf("\nValues of local copy after swapping\n");
    printf("\nValue of a = %d, Value of b = %d\n", a, b);
}
```

Test Run

```
Enter value for x : 10
Enter value for y : 12
Before calling swap function
Value of x = 10, Value of y = 12
Values received from the main function
Value of a = 10, Value of b = 12
Values of local copy after swapping
Value of a = 12, Value of b = 10
After returning from swap function
Value of x = 10, Value of y = 12
```

You must have observed that any change made to the formal arguments have not effect on the actual arguments.

6.10.2 Call by Reference

In this approach, the addresses of the actual arguments are used in the function call. The actual argument can be variables only.

The formal arguments are declared as pointers to *types* that match the data types of the actual arguments.

When the function is called, the addresses of the actual arguments are substituted to the corresponding formal arguments which are pointers, and then the control is transferred to the function.

The local variables of the function are created and after that the statements that define the task to be the function are executed.

If the called function is supposed to return a value, it is returned via *return* statement.

Following points must be noted about passing arguments using call by reference mechanism:

1. The actual arguments can only be variables.
2. When the control is transferred from the calling function to the called function, the memory for local variables is allocated, and the statements in the function body are executed.
3. As soon as the called function finishes its execution, the memory allocated for its local variables is de-allocated, and finally the control is transferred back to the calling function.
4. Any change made to the formal arguments will have immediate effect on actual arguments, since function will be working on actual arguments through pointers.

Following function illustrates mechanism of passing arguments by reference (address).

```
void swap( int *a, int *b )
{
    int temp;
    temp  = *a;
    *a = *b;
    *b = temp;
}
```

The above function will be accessed as

```
swap(&x,&y);
```

where *x* and *y* are actual arguments in the calling function.

Listing 6.2

```
/*
    Program to illustrate the passing of arguments by reference.
   It calls a function swap() that interchanges values of arguments.
*/

#include <stdio.h>
void swap(int *a, int *b);  /* function prototype */

int main()
{
   int x, y;
   printf("\nEnter value for x : ");
   scanf("%d", &x);
   printf("\nEnter value for y : ");
   scanf("%d", &y);
   printf("\nBefore calling swap function\n");
   printf("\nValue of x = %d, Value of y = %d\n", x, y);
   swap(&x,&y);  /* function call   */
   printf("\nAfter returning from swap function\n");
   printf("\nValue of x = %d, Value of y = %d\n", x, y);
   return 0;
}
void swap( int *a, int *b )
{
   int temp;
   printf("\nValues received from the main function\n");
   printf("\nValue of *a = %d, Value of *b = %d\n", *a, *b);
   temp  = *a;
   *a = *b;
   *b = temp;
   printf("\nValues of local copy after swapping\n");
   printf("\nValue of *a = %d, Value of *b = %d\n", *a, *b);
}
```

Test Run

```
Enter value for x : 10
Enter value for y : 12
```

```
Before calling swap function
Value of x = 10, Value of y = 12
Values received from the main function
Value of *a = 10, Value of *b = 12
Values of local copy after swapping
Value of *a = 12, Value of *b = 10
After returning from swap function
Value of x = 12, Value of y = 10
```

You must have observed that any change made to the formal arguments is reflected back in the actual arguments.

Next program is another example to demonstrate the passing of arguments using call by reference.

Listing 6.3

```
/* Program that calls function example() to find the average of
   two numbers, largest & smallest of these numbers. The
   average is retuned via return statement while largest and
   smallest numbers are returned via formal arguments
*/
#include <stdio.h>
/* function prototype  */
float example(float x, float y, float *big, float *small);
int main()
{
   float x, y, avg, larger, smaller;
   printf("\nEnter value for x : ");
   scanf("%f", &x);
   printf("\nEnter value for y : ");
   scanf("%f", &y);
   avg = example(x,y,&larger,&smaller); /*  function call  */
   printf("\nAverage of %.2f and %.2f is %.2f\n", x, y, avg);
   printf("\nLarger number is %.2f\n", larger);
   printf("\nSmaller number is %.2f\n", smaller);
   return 0;
}
/*
   Function to compute the average of two numbers and find the
   largest and smallest of these numbers
*/
float example(float x, float y, float *big, float *small)
{
    float average;
```

```
    average = ( x + y ) / 2;
    if ( x > y ) {
       *big = x;
       *small = y;
    } else {
       *big = y;
       *small = x;
    }
    return average;
}
```

Test Run

```
Enter value for x : 10.5
Enter value for y : 12.5
Average of 10.50 and 12.50 is 11.50
Larger number is 12.50
Smaller number is 10.50
```

6.10.3 Comparison between Call by Value and Call by Reference

Table 6.1 highlights the key points of differences regarding the prototype, definition and function call.

Table 6.1: Difference between call by value & call by reference Part-1

Call by Value	Call by Reference
Function Prototype: `void swap(int a, int b);`	Function Prototype: `void swap(int *a, int *b);`

Function Definition:
`void swap(int a, int b)`
`{`
`int temp;`
`temp = a;`
`a = b;`
`b = temp;`
`}`
	Function Definition:
	`void swap(int *a, int *b)`
	`{`
	`int temp;`
	`temp = *a;`
	`*a = *b;`
	`*b = temp;`
	`}`

Function Call: `swap(x, y);`	Function Call: `swap(&x, &y);`

Table 6.2 highlights the key points of differences regarding the nature of actual arguments, nature of formal arguments and the impact of any change/modification of formal arguments on actual arguments.

Table 6.2: Difference between call by value & call by reference Part-2

Call by Value	Call by Reference
Actual arguments can be constants, variables, or expressions.	Actual arguments can only be variables.
Formal arguments are ordinary variables.	Formal arguments are pointer variables.

Call by Value	Call by Reference
Values of actual arguments are substituted in formal arguments.	Addresses of the actual arguments are substituted in formal arguments.
Any change made to formal arguments will have no effect on actual arguments, since the function will only be using the local copy of the arguments.	Any change made to pointer variables will have immediate effect on actual arguments, since the function will be working on actual arguments through address.

6.10.4 Passing One-Dimensional Array as Argument

Note that instead of passing values of all elements of an array to a function, only address of the array is passed. As you know, in C, the name of an array is base address, *i.e.*, address of the first element of the array.

```
/*  main function   */
#include <stdio.h>
void fun( int x[], int m );
void main()
{
   int a[10], n;
   /*  other local declarations */

   func(a, n); /* function call */
   /*  other statements         */
}
```

```
/*  function definition   */
void fun( int x[], int m )
{
   /*  local declarations */
   /*  other statements   */
}
```

Fig. 6.4: Passing one-dimensional array as an argument to a function

Because the name of the array is in fact its address, therefore, passing the array name allows called function to refer to the array back in the calling function.

Listing 6.4

```
/* Program to demonstrate the passing of an array as an argument  */
#include <stdio.h>
int largest( int x[], int n );    /*  function prototype */
int main(void)
{
   int a[10]={12,15,20,17,25,50,11,10,8,13};
   int i;
   printf( "Largest element of array = %d\n", largest(a,10));
   return 0;
}
/* function that returns largest element of the array   */
```

```
int largest( const int x[], int n )
{
   int big, i;
   big = x[0];
   for ( i = 1; i < n; i++ ) {
      if ( x[i] > big )
         big = x[i];
   }
   return big;
}
```

Test Run

```
Largest element of array = 50
```

6.10.5 Passing Two-Dimensional Array as Argument

When we pass two-dimensional array to a function, we use the array name as we did with one-dimensional array. The formal argument in the argument list in the called function header must indicate the array has two dimensions. This is done by including two sets of brackets, one for each dimension. The first pair of bracket can be empty but in the second pair of brackets we need to specify the size that is equal to the size of the corresponding dimension of the actual array. During function call, only the base address of the array is passed.

```
/*  main function   */
#include <stdio.h>
void fun(int x[][4], int m, int n);
void main()
{
   int a[4][4];
   /*  other local declarations  */
   func(a,m,n); /* function call */
   /*  other statements          */
}
```

```
/*  function definition    */
void fun(int x[][4], int m, int n)
{
   /*  local declarations  */
   /*  other statement     */
}
```

Fig. 6.5: Passing two-dimensional array as an argument to a function

Listing 6.5

```
/*
   Program to add matrix A(mxn) and matrix B(mxn).
   This program uses function to read, add and display matrices.
*/
#include<stdio.h>
```

```
#define ROWS 10
#define COLS 10
/* function declarations */
void readMatrix(int a[][COLS], int m, int n);
void printMatrix(int a[][COLS], int m, int n);
void addMatrices(int a[][COLS], int b[][COLS], int c[][COLS],
                 int m, int n);
int main()
{
    int a[ROWS][COLS], b[ROWS][COLS], c[ROWS][COLS];
    int n, m;
    char ch;
    printf( "Enter size of matriices as mxn: " );
    scanf( "%d%c%d", &m, &ch, &n);
    readMatrix(a,m,n);            /* input matrix A(mxn) */
    readMatrix(b,m,n);            /* input matrix B(mxn) */
   addMatrices(a,b,c,m,n);  /* add matrix A(mxn) & matrix B(mxn) */
    printf( "\nSum A+B is\n\n" );
    printMatrix(c,m,n);      /* output matrix C(mxn) */
    return 0;
}
/* function that reads a matrix A(mxn)  */
void readMatrix(int a[][COLS], int m, int n)
{
   int i, j;
   printf("\nEnter %d elements of matrix A row-wise\n", m*n );
   for ( i = 0 ; i < m; i++ ) {
      for ( j = 0; j < n; j++ ) {
         scanf( "%d", &a[i][j] );
      }
   }
}
/* function that displays a matrix 'a' of order mxn  */
void printMatrix(int a[][COLS], int m, int n)
{
    int i, j;
    for ( i = 0 ; i < m; i++ ) {
      for ( j = 0; j < n; j++ ) {
           printf( "%4d", a[i][j] );
```

```
        }
        printf ( "\n" );
    }
}
/*
   function that adds A(mxn) & b(mxn), and
   stores the sum in matrix c(mxn)
*/
void addMatrices(int a[][COLS], int b[][COLS], int c[][COLS],
                 int m, int n)
{
    int i, j;
    for ( i = 0 ; i < m; i++ )
    {
        for ( j = 0 ; j < n; j++ )
        {
            c[i][j] = a[i][j] + b[i][j];
        }
    }
}
```

Test Run

```
Enter size of matrices as mxn : 3x3
Enter 9 elements of matrix A row-wise
1 3 2
4 5 1
6 5 8
Enter 9 elements of matrix A row-wise
7 3 5
2 7 3
4 2 1
Sum A+B is
  8   6  7
  6  12  4
 10   7  9
```

6.10.6 Passing String as Argument

A string can be passed as an argument to a function using subscripted notation for arrays. In a function definition, the formal parameter is declared as an array of characters. The following program illustrates the passing of a string to a function.

Listing 6.6

```
/* Program to illustrates passing of a string to a function  */

#include<stdio.h>
#include<string.h>

void fun( char temp[] );    /*  function prototype  */

int main()
{
   char str[] = "Sample string";
   fun(str);
   return 0;
}
void fun( char temp[] )
{
   printf( "String passed to fun : " );
   puts( temp );
   printf( "and its length is : %d\n", strlen(temp) );
}
```

Test Run

```
String passed to fun : Sample string
and its length is : 13
```

ILLUSTRATIVE EXAMPLES

To have hands on defining functions, let use consider few more examples.

Example 6.4: Write a function, say *int factorial(int n)*, to find the factorial of a positive integer number *n*. Using this definition of factorial function, write a program to compute the Binomial coefficient ${}^{n}c_{r}$ defined as

$$ {}^{n}c_{r} = \frac{n!}{r!(n-r)!} $$

Listing 6.7

```
/*  Program to compute Binomial coefficient  */
#include <stdio.h>
int factorial(int n);  /*  function prototype  */
int main()
```

```
{
    int ncr, n, r;
    printf( "\nEnter value of n: " );
    scanf( "%d", &n );
    printf( "\nEnter value of r: " );
    scanf( "%d", &r );
    ncr = factorial(n)/(factorial(r)*factorial(n-r));
    printf( "\nValue of ncr = %d\n" , ncr);
    return 0;
}
/*  Function to compute factorial of a +ve integer number  */
int factorial(int n)
{
   int prod = 1, i;
   for ( i = 1; i <= n; i++ )
      prod = prod * i;
   return prod;
}
```

Test Run

```
Enter value of n: 5
Enter value of r: 3
Value of ncr = 10
```

This program calls the factorial function thrice with argument *n*, *r*, (*n*–*r*) respectively, and then using these values computes the binomial coefficient.

Example 6.5: *Write a function, say int computeHCF(int m, int n), that returns HCF of m & n. Using this definition write a complete program to compute the HCF of two given positive integers m and n.*

Listing 6.8

```
/*
    Program to compute HCF of two given positive integers (m and n)
*/
#include <stdio.h>
int computeHCF( int m, int n );   /*  function prototype */
int main()
{
    int m, n, hcf;
    printf( "\nEnter value of n: " );
    scanf( "%d", &n );
```

```
    printf( "\nEnter value of m: " );
    scanf( "%d", &m );
    printf( "\nEnter value of n: " );
    scanf( "%d", &n );
    hcf = computeHCF(m,n);
    printf( "\nValue of HCF = %d\n" , hcf);
    return 0;
}
/* Function to compute HCF of two positive integer numbers */
int computeHCF( int m, int n )
{
   int r;
   while ( 1 )
   {
      r = m % n;
      if ( r == 0 )
         return n;
      m = n;
      n = r;
   }
}
```

Test Run

```
Enter value of m: 125
Enter value of n: 35
Value of HCF = 5
```

Examples 6.6: *Write a function, say int smallestDigit(int n), that returns the smallest digit in number n. Use this function in a program to demonstrate its use.*

Listing 6.9

```
/*
   Program to find smallest digit in a positive integer number
*/
#include <stdio.h>
int smallestDigit(int n);    /* function prototype */

int main()
{
    int n;
    printf( "\nEnter positive integer number : " );
    scanf( "%d", &n );
```

```
    printf( "\nSmallest digit in %d is %d\n", n, smallestDigit(n));
    return 0;
}
/* function that returns smallest digit in a positive integer number */
int smallestDigit(int n)
{
    int sd = 9;
    int d;
    while ( n > 0 )
    {
       d = n % 10;
       if ( d < sd )
          sd = d;
       n = n / 10;
    }
    return sd;
}
```

Test Run

```
Enter positive integer number : 23145
Smallest digit in 23145 is 1
```

Examples 6.7: *Write a function, say int isPrime(int n), that returns value 1 if n is prime number otherwise returns 0. Use this function in a program to test whether the given natural number is prime number or not.*

Listing 6.10

```
/*
   Program to test whether the given natural number is
   prime number or not.
*/
#include<stdio.h>
#include<math.h>
int isPrime(int n);  /*  function prototype  */
int main()
{
   int n;
   printf( "\nEnter a positive integer number: " );
   scanf("%d", &n);
   if ( isPrime(n) == 1 )
      printf( "\n%d is a prime number.\n", n );
   else
      printf( "\n%d is not a prime number.\n", n );
   return 0;
}
```

```
/*
    definition of function that tests whether the given positive
    integer number 'n' is prime number
*/
int isPrime(int n)
{
   int k, m;
   if ( ( n > 2 ) && ( ( n % 2 ) == 0 ) ) {
      return 0;
   }
   m = sqrt( n );
   for ( k = 3; k <= m; k += 2 )
   {
       if ( n % k == 0 )
       {
           return 0;
       }
   }
   return 1;
}
```

Test Runs

First Run

```
Enter a positive integer number: 43
43 is a prime number.
```

Second Run

```
Enter a positive integer number: 92
92 is not a prime number.
```

UNIT SUMMARY

In this chapter, we have learned that

- Many problems that arise because of the size and complexity of software can be handled efficiently using modular design approach.
- Each function is complete and independent in its own.
- Each function handled one aspect of the complex problem.
- Value can be returned from a function using *return* statement.
- A Function can return any type of value. This includes pointers as well.
- Arguments can be passed to a function either using call by value approach or call by reference approach.
- A function prototype, like variable declaration, is a declaration that specifies the return type, name, and type of formal arguments.

EXERCISE

Subjective Questions

1. What is the difference between a user-defined function and library function?
2. When is the execution of the function terminated?
3. How many arguments can be passed to a function?
4. What are the rules regarding the relationship between formal arguments and actual arguments?
5. How is a function invoked?
6. How many times can a function be called?
7. What is wrong with the following function definition?

```
testFunction( int k ) {
   float temp = 5.25;
   temp = k / 2.0;
   return temp;
}
```

8. Find the errors, if any, in the following function definition

```
void fun( int x, int y ) {
   int z;
   /*  some statement  */
   return z;
}
```

9. Find the errors, if any, in the following function definition

```
void fun( int x, y ) {
   int z;
   /*  some statement  */
   return;
}
```

10. Find the errors, if any, in the following function definition

```
int fun1( int x, int y ) {
   int z;
   /*  some statement  */
   int fun2( int t ) {
      return (t-2);
   }
   /*  some more statements  */
   return z;
}
```

11. What will be the output of the following program?

```
void main() {
   float r = 5.0, c;
   float func( float t );  /* function prototype   */
   c = func( r );
   printf( "\nValue returned by function" );
   printf( " = %d\n", c);
}
float func( float t ) {
   float temp;
   temp = t * t / 2;
   return 5.0;
}
```

12. Find the errors, if any, in the following function declarations:

(*a*) `int diff(int x, y);` (*b*) `void fun(void, void);`

(*c*) `float diff(float, float);` (*d*) `int test(float x, int y)`

Programming Problems

1. Write a function, say *int sumOfDigits(int number)*, that returns the sum of digits of a positive integer number.
2. Write a function, say *float absValue(float value)*, that returns the absolute value of a number of type *float*.
3. Write a function, say *int round(float x)*, that returns rounded value of *x* to nearest integer value.
4. Write a function, say *int sumOfN(int n, int m)*, which computes the sum of *n* integers starting with *m*th integer, *i.e.*, $m + (m + 1) + (m + 2) + \ldots + (m + n - 1)$.
5. Write a function, say *int isPrime(int n)*, that returns 1 if number *n* is prime else returns 0.
6. A number is a palindrome if it is the same number when read forward or backward. For example, numbers 1991, 1001 and 1221 palindrome numbers. Write a function, say *int isPalindrome(unsigned int k)*, that returns value 1 if the number *k* is palindrome else returns value 0.
7. A positive integer number IJK is said to be *well-ordered* if I < J < K. For example, number 138 is called *well-ordered* because the digits in the number (1, 3, 8) increase from left to right, *i.e.*, 1 < 3 < 8. Number 365 is not *well-ordered* because 6 is larger than 5. Write a function, say *int isWellOrdered(unsigned int k)*, that returns value 1 if the *k* is a *well-ordered* number else returns 0.
8. Write a function, say *int largestDigit(int x)*, that returns the largest digit in number *x*.
9. Write a function, say *int unitDigit(int n)*, that returns the unit digit of a number represented in argument *n*.
10. Write a function, say *int isLeapYear(int year)*, that returns value 1 if the argument *year* represents a leap year else return value 0.

11. Write a function, say *int isValidDate(int dd, int mm, int yyyy)*, that returns value 1 if the date is valid else returns value 0.

12. Write a function, say *int isTrianglePossible(int a, int b, int c)*, that returns value 1 if the triangle can be passed using *a*, *b*, and *c* as its sides else returns value 0.

13. Write a function, say *int countDigits(int n)*, that return the number of digits in *n*, *i.e.*, size of number *n*.

Multiple Choice Questions

1. Which of the following statement about functions is false?
(*a*) More than one function is allowed in a program unit.
(*b*) A function can call another function.
(*c*) A function can call itself.
(*d*) Constants can appear in the formal argument list.

2. Which of the following is not a valid reason for using functions?
(*a*) They use less memory than repeating the same code.
(*b*) They keep different program activities separate.
(*c*) They run faster.
(*d*) They keep variables safe from other parts of the program

3. What is the default return type of a function?
(*a*) void (*b*) int
(*c*) float (*d*) char

4. The program execution starts from the
(*a*) *main()* function (*b*) function which is defined first
(*c*) function that is defined last (*d*) depends on compiler

5. The C language allow arguments to be passed
(*a*) only call by value (*b*) only call by reference
(*c*) both *a* & *b* (*d*) depends on compiler

6. Consider the following program

```
swap(int i, int j)
{
   i = i + j;
   j = i - j;
   i = i - j;
}
void main()
{
    int i = 5, j = 10;
    swap(i, j);
    printf( "\n%d, %d", i, j );
}
```

What will happen when we attempt to compile & run the program?

(*a*) Program will fail to compile because same variable names cannot be used in *main()* and *swap()* functions.

(*b*) Program will fail to compile because return type of the swap function is not specified.

(*c*) Program will compile and execute giving output as 5, 10.

(*d*) Program will compile and execute giving output as 10, 5.

7. Identify the correct statement

(*a*) A function can be defined more than once in a program.

(*b*) One function cannot be defined within another function definition.

(*c*) All functions must be in the same file.

(*d*) Function should appear in the order they are called in the main function.

8. Consider the following function definition, and identify the correct statement

```
myFunction( a + b, c, d, 5 )
int a, b, c, d;
{
    int sum;
    sum = a + b +  c + d + 5;
    return sum;
}
```

(*a*) Function will fail to compile because expression is not permitted as formal argument.

(*b*) Function will fail to compile because constant is not permitted as formal argument.

(*c*) Function will fail to compile because return type of the function is omitted.

(*d*) Both (*a*) and (*b*).

9. Which of the following is a part of the function header?

(*a*) Function name (*b*) Return type

(*c*) Argument list (*d*) All of the above

10. Which of the following is a complete function?

(*a*) `int fun();`

(*b*) `int fun(int x) { return x+1; }`

(*c*) `void fun(int) { print(“Hello”); }`

(*d*) `void fun(x) { print(“hello”); }`

ANSWERS																			
1.	(*d*)	2.	(*c*)	3.	(*b*)	4.	(*a*)	5.	(*c*)	6.	(*c*)	7.	(*b*)	8.	(*d*)	9.	(*d*)	10.	(*b*)

PRACTICALS

1. Write a program to find the largest element, smallest element, and average of elements of an array named *a* with *n*(≤50) element using following user-defined functions

(*a*) Function, say *findLargest(int a[], int n),* to find the largest element of the array

(*b*) Function, say *in findSmallest(int a[], int n)*, to find the smallest element of the array

(*c*) Function, say *float findAverage(int a[], int n)*, to find the average of the elements of the array

Listing 6.11

```
/*
   Program to find the largest element, smallest element,
   and average of elements of array using functions
*/
#include<stdio.h>
/*  function prototypes  */
int findLargest(int a[], int n);
int findSmallest(int a[], int n);
float findAverage(int a[], int n);
int main()
{
   int a[50];
   int i, n;
   printf("\nEnter size of array n(<=50) : ");
   scanf("%d", &n);
   printf("\nEnter %d elements of array\n\n");
   for ( i = 0; i < n; i++ )
   {
      scanf("%d", &a[i]);
   }
   printf("\nSmallest element    = %d", findSmallest(a,n));
   printf("\nLargest element     = %d", findLargest(a,n));
   printf("\nAverage of elements = %.2f", findAverage(a,n));
   return 0;
}

int findLargest(int a[], int n)
{
   int i, max;
   max = a[0];
```

```
    for ( i = 1; i < n; i++ ) {
        if ( a[i] > max )
            max = a[i];
    }
    return max;
}
int findSmallest(int a[], int n)
{
    int i, min;
    min = a[0];
    for ( i = 1; i < n; i++ ) {
        if ( a[i] < min )
            min = a[i];
    }
    return min;
}

float findAverage(int a[], int n)
{
    int i, sum;
    float avg;
    sum = 0;
    for ( i = 0; i < n; i++ ) {
        sum = sum + a[i];
    }
    avg = (float)sum/n;
    return avg;
}
```

Test Run

```
Enter size of array n(<=50) : 10
Enter 10 elements of array
25 20 40 32 10 15 45 50 30 24
Smallest element    = 10
Largest element     = 50
Average of elements = 29.10
```

2. Write a function, say *int isPrime(int n)*, that returns value 1 if *n* is prime number otherwise returns 0. Use this function in a program to print first *m* prime numbers

Listing 6.12

```
/*
   Program to print first 'm' prime numbers using a function
*/
#include<stdio.h>
#include<math.h>
int isPrime(int n);  /*  function prototype  */
int main()
{
   int m, num = 2, count = 0;
   printf( "\nEnter value for m : " );
   scanf("%d", &m);
   printf( "\nFirst %d prime number are\n\n", m );
   while ( count < m )
   {
       if ( isPrime(num) == 1 ) {
           count++;
           printf( "%d  ", num );
       }
       num++;
   }
   printf( "\n" );
   return 0;
}
/*
    definition of function that tests whether the given
    positive integer number 'n' is prime number
*/
int isPrime(int n)
{
   int k, m;
   if ( ( n > 2 ) && ( ( n % 2 ) == 0 ) )
   {
      return 0;
   }
```

```
    m = sqrt( n );
    for ( k = 3; k <= m; k += 2 )
    {
        if ( n % k == 0 )
        {
            return 0;
        }
    }
    return 1;
}
```

Test Run

```
Enter value for m : 10
First 10 prime number are
2  3  5  7  11  13  17  19  23  29
```

3. Write a function, say *int product(int a, int b)*, that returns product of two numbers *a* and *b*. Use this function in a program to find the product of two given numbers.

Listing 6.13

```
*/
   Program to find product of two numbers using function
*/
#include <stdio.h>
/*  function prototype  */
int product(int a, int b);
int main()
{
   int m, n, result;
   printf("\nEnter first number : ");
   scanf("%d", &m);
   printf("\nEnter second number : ");
   scanf("%d", &n);
   result = product(m, n);
   printf("\nProduct of %d and %d = %d\n", m, n, result);
   return 0;
}
/*  definition of functio that returns product of two numbers  */
int product(int a, int b)
{
```

```
    int temp = 0;
    while (b != 0)
    {
        temp += a;
        b--;
    }
    return temp;
}
```

Test Run

```
Enter first number : 15
Enter second number : 12
Product of 15 and 12 = 180
```

KNOW MORE

The use of functions enables the developer to create programs for solving large and complex program that are easier to code, easy to debug, and easy to modify.

The teacher is expected to develop an understanding among the students about the benefits of using functions and various aspects related with the development of functions.

The teacher should demonstrate the use of functions by taking examples from real-life situations, and creating C programs to solve them.

REFERENCES & SUGGESTED READINGS

1. R. S. Salaria, Problem Solving & Programming in C, Khanna Book Publishing Co(P) Ltd., New Delhi.
2. E. Balagurusamy, Programming in ANSI C, Tata McGraw Hill, New Delhi.
3. Yashavant Kanetkar, Let Us C, BPB Publications, New Delhi.
4. Byron Gottfried, Programming with C, Schaum's Outlines.
5. https://onlinecourses.nptel.ac.in/noc21_cs01/preview
6. https://ocw.mit.edu/courses/intro-programming/
7. https://www.programiz.com/c-programming
8. https://www.javatpoint.com/c-programming-language-tutorial

7 Recursion

UNIT SPECIFICS

This unit discusses the topics related to recursion. Recursion is one of the important concepts in mathematics & computer science and is used extensively in solving many real-life problems. This unit explains various aspects of recursion and demonstrates their use with suitable examples.

RATIONALE

In real-life situations, many problems may have iterative solution as well as recursive solution. Then the important question arises which one to prefer. The main reason we use recursion is to simplify an algorithm into terms easily understood by most people. It's important to note here that the purpose of recursion (rather than its benefit) is to make our code easier to read, and easier to reason with. However, it is important to be aware that recursion is not a mechanism we can use to optimize our code for performance - if anything; it can have an adverse effect on performance compared to an equivalent function written iteratively.

To have a short sound bite to remember, we can say that *"Recursive functions optimize legibility for developers; iterative functions optimize performance for computers."*

Usefulness of recursion can be realized and appreciated when we actually try to write a code that resembles a real life scenario.

This unit will help the student to understand the various aspects related to recursion and its implementation in C using recursive functions.

PRE-REQUISITES

- Condition branching
- Arrays
- User-defined functions

UNIT OUTCOMES

Upon completion of the unit, students will be able to

U7-O1: explain the concept of recursion

U7-O2: explain concept of base case and recursive step

U7-O3: demonstrate defining and using recursive functions for selected recursive problems

U7-O4: explain and implement Quick sort algorithm

U7-O5: explain and implement Merge sort algorithm

U7-O6: develop modular programs using recursive function for solving real-life problems

Unit 7 Outcomes	EXPECTED MAPPING WITH COURSE OUTCOMES (1- Weak Correlation; 2- Medium correlation; 3- Strong Correlation)							
	CO-1	CO-2	CO-3	CO-4	CO-5	CO-6	CO-7	CO-8
U7-O1	1	-	-	-	-	-	-	-
U7-O2	1	-	-	-	-	-	-	-
U7-O3	-	-	-	-	2	-	-	-
U7-O4	-	-	-	-	-	-	3	-
U7-O5	-	-	-	-	-	-	3	-
U7-O6	-	-	-	-	-	-	3	-

7.1 INTRODUCTION

Recursion is a powerful concept in the development of programs in which a function has the ability to refer to itself to solve a problem. This control technique, called *recursion* is an important concept in computer science and is convenient for a variety of problems that would be difficult to solve using iterative constructs such as *for, while* and *do – while* loops.

Recursion is used extensively in solving many real-life problems.

Recursive functions can be directly implemented in almost all modern high-level programming language such C/C++/C#/Java/Python.

This unit introduces recursive functions and illustrates their use through various illustrative examples.

7.2 RECURSIVE FUNCTIONS

A recursive function is a function whose definition is based upon itself, *i.e.*, which calls itself.

A recursive function is defined in terms of *base case* and *recursive step*.

1. **Base Case:** The result is computed immediately with the given inputs to the function, *i.e.*, there is value of argument(s), for which the function does not call itself.

   In other words, the base case is the solution to the "simplest" possible problem.

   Consider an example of finding maximum value in a list of numbers. The maximum value in a list is either the first number or the biggest of the remaining numbers.

   The the base case in the problem is that the list had only one number, and by definition, if there is only one number, it is the largest.

2. **Recursive Step:** The result is computed with the help of one or more *recursive calls* to the function, but with the argument(s) somehow reduced in size, *i.e.*, closer to a base case.

   In above example of finding maximum value in a list of numbers, the recursive step is to find maximum value in the remaining list of numbers, i.e., list of reduced size (1 less that the earlier size).

Let us take few well known examples of recursive functions.

7.2.1 Factorial Function

The factorial of a positive number n, written as $n!$, is the product of the positive integers from 1 to n:

$n! = 1 \times 2 \times 3 \times \ldots \times (n-2) \times (n-1).n$ where as $0! = 1$

Thus, definition of the iterative version of the factorial function can be written as

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ \prod_{i=1}^{n} i & \text{if } n > 0 \end{cases}$$

From above definition, we have

$0! = 1$
$1! = 1$
$2! = 1 \times 2 = 2$
$3! = 1 \times 2 \times 3 = 6$
$4! = 1 \times 2 \times 3 \times 4 = 24$
$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$ and so on.

Observe that

$4! = 4 \times 3! = 4 \times 6 = 24$ and $5! = 5 \times 4! = 5 \times 24 = 120$

And this is true for every positive integer n; that is

$n! = n \times (n-1)!$

Thus, the factorial function can be defined as

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times n-1)! & \text{if } n > 0 \end{cases}$$

This definition of $n!$ is recursive, since it refers to itself when it uses $(n-1)!$.

Listing 7.1

```
/*
    Program to print the factorials of first 'n' natural numbers
    using recursive function
*/
#include <stdio.h>
int factorial( int n );      /*  function prototype   */
int main()
{
   int i, n;
   printf( "\nEnter value of n: " );
   scanf( "%d", &n );
   for ( i = 1; i <= n; i++ ) {
       printf( "\n%d! = %d", i, factorial(i) );
   }
   return 0;
}
```

```
/*
   Recursive function to compute factorial of a number
*/
int factorial( int n )
{
  if ( n == 0 )
     return 1;
  else
     return ( n * factorial( n – 1 ) );
}
```

Test Run

```
Enter value of n: 6
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
```

7.2.2 Fibonacci Numbers

A very important sequence, *Fibonacci sequence,* usually denoted by F_0, F_1, . . ., F_n, is as follows:

0, 1, 1, 2, 3, 5, 8, 13, . . .

That is, $F_0 = 0$ and $F_1 = 0$ and subsequent terms are sum of the two preceding terms.

For example, the next term in the above sequence is

$$8 + 13 = 21$$

A formal recursive definition for Fibonacci number F_n is:

$$fib(n) = \begin{cases} n & \text{if } n \le 1 \\ fib(n-1) + fib(n-2) & \text{if } n > 1 \end{cases}$$

Notice that the recursive definition of the Fibonacci numbers differs from the recursive definitions of the factorial function in the sense that it refers to itself twice.

Listing 7.2

```
/*
    Program to print first 'n' terms of Fibonacci sequence
    using recursive functions
*/
#include <stdio.h>
int fib( int n );            /*  function prototype   */
int main()
{
```

```
    int i, m;
    printf( "\nEnter value of m: " );
    scanf( "%d", &m );
    for ( i = 0; i < m; i++ )
    {
        printf( "%d ", fib(i) );
    }
    return 0;
}

/*
    Recursive function to compute Fibonacci number 'n'
*/
int fib( int n )
{
  if ( n <= 1 )
     return n;
  else
     return ( fib(n-1) + fib(n-2) );
}
```

Test Run

```
Enter value of m: 8
0   1   1   2   3   5   8   13
```

7.2.3 Ackermann Function

The *Ackermann* function is defined recursively, for all non-negative values of m and n, as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } n = 0 \\ A(m-1, A(m,n-1)) & \text{Otherwise} \end{cases}$$

Listing 7.3

```
/*
    Program to compute Ackermann function
*/
#include <stdio.h>
int ackermann(int m, int n);  /*  function prototype   */
int main()
{
  int m, n;
   printf( "\nEnter value of m : " );
```

```
    scanf( "%d", &m );
    printf( "\nEnter value of n : " );
    scanf( "%d", &n );
    printf( "\nA(%d,%d) = %d\n", m, n, akermann(m,n) );
    return 0;
}
/*  function definition to compute ackermann function A(m,n)  */
int ackermann(int m, int n)
{
  if ( m == 0 )
      return (n+1);
  else if ( n == 0 )
      return ackermann(m-1, 1);
  else
      return ackermann(m-1, ackermann(m,n-1));
}
```

Test Run

```
Enter value of m : 1
Enter value of n : 3
 A(1,3) = 5
```

7.3 QUICK SORT ALGORITHM

Quicksort is a sorting algorithm that uses the idea of *divide and conquer*. This algorithm finds the element, called *pivot*, that partitions the array into two halves in such a way that the elements in the left sub array are less than and the elements in the right sub array are greater than the partitioning element. Then these two subarrays are sorted separately. This procedure is recursive in nature with the *base case* – the number of elements in the array are not more than 1.

Suppose variable *start* and *end* represents the index of the first and last element of the array, the quicksort can be defined recursively as

```
If ( start < end ) then
   Partition the array into two halves
   Quicksort the left half
   Quicksort the right half
Endif
```

Following are some of the variations of the quicksort algorithm, depending on the way pivot element is selected:

- First element as the pivot
- Last element as the pivot
- Random element as the pivot
- Median as the pivot

We will consider the last element as the pivot in our illustration.

To begin with, we set

```
pIndex = start
pivot = a[end]
```

Now, we iterate from *start* to (*end*-1) using index variable *i*, and do the following in each iteration:

```
If ( a[i] < pivot ) then
     Swap a[i] and a[pIndex]
     Increment pIndex
Endif
```

and, finally

```
swap a[pIndex] and a[end]
```

Put together, the following function accomplishes the task of partitioning.

```
int partition(int a[], int start, int end)
{
   int i, temp;
   int pIndex = start;
   int pivot  = a[end];
   for( i = start; i < end; i++ )
   {
      if ( a[i] < pivot )
      {
          temp = a[i];
          a[i] = a[pIndex];
          a[pIndex] = temp;
          pIndex++;
      }
   }
   temp = a[end];
   a[end] = a[pIndex];
   a[pIndex] = temp;
   return pIndex;
}
```

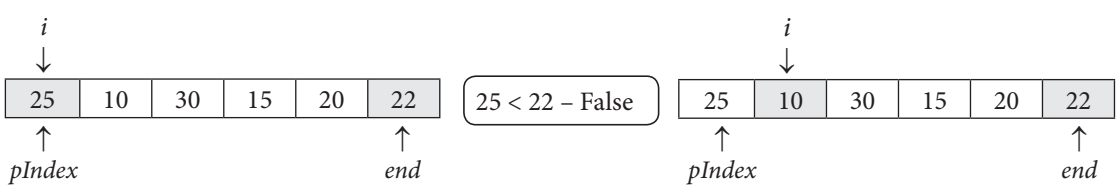
Example 7.1: *To illustrate the partitioning process, consider the following array*

25 10 30 15 20 28

Solution: In the given array, *start* = 0, *end* = 5

Here the last element (22) is taken as pivot.

To begin with , we take *pIndex* = *start*, *i* = *start*



Step 1

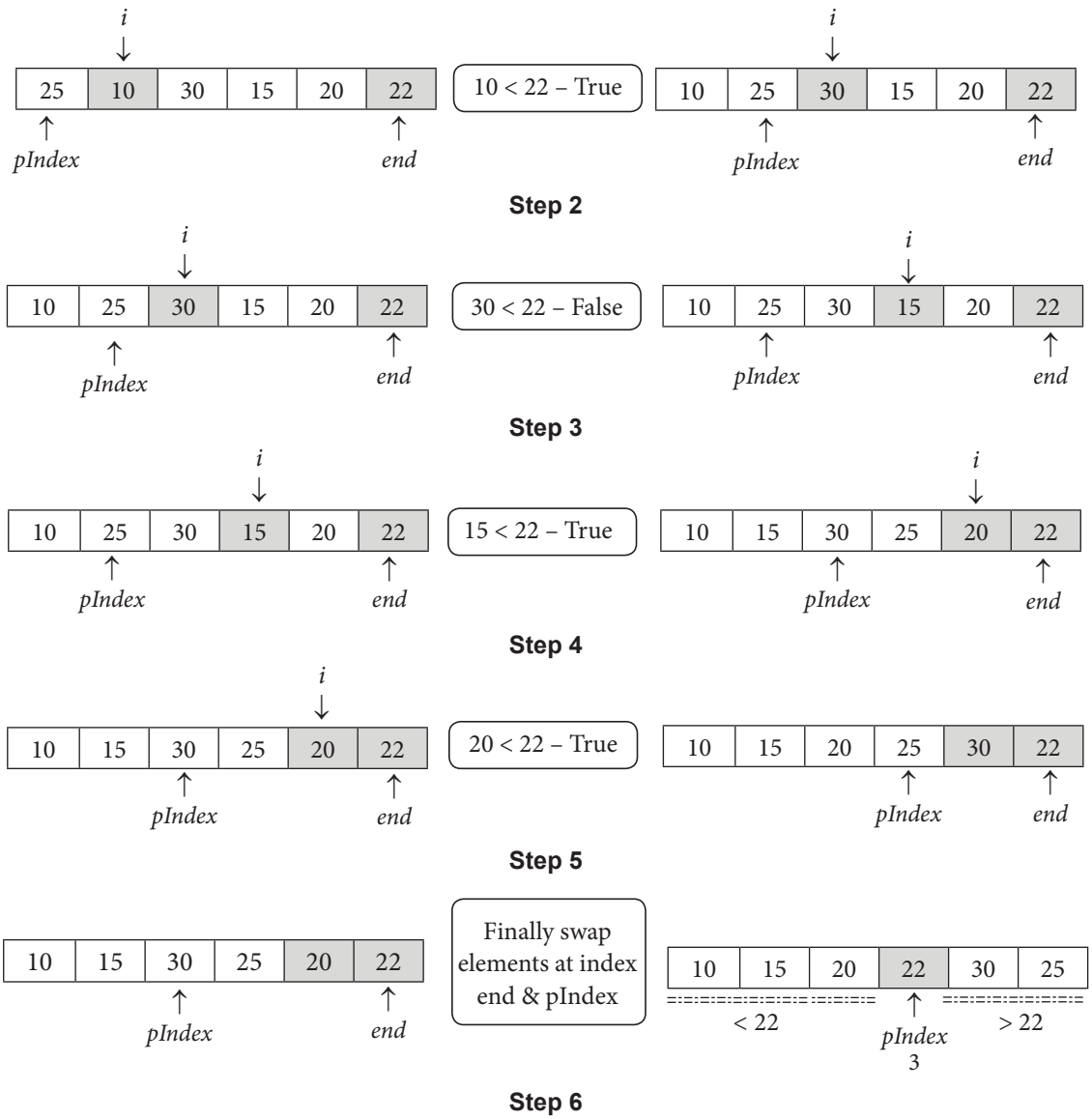


Fig. 7.1: Illustration of partitioning of array

The pivot element 22 is placed in its final position and it divides the array into two sub arrays as

10 15 15 and 30 25

As you can see, the elements in the left sub array are smaller than 22, and elements in the right sub array are greater than 22.

This means that element 22 is correctly placed in its final position and it partitions the remaining elements in two sub arrays where elements in the left sub array are less than it and elements in the right sub array are greater it.

The above partitioning step is repeated with each sub array containing 2 or more elements. Since, we can work on one sub array at a time, we must be able to keep track of the second sub array. This task is accomplished either using stack explicitly (iterative implementation) or implicitly (recursive implementation). Both the implementations are given in this section.

Listing 7.4

```
/*
    Program to sort an array of integers in ascending order using
    Quick sort method
*/
#include<stdio.h>
/*  function prototypes  */
void quickSortRecursive(  int a[], int lb, int ub );
int partition(int a[], int start, int end);

int main()
{
   int i, n, a[20];
   printf( "\nEnter size of array n(<=20) : " );
   scanf( "%d", &n );
   printf( "\nEnter %d integer elements of array\n\n", n );
   for ( i = 0; i < n; i++ )
      scanf( "%d", &a[i] );
   quickSortRecursive( a, 0, n-1 );
   printf( "\n\nSorted list of elements\n\n" );
   for ( i = 0; i < n; i++ )
      printf( "%d ", a[i] );
   printf( "\n");
} /*---- end of main function ----*/

void quickSortRecursive( int a[], int lb, int ub )
{
   int pIndex;
   if ( lb < ub )
   {
      pIndex = partition( a, lb, ub);
      quickSortRecursive( a, lb, pIndex-1 );
      quickSortRecursive( a, pIndex+1, ub );
  }
}
int partition(int a[], int start, int end)
{
   int i, temp;
   int pIndex = start;
```

```
    int pivot  = a[end];
    for(i = start; i < end; i++)
    {
       if (a[i] < pivot)
       {
           temp = a[i];
           a[i] = a[pIndex];
           a[pIndex] = temp;
           pIndex++;
       }
    }
    temp = a[end];
    a[end] = a[pIndex];
    a[pIndex] = temp;

    return pIndex;
}
```

Test Run

```
Enter size of array n(<=20) : 6
Enter 6 integer elements of array
25 10 30 15 20 28
Sorted list of elements
10 15 20 25 28 30
```

7.4 MERGE SORT ALGORITHM

Merge sort is another sorting algorithm that uses the idea of *divide and conquer*. This algorithm divides the array into two halves, sorts them separately, and then merges them.

This procedure is recursive, with the *base case* — the number of elements in the array are just 1.

Suppose variable *beg* and *end* represents the index of the first and last element of the array respectively, the merge sort can be defined recursively as

```
If ( beg < end ) then
   Divide the list into two halves
   Mergesort the left half
   Mergesort the right half
   Merge the two-sorted halves into one sorted list
Endif
```

Example 7.2: *To illustrate the working of the merge sort method, consider the following array a with 7 elements as*

33, 26, 35, 29, 18, 10, 24

Solution: The first step of mergesort is to divide the array into two sub arrays. Thus we divide the array into

33, 26, 35, 29 and 18, 10, 24

and first consider the left sub array. It is again divided into two sub arrays as

33, 26 and 35, 29

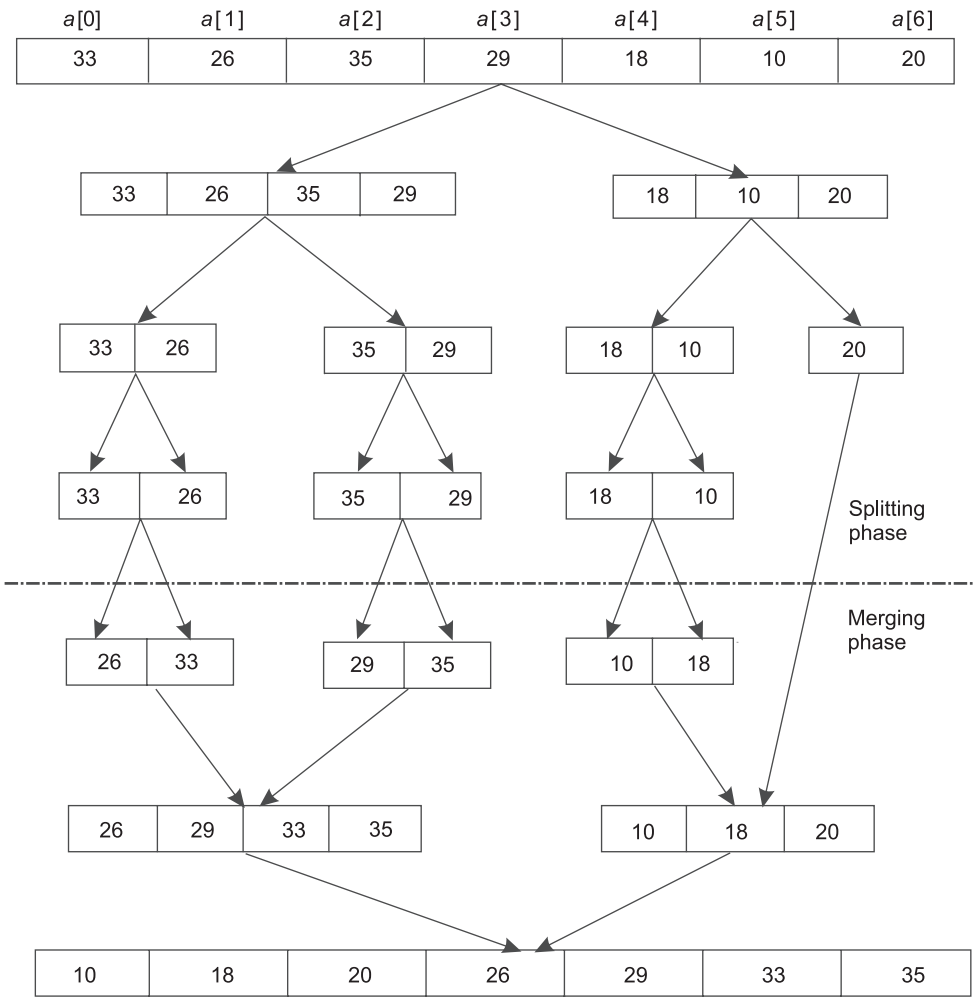


Fig. 7.2: Illustration of successive steps of Merge sort

For each of these sub arrays, we again apply the same method, dividing each of these into sub arrays of one element each. Sub arrays of size one, of course, require no sorting. Finally, we start merging the sub arrays to obtain a sorted array.

The sub arrays 33 and 26 merge to give the sorted array 26, 33 and the sub arrays 35 and 29 merge to give sorted array 29, 35.

At the next step, we merge these two-sorted sub arrays of size two to obtain a sorted array of size four as

26, 29, 33, 35

Now the left half of the given array is sorted, we do the same steps on the right half. First, we divide it into two sub arrays as

18, 10 and 24

The first of these is divided into two sub arrays of size one, which are merged to give 10, 18. The second sub array, 24, has size one, so needs no sorting. Now these are merged to give the sorted array

10, 18, 24

Finally, the sorted sub arrays of size four and three are merged to give

10, 18, 24, 26, 29, 33, 35

The above sorting process can also be visualized as shown in Fig. 7.2.

Listing 7.5

```
/*
   Program to sort an array of integers in ascending order using
   merge sort method
*/
#include<stdio.h>
/*----  function prototype  ----*/
void mergeSortMethod(int a[], int beg, int end );
void mergingSortedSubArrays(int a[],int lb, int lr,int rb, int  rr);
int main()
{
   int i, n, a[20];
   printf( "\nEnter size of array n(<=20) : " );
   scanf( "%d", &n );
   printf( "\nEnter %d integer elements of array\n\n", n );
   for ( i = 0; i < n; i++ )
      scanf( "%d", &a[i] );
   mergeSortMethod( a, 0, n-1 );
   printf( "\n\nSorted list of elements\n\n" );
   for ( i = 0; i < n; i++ )
   {
      printf( "%d ", a[i] );
   }
   printf( "\n");
} /*---- end of main function ----*/
```

```
void mergeSortMethod( int a[], int beg, int end )
{
   int mid;
   if ( beg < end )
   {
       mid = ( beg + end ) / 2;
       mergeSortMethod( a, beg, mid );
       mergeSortMethod( a, mid+1, end );
       mergingSortedSubArrays( a, beg, mid, mid+1, end );
   }
}
void mergingSortedSubArrays(int a[], int lb, int lr, int rb, int rr)
{
   int na, nb, nc, k, c[MAX];
   na = lb;
   nb = rb;
   nc = lb;
   while ( ( na <= lr ) && ( nb <= rr ) )
   {
        if ( a[na] < a[nb] )
           c[nc] = a[na++];
        else
           c[nc] = a[nb++];
        nc++;
   }
   if ( na > lr ) {
      while ( nb <= rr )
          c[nc++] = a[nb++];
   } else {
      while ( na <= lr )
          c[nc++] = a[na++];
   }
   for ( k = lb; k <= rr; k++ )
      a[k] = c[k];
}
```

Test Run

```
Enter size of array n(<=20) : 7
Enter 7 integer elements of array
33 26 35 29 18 10 20
Sorted list of elements
10 18 20 26 29 33 35
```

7.5 RECURSION, ITERATION OR . . .?

When there are number of ways of solving a problem, the obvious question that arises is *"Which one should be selected?"*

There is no set of pre-defined rules to select a particular way; however, we must consider the following issues:

- Processing time taken.
- Computer memory used.
- Time taken to develop the program.
- Time taken to debug the program.
- Time to maintain the program.

As a generalization, recursive solutions tend to do well in categories 3, 4, and 5. Because recursive solutions tend to be simple and small, the time required to modify the programs later is less than the time needed for non-recursive solution of the same problem, if it does exists.

On the other hand, in general, recursive solutions do not tend to do well in their use of processing time and the amount of computer memory they require.

Table 7.1 highlights key points of comparison of recursion and iteration

Table 7.1: Comparison: Recursion vs Iteration

Criteria	Recursion	Iteration
Mode of Implementation	Using function call(s) to itself	Using loops
State	Defined by the argument value(s) stored in stack	Defined by the value of the control variable
Progression	Function state converges towards the base case	Value of control variable moves towards the final value
Termination	Base case is reached	Control variable satisfies the condition
No Termination State	Infinite recursive calls may occur due to some mistake in specifying the base case, and as a result the function keeps calling itself, which may lead system to crash.	Infinite loop due to mistake in assignment, increment, or terminating condition, and will result in program execute endlessly
Code size	Tends to be very small	Tends to be large
Execution	Execution is slower	Execution is faster

ILLUSTRATIVE EXAMPLES

Example 7.3: *Write a program to find the sum of first n natural numbers using recursion.*

The function to find the sum of first n natural numbers, say findSum(n), can be recursively defined as

$$\text{findSum}(n) = \begin{cases} 1 & \text{if } n = 1 \\ n + \text{findSum}(n-1) & \text{if } n > 1 \end{cases}$$

Listing 7.6

```
/*
   Program to find the sum of first 'n' natural numbers
   using recursion
*/
#include<stdio.h>
int findSum(int n);   /*  function prototype  */
int main()
{
   int n, sum;
   printf("\nEnter value for n : ");
   scanf("%d", &n);
   sum = findSum(n);
   printf("\nSum of first %d natural numbers = %d\n\n", n, sum);
   return (0);
}
/*
    definition of recursive function that returns
    sum of first 'n' natural numbers
*/
int findSum(int n)
{
   if ( n == 1 )
      return 1;
   else
      return ( n + findSum(n-1) );
}
```

Test Run

```
Enter value for n : 10
Sum of first 10 natural numbers = 55
```

Example 7.4: *Write a program to compute positive exponential power* (x^n) *using recursion.*

The positive exponential power function (x^n) can be recursively defined as

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \times x^{n-1} & \text{if } n > 0 \end{cases}$$

Listing 7.7

```
/*
   Program to find the value of the positive exponential power
   function using recursion
*/
#include<stdio.h>
```

```
float power(float x, int n);  /*  function prototype  */
int main()
{
   int n;
   float x;
   printf("\nEnter value for x : ");
   scanf("%f", &x);
   printf("\nEnter value for n : ");
   scanf("%d", &n);
   printf("\nValue of power function = %.2f\n\n", power(n));
   return (0);
}
/*
    definition of recursive function that returns value
    positive exponential power function (x^n)
*/
float power(float x, int n)
{
   if ( n == 0 )
      return 1;
   else
      return ( x * power(x,n-1) );
}
```

Test Run

```
Enter value for x : 5.25
Enter value for n : 2
Value of power function = 27.56
```

Example 7.5: *Write a program to find the greatest common divisor (GCD) to natural numbers m and n using recursion.*

The GCD function can be recursively defined as

$$\gcd(m,n) = \begin{cases} n & \text{if } m\%n = 0 \\ \gcd(n, m\%n) & \text{if } m\%n \neq 0 \end{cases}$$

Listing 7.8

```
/*
    Program to compute GCD of natural numbers m and n
    using recursion
*/
int gcd( int m, int n );   /*  function prototype */
void main()
{
```

```
    int m, n;
    printf( "\nEnter value of m : " );
    scanf( "%d", &m );
    printf( "\nEnter value of n : " );
    scanf( "%d", &n );
    printf( "\nValue of GCD(%d,%d) = %d\n", m, n, gcd(mn,n) );
}
/*
     definition of recursive function that returns GCD of
     natural numbers 'm' and 'n'
*/
int gcd( int m, int n )
{
    if ( m % n == 0 )
       return n;
    else
       return gcd( n, m % n );
}
```

Test Run

```
Enter value of m: 35
Enter value of n: 125
Value of GCD(35,125) = 5
```

Example 7.6: *Write a program to convert a given decimal number n to its equivalent binary number using recursion.*

Listing 7.9

```
/*
   Program to convert decoml number to equivalent binary number
   function using recursion
*/
#include<stdio.h>
void convert(int n);  /*  function prototype  */
int main()
{
   int n;
   printf("\nEnter decimal number : ");
   scanf("%d", &n);
   printf("\nBinary equivalent of %d = ",n);
   convert(n);
   printf("\n");
```

```
    return (0);
}
/*
    definition of recursive function that convert
    decimal number to binary
*/
void convert(int n)
{
   if ( n > 0 )
   {
      convert(n/2);
      printf("%d", n%2);
      }
}
```

Test Run

```
Enter decimal number : 105
Binary equivalent of 105 = 1101001
```

UNIT SUMMARY

In this chapter, we have learned that

- Recursion is a powerful concept in the development of programs.
- A function that calls itself is known as a *recursive function*.
- Recursive functions can be directly implemented in C.
- The popular examples of recursive functions include factorial & Fibonacci numbers.
- Quick sort is a sorting algorithm that uses the idea of *divide and conquer*. It finds the element, called *pivot*, that partitions the array into two halves in such a way that the elements in the left sub array are less than and the elements in the right sub array are greater than the partitioning element. Then these two subarrays are sorted separately.
- Merge sort is another sorting algorithm that uses the idea of *divide and conquer*. This algorithm divides the array into two halves, sorts them separately, and then merges them.
- Recursive solutions generally take less time to develop and debug the program, and are easy to maintain than iterative solutions.
- On the other side, recursive solutions take more processing time and consume more memory than iterative solutions.

EXERCISE

Subjective Questions

1. What is a recursion?
2. When a recursive function said to be well-defined?
3. What do you mean by base case?
4. How do a recursive function compare with iterative function?
5. Is it possible to convert a recursive function to iterative function? If yes, give an example.
6. Is it possible to convert an iterative function to recursive function? If yes, give an example.
7. Describe in brief the working of quick sort algorithm.
8. Describe in brief the working of merge sort algorithm.
9. How do recursion compare to iteration?

Multiple Choice Questions

1. Which of the following problems can't be solved using recursion?
   (*a*) Factorial of a number
   (*b*) *n*th fibonacci number
   (*c*) Length of a string
   (*d*) Problems without base case
2. In recursion, the condition for which the function will stop calling itself is _____ .
   (*a*) Best case
   (*b*) Worst case
   (*c*) Base case
   (*d*) There is no such condition
3. Which of the following statements is true?
   (*a*) Recursion is always better than iteration
   (*b*) Recursion uses more memory compared to iteration
   (*c*) Recursion uses less memory compared to iteration
   (*d*) Iteration is always better and simpler than recursion
4. What will happen when the below code snippet is executed?

```
void fun()
{
    fun();
}
int main()
{
    fun();
    return 0;
}
```

   (*a*) The code will be executed successfully and no output will be generated
   (*b*) The code will be executed successfully and random output will be generated

(*c*) The code will show a compile time error

(*d*) The code will run for some time and stop when the stack overflows

5. What is the output of the following code?

```
void fun(int n)
{
    if(n == 0)
    return;
    printf("%d ",n);
    fun(n-1);
}
int main()
{
    fun(10);
    return 0;
}
```

(*a*) 10 (*b*) 1 (*c*) 10 9 8 … 1 0 (*d*) 10 9 8 … 1

6. What is the base case for the following code?

```
void fun(int n)
{
     if (n == 0)
     return;
     printf("%d ",n);
     fun(n-1);
}
int main()
{
     fun(10);
     return 0;
}
```

(*a*) return (*b*) printf("%d ", n) (*c*) if(n == 0) (*d*) fun(n-1)

7. What will be the output of the following C code?

```
#include<stdio.h>
int main()
{
    printf("Hello");
    main();
    return 0;
}
```

(*a*) Hello is printed once (*b*) Hello infinite number of times

(*c*) Hello is not printed at all (*d*) 0 is returned

8. What will be the output of the following C code?

```
fun(int x)
{
    int b;
    if(x==1)
        return 1;
    else
        b=x*fun(x-1);
        return b;
}
int main()
{
    int n;
    n=fun(4);
    printf("%d",n);
    return 0;
}
```

(*a*) 24 (*b*) 4 (*c*) 12 (*d*) 10

9. What will be the output of the following C code?

```
int fun(int x)
{
    if(x==2)
       return 2;
    else
    {
        printf("+");
        fun(x-1);
    }
}
int main()
{
    int n,i;
    n=fun(6);
    printf("%d",n);
    return 0;
}
```

(*a*) ++++2 (*b*) +++++2 (*c*) +++++ (*d*) 2

10. How many times is 'a' printed when the following C code is executed?

```
int fun(int b)
{
    if(b==0)
        return 0;
    else
    {
        printf("a");
        fun(b--);
    }
}
int main()
{
    int a;
    a=fun(10);
    printf("%d",a);
    return 0;
}
```

(*a*) 9 times (*b*) 10 times

(*c*) 0 times (*d*) Infinite number of times

11. What will be the output of the following C code?

```
int f(int n)
{
    if(n>0)
        return(n+f(n-2));
}
main()
{
    int n=10;
    int f(int n);
    printf("%d",f(n));
}
```

(*a*) 10 (*b*) 80 (*c*) 30 (*d*) Error

12. Find the output of the following pseudo-code if $x = 4$ and $y = 5$:

```
int fun(int x, int y)
{
   if(x > 1)
      fun(x - 2, y + 2);
   printf("%d", y);
}
```

(*a*) 5 6 (*b*) 7 6 5 (*c*) 9 7 5 (*d*) 5 7 9

13. What will be the output of the following pseudocode for n=2?

```
int fun ( int n)
{
       if ( n == 4 )
       return n;
   else
       return 2*fun(n+1);
}
```

(*a*) 2 (*b*) 16 (*c*) 8 (*d*) 4

14. What will be the output of the following pseudocode for input n =134?

```
        int fun (int n)
{
   static int a = 0;
   if ( n > 0 ) {
       a = a + 1;
       fun(n/10);
   }
   else
       return a;
}
```

(*a*) 8 (*b*) 3 (*c*) 2 (*d*) 431

15. What will be the output of the following C program?

```
int f( int x )
{
if ( x <= 0 )
return 1;
return f(x-1) + x;
}
int main()
{
printf("%d", f(5) );
return 0;
}
```

(*a*) 12 (*b*) 16 (*c*) 15 (*d*) 11

16. What does the following function print for n = 25?

```
void fun(int n)
{
    if (n == 0)
```

```
        return;
    printf("%d", n%2);
    fun(n/2);
}
```

(*a*) 11001 (*b*) 11111 (*c*) 00000 (*d*) 10011

17. Consider the following C recursive function fun(x, y). What is the value of fun(4, 3)?

```
int fun(int x, int y)
{
   if (x == 0)
       return y;
   return fun(x - 1,  x + y);
}
```

(*a*) 13 (*b*) 12 (*c*) 10 (*d*) 9

18. What does the following C function print for $n = 25$?

```
void fun(int n)
{
    if (n == 0)
       return;
    fun(n/2);
    printf("%d", n%2);
}
```

(*a*) 11001 (*b*) 11111 (*c*) 00000 (*d*) 10011

19. Select the correct output.

```
int rec(int num)
{
    return (num) ? num%10 + rec(num/10):0;
}
int main()
{
    printf("%d",rec(4567));
    return 0;
}
```

(*a*) 4 (*b*) 12 (*c*) 22 (*d*) 21

20. Select the correct output.

```
int doSomething(int a, int b)
{
 if (b==1)
    return a;
 else
```

```
    return a + doSomething(a,b-1);
}
int main()
{
    int k;
    k = doSomething(2,3);
    printf("%d", k);
    return 0;
}
```

(*a*) 4 (*b*) 6 (*c*) 5 (*d*) 7

ANSWERS									
1.	(*d*)	2.	(*c*)	3.	(*b*)	4.	(*d*)	5.	(*d*)
6.	(*c*)	7.	(*b*)	8.	(*a*)	9.	(*a*)	10.	(*d*)
11.	(*c*)	12.	(*c*)	13.	(*b*)	14.	(*b*)	15.	(*b*)
16.	(*d*)	17.	(*a*)	18.	(*a*)	19.	(*c*)	20.	(*b*)

Programming Problems

1. Write a recursive function that prints first *n* natural number in the reverse order. For example, if *n* = 10, output should be 10 9 8 7 6 5 4 3 2 1.
2. Write a recursive function to find the largest element of an array.
3. Write a recursive function to reverse the order of elements of an array.
4. Write a recursive function to find the greatest common divisor of two natural numbers.
5. Write a recursive function to find the product of two natural numbers.
6. Write a recursive function to find factorial of a natural number.
7. Write a recursive function to find the sum of digits of a natural number.
8. Write a recursive function to find the convert a decimal number to binary number.

PRACTICALS

1. Write a program to find the factorial of a natural number *n* using recursion.
   Refer to Listing 7.1
2. Write a program to find the *n*th Fibonacci number using recursion.
   Refer to Listing 7.2
3. Write a program to convert a decimal number to its equivalent binary number using recursion.
   Refer to Listing 7.9

KNOW MORE

The topic of recursion is another important topic in problem solving and is used extensively to solve many real-life problems.

The teacher is expected to develop an understanding about the concepts of recursion, and to solve problem using recursive function with students participation.

REFERENCES & SUGGESTED READINGS

1. R. S. Salaria, Problem Solving & Programming in C, Khanna Book Publishing Co(P) Ltd., New Delhi.
2. E. Balagurusamy, Programming in ANSI C, Tata McGraw Hill, New Delhi.
3. Yashavant Kanetkar, Let Us C, BPB Publications, New Delhi.
4. Byron Gottfried, Programming with C, Schaum's Outlines.
5. https://onlinecourses.nptel.ac.in/noc21_cs01/preview
6. https://ocw.mit.edu/courses/intro-programming/
7. https://www.programiz.com/c-programming
8. https://www.javatpoint.com/c-programming-language-tutorial

8 Structures

UNIT SPECIFICS

This unit discusses the topics related to structures. Structures are used to store data pertaining to an entity, where an entity can be an employee, customer, vehicle or book. This unit explains the various aspects of structures and demonstrates their use with suitable examples.

RATIONALE

There can be real-life problems where we may have to deal with the collection of data where individual data items can belong to different data types. For example, in an educational institution, there is a need to store information of students that may include *roll number*, *name*, *father's name*, *mother's name*, *date of birth*, *email-id*, *mobile number*, *address*, *details of marks of each subject of each semester, fee details*, etc. These data items belong to different data types such as integer, long integer, string, float, etc.

Therefore, we need a mechanism to hold this kind of data collection as a single unit, and the answer to this question is *the structure*.

This unit will help the students to understand the various aspects of structures, and develop programs using structures to solve real-life problems.

PRE-REQUISITES

- Basic data types
- Conditional Branching
- Looping
- Arrays and strings
- Functions

UNIT OUTCOMES

Upon completion of the unit, students will be able to

U8-O1: explain the concept of structure

U8-O2: explain different ways of defining a structure

U8-O3: declare and initialize structure variables

U8-O4: declare and initialize the array of structures

U8-O5: develop programs for real-life problems using structures

Unit 8 Outcomes	EXPECTED MAPPING WITH COURSE OUTCOMES (1- Weak Correlation; 2- Medium correlation; 3- Strong Correlation)							
	CO-1	CO-2	CO-3	CO-4	CO-5	CO-6	CO-7	CO-8
U8-O1	-	-	-	-	-	1	-	-
U8-O2	-	-	-	-	-	2	-	-
U8-O3	-	-	-	-	-	2	-	-
U8-O4	-	-	-	-	-	2	-	-
U8-O5	-	-	-	-	-	3	-	-

8.1 INTRODUCTION

You can think of a structure as an array of closely related items. However, unlike arrays, a structure is a collection of related data items that can belong to different data types. The data items that make a structure occupy contiguous memory locations, the same way elements of an array are stored in contiguous memory locations.

Structures can be used to organize complex data in a more meaningful way.

Suppose we want to store information of employees in an organization. This information may include the employee's name (character array), department code (integer number), salary (floating-point number), and so forth. And we want our program to deal with them as elements of an array.

One possible approach is to use several arrays — a character array for names, an integer array for department code, a floating-point array for salary, and so forth. And using a common index we can access the information for a particular employee. Still this approach obscures the fact that we deal with data items related to a single entity, *an employee* in our example.

To solve this sort of problem, the C language provides a special data type: *the structure*. As already mentioned, you can think of a structure as a group of data items of different data types held together in a single unit. In our example, the structure would consist of the employee's name, department number, salary, and so forth.

Here are few more examples of structures

Name of Structure	Probable Constituent Data Items
Date	day, month, year
Time	hours, minutes, seconds
Book	title, author, publisher, price
Address	name, house number, locality, city, state, pincode
Student	roll number, name, father's name, grade
Customer	name, address, mobile no
Inventory	item code, description, quantity, unit price
Employee	employee id, name, address, mobile no

In this unit, we learn about various aspects associated with structures.

8.2 DEFINING A STRUCTURE

The C language has two ways to define a structure: *tagged structure* and *type-defined structure*.

Tagged Structure

The first way to define a structure is to use a tagged structure. The syntax for declaring a tagged structure is

```
struct STUDENT
{
    . . .
    Field list
    . . .
};
```

(*a*) Format

```
struct STUDENT
{
    int rollNo;
    char name[20];
    char fname[20];
    char grade;
};
```

(*b*) Example

Fig. 8.1: Defining a tagged structure

The tagged structure starts with the keyword *struct*. The next element in the definition is *tag*. The tag is the identifier for the structure, and will be used to declare variable, arguments of the functions, and return types for the functions.

If we conclude the structure definition with a semicolon after the closing brace, no variables are declared. In this case, the structure is simply a type template with no associated storage.

Type-defined Structure

The more powerful way to define a structure is by using type definition *typedef*. The syntax for declaring a type-defined structure is

```
typedef struct
{
    . . .
    Field list
    . . .
} TYPE;
```

(*a*) Format

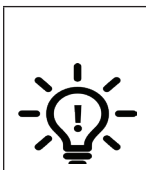
```
typedef struct
{
    int rollNo;
    char name[20];
    char fname[20];
    char grade;
} STUDENT;
```

(*b*) Example

Fig. 8.2: Defining a type-defined structure

The type-defined structure differs from tagged structure in the following aspects:

- The keyword *typedef* is used before the *struct* keyword.
- Identifier after the *struct* keyword is not mandatory as is the case with tagged structure.
- The identifier is required after the closing brace and before the semicolon.
- As we will see later, the variable can be declared with the declaration of the tagged structure whereas it cannot be with type-defined structures.



A type definition, *typedef*, give a name to a data type by creating a new type that can be used anywhere a type is permitted. The syntax for type definition is

```
typedef dataType IDENTIFIER;
```

where *dataType* is either built-in data type or user-defined data type, and *IDENTIFIER*, usually in uppercase, is the new and convenient name for the *dataType*. The *typedef* keyword tells the compiler to recognize the *IDENTIFIER* as synonymous of *dataType*.

Following are few more examples of definitions of structures:

```
struct Date
{
  int day;
  int month;
  int year;
};

struct Time
{
  int hours;
  int minutes;
  int seconds;
};

struct Book
{
  char title[25];
  char author[25];
  char publisher[15];
  float price;
};

struct Customer
{
  char name[25];
  char address[80];
  long mobile_no;
};

struct Inventory
{
  char item_code[25];
  char desc[25];
  int qty;
  float unit_price;
};

struct Address
{
  char name[25];
  int house_no;
  char locality[25];
  char city[15];
  char state[15];
  long pincode;
};
```

8.3 DECLARING STRUCTURE VARIABLES

Once a structure is defined, we can declare variables using these declarations. Usually, the definition is placed in the global area of the program, *i.e.*, before the *main()* function, so that it is visible to all the functions in the program. The variables, on the other hand, are usually declared in the local area of the functions or in the argument list in the functions headers. The following statements shows the way structure variables are declared.

For tagged structure, the variable is declared as

```
struct STUDENT aStudent;
```

For type-defined structure, the variable is declared as

```
STUDENT bStudent;
```

By comparing both the declarations, you will find the type-defined structure is handier in the declarations of the variable because you don't need to use the *struct* keyword. This can be quite a time and effort-saving if the variable declarations are required at many places in the program.

Further note that, in the case of tagged structure, the variable declaration can be combined with its definition as shown below.

```
struct EMPLOYEE
{
  int   code;
  char  name[25];
  char  department[15];
  float salary;
} aEmployee, bEmployee;
```

Usually, this approach of declaring variables is not recommended, for the following reasons:

1. Since structure definition is placed in the global area, and so the variables will also be global.
2. If we want to declare local variables using this approach, we may have to place the structure definition in local scope, making the definition of structure invisible to other functions.

8.4 INITIALIZING STRUCTURES

Like simple variables and arrays, structure variables can also be initialized at the time of declaration. The format used is quite similar to that used to initialize arrays.

The following segment illustrates this

```
typedef struct
{
  int   rollno;
  char  name[25];
  int   age;
  float height;
} STUDENT;
STUDENT aStudent = { 1000, "Surbhi", 18, 5.6 };
```

8.5 ACCESSING STRUCTURE ELEMENTS

Individual structure elements of structure variables can be accessed using the *dot operator* '.' This dot operator is also referred to as *membership operator*.

The syntax for accessing a structure element is as follows

```
sname.vname
```

where *sname* is the name of the structure variable, and *vname* is the name of the data field.

Consider the following statements

```
struct Student
{
  int  rollno;
  char name[25];
  char fname[25];
  char grade;
};
Student aStudent;;
```

The individual elements of structure variable *aStudent* will be accessed as shown below:

```
aStudent.rollno        // reference to element rollno
aStudent.name          // reference to element name
aStudent.fname         // reference to element fname
aStudent.grade         // reference to element grade
```

8.6 ASSIGNING OF STRUCTURES

A structure is an entity that can be treated as a whole. There is only one operation that can be performed on the structure as a whole is that a structure can be assigned/copied into another structure of the same type using the assignment operator.

Again consider the following statements

```
typedef struct
{
  int   rollno;
  char  name[25];
  int   age;
  float height;
} STUDENT;

STUDENT aStudent, bStudent;
```

The following assignment statement

```
aStudent = bStudent;
```

assigns the contents of structure *bStudent* to *aStudent*.

It is a unique feature of structures as compared to arrays, where in order to copy one array to another array of the same type, we have to set up a loop to copy element-by-element.

This is rather an amazing capability when you think about it. When you assign one structure to another, all the values in the structure are assigned to corresponding structure elements. Simple assignment statements cannot be used this way for arrays. That is, to assign one array to another, we have to assign element by element.

8.7 READING/WRITING STRUCTURES

We can read into and write data from a structure element in the same manner as we do with simple variables.

For example, the elements of *aStudent* structure, we can read data, interactively, from the keyboard and place in the *aStudent* structure using the following statements

```
printf("\nEnter roll number of the student : ");
scanf("%d", &aStudent.rollno);
printf("\nEnter name of the student : ");
scanf("%d", aStudent.name);
printf("\nEnter age of the student : ");
scanf("%d", &aStudent.age);
printf("\nEnter height of the student : ");
scanf("%d", &aStudent.height);
```

Note that in the *scanf()* function, the expression

```
&aStudent.rollno
```

is interpreted by the compiler as

```
&(aStudent.rollno)
```

because of higher precedence of direct selection operator over *addressof* operator (&), and that is what exactly is required.

Listing 8.1

```
/*
    Program to demonstrate input/output of structures
*/

#include<stdio.h>
#include<sting.h>

struct EMPLOYEE
{
   int code;
   char name[25];
   char department[15];
   float salary;
};

int main()
{
   struct EMPLOYEE aEmployee;
   printf("Enter employee's code: " );
```

```
    scanf( "%d", &aEmployee.code );
    fflush(stdin);
    printf("Enter employee's name: " );
    gets( aEmployee.name );
    printf("Enter employee's department: " );
    gets( aEmployee.department );
    printf("Enter employee's salary: " );
    scanf("%f", &aEmployee.salary );
    printf("\n\nParticulars of employee as" );
    printf(" entered by user\n" );
    printf("\nEmployee's code: %d", aEmployee.code );
    printf("\nEmployee's name: %s", aEmployee.name );
    printf("\nEmployee's department: %s",aEmployee.department);
    printf("\nEmployee's salary: %.2f\n", aEmployee.salary );
    return 0;
}
```

Test Run

```
Enter employee's code: 826
Enter employee's name: Rajesh Kumar
Enter employee's department: Computer Science
Enter employee's salary: 20000

Particulars of employee as entered by user

Employee's code: 826
Employee's name: Rajesh Kumar
Employee's department: Computer Science
Employee's salary: 20000.00
```

8.8 ARRAYS OF STRUCTURES

In many practical situations, we need to create an array of structures. As an example, we need to create an array of students to work with a group of students in a class. By putting the data of students in an array of structures, we can quickly and efficiently process the student's data.

An array of structures to store the data of students will be declared as

```
STUDENT students[50];
```

8.8.1 Accessing Elements of Array of Structures

Individual elements of a structure in an array of structures are accessed by referring to structure variable name, followed by index, direct selection operator, and ending with the desired structure element.

In our example of students of a class, the name of the *i*th student can be accessed as follows:

```
students[i].name;
```

8.8.2 Initializing Array of Structures

Like any other array, an array of structures can also be initialized. Since each element of array structures is a different structure, we need to include elements of each structure in a separate set of braces as shown:

```
STUDENT students[50] = { { 1000, "Monika",{56,76,85,69}, 'A' },
                         { 1001, "Ram", {50,66,70,60}, 'B' },
                                 :
                         { 1049, "Raju", {65,62,68,59}, 'B' },
                       };
```

The following program illustrates the input, processing and output of an array of structures.

Listing 8.2

```
/*
   Program to illustrate processing of array of structures by
   storing list of students in memory, and computes their grades
*/

#include <stdio.h>

typedef struct {
   int  rollno;
   char sname[25];
   int  marks[4];
   char grade;
} STUDENT;

int main()
{
   STUDENT students[50];
   int totalMarks, i, j, n;
   printf( "\n\nEnter number of students in a class : " );
   scanf( "%d", &n );
   for ( i = 0; i < n; i++ )
   {
      printf("\nParticulars of student #%d\n\n", i+1);
      printf( "Enter students' roll number : " );
      scanf("%d", &students[i].rollno );
      printf( "\nEnter students' name : " );
      gets( students[i].sname );
      printf( "\nEnter students marks in four subjects\n\n" );
      for ( j = 0; j < 4; j++ )
         scanf("%d", &students[i].marks[j] );
   }
   for ( i = 0; i < n; i++ )
   {
```

```
      totalMarks = 0;
      for ( j = 0; j < 4; j++ )
            totalMarks += students[i].marks[j];
      if ( totalMarks > 75 )
            students[i].grade = 'A';
      else if ( totalMarks > 60 )
            students[i].grade = 'B';
      else if ( totalMarks > 50 )
            students[i].grade = 'C';
      else if ( totalMarks > 40 )
            students[i].grade = 'D';
      else
            students[i].grade = 'F';
   }
   printf("\nPerformance of students\n\n");
   printf("%-10s", "Roll No.");
   printf("%-30s", "Name");
   printf("%6s\n", "Grade");
   for ( i = 0; i < n; i++ )
   {
      printf("%-10d", students[i].rollno);
      printf("%-30s", students[i].sname);
      printf("%4c\n", students[i].grade);
   )
   return 0;
}
```

Test Run

```
Enter number of students in a class : 3

Particulars of student #1

Enter students' roll number : 1000
Enter students' name : Ram Kumar
Enter students marks in four subjects
98 99 89 88

Particulars of student #2

Enter students' roll number : 1001
Enter students' name : Mohit Kumar
Enter students marks in four subjects
65 55 45 60
```

```
Particulars of student #3

Enter students' roll number : 1002
Enter students' name : Jaspreet Singh
Enter students marks in four subjects
41 44 45 50

Performance of students

SNo.           Student's Name     Grade
  1                     Sonu         A
  2                   bhollu         C
  3                 jaspreet         D
```

8.9 PASSING STRUCTURE TO A FUNCTION

In order to be fully useful, we must be able to pass structures to functions as arguments and returning them from functions.

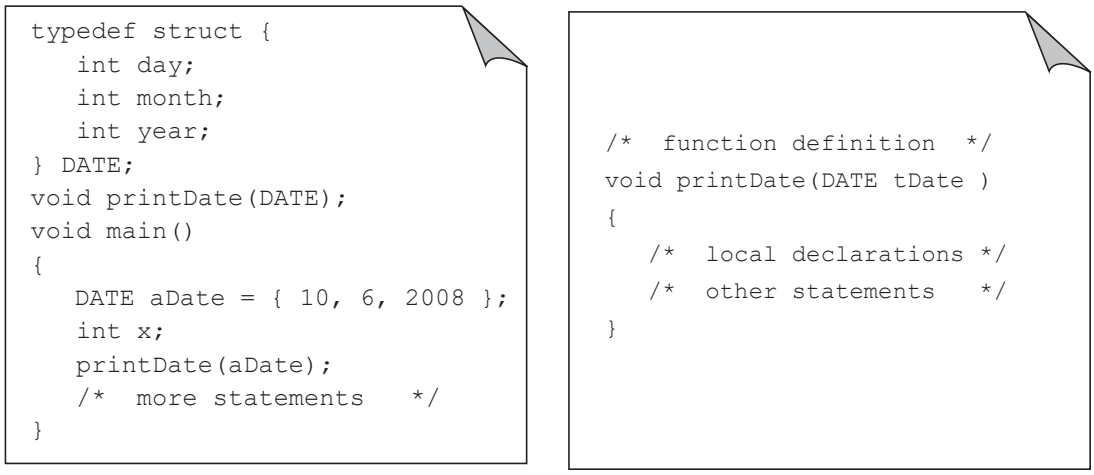


Fig. 8.3: Passing structure to a function

Listing 8.3

```
/*
    Program to illustrate the passing of a structure to a function
*/
#include <stdio.h>
typedef struct
{
   int day;
```

```
   int month;
   int year;
} DATE;
void printDate( DATE aDate );       /*  function declaration  */
void main()
{
   DATE tDate = { 10, 6, 2008 }; /* initialize a structure  */
   printDate( tDate );                  /*  function call       */
}
void printDate( DATE aDate )
{
   printf( "\nDate in format dd/mm/yyyy: %02d/%02d/%2d\n",
           aDate.day, aDate.month, aDate.year );
}
```

Test Run

```
Date in format dd/mm/yyyy: 10/06/2008
```

8.10 FUNCTION RETURNING A STRUCTURE

A function can also return a value of structure type as well via *return* statement as illustrated below.

```
typedef struct
{
   int day;
   int month;
   int year;
} DATE;
DATE getDate(void);
void main()
{
   DATE aDate;
   int x;
   aDate = getDate();
   /*  more statement   */
}
```

```
/*  function definition   */
DATE getDate(void)
{
   DATE tDate;
   /*  local declarations */
   /*  more statements    */
   return tDate;
}
```

Fig. 8.4: Function returning a structure

Listing 8.4

```
/*
    Program to illustrate function returning a structure
*/
```

```
#include <stdio.h>
typedef struct
{
   int day;
   int month;
   int year;
} DATE;
DATE getDate( void );       /*  function declaration   */
void main()
{
   DATE tDate;
   printf( "Enter date in format dd/mm/yyyy : " );
   tDate = getDate();
   printf( "\nDate entered by you: %02d/%02d/%2d\n",
          tDate.day, tDate.month, tDate.year );
}
DATE getDate( void )
{
   DATE xDate;
   scanf( "%d/%d/%d", &xDate.day, &xDate.month, &xDate.year );
   return xDate;
}
```

Test Run

```
Enter date in format dd/mm/yyyy: 10/6/2008
Date entered by you: 10/06/2008
```

ILLUSTRATIVE EXAMPLES

Example 8.1: *Given the following declaration*

```
struct STUDENT
{
  int  rollNo;
  char name[31];
  int   marks;
};
```

to represent the information of a student.

Using this structure, write a function that takes a list of students, in the form of an array, as its argument, and sorts them in the descending order of the marks secured by the students.

Using the above structure and function, write a program that reads the data for *n* students and displays them in descending order of marks secure by the students.

Listing 8.5

```
/*
   Program to the given information of students in
   descending order of the marks secured by them
*/

#include <stdio.h>
typedef struct
{
  int  rollno;
  char name[20];
  int  marks;
} STUDENT;

int main()
{
  STUDENT st[50], temp;
  int i, j, n;
  printf( "\nEnter number of students : " );
  scanf( "%d", &n );

  for ( i = 0; i < n; i++ )
  {
     printf("\nParticulars of student #%d\n\n", i+1);
     printf( "Enter students' roll number : " );
     scanf("%d", &st[i].rollno );
     fflush(stdin);
     printf( "Enter students' name : " );
     gets(st[i].name );
     printf( "Enter students' marks : " );
     scanf("%d", &st[i].marks );
  }
  for ( i = 1; i < n; i++ )
  {
      for ( j = 0; j < n-i; j++ )
      {
          if ( st[j].marks < st[j+1].marks )
          {
```

```
                temp = st[j];
                st[j] = st[j+1];
                st[j+1] = temp;
            }
        }
    }

    printf("\nStudent's Information after sorting\n\n");
    printf("%-10s", "Roll No.");
    printf("%-20s", "Name");
    printf("%6s\n\n", "Marks");

    for ( i = 0; i < n; i++ )
    {
       printf("%-10d", st[i].rollno);
       printf("%-20s", st[i].name);
       printf("%4d\n", st[i].marks);
    }
    return 0;
}
```

Test Run

```
Enter number of students : 5

Particulars of student #1

Enter students' roll number : 1000
Enter students' name : Ravi
Enter students' marks : 80

Particulars of student #2

Enter students' roll number : 1001
Enter students' name : Shankar
Enter students' marks : 79

Particulars of student #3

Enter students' roll number : 1002
Enter students' name : Ankit
Enter students' marks : 65
```

```
Particulars of student #4

Enter students' roll number : 1003
Enter students' name : Rupinder
Enter students' marks : 75

Particulars of student #5

Enter students' roll number : 1004
Enter students' name : Mehak
Enter students' marks : 88

Student's Information after sorting

Roll No.  Name               Marks

 1004     Mehak               88
 1000     Ravi                80
 1001     Shankar             79
 1003     Rupinder            75
 1002     Ankita              65
```

Example 8.2: *Write a program to add two distances given in inch-feet system using Structures.*

Listing 8.6

```
/*
   Program to add two distances given in inch-feet system
   using structures and functions
*/

#include <stdio.h>

typedef struct
{
   int feets;
   int inches;
} Distance;

/*  function prototypes  */
Distance getInput(void);
Distance addDistances(Distance d1, Distance d2);
void printDistance(Distance d);
int main()
{
   Distance d1, d2, d3;
```

```
    printf("\nEnter the first distance in feets and inches\n");
    d1 = getInput();
    printf("\nEnter the second distance in feets and inches\n");
    d2 = getInput();

    d3 = addDistances(d1, d2);

    printf("\nSum of given distances\n\n");
    printDistance(d3);

    return 0;
}
Distance getInput()
{
    Distance temp;
    printf("\n\tEnter feets : ");
    scanf("%d", &(temp.feets));
    printf("\n\tEnter inches : ");
    scanf("%d", &(temp.inches));
    return temp;
}
Distance addDistances(Distance d1, Distance d2)
{
    Distance temp;
    temp.feets = d1.feets + d2.feets;
    temp.inches = d1.inches + d2.inches;
    if ( temp.inches >= 12 )
    {
       temp.feets += 1;
       temp.inches -= 12;
    }
    return temp;
}
void printDistance(Distance d)
{
    printf("\n\tFeets = %d", d.feets);
    printf("\n\tInches = %d", d.inches);
}
```

Test Run

```
Enter the first distance in feets and inches
   Enter feets : 4
   Enter inches : 8
```

```
Enter the second distance in feets and inches
   Enter feets : 5
   Enter inches : 7

Sum of given distances
    Feets = 10
    Inches = 3
```

Example 8.3: *Use a structure to model a complex number. Using this structure write functions to accomplish the following tasks:*

- A function that returns the structure read from the keyboard.
- A function that takes two structures using call by value mechanism as its arguments and return their sum.
- A function that takes single structure using call by value mechanism as its arguments and prints it.

Use these functions in the program to demonstrate their work.

Listing 8.7

```
/*
   Program to demonstrate operations on complex number using
   structure & functions
*/

#include <stdio.h>

typedef struct
{
   float real;
   float imag;
} COMPLEX;

COMPLEX input(void);        /*  function declarations  */
void output(COMPLEX);
COMPLEX add(COMPLEX, COMPLEX);
COMPLEX multiply(COMPLEX, COMPLEX);
int main()
{
   COMPLEX n1, n2, n3, n4;
   printf("\nProgram to add two complex numbers\n\n");
   printf("\nEnter the first complex number\n");
   n1 = input();
   printf("\nEnter the second complex number\n");
   n2 = input();
   n3 = add(n1,n2);
```

```
    printf("\nSum of the given complex numbers\n\n");
    output(n3);
    n4 = multiply(n1,n2);
    printf("\nProduct of the given complex numbers\n\n");
    output(n4);
    return 0;
}
COMPLEX input(void)
{
    COMPLEX t;
    printf(«\nEnter real part : «);
    scanf(«%f», &t.real);
    printf(«Enter imaginary part : «);
    scanf(«%f», &t.imag);
    return t;
}
void output(COMPLEX t)
{
    printf(«Real part : %.2f», t.real);
    printf(«\nImaginary part : %.2f», t.imag);
}
COMPLEX add(COMPLEX a, COMPLEX b)
{
    COMPLEX t;
    t.real = a.real + b.real;
    t.imag = a.imag + b.imag;
    return t;
}
COMPLEX multiply(COMPLEX a, COMPLEX b)
{
    COMPLEX t;
    t.real = a.real * b.real – a.imag * b.imag;
    t.imag = a.real * b.imag + a.imag * b.real;
    return t;
}
```

Test Run

```
Program to add two complex numbers
Enter the first complex number
Enter real part : 1.0
Enter imaginary part : 2.5
Enter the second complex number
Enter real part : 2.5
Enter imaginary part : -1.5
```

```
Sum of the given complex numbers
Real part : 3.50
Imaginary part : 1.00
Product of given complex numbers
Real part : 6.25
Imaginary part : 4.75
```

UNIT SUMMARY

In this chapter, we have learned that

- A structure is a collection of related elements, which can be of different types, having a single name.
- Each element of the structure is called a field.
- There are two ways to declare a structure: *tagged structure* and *type-defined structure*.
- A structure can be initialized when it is declared and defined. The list of comma- separated values is enclosed in pair of braces.
- We can access the fields of the structure using direct selection, dot operator (.).
- Unlike arrays and string, a structure can be assigned to another structure using an assignment operator.
- A structure can be passed as an argument to a function.
- A function can return a structure.

EXERCISE

Subjective Questions

1. What is the main reason for using structures?
2. How is a structure different from an array?
3. What is the structure tag and what is its purpose?
4. What are different ways of defining a structure? Give example in each case.
5. Is there thing wrong with the following?

```
struct Time
{
   int hrs;
   int mts;
   int secs;
}
time t1;
```

6. What is wrong with the following, if any?

```
struct Time
{
   int hrs =10;
   int mts = 20;
   int secs = 35;
} t1, t2, t3;
```

7. Identify the errors, if any, in the following

```
struct
{
   int hrs;
   int mts;
   int secs;
} time;
time t1, t2, t3;
```

8. Give the output of the following program:

```
#include <stdio.h>
struct MyBox {
   int Length, Breadth, Height;
};
void Dimension( MyBox M )
{
   printf("\n%d x %d x %d\n",M.Length,M.Breadth,M.Height);
}
void main()
{
   MyBox B1 = { 12, 20, 8 }, B2, B3;
   ++B1.Height;
   Dimension(B1);
   B3 = B1;
   ++B3.Length;
   B3.Breadth++;
   Dimension(B3);
   B2 = B3;
   B2.Height += 5;
   B2.Length--;
   Dimension(B2);
}
```

Multiple Choice Questions

1. A structure is

(*a*) Scalar data type
(*b*) Derived data type
(*c*) Primitive data type
(*d*) Use-defined data type

2. Which of the following statement is true about structures?

(*a*) The elements of the structure must be of different type.

(*b*) Structure tag is mandatory in all cases.

(*c*) While declaring a variable of structure type, you need to prefix keyword *struct* before the tag name.

(*d*) Structure element are accessed using the star (*) operator.

3. A structure is a data type in which

(*a*) Each element must have the same type. (*b*) Elements can be of a different type.

(*c*) Each element must be of pointer type. (*d*) None of above

4. Consider the following declaration

```
struct ex {
   char cVvar;
   int iVar;
   long lVar;
};
```

What value will be returned by the *sizeof* operator called on *ex*?

(*a*) 4 (*b*) 7 (*c*) 1 (*d*) 6

5. Keyword used to create a structured data type is

(*a*) structure (*b*) structr (*c*) struct (*d*) struc

6. The operator used to access the structure member is

(*a*) * (*b*) . (*c*) & (*d*) []

7. Consider the statement: The size of a *struct* is always equal to the sum of its members.

(*a*) Valid (*b*) Can't say

(*c*) Invalid (*d*) Compiler dependent

8. Consider the statement: The size of a *struct* is always equal to the sum of its members.

```
#include<stdio.h>
struct st
{
     int x;
     static int y;
};

int main()
{
     printf("%d", sizeof(struct st));
     return 0;
}
```

(*a*) 4 (*b*) 8

(*c*) Runtime error (*d*) Compiler time error

9. Which of the following operator can be applied on structure variables?

(*a*) == (*b*) = (*c*) > (*d*) +

10. Presence of code like "s.b = 10" indicates a _____

(*a*) ordinary variable name (*b*) double data type

(*c*) structure (*d*) syntax error

11. What will be the output of the following program?

```
struct Test
{
   char str[20];
};
void main()
{
   struct Test st1, st2;
   strcpy(st1.str, "C Quiz");
   st2 = st1;
   st1.str[0] = 'S';
   cout << st2.str;
}
```

(*a*) C Quiz (*b*) S Quiz

(*c*) Runtime error (*d*) Compile time error

12. What is actually passed if you pass a structure variable to a function?

(*a*) Reference of structure variable (*b*) Copy of structure variable

(*c*) Starting address of structure variable (*d*) Ending address of structure variable

13. Comment on the output of the following C code.

```
struct temp {
  int a;
  int b;
  int c;
};
main()
{
  struct temp p[] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
}
```

(*a*) Creates an array of structure of size 3

(*b*) Creates an array of structure of size 9

(*c*) Illegal assignment to members of structure

(*d*) Illegal declaration of a multidimensional array

14. Which of the following cannot be a structure member?

(*a*) Array (*b*) Structure (*c*) String (*d*) Function

15. Which of the following is a properly defined struct?

(*a*) `struct {int a;}` (*b*) `struct a_struct {int a;};`

(*c*) `struct a_struct int a;` (*d*) `struct a_struct {int a;}`

ANSWERS									
1.	(*d*)	2.	(*c*)	3.	(*b*)	4.	(*b*)	5.	(*c*)
6.	(*b*)	7.	(*a*)	8.	(*d*)	9.	(*b*)	10.	(*c*)
11.	(*a*)	12.	(*b*)	13.	(*a*)	14.	(*d*)	15.	(*b*)

Programming Problems

1. Design a structure named *distance* to store a length in *yard*, *feet*, and *inches*. Using this structure, write a program that accepts two measurements from the user as represented by these structures and prints the absolute difference between these measurements.

2. Design a structure named *student* to store data about a student which contains the following elements — *rollno* of type *int*, *name* an array of type *char* of size 20, *college* an array of type *char* of size 40, and *score* of type *float*. Assume that there are not more than 100 students. Write a program to input the data about students, and output the stored data according to the merit of the students.

3. Consider the following declaration:

```
struct EMPLOYEE
{
   int code;
   char name[31];
   float salary;
};
```

Given that there are n ($\leq$50) employees in an organization. Write a program that output the details of the employee having the highest salary.

4. Create a structure named *Time* to model a time in 12 hour system having three fields as *hours*, *minutes* and *seconds*. Write a function called *elapsedTime* that takes two arguments, the *start time* and the *end time*. The function should return a *Time* structure containing the time elapsed between the two arguments. Write a program to test the above function.

5. A point in plane in Cartesian system can be represented by its coordinates x and y. We can create a structure to represent a point in plane as shown below:

```
struct POINT
{
    int x;
    int y;
};
```

Write a function that accepts the structure representing a point and returns an integer value 1, 2, 3 or 4 indicating the quadrant in which the point is located. Write a program to test the above function.

6. Define a structure that contains three members – *the name of country*, *name of the capital*, and *per capita income*. Using this structure, write a program to list the countries along with their capitals in the decreasing order of their per capita income.

PRACTICALS

1. Write a program to simulate a digital clock (in 24 hours format) using a structure to represent the time.

Listing 8.8

```
#include <stdio.h>
/* typedefined structure to represent clock time */
typedef struct {
    int hh;
    int mm;
    int ss;
} CLOCK;
int main()
{
    CLOCK myClock = { 12, 40, 55 };
    int i;
    printf("\nPress any key to stop the clock\n");
    while ( !kbhit() )
    {
       /*  increment clock  */
       (myClock.ss)++;
       if ( myClock.ss == 60 ) {
          myClock.ss = 0;
          (myClock.mm)++;
          if ( myClock.mm == 60 ) {
             myClock.mm = 0;
             (myClock.hh)++;
             if ( myClock.hh == 24 ) {
                 myClock.hh = 0;
             }
          }
        }
        /*  show clock  */
        printf("%02d:%02d:%02d\n",
               myClock.hh, myClock.mm, myClock.ss);
    }
    return 0;
}
```

Test Run

```
 Press any key to stop the clock
   12:40:56
   12:40:57
   12:40:58
   12:40:59
   12:41:00
   12:41:01
   12:41:02
```

2. Write a program to find the difference between start time and stop time (in 12 hours format) using a structure to represent the time.

Listing 8.9

```
#include <stdio.h>
typedef struct {
   int hh;
   int mm;
   int ss;
} TIME;
/*  function prototype  */
TIME differenceTime( TIME t1, TIME t2);

int main()
{
   TIME startTime, stopTime, diffTime;
   printf("Enter the start time in format hh mm ss : ");
   scanf("%d %d %d", &startTime.hh, &startTime.mm, &startTime.ss);
   printf("Enter the stop time in format hh:mm:ss : ");
   scanf("%d %d %d", &stopTime.hh, &stopTime.mm, &stopTime.ss);
   /*  functiona call  */
   diffTime = differenceTime(startTime, stopTime);
   printf("\nTime Difference: %d:%d:%d\n", diffTime.hh,
            diffTime.mm, diffTime.ss);
   return 0;
}
/* function that return difference between time periods */
TIME differenceTime( TIME t1, TIME t2)
{
```

```
    TIME temp;
    if ( t1.ss > t2.ss ) {
       --t2.mm;
       t2.ss += 60;
    }
    temp.ss = t2.ss - t1.ss;
    if ( t1.mm > t2.mm ) {
       --t2.hh;
       t2.mm += 60;
    }
    temp.mm = t2.mm - t1.mm;
    temp.hh = t2.hh - t1.hh;
    return temp;
}
```

Test Run

```
Enter the start time in format: hh mm ss : 5 20 30
Enter the stop time in format: hh mm ss : 7 15 10
Time Difference: 1:54:40
```

KNOW MORE

The structure is a user-defined type. There are many problems where the basic types and arrays derived from basic types are not able to model real-life problems. In that case we need to create a new data type, called user-defined type, which is a structure in C.

In most of the real-life applications that need to record information about entities in real-life, use of structure is the only way. That signifies the importance of structures in solving real-life problems.

The teacher is expected to develop an understanding of the concepts of structures and demonstrate the use of use of structures by taking suitable examples possibly from real life.

REFERENCES & SUGGESTED READINGS

1. R. S. Salaria, Problem Solving & Programming in C, Khanna Book Publishing Co(P) Ltd., New Delhi.
2. E. Balagurusamy, Programming in ANSI C, Tata McGraw Hill, New Delhi.
3. Yashavant Kanetkar, Let Us C, BPB Publications, New Delhi.
4. Byron Gottfried, Programming with C, Schaum's Outlines.
5. https://onlinecourses.nptel.ac.in/noc21_cs01/preview
6. https://ocw.mit.edu/courses/intro-programming/
7. https://www.programiz.com/c-programming
8. https://www.javatpoint.com/c-programming-language-tutorial

9 Pointers

UNIT SPECIFICS

This unit discusses the topics related to pointers. Pointers represent one of the important topics of C language. The real power of C language can be exploited by the judicious use of pointers. This unit explains various aspects of pointers and demonstrates their use with suitable examples.

RATIONALE

Under normal circumstances, every C program is allocated memory for storage of data and the program instructions. The amount of memory required for the storage of both is determined at the time of compilation, and it is fixed and cannot be changed during execution of the program.

Practically, the memory required for the instruction remains fixed every time you execute the program, however, the size of the data can vary from one problem instance to another.

This issue of varying size of data, dynamic nature of data, can only be resolved by making modification to declaration of data items of the program, and building the program again.

In case, you don't have access to source code, then this approach will not work.

The pointers provide a better alternate to handle this issue, as by using pointer, memory can be allocated for data items as per the actual requirement at execution time.

In addition, use of pointers offers benefits in following circumstances:

- Passing arguments to a function efficiently.
- Called function can return more than one value to the calling functions via arguments.
- Program can access any memory location of the installed memory as well any device connected which is addressable.
- Building complex data structures to model complex real-life problems.

In the nutshell, we can say pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language.

This unit will help the student to understand the various aspects related to pointers and develop programs using pointers to solve real-life problems efficiently.

PRE-REQUISITES

- Basic data types
- Operators
- Arrays and String
- Functions
- Structures

UNIT OUTCOMES

Upon completion of the unit, students will be able to

U9-O1: comprehend the world of pointers, one of the decadent feature of C language

U9-O2: declare and initialize pointers

U9-O3: access the data by dereferencing a pointer

U9-O4: explain the permissible operations on pointers

U9-O5: explain the usefulness of pointers in passing arguments to a function

U9-O6: explain the concept of dynamic memory allocation

U9-O7: write efficient programs using pointers to solve variety of problems

Unit 9 Outcomes	EXPECTED MAPPING WITH COURSE OUTCOMES (1- Weak Correlation; 2- Medium correlation; 3- Strong Correlation)							
	CO-1	CO-2	CO-3	CO-4	CO-5	CO-6	CO-7	CO-8
U9-O1	-	-	1	-	-	-	-	-
U9-O2	-	-	1	-	-	-	-	-
U9-O3	-	-	1	-	-	-	-	-
U9-O4	-	-	-	-	-	2	-	-
U9-O5	-	-	-	-	2	-	-	-
U9-O6	-	-	-	-	-	2	-	-
U9-O7	-	-	-	-	-	3	-	-

9.1 INTRODUCTION

Pointers are one of the most important features of C language. Pointers, by most people, are regarded as one of the most difficult topics in C. Let me tell you frankly, it is more a myth than a fact. I would prefer to say that pointers are one of the most delicate aspects of C language, and you know; delicate things need cautious handling. Therefore, pointers need careful handling.

There are many reasons for using pointers; some of them are enumerated below:

- A pointer allows a function or a program to access a variable (better we say memory location) outside the preview of function or a program. Using a pointer, your program can access any memory location in the computer's memory.
- Since using *return* statement, a function can only pass back a single value to the calling function, pointers allows a function to pass back more than one value by writing them into memory locations that are accessible to the calling function.
- Using pointers, arrays and structures can be handled in more efficient way.
- Without pointers, it will be impossible to create complex data structures, such as *linked lists*, *trees* and *graphs*.

- To communicate with operating system about memory. For example, the operator *new* returns the location of free memory block by using a pointer and the operator *delete* returns the memory block pointed to by a pointer to the operating system.

In totality, we can say that the real power of C language lies in the judicious use of pointers. Therefore, every reader of C must learn and master the art of using pointers.

In this unit, we will learn about various aspects related to pointers in C language.

9.2 IDEA OF POINTERS

Computer's memory is organized as a linear collection of bytes or words. For Intel processors, memory is byte-organized whereas for most of the non-Intel processors, memory is word-organized. In byte-organized memory, each memory location is one byte, whereas in word-organized, each memory location is of one word. Each memory location is assigned a unique number called *location number* or *address*. Fig. 9.1 shows the schematic of memory organization.

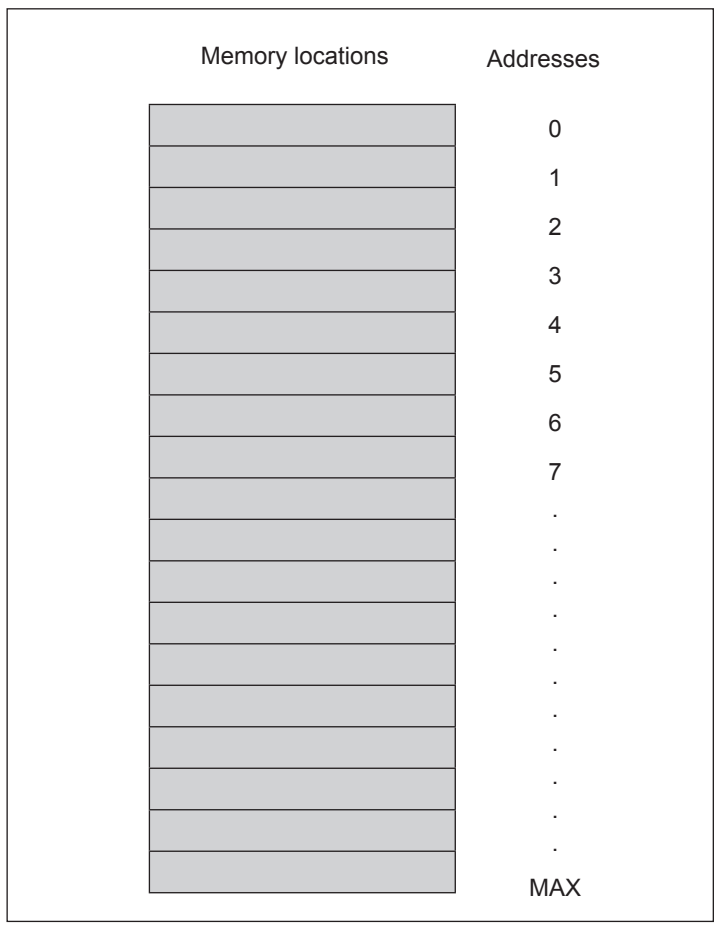


Fig. 9.1: Memory Organization

We know that one byte is equal to 8 bits, whereas one word may be made up of two or more bytes. A word measures the capacity of processor in terms of the size of operands that it can process (*i.e.* add, subtract, multiply, divide, compare, etc.) in single operation. When we say that a processor is a 16-bit processor, its word size is 16-bits (2 bytes) and it can process 16-bit operands in single operations. For large size operands, it needs more operations.

A *memory cell* is either of one byte/word or more contiguous bytes/words. Each cell can store one value at a given time. The address of a cell is always the address of the first byte/word.

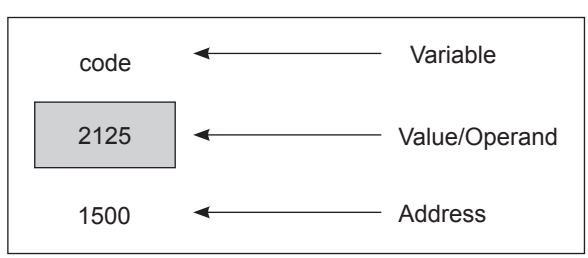


Fig. 9.2: Representation of a variable in memory

When you give a command to run a program, the operating system first finds a free block of requisite size from computer (main) memory, and then loads the program into that

block. Even in that block, usually, the program's code is loaded in one part and the program's data in another part of the block. Each of the data operand is stored in a cell and the system associates each of the variables with these addresses. This is illustrated in Fig. 9.2.

To access the data operand either we use either variable name or use the memory cell's address. Since these addresses are simply positive integer numbers, they can also be assigned to some variables stored in memory, like any other variable. Such variables that hold addresses are known as a *pointer*. A pointer is, therefore, nothing but a variable that contains address of another variable in memory.

Since pointer is a variable, its value is also stored in memory in another address. Suppose we assign the address of variable *code* to a variable *ptr*. The link between the variable *ptr* and *code* can be visualized as shown in Fig. 9.3.

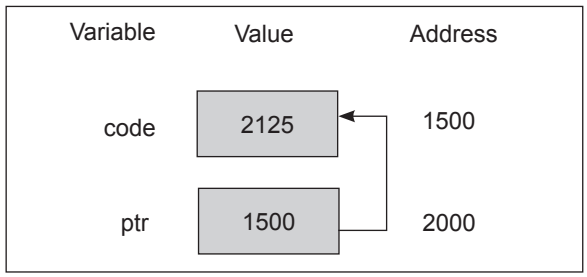


Fig. 9.3: Pointer as a Variable

Since the value of the variable *ptr* is the address of the variable *code*, we can access the value of variable *code* by using the value of variable *ptr*. Therefore, we say that the variable *ptr* points to the variable *code*, and hence *ptr* gets the name pointer.

9.3 ACCESSING ADDRESS OF A VARIABLE

The actual address of a variable is a system dependent. During compilation and linking, addresses are assumed to be relative to some base address, usually 0. When the operating system loads the program in a free block, all the address are transformed with relative to address of the first memory location of the free block. Hence, we do not know the address of a variable. Therefore, a question arises — *how can we determine the address of a variable*? This is done with the help of *address of* operator (&).

Consider the following statement

```
ptr = &code;
```

It will assign the number 1500 (address of variable *code*) to the variable *ptr*.

The following restrictions must be observed:

1. The address of operator can only be used with variables or an array element. The following are illegal use of *address of* operator:
   (*a*) `&1200` (pointing at constant expression)
   (*b*) `&(a/b)` (pointing at expression)
2. Pointer variable can be initialized only with the use of *address of* operator or by assigning the value of another compatible pointer variable. The following way of initializing a pointer variable *ptr* is illegal:

   ```
   ptr  = 1500;
   ```

   Though, the number 1500 happen to be the address of variable code.

9.4 DECLARING A POINTER

Declaring a pointer variable is similar to the declaration of a standard variable but the name of pointer variable is prefixed with * symbol.

The syntax for declaring a pointer variable

```
datatype *pointerVariableName
```

For example, the statement

```
int *ptr;
```

declare a pointer variable *ptr* that that can be used to store address of any integer variable.

9.5 ASSIGNING ADDRESS TO A POINTER

Once a pointer has been declared, it must be assigned address of a variable. Like standard variables, a pointer variable will take a garbage value once declared. Therefore, before use, a pointer variable must be assigned the address of a variable.

The syntax for assigning address to a pointer variable

```
pointerVariableName = &variableName;
```

For example, the statement

```
int a, *ptr;
```

declares *a* as a normal integer variable and *ptr* is an integer pointer variable.

To assign the address of variable *a* to pointer variable *ptr*, we use the following statement

```
ptr = &a;
```

Here we say that pointer variable *ptr* is pointing to variable *a*.

9.6 ACCESSING VARIABLE USING A POINTER

Once a pointer variable has been assigned the address of a variable, the question remains - how to access the value of a variable using the pointer variable. This is done by prefixing * with the pointer variable.

The syntax for accessing variable using a pointer variable

```
*pointerVariableName
```

Listing 9.1

```
/*
   Program to illustrate the use of pointer variable
*/
#include<stdio.h>
int main()
{
   int *ptr, x = 10;
   ptr = &x;
   printf(«Value of variable x = %d\n», x );
   printf(«Value of variable pointed to by ptr = %d\n»,*ptr );
```

```
    printf(«Address of variable x = %u\n», ptr );
    return 0;
}
```

Test Run on Turbo C++ Complier

```
Value of variable x = 10
Value of variable pointed to by ptr = 10
Address of variable x = 65524
```

9.7 POINTER ARITHMETIC

The C language is very consistent in its approach to pointer arithmetic. The integration of pointers, arrays, and pointer arithmetic is one of the strengths of the C language. In this section, we will briefly describe the various operations permissible on pointers.

The following operations are permitted on pointers.

1. **Addition of a number to a pointer variable.**

   Suppose *p* is a pointer variable pointing to an element of integer type, then the statement

   p++; or ++p;

   increments the value of *p* by a factor of 2, so that it points to the following location that holds another value of integer type. This increment factor will be 1 for *character*, 4 for *long integer* and *float*, and 8 for *long float*. The statement

   ```
   p += i;
   ```

   where *i* is either a positive integer constant or an integer variable having positive value, increments *p* such that it points to the *i*th location beyond the location to which it is currently pointing.

2. **Subtraction of a number from a pointer variable.**

   Suppose *p* is a pointer variable pointing to an element of integer type, then the statement

   ```
   p--;  or --p;
   ```

   decrements the value of *p* by a factor of 2, so that it now points to the location preceding the current location. The statement

   ```
   p -= i;
   ```

   where *i* is either a positive integer constant or an integer variable having positive value, decrements *p* such that it points to the *i*th location before the location to which it is currently pointing.

3. **Subtraction of one pointer variable from another.**

   One pointer variable can be subtracted from another provided both point to the same data type. The difference of the two indicates the number of bytes separating the corresponding elements.

In addition to these arithmetic operations, pointer variables can be compared with one another, provided they are compatible, *i.e.*, point to variables of same data type.

The following arithmetic operations on pointers are not permitted.

- Addition of two pointer variables.
- Multiplication of a pointer variable by a number.
- Division of a pointer variable by a number.

9.8 POINTERS AS FUNCTION ARGUMENTS

In *Chapter 6*, we had mentioned that the arguments to a function could be passed using either *call by value* or *call by address* mechanism. When we use call by value mechanism, the formal arguments of the function are declared as simple variable. At execution time the values of the actual arguments are copied into formal arguments (formal arguments are local to the function), and the execution of the function begins.

However, when we use call by reference mechanism, the formal arguments are declared as pointers. At execution time, the addresses of the actual arguments are copied into formal arguments and the function's execution begins.

Listing 9.2

```
/*
   Program to illustrate pointers as function arguments
*/
#include<stdio.h>
/* function prototype */
void swap( int *x, int *y )
int main()
{
   int a = 10, b = 20;
   /* function prototype */
   void interchange( int *, int *);
   printf( "\nValue of a = %d and b = %d", a, b );
   printf( " before function call\n"  );
   /* function call  */
   swap ( &a, &b);
   printf( "\nValue of a = %d and b = %d", a, b );
   printf( " after function call\n"  );
   return 0;
}
/* function definition */
void swap( int *x, int *y )
{
  int temp;
  temp = *x;
  *x = *y;
  *y = temp;
}
```

Test Run on Turbo C++ Complier

```
Value of a = 10 and b = 20 before function call
Value of a = 20 and b = 10 after function call
```

9.9 POINTERS AND STRUCTURES

As we can use pointers with other data types, we can also use pointers for structure variables. Consider the following declarations

```
typedef struct
{
   int code;
   char name[20];
   int  dept_code;
   float  salary;
} EMPLOYEE;
EMPLOYEE emp, *ptr;
```

These declarations create and user-defined data type and given it name *EMPLOYEE* and declares *emp* as variable of type employee and *ptr* as pointer variable to type *EMPLOYEE*.

The following statement

```
ptr = &emp;
```

can be used to assign the address of variable *emp* to pointer variable *ptr*. Now we can access the members of the structure variable *emp* using pointer variable *ptr*.

The elements of structure are accessed using *arrow operator* '–>' which combines two characters, hyphen '–' and greater-than '>'.

The syntax for accessing members of structure using arrow operator is:

```
ptr–>vname
```

where *ptr is* pointer variable pointing to a structure, and *vname* is an element of the structure.

Using arrow operator, the elements of structure of type *employee* pointed to by pointer variable *ptr* can be accessed as

```
ptr->code      ptr->name      ptr->dept_code      ptr->salary
```

9.9.1 Self-referential Structures

A self-referential structure is a structure that includes at least one element that is a pointer to its own type. These structures find their application in building complex data structure such as *linked lists*, *trees* and *graphs*. Following is one such application of self-referential structure.

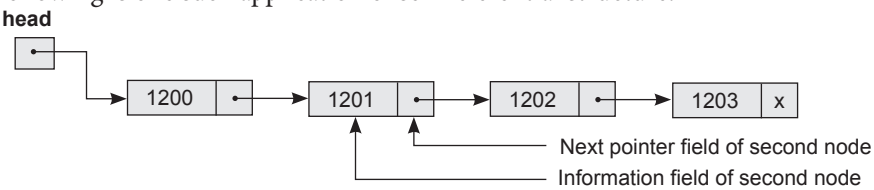


Fig. 9.4: Linear linked list of integer values with nodes 4

To represent the above linear linked list in memory, we need following declarations

```
struct NODE
{
   int info;
   struct NODE *next;
};
struct NODE *head;
```

9.10 DYNAMIC MEMORY ALLOCATION

The memory requirements for the instructions are always fixed. However, there may be situation when the exact memory requirements for the data may not be known in advance. The data requirements may vary from one program execution to another program execution, *i.e.*, memory requirements for the data are dynamic.

In such cases when the amount of memory is not known before hand for a particular data item(s), then it is always better to allocate memory at the execution time, *i.e.*, when the program is running. Thus, the memory allocation at execution time is known as *dynamic memory allocation*.

Every program is provided with a pool of unallocated memory that it can utilize during execution. This pool of unallocated memory is known as *free store*. Therefore, whenever the memory of the requisite size (amount) is required by the program, it is taken from the free store. When the previously allocated memory is not required further, it is returned to the free store.

The C language provides library functions, called *dynamic memory management functions*, to allocate and de-allocate memory at execution time, *i.e.*, dynamically.

Table 9.1: Memory Management Functions

Function Name	Description
`malloc`	Allocates memory from free store.
`free`	Deallocates a previously allocated.

By allocation, we mean that your program can obtain as much memory as required by your program even during execution of your program.

By de-allocation, we mean that the memory acquired dynamically can be released at any time during your program execution.

It is important to note that memory allocated dynamically, must be de-allocated before your program finishes its execution. Otherwise, even if your program terminates, memory allocated dynamically is never released automatically, and from operating system point of view that memory is still in use.

Therefore, if you run the same program many times, many users are using your program concurrently, the operating system may run out of memory.

9.10.1 Allocating Memory

The *malloc()* function takes one argument that specifies the size of the block in bytes. The function allocates the memory of requisite size from heap and returns a pointer to the allocated memory on success or a NULL pointer (0) in case of failure. In case of failure, the program should make a safe exit.

The pointer returned is of type *void* which need to be casted into a pointer of desired type. The memory allocated is left un-initialized, *i.e.*, all locations get the garbage value.

The syntax for use of *malloc()* function is

```
datatype *ptr
ptr = (datatype *) malloc( n * sizeof (datatype) );
```

where *n* is the number of elements.

For example,

```
int *ptr
ptr = (int *) malloc( 10 * sizeof(int) );
if ( ptr == NULL )
{
   printf("\nMemory allocation failed\n");
   return 1;
}
```

Since the size of *int* is 2-bytes for 16-bit C Compiler and 4-bytes for 32-bit C Compiler, therefore, this statement will allocate block of 20/40 bytes of memory. And, the pointer *ptr* holds the address of the first byte in the allocated memory.

9.10.2 De-allocating Memory

The *free()* function de-allocates a dynamically allocated block of memory. The syntax for use of *free()* function is

```
free(ptr);
```

It takes one argument that specifies the pointer to the allocated block. It is important to note that only the allocated block is de-allocated, the pointer variable is not deleted.

Listing 9.3

```
/*
   Program to illustrate the allocation of memory dynamically
   for one-dimensional array at execution time.
   Program takes an array of arbitrary size as input and finds the
   Largest element of the array
*/
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int  *a, n, i, max;
   printf( "\nEnter size of 1D array : "  );
   scanf( "%d", &n );
   /* dynamically allocate memory for array */
   a = (int *) malloc(n*sizeof(int));
```

```
    if ( a == NULL )
    {
        printf("\nMemory allocation failed.\n");
        return 1;
    }

    printf( "\nEnter %d elements of array\n", n );
    for ( i = 0 ; i < n ; i++ )
       scanf( "%d", &a[i] );

    max = a[0];
    for ( i = 1; i < n; i++ )
    {
       if ( a[i] > max )
           max = a[i];
    }
    printf( "\nLargest element of array = %d\n", max );
    /* de-allocate memory */
    free(a);
    return 0;
}
```

Test Run

```
Enter size of 1D array : 10
Enter 10 elements of array
10 35 12 9 45 11 50 20 44 32
Largest element of array = 50
```

ILLUSTRATIVE EXAMPLES

Example 9.1: *Write a program to print the elements of array using pointers.*

Listing 9.4

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int  a[5] = { 1, 2, 3, 4, 5 };
   int i, *ptr;
   ptr = a;
   printf("\nElements of array\n\n");
```

```
    for ( i = 0; i < 5; i++ )
        printf("%d  ", *(ptr+i));
    printf("\n\n");
    return 0;
}
```

Test Run

```
Elements of array
1  2  3  4  5
```

Example 9.2: *Write a program to demonstrate passing of 1D array to a function using pointer.*

Listing 9.5

```
#include<stdio.h>
#include<stdlib.h>
void fun(int *, int);  /* function prototype  */
int main()
{
   int  a[10] = { 5, 2, 1, 7, 9, 10, 4, 6, 8, 3 };
   /*  call to function  */
   fun(a,10);
   return 0;
}
/* function definition  */
void fun(int *p, int n)
{
   int i;
   printf("\nElements of array\n\n");
   for ( i = 0; i < n; i++ )
       printf("%d  ", p[i]);
   printf("\n\n");
}
```

Test Run

```
Elements of array
5 2 1 7 9 10 4 6 8 3
```

UNIT SUMMARY

In this chapter, we have learned that

- Pointer is a variable that holds an address of an operand.
- Pointers offers many advantages such as returning more than one value from a function, building complex data structures, communicating with the H/W, etc.

- An addition operation on pointers is permitted. A subtraction operation is permitted in the restricted sense that it must give positive difference. Multiplication and division operation is not permitted.
- Pointers are used to pass arguments by address, mechanism better known as *call by reference*.
- A self-referential structure is a structure that includes at least one element that is a pointer to itself.
- Memory allocation can be done in two ways – *statically* and *dynamically*.
- Static memory allocation is that which is done at compilation time.
- Dynamic memory allocation is that which is done at execution time.
- Every program is provided with a pool of unallocated memory that it can utilize during execution. This pool of unallocated memory is known as *heap* or *free store*.
- The process of managing the memory at execution time is known as *dynamic memory management*.
- The *malloc()* is used to allocate memory dynamically.
- The *free()* is used to de-allocate previously allocated memory by *malloc()* function.
- An initialized pointer is referred to as *dangling/wild pointer*, as it can end up pointing anywhere in memory.
- Whenever there is attempt to write on zero address (also known as *Null pointer*) or the system will flag and a message "*Null pointer assignment*" on termination of the program.
- Memory leak is a kind of situation when memory is allocated in a called function but it is not deallocated in that function after it use. As a result, when the called function completes its execution, the pointer variable is destroyed and there is no means to reach that allocated memory block.
- Allocation failure is a situation when the system is not able to find a requisite block of free memory. The system indicates this situation by returning a *NULL* pointer.

EXERCISE

Subjective Questions

1. How can we get the address of a variable?
2. What is a pointer variable?
3. How is a pointer variable declared?
4. How can the values pointed to by a pointer variable be accessed?
5. What are the various types of pointers?
6. What are the operations permissible on pointers?
7. What are the operations, which are not permissible on pointers?

Multiple Choice Questions

1. The address of a variable can be obtained using ___ operator.
(*a*) * (*b*) & (*c*) ? (*d*) ->

2. Which of the following operator is known as *indirection* (*dereference*) operator?
(*a*) & (*b*) << (*c*) ^ (*d*) *

3. Which of the following operator is known as *address of* operator?
(*a*) & (*b*) * (*c*) -> (*d*) **

4. Identify the correct declarations of pointer variables *ptr1*, *ptr2*.
(*a*) `int ptr1, *ptr2;` (*b*) `int *ptr1, ptr2;`
(*c*) `int ptr1, ptr2;` (*d*) `int *ptr1, *ptr2;`

5. The operators exclusively used with pointers are
(*a*) * and / (*b*) & and * (*c*) & and ^ (*d*) * and +

6. The operand of *addressof* operator can be a/an
(*a*) constant (*b*) array element (*c*) expression (*d*) None

7. Pointer variables may be assigned
(*a*) address value represented in hexadecimal notation
(*b*) address value represented in octal notation
(*c*) address of another variable
(*d*) address value represented in binary notation

8. Operand of *indirection* (*dereference*) operator is
(*a*) pointer variable (*b*) ordinary variable (*c*) integer constant (*d*) none of above

9. Identify the operator that is not used with pointers
(*a*) -> (*b*) & (*c*) * (*d*) >>

10. When *addressof* operator (&) is prefixed to a variable, it yields
(*a*) variable's value (*b*) variables data type
(*c*) variable's address (*d*) none of above

11. Which of the following operation is not permitted on pointers?
(*a*) incrementing a pointer variable
(*b*) adding a number to a pointer variable
(*c*) division of a pointer variable by a number
(*d*) difference of two pointer variables

12. What is the value of *x* after executing the following segment?

```
int x = 5, *p;
p = &x;
*p = 7;
```

(*a*) 5 (*b*) 7

(*c*) Undefined (*d*) None

13. What will be the output of the following segment?

```
int a[5] = {1,2,3,4};
 int *p = a;
 printf("%d",*(p+2));
```

(*a*) 4 (*b*) 0 (*c*) 3 (*d*) 2

14. Consider the following program segment:

```
int *p, a[] = { 1, 2, 3, 4, 5 };
p = &a[2];
printf("%d", p[-1]);
```

Which of the following is correct about above program segment?

(*a*) Compiler time error because *p* is not an array

(*b*) Output will be 2

(*c*) Compiler time error because subscript cannot be negative

(*d*) None of above

15. Consider the following program segment:

```
const int i = 5;
int *p;
i = 20;
p = &i;
printf("\n%d, %d\n", *p, i);
```

Which of the following is correct about above program segment?

(*a*) Output will be 20, 20

(*b*) Output will be 5, 20

(*c*) Output will be 20, 5

(*d*) Compile time error because the value of variable *i* cannot be changed.

ANSWERS															
1.	(*b*)	2.	(*d*)	3.	(*a*)	4.	(*d*)	5.	(*b*)	6.	(*b*)	7.	(*c*)	8.	(*a*)
9.	(*d*)	10.	(*c*)	11.	(*c*)	12.	(*b*)	13.	(*c*)	14.	(*b*)	15.	(*d*)		

Programming Problems

1. Write a program to copy one array to other using pointers.

2. Write a program to reverse an array using pointers.

3. Write a program to find length of string using pointer.

4. Write a program to find reverse of a string using pointers.

5. Write a program to compare two strings using pointers.
6. Write a function that return smallest element, largest element, and average of elements of a given array using pointers. Demonstrate the use of this function.
7. Write a function that receives a floating point number and return its integral part and fractional part using pointers. Demonstrate the use of this function.
8. Write a program to find whether the array is having duplicate elements or not using pointers.
9. Write a program to remove duplicate elements from an array using pointers.
10. Write a program to demonstrate the handling of a structure using a pointer.
11. Write a function *replace()* whose prototype is given as

```
int replace(char *str, char c1, char c2);
```

that replace every occurrence of character c1 with character c2, and returns the total replacements made. Write a program to test the above function.

12. Consider the following structure definition:

```
struct BOX
{
   char make[25];
   float length, breadth, height;
};
```

Write a function that takes address of the *BOX* structure and returns its volume.

13. Consider the following declaration:

```
struct EMPLOYEE
{
   int code;
   char name[31];
   float salary;
};
```

Given that there are n ($\leq$50) employees in an organization. Write a program, using pointer notation to access the data of an employee that output the details of the employee having highest salary.

14. Consider the following declaration:

```
char *names[]={ "Vimal","Amit","Anuj","Rohit","Abhijit" };
```

Write a program to produce the following output using pointers.

```
char *names[]={ "tijihbA","tihoR","junA","timA","lamiV" };
```

15. Write a function *substr()* whose prototype is given as

```
char *substr(char *str1, char *str2);
```

that scan the first string for the occurrence of a second string, and returns a pointer to the element in the first string where the second string begins. However, if the second string does not occur in the first string, the function should return NULL. Write a program to test the above function.

PRACTICALS

1. Write a program to implement linear linked lists with limited number of operations.

Listing 9.6

```
/*
   Program to demonstrate the use of pointers and structures
   by implementing a linear linked list, where the insertion
   and deletions operations are limited to beginning of the l
   inked list,for simplicity.
   Program used separate function to perform each operation.
*/
#include <stdio.h>
#include <stdlib.h>
typedef struct nodeType
{
    int info;
    struct nodeType *next;
} NODE;
/*-------   function prototypes   -------*/
void traverse(NODE *);
void search(NODE *, int);
NODE *insert(NODE * int);
NODE *delete(NODE *);
int main()
{
    NODE *head = NULL;
    int choice, element, after;
    while ( 1 )
    {
        printf( "\n\n         Options available \n" );
        printf( "++++++++++++++++++++++++++++++ \n\n" );
        printf( "  1.  Insert\n" );
        printf( "  2.  Delete\n" );
        printf( "  3.  Search\n" );
        printf( "  4.  Traverse\n" );
        printf( "  5.  Exit\n\n" );
        printf( "Enter your choice ( 1-5 ) : " );
        scanf( "%d", &choice );
        switch ( choice )
        {
```

```
                case 1 : printf( "\nEnter element : " );
                         scanf( "%d", &element );
                         head = insert( head, element );
                         break;
                case 2 : head = delete(head);
                         break;
                case 3 : printf( "\nEnter element to search : " );
                         scanf( "%d", &element );
                         search( element );
                         break;
                case 4 : if ( head == NULL )
                             printf( "\nList is empty ... " );
                         else
                             traverse(head);
                         printf("\nPress any key to continue...");
                         getch();
                         break;
                case 5 :
                        printf("\nProgram terminated on success\n");
                        return 0;
            }
        }
}  /****** end of main function ******/

/*
   function to traverse and print elements of the linked list
*/
void traverse(NODE *head)
{
    NODE *ptr = head;
    printf("\n\nLinked list\n\n");
    printf("\thead" );
    while ( ptr != NULL )
    {
        printf( " -> %d", ptr->info );
        ptr = ptr->next;
    }
}
```

```
/*
   function to search a given element in the linked list
*/
void search(NODE *head, int item)
{
    NODE *ptr = head;
    while ( ptr != NULL )
    {
       if ( item == ptr->info )
       {
           printf("\nElement %d found in linked list\n", item);
           return;
       }
       ptr = ptr->next;
    }
    printf("\nElement %d not found in linked list\n", item);
}
/*
   function to insert node in the beginning of the linked list
*/
NODE *insert(NODE *head, int item )
{
   NODE *ptr;
   ptr = ( NODE * ) malloc( sizeof( NODE ) );
   ptr->info = item;
   ptr->next =  head;
   head = ptr;
   return ptr;
}
/*
   function to delete node from the beginning of the linked list
*/
NODE *delete(NODE *head)
{
    NODE *ptr;
    if ( head == NULL ) {
        printf("\nList is empty ... ");
        return;
    }
```

```
    printf("\nElement %d deleted from list\n", head->info);
    ptr = head;
    head = head->next;
    free(ptr);
    return head;
}
```

Test Run

```
          Options available
++++++++++++++++++++++++++++++++
  1.  Insert
  2.  Delete
  3.  Search
  4.  Traverse
  5.  Exit
Enter your choice ( 1-5 ) : 1
Enter element : 1
          Options available
++++++++++++++++++++++++++++++++
  1.  Insert
  2.  Delete
  3.  Search
  4.  Traverse
  5.  Exit
Enter your choice ( 1-5 ) : 1
Enter element : 15
        Options available
++++++++++++++++++++++++++++++++
  1.  Insert
  2.  Delete
  3.  Search
  4.  Traverse
  5.  Exit
Enter your choice ( 1-5 ) : 1
Enter element : 10
         Options available
++++++++++++++++++++++++++++++++
  1.  Insert
  2.  Delete
  3.  Search
```

```
    4.  Traverse
    5.  Exit
  Enter your choice ( 1-5 ) : 1
  Enter element : 25
          Options available
  +++++++++++++++++++++++++++++++++
    1.  Insert
    2.  Delete
    3.  Search
    4.  Traverse
    5.  Exit
  Enter your choice ( 1-5 ) : 4
  Linked list
    head -> 25 -> 10 -> 15 -> 1
  Press any key to continue ...
          Options available
  +++++++++++++++++++++++++++++++++
    1.  Insert
    2.  Delete
    3.  Search
    4.  Traverse
    5.  Exit
  Enter your choice ( 1-5 ) : 3
  Enter element to search : 15
  Element 15 found in linked list
          Options available
  +++++++++++++++++++++++++++++++++
    1.  Insert
    2.  Delete
    3.  Search
    4.  Traverse
    5.  Exit
 Enter your choice ( 1-5 ) : 2
  Element 25 deleted from list
          Options available
  +++++++++++++++++++++++++++++++++
    1.  Insert
    2.  Delete
    3.  Search
```

```
  4.  Traverse
  5.  Exit
Enter your choice ( 1-5 ) : 4
Linked list
  head -> 10 -> 15 -> 1
Press any key to continue ...
        Options available
++++++++++++++++++++++++++++++++
  1.  Insert
  2.  Delete
  3.  Search
  4.  Traverse
  5.  Exit
Enter your choice ( 1-5 ) : 5
Program terminated on success\n");
```

KNOW MORE

Pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language. In the beginning, use of pointers may appear a little confusing and complicated, but once you understand the concept, you will be able to do a lot with C language.

The teacher is expected to understand the concepts of pointers, and the various aspects related to handling of pointers in C language.

The teacher should also demonstrate the use of pointer by taking appropriate examples and create C programs to solve them with the participation of students.

REFERENCES & SUGGESTED READINGS

1. R. S. Salaria, Problem Solving & Programming in C, Khanna Book Publishing Co(P) Ltd., New Delhi.
2. E. Balagurusamy, Programming in ANSI C, Tata McGraw Hill, New Delhi..
3. Yashavant Kanetkar, Let Us C, BPB Publications, New Delhi.
4. Byron Gottfried, Programming with C, Schaum's Outlines.
5. https://onlinecourses.nptel.ac.in/noc21_cs01/preview
6. https://ocw.mit.edu/courses/intro-programming/
7. https://www.programiz.com/c-programming
8. https://www.javatpoint.com/c-programming-language-tutorial

10 File Handling

UNIT SPECIFICS

This unit discusses the topics related to files. Files provide a means for long term storage of data. Data can be read from file or written to a file using few commands. This unit explains various aspects of files and demonstrates their use with suitable examples.

RATIONALE

In real-life problems, we have to deal with any of the following situations:

- The volume of input data is enormous.
- Same data may need to be processed multiple times
- The output of one program may have to use as input for another program
- Data may be machine-generated, which is not in human-readable

In all these situations, the traditional way of entering data from the keyboard and producing the results on the computer screen during the program execution is almost impractical.

The better solution is that input data can be stored in a file, called a data file, and then the program can be directed to read data from that file. Likewise, the program can also be directed to write the output of a program to a data file.

These all operations dealing with the files are referred to as file handling.

This unit will help the student understand the various aspects of files and develop programs using files to solve real-life problems.

PRE-REQUISITES

- Standard Input/Output
- Unformatted and formatted input/output
- Predefined files/steams

UNIT OUTCOMES

Upon completion of the unit, students will be able to

U10-O1: explain the concept of the data file and its usefulness

U10-O2: demonstrate the opening and closing of files

U10-O3: perform read and write operations on files

U10-O4: write efficient programs for file handling

Unit 10 Outcomes	EXPECTED MAPPING WITH COURSE OUTCOMES (1- Weak Correlation; 2- Medium correlation; 3- Strong Correlation)							
	CO-1	CO-2	CO-3	CO-4	CO-5	CO-6	CO-7	CO-8
U10-O1	1	-	-	-	-	-	-	-
U10-O2	-	-	1	-	-	-	-	-
U10-O3	-	-	1	-	-	-	-	-
U10-O4	-	-	-	-	-	2	-	-

10.1 INTRODUCTION

A file is a collection of related data. The primary purpose of the file is to keep a record of data. Since computer memory is volatile (*i.e.*, memory contents are lost when the computer is shut down), we need to store the data for later use. In addition, the volume of data can be significant to reside entirely in memory at a given time. Therefore, our programs must have the ability to read and write a portion of the data while the rest of the data remain in the file.

Files to be used by the programs are generally stored on the hard disk. When the program reads, the data is moved from file to memory; it moves from memory to file when it writes. This data moved data uses a particular work area known as a buffer, i.e., a buffer is a temporary storage area that holds data while moving to or from memory.

A program in the C language can read and write files in a variety of ways. Each of these ways is straightforwardly described in this unit and illustrated with well-designed programming examples.

10.2 TYPES OF FILES

There are two types of files in C – *text file* and *binary file*.

- **Text File** - A **text file** stores data in alphabets, digits, and other special symbols by storing their ASCII values and are in a human-readable format. When you open these files, you can see all the contents within the file as plain text. You can easily edit or delete the contents. These files take minimum effort to maintain, are easily readable, provide the least security, and take bigger storage space.
- **Binary File** - A **binary file** store data as a sequence of bytes which are not in a human-readable format. They can only be created by using programs. They can hold a large amount of data, are not readable easily, and provides better security than text files.

Table 10.1: Text File VS Binary File

Text File	Binary File
Data is stored using human-readable characters.	Data is stored in the same format as it is stored in memory.
Each line of data ends with a newline character.	There is no newline character.
There is a unique character called end-of-file (EOF) at the end of the file.	There is an end-of-file marker at the end of the file.

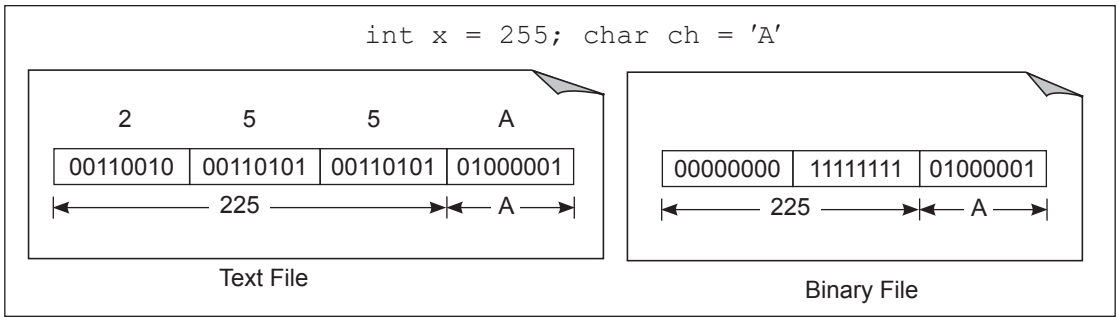


Fig. 10.1: Illustration of storage of data in files

10.3 STEPS IN PROCESSING A FILE

The processing of a file involves the following steps:

- **Opening a file –** The first step in file processing is to open the file in an appropriate mode. If you want to read from a file in your program, the file must be opened in the read/input mode. Similarly, if you want to write the output of your program to a file, the file must be opened in the write/output mode. However, when you want to read from and write onto the same file; then, the file must be opened in read-write mode.
- **Reading from or writing onto a file** – Once a file is opened successfully, data can be read from a file or written to a file in various ways.
- **Closing the file –** This is the final step in file processing. Once we have completed the reading and writing of the files, each file must be closed.

10.3.1 Opening a File

Before a program can write to a file or read from a file, the program must open it. Opening a file establishes an understanding between the program and the operating system. This provides the operating system with the *file's nam*e and the *mode* in which the file is to be opened, *i.e.*, whether for reading, writing, or appending.

Communication areas are set up between the file and the program. One of these areas is a structure of type *FILE*, declared in header file *stdio.h* that holds information about the file.

We need to declare one file pointer for every file using the following declaration

```
FILE *fp;
```

Next, we can open the file using *fopen()* function as

```
fp = fopen( "filename", "mode" );
```

The *fopen()* function request the operating system to open a file named *filename* in given *mode*. It returns a pointer to FILE structure if the request is granted; otherwise returns a *NULL* pointer.

Therefore, a program can check the value returned by the *fopen()* function to determine whether the file is opened successfully or not. Hence, it is always better to write a program segment responsible for opening a file as.

```
fp = fopen ("filename", "mode" );
if ( fp == NULL )
{
   printf( „\nUnable to open file.\n" );
   return 1;
}
```

Each file opened will have its own *FILE* structure, with a pointer to it. Any program can open any number of files.

Table 10.2: File opening modes

Mode	Description
r	Open a text file for reading. The file must already exist.
w	Open a text file for writing. If the file already exists, its contents are overwritten. If it does not exist, it will be created.
a	Open a text file for append. Data will be added to the end of the file. If the file does not exist, it will be created.
r+	Open a text file for both reading and writing. The file must already exist.
w+	Open a text file for both reading and writing. If the file exists, its contents are overwritten. If it does not exist, it will be created.
a+	Open a text file for both reading and appending. If the file does not exist, it will be created.
rb	Open a binary file for reading. The file must already exist.
wb	Open a binary file for writing. If the file already exists, its contents are overwritten. If it does not exist, it will be created.
ab	Open a binary file for append. Data will be added to the end of the file. If the file does not exist, it will be created.
rb+	Open a binary file for both reading and writing. The file must already exist.
wb+	Open a binary file for both reading and writing. If the file exists, its contents are overwritten. If it does not exist, it will be created.
ab+	Open a binary file for both reading and appending. If the file does not exist, it will be created.

10.3.2 Closing a File

When a program has finished with the reading/writing of a file, it must be closed. This task is carried out by using the *fclose()* library function, whose syntax is

```
fclose( fp );
```

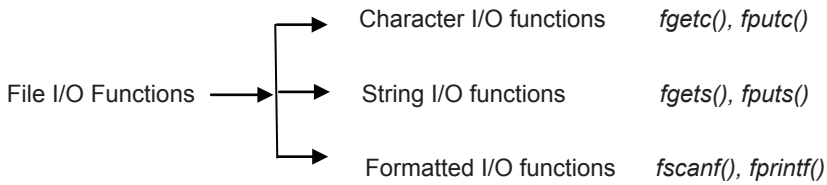
Where *fp* is a file pointer associated with a file that is to be closed.

Closing a file explicitly has two effects. These are

- Any data remaining in the buffer is written to the file. It may be noted that the buffer used is invisible to the programmer.
- Frees the communication areas used by the particular file so that they are available for other files. These areas include the *FILE* structure and the buffer itself.

10.3.3 Reading and Writing of Text Files

There is a variety of ways of reading data from and writing data to a text file. These are



10.3.3.1 Reading and Writing using Character I/O Functions

Using character I/O functions, data can be read or written one character at a time. This is analogous to the way functions *putchar(),* and *getchar()* write data to screen and read data from the keyboard.

Writing to a file

Once the program has established a line of communication with a particular file by opening it, it can write to it. The syntax of function that writes one character at a time is

```
fputc( ch, fp );
```

Where *ch* is a character variable or constant, and *fp* is a file pointer.

The following listing shows the example of a program that accepts a line of text from the keyboard, one character at a time, and writes into a disk file named *file1.dat*.

Listing 10.1

```
/*
   Program to demonstrate writing of one character at a time to file
*/
#include<stdio.h>
int main()
{
   FILE *fp;
   char ch;
   fp = fopen( "file1.dat", "w" );
   if ( fp == NULL ) {
      printf( "\nUnable to open file1.dat\n" );
      return 1;
   }
   printf( "\nType a line of text, when finished" );
   printf( ", when finished hit Enter key\n" );
   while ( ( ch = getche() ) != '\r' )
      fputc ( ch, fp );
   fclose( fp );
   return 0;
}
```

When the above program is executed, it prompts the user to type a line of text. When you have finished, you hit the Enter (↵) key to terminate the program.

Test Run

```
C is a powerful procedural language. It provides both low-level as well
as high-level features. ↵
```

Reading from a file

If the program can write to a file, it should also be able to read from a file. The syntax of the function that reads and returns one character at a time is

```
ch = fgetc( fp );
```

Where *ch* is a character variable, and *fp* is a file pointer.

The *fgetc()* function returns the character read from the file or the end-of-file (EOF) character if it has reached the end of the file.

The following listing shows the example of a program that reads one character from a file and writes onto the screen.

Listing 10.2

```
/*
    Program to demonstrate reading of character at a time from a file
*/
#include<stdio.h>
int main()
{
   FILE *fp;
   char ch;
   fp = fopen( "file1.dat", "r" );
   if ( fp == NULL ) {
      printf( "\nUnable to open file1.dat\n" );
      return 1;
   }
   printf( "\nContens of file are:\n\n" );
   while ( ( ch = fgetc (fp) ) != EOF )
      putchar( ch );
   fclose( fp );
   return 0;
}
```

When the above program is executed, it reads the contents of file *file1.dat*, one character at a time, and writes onto the screen till it encounters the end-of-file (EOF).

Test Run

```
Contents of the file are:
C is a powerful procedural language. It provides both low-level as well
as high-level features.
```

10.3.3.2 Reading and Writing using String I/O functions

Using string I/O functions, data can be read or written in the form of a string of characters. Reading and writing strings of characters is as easy as individual characters.

Writing to a file

The syntax of the function that writes a string of characters at a time is

```
fputs( str, fp );
```

Where *str* is an array of characters or a string constant, and *fp* is a file pointer.

The following listing shows the example of a program that accepts a series of strings from the keyboard and writes them onto a disk file.

Listing 10.3

```
/*
   Program to demonstrate writing of strings to a file
*/
#include<stdio.h>
#include<string.h>
int main()
{
   FILE *fp;
   char str[81];
   fptr = fopen( "file2.dat", "w" );
   if ( fp == NULL ) {
      printf( "\nUnable to open file2.dat\n" );
      return 1;
   }
   printf( "\nEnter a set of strings, press just" );
   printf( " Enter key to finish\n" );
   while ( strlen( gets( str ) ) > 0 )
   {
      fputs( str, fp );
      fputs( "\n", fp );
   }
   fclose( fp );
   return 0;
}
```

Test Run

```
C is a powerful procedural language. ↵
It is a middle-level language. ↵
All Programs are tested using Turbo C Compiler. ↵
↵
```

When the above program is executed, it accepts a set of strings terminated by hitting *Enter* key. When you have finished, hit the *Enter* key initially without entering anything, which is taken as a string of length 0, *i.e.*, *null string*, to terminate the program.

In the above program, we have set up an array of characters to store the string. The *fputs()* function then writes the contents of the array to the disk. Since the *fputs()* function does not automatically add a newline character to the end of the string, we must explicitly make it easier to read the string back from the file.

Reading from a file

The syntax of the function that reads strings from a file is

```
fgets( str, n, fp );
```

Where *str* is an array of characters and specifies the address where the string is stored, *n* is the maximum length of the input string, and *fp* is a file pointer.

The *fgets()* function returns a *NULL* value when it reads end-of-file *EOF*.

The following listing shows an example of a program that reads one string at a time from a file and writes onto the screen.

Listing 10.4

```
/*
   Program to demonstrate reading of strings from a file
*/
#include<stdio.h>
#include<string.h>
int main()
{
   FILE *fp;
   char str[81];
   fp = fopen( "file2.dat", "r" );
   if ( fp == NULL ) {
      printf( "\nUnable to open file2.dat\n" );
      return 1;
   }
   printf( "\nContens of file are:\n\n" );
   while ( fgets( str, 80, fp ) != NULL ) {
      puts( str );
   }
   fclose( fp );
   return 0;
}
```

When the above program is executed, it reads the contents of *file3.dat* one string at a time and writes them onto the screen till it encounters EOF.

Test Run

```
Contents of the file are:
C is a powerful procedural language.
It is a middle-level language.
All Programs are tested using Turbo C Compiler.
```

10.3.3.3 Reading and Writing using Formatted I/O functions

So far, we have considered reading and writing characters, strings, and integer numbers. *What about the real numbers*? And *What about the mixed type of data*? For example, suppose that we want to store information about an agent comprising his name (a string), code number (an integer number), and height (a natural number). We want to create a data file for a given list of agents. This can be done using formatted I/O functions.

Writing to a file

The syntax of the function that writes formatted data to a file is

```
fprintf( fp, "format-string" , ditems );
```

Where *fp* is a file pointer, and *ditems* is a list of variables to be written to a file. The *fprintf()* is similar to the *printf()* function; the only difference is that *printf()* function writes formatted data onto the screen.

Listing 10.5

```
/*
   Program to demonstrate writing of formatted data to a file
*/
#include<stdio.h>
int main()
{
   FILE *fp;
   char yes_no;
   char name[41];
   int code;
   float height;
   fp = fopen( "file3.dat", "w" );
   if ( fp == NULL ) {
      printf( "\nUnable to open file3.dat\n" );
      return 1;
   }
   while(1)
   {
      printf( "\nEnter name, code number, height" );
      scanf( "%s,%d,%f", name, &code, &height );
```

```
        fprintf( fp,"%s,%d,%f", name, code, height );
        printf( "Any more input y/n?: " );
        yes_no = tolower( getche() );
        fflush ( stdin );
        if ( yes_no == 'n' )
           break;
    }
    fclose( fp );
    return 0;
}
```

Test Run

```
Enter name, code number, height
Geetu, 10, 160.25
Any more input y/n?: y
Enter name, code number, height
Shivani, 11, 158.5
Any more input y/n?: y
Enter name, code number, height
Vijay, 12, 165.5
Any more input y/n?: y
Enter name, code number, height
Gurpreet, 13, 166.25
Any more input y/n?: n
```

The above information is now in a file named *file3.dat*. If we attempt to look at it using the TYPE command, the entire output will be on the same line as there are no *newlines* in the data. To format the output more conveniently, we can write a program specifically to read the above file using formatted input.

Reading from a file

The syntax of the function that reads formatted data to a file is

```
fscanf( fp, "format-string" , ditems );
```

Where *fptr* is a file pointer, and *ditems* is a list of addresses where the values read from a file are to be stored.

Listing 10.6

```
/*
   Program to demonstrate writing of formatted from a file
*/
#include<stdio.h>
int main()
```

```
{
   FILE *fp;
   char yes_no, char name[41];
   int code;
   float height;
   fp = fopen( "file3.dat", "r" );
   if ( fp == NULL ) {
      printf( "\nUnable to open file3.dat\n" );
      return 1;
   }
   printf( "\nContens of file are:\n\n" );
   while( fscanf(fp,"%s,%d,%f", name, &code, &height) != EOF ) {
      printf( "%s %d %f\n", name, code, height );
   }
   fclose( fp );
   return 0;
}
```

Test Run

```
Contents of file are:
Geetu 10 160.25
Shivani 11 158.5
Vijay 12 165.5
Gurpreet 13 166.25
```

10.3.4 Reading and Writing of Binary Files

The C language provides *record I/O*, sometimes called *block* I/O, functions to read and write data to binary files.

Record I/O writes numbers to disk files in binary format so that integers are stored in two bytes; long integers are stored in four bytes, single-precision floating-point numbers in four bytes, and double-precision floating-point numbers in eight bytes — *the same format used to store numeric data in memory*.

Record I/O also permits reading and writing of data simultaneously; the process is not limited to a single character or string or a few values. Arrays, structures, the array of structures can be read and written as a unit.

Writing to a file

The syntax of the function that writes a block of data at a time is

```
fwrite( ptr, m, n, fp );
```

Where *ptr* is an address of an array or a structure to be written, *m* is the size of an array or a structure, *n* is the number of such arrays or structures to be written, and *fp* is a file pointer of an opened in binary mode for writing.

After writing the block, the *fwrite()* function returns the number of data items actually written. If the number of items written is less than requested, it means some error has occurred.

Writing Arrays

Suppose we want to store an integer array having 10 elements to a file named *file4.dat*.

Listing 10.7

```
/*
   Program to demonstrate writing of an entire array to a file
*/

# include <stdio.h>
int main()
{
   FILE *fp;
   int i, a[10];
   fp = fopen( "file4.dat", "wb" );
   if ( fp == NULL ) {
      printf( "\nUnable to open file4.dat\n" );
      return 1;
   }
   printf( "\nEnter ten values\n" );
   for ( i = 0; i <= 10; i++ )
       scanf( "%d", &a[i] );
   fwrite(a, sizeof(a), 1, fp); /* write entire array to file */
   fclose( fp );
   return 0;
}
```

When executed, this program will prompt the user to enter ten integer values and then writes the entire array to the disk file named *file4.dat*.

Writing Structures

Suppose we want to write a structure named *agent* whose elements are — *name* (maximum of 40 characters), *code* (maximum of 5 characters), and *height* (a real number) to a disk file named *file5.dat*.

Listing 10.8

```
/*
   Program to demonstrate writing of an entire structure to a file
*/
#include<stdio.h>
struct
{
   char name[41];
```

```
    char code[6];
   float height;
} agent;
int main()
{
   FILE *fp;
   char yes_no, numstr[40];
   fp = fopen( "file5.dat", "wb" );
   if ( fp == NULL ) {
      printf( "\nUnable to open file5.dat\n" );
      return 1;
   }
   while ( 1 ) {
      printf( "\nEnter name : " );
      gets( agent.name );
      printf( "\nEnter code number : " );
      gets( agent.code );
      printf( "\nEnter height : " );
      gets( numstr );
      agent.height = atof ( numstr );
      fwrite( &agent, sizeof(agent), 1, fp );
      printf( "Any more input y/n?: " );
      yes_no = tolower( getche() );
      fflush( stdin );
      if ( yes_no == 'n' )
          break;
   }
   fclose( fp );
   return 0;
}
```

When executed, this program will prompt the user to enter the particulars of an agent, stores them in a structure, and then writes this structure to the disk file named *file5.dat* in a single write operation.

Instead of a structure variable, an agent is an array of structures with *n* elements, and we want to write the entire array in a single write operation.

This task can be accomplished by writing the *fwrite()* function as

```
fwrite( agent, sizeof(agent[0]), n, fp );
```

The above program demonstrates another way of performing input of numeric data. The numeric data can also read as a string and converted later on to the appropriate value. The library function *atoi()* converts an integer string to an integer value, whereas the function *atof()* converts a real string to a real number. Their counterparts are *itoa()* and *fcvt()*, which convert an integer value and a real value to the

appropriate string. These functions are defined in header files *stdlib.h*. The question may arise — *Why are these conversions required*?

There are two reasons for this.

- Sometimes we may need to combine a numeric value with a string, and this will be possible only if the numeric value is converted into a string.
- The C system is very inconsistent with the input of real numbers with the *scanf()* function. Sometimes it gives an error message — *floating point format not linked*.

Another function used in the above program is *fflush()* function. This function is used to flush the undesired data that may be left in the keyboard buffer. You may have observed that sometimes the system did not stop for a particular input in response to a call to some input function. This is because of some undesired data that is left in the keyboard buffer. So we can flush it before taking the next input using the *fflush()* function.

Reading from a file

The syntax of the function that reads a block from a file is

```
fread( ptr, m, n, fp );
```

Where *ptr* is an address of an array or a structure where the block will be stored after reading, *m* is the size of an array or a structure to be read, *n* is the number of such arrays or structures to be read, and *fp* is a file pointer of a file opened in binary mode for reading.

The *fread()* function returns the number of actual data items read. The usage of this function is illustrated in the following program.

Listing 10.9

```
/*
   Program to demonstrate reading of entire structure from a file
*/
#include<stdio.h>
struct
{
   char name[41];
   char code[6];
   float height;
} agent;
int main()
{
   FILE *fp;
   int record_no = 0;
   fp = fopen( "file5.dat", "rb" );
   if ( fp == NULL )
   {
      printf( "\nUnable to open file5.dat\n" );
      return 1;
```

```
    }
    while ( fread(&agent,sizeof(agent),1,fp) > 0 )
    {
       printf( "\nRecord #%d\n", record_no );
       printf( "\nName : %s", agent.name );
       printf( "\nCode number : %s", agent.code );
       printf( "\nHeight : %.2f", agent.height );
       record_no++;
       printf("\n\nPress any key to see next record..." );
       getch();
    }
    fclose( fp );
    return 0;
}
```

The above program reads the data file created by the program listing 10.8. It reads one structure at a time and displays it onto the screen. The *while* loop will terminate when the *fread()* function returns value 0, which means it cannot read a block. This indicates the end of the file.

10.4 FILE POSITIONING FUNCTIONS

The C language provides the following functions to handle file positioning operations:

- Function *rewind()* to set the file pointer to the beginning of the file.
- Function *ftell()* to know the current position of the file pointer in the file.
- Function *seek()* to change the position of the file pointer in the file.

10.4.1 Rewind File: *rewind()* Function

One way of positioning the file pointer to the beginning of the file is to close the file and then re-open it again. However, we can accomplish the same task without closing the file using the *rewind()* function. This function positions the file pointer at the beginning of the file. This function sounds very much like rewinding the audio or video cassette in order to listen or watch the cassette from the beginning.

The syntax of *rewind()* function is

```
rewind(fp);
```

where *fp* is a file pointer for the currently opened file.

10.4.2 Current Location: *ftell()* Function

In some situations, it may be required to find the current location of the file pointer within the file. The *ftell()* function lets us know the current position of the file pointer.

The syntax of the *ftell()* function is

```
long k = ftell(fp);
```

where *fp* is a file pointer for the currently opened file, note that the *ftell()* function returns a long integer. This is necessary because many files may have more than 32767 bytes of data.

Recall that the C I/O system considers files as streams of bytes of data. It measures the position in the file by the number of bytes relative to zero, *i.e.*, from the beginning of the file. When the file pointer is at the beginning of the file, the *ftell()* function returns 0. If the file pointer is at the second byte, the *ftell()* function returns value 1.

10.4.3 Set Position: *fseek()* Function

To read a data item from anywhere in a file, we have to move the file pointer to the beginning of that data item. To accomplish this task, we can use the fseek() function.

The syntax of the *fseek()* function is

```
fseek(fp, offset, wherefrom );
```

The *fseek()* function takes three arguments, where the first argument *fp* is a file pointer, the second argument *offset* is a variable of type *long integer* that specifies the number of bytes by which file pointer is to move. The third argument *wherefrom* specifies from which position the *offset* is measured.

The various values for argument *wherefrom* are listed in Table 10.3.

Table 10.3: Various Values of *wherefrom* for *fseek()* function

Mode	Offset is measured from
SEEK_BEG	Beginning of the file
SEEK_CUR	Current position of the file
SEEK_END	End of the file

Table 10.4: Some examples illustrating the use of the *fseek()* function

Seek Call	Action performed
`fseek(fp, 0, SEEK_BEG);`	Moves the file pointer fptr to the beginning of the file. If the file pointer is currently at the beginning of the file, it results in no action.
`fseek(fp, n, SEEK_BEG);`	Moves the file pointer fptr forward by n bytes, i.e., moves the file pointer to (n+1) the bytes in the file.
`fseek(fp, -n, SEEK_CUR);`	Moves the file pointer fptr backward by n bytes from the current position.
`fseek(fp, 0, SEEK_END);`	Moves the file pointer fptr to the end of the file. If the file pointer is currently at the end of the file, it results in no action.
`fseek(fp, 0, SEEK_END);`	Moves the file pointer fptr to the end of the file. If the file pointer is currently at the end of the file, it results in no action.
`fseek(fp, n, SEEK_CUR);`	Moves the file pointer fptr forward by n bytes from the current position.
`fseek(fp, 1, SEEK_CUR);`	Moves the file pointer fptr to the next byte.
`fseek(fp, -1, SEEK_CUR);`	Moves the file pointer fptr to the previous byte.
`fseek(fp, m, SEEK_END);`	Moves the file pointer fptr backward by m bytes from the end of the file.

10.5 FILE STATUS FUNCTIONS

The C language provides the following functions to handle file status queries:

- Function *feof()* to test end of the file.
- Function *ferror()* to test error.
- Function *clearerr()* to clear error.

10.5.1 Test End of File: *feof()* Function

The *feof()* function is used to check if the end of the file has been reached. If the file pointer is at the end, *i.e.*, all data have been read, the function returns 1. If the end of the file is not reached, it returns value 0.

The syntax of *feof()* is

```
feof(fp);
```

where *fp* is a file pointer for the currently opened file.

10.5.2 Test Error: *ferror()* Function

The *ferror()* function is used to test the error status of the file. As mentioned earlier, errors can be created for many reasons, ranging from bad media (disk, CD, etc.) to illegal operations such as reading a file in the write state.

Suppose the *ferror()* function returns 1 if an error has occurred after a file operation. If no error has occurred, the file ferror() returns value 0.

The syntax of *ferror()* is

```
ferror(fp);
```

where *fp* is a file pointer for the currently opened file.

It is important to note here that testing for an error does not reset the error condition. Once an error has occurred, it can only return to the standard read or write state after clearing the error state using the *clearer()* function described next.

13.5.3 Clear Error: *clearerr()* Function

When an error occurs, the subsequent calls to the *ferror()* function return 1 until the error status of the file is reset. The *clearerr()* function is used for this purpose.

The syntax of *clearerr()* is

```
clearerr(fp);
```

where *fp* is a file pointer for the currently opened file.

It is important to note here that we have not necessarily cured the problem even though we have cleared the error. We may find that subsequent read or write operations may return to the error state.

ILLUSTRATIVE EXAMPLES

Example 10.1: *Program to find the size of a given file, where the user provides the file's name.*

To know the size of the specified file, position the file pointer at the end of the file using the *fseek()* function and access the value of the file pointer using the *ftell()* function that gives the size of the file.

Listing 10.10

```
/*
   Program to find the size of a given file
*/
#include <stdio.h>
int main()
{
   FILE *fptr;
   char fname[30];
   printf("\nEnter name of file : ");
   gets(fname);
   fptr = fopen(fname, "r");
   if (!fptr) {
      printf("\nFile %s does not exist\n", fname);
      return 1;
   }
   fseek(fptr, 0L, 2);
   printf("\nSize of file = %ld bytes.\n", ftell(fptr));
   fclose(fptr);
   return 0;
}
```

Test Run

```
Enter name of file : fsize.c
Size of file = 443 bytes.
```

The above test run shows that the size of the program file (source code) created in Listing 10.7 is 443 bytes.

Example 10.2: *Write a program that reads some text from the keyboard and writes it into a TEXT file. The program then reads this file and displays its contents on the screen.*

Listing 10.11

```
/*
  Program that creates a file and then reads its contents
  and display them on the computer screen
*/
#include <stdio.h>
int main()
{
   char ch;
   FILE *fp1;
   fp1 = fopen("TEXT", "w");    /*  open file for writing  */
   if ( fp1 == NULL ) {
      printf("\nUnable to open file TEXT for writing\n");
      return 1;
```

```
    }
    printf("\nType some text and terminate");
    printf(" the input by Enter key\n\n");
    while ( ( ch = getche() ) != '\r' )
      fputc(ch, fp1);
    fclose(fp1);   /*  close file  */
    fp1 = fopen("TEXT", "r");    /*  open file for reading  */
    if ( fp1 == NULL ) {
       printf("\nUnable to open file TEXT for reading\n");
       return;
    }
    printf("\n\nContents of file TEXT are\n\n");
    while ( !feof(fp1) ) {
       ch = fgetc(fp1);
       putchar(ch);
    }
    fclose(fp1);
    return 0;
}
```

Test Run

```
Type some text and terminate the input by Entering key
Don't do anything with others that you wish others should not
do with you. ↵
Contents of file TEXT are
Don't do anything with others that you wish others should not
do with you.
```

Example 10.3: *Write a program to copy the contents of one file to another file byte-by-byte. The user provides the names of the files.*

Listing 10.12

```
/*
    Program to copy the contents of a given file to another file.
*/
#include <stdio.h>
int main()
{
   char ch;
   FILE *fp1, *fp2;
   char file1[30], file2[30];
   printf("\nEnter name of source file : ");
   gets(file1);
   printf("\nEnter name of destination file : ");
   gets(file2);
```

```
    fp1 = fopen(file1, "r");
    if ( fp1 == NULL ) {
        printf("\nUnable to open file: %s for reading\n", file1);
        return 1;
    }
    fp2 = fopen(file2, "w");
    if ( fp2 == NULL ) {
        printf("\nUnable to open file: %s for writing\n", file2);
        return 1;
    }
    while ( !feof(fp1) ) {
        ch = fgetc(fp1);
        fputc(ch, fp2);
    }
    printf("\nFile copied successfully...\n");
    fclose(fp1);
    fclose(fp2);
    return 0;
}
```

Test Run

```
Enter the name of the source file: file_copy.c
Enter the name of destination file: temp
File copied successfully...
```

You can verify that the contents of file *temp* will be the same as that of *file_copy.c*.

UNIT SUMMARY

In this chapter, we have learned that

- If the volume of the input data is vast, it can be best handled using data files.
- If the data generated by some instrument is in machine-readable form (binary form), you have to use a data file to pass on this data to your program.
- If the output volume is immense and can't be viewed on the screen properly, it is better to write the output to a data file that you can refer to any time without re-executing your program.
- If the output generated by one program is to be used as input for another program, again, the use of data files will be advantageous.
- A data file can be a text file or a binary file.

EXERCISE

Subjective Questions

1. Name the various systems for performing file I/O in C.
2. Is it is advisable to close a file explicitly opened for writing?
3. Name the function to close a file.
4. What information system gets by opening a file?
5. What do you mean by a *file pointer*?
6. How a file opened with the *fopen()* function is referred to in a program?
7. Suppose a program wants to examine every byte of a file; which mode will be more appropriate?
8. What do you mean by term *offset* in reference to a file?
9. What is the task performed by the *fseek()* function?
10. What is the task performed by the *ftell()* function?
11. Which value is returned by the *fopen()* if some error occurs in opening a file?
12. What is the difference between *w+* and *r+* modes?
13. What a *rewind()* function does?

Multiple Choice Questions

1. Which of the following is a default file pointer?
   (*a*) stdin (*b*) stdout
   (*c*) stderr (*d*) All of the above
2. The function *fopen ()* when fails to open file, then it returns value
   (*a*) NULL (*b*) −1
   (*c*) Null (*d*) void
3. The function *flose()* is, usually, used to
   (*a*) delete a file (*b*) disconnect file from program
   (*c*) read data from a file (*d*) write data to a file
4. Which of the following values is not a value of *from* in function `seek(fptr,offset,from)`?
   (*a*) 2 (*b*) 0
   (*c*) 1 (*d*) EOF
5. The task performed by function `fseek(fptr,0,0)` is
   (*a*) fclose (fptr) (*b*) ftell (fptr)
   (*c*) search (fptr) (*d*) rewind (fptr)
6. Which of the following is not a valid file opening mode?
   (*a*) r+ (*b*) +r
   (*c*) r (*d*) rb

7. If letter *b* is suffixed with the file opening mode, what does it convey?

(*a*) file is to be opened for reading only (*b*) file is to be opened for writing only

(*c*) file is to be opened for both read-write (*d*) file is a binary file

8. Consider the following code segment

```
FILE *fp;
fp =  fopen("inventory.dat", "rb");
```

What is the role of letter *b* in "*rb*"?

(*a*) Open *inventory.dat* file in binary mode for reading.

(*b*) Open *inventory.dat* file reading and writing.

(*c*) Create a new file *inventory.dat* for writing.

(*d*) Open *inventory.dat* file in binary mode for writing and reading.

9. Consider the following code segment

```
FILE *fp;
fp =  fopen("notes.txt", "r+");
```

Which of the following operations can be performed on the *notes.txt* file?

(*a*) Reading (*b*) Writing

(*c*) Appending (*d*) All of the above

10. FILE is of type _____ .

(*a*) *int* type (*b*) *struct* type

(*c*) *string* type (*d*) *structure* type

11. If there is any error while opening a file, fopen will return _____ .

```
FILE *fp;
```

(*a*) null (*b*) NULL (*c*) Null (*d*) EOF

12. Which of the following library function can be used to detect end-of-file?

(*a*) eof() (*b*) isend() (*c*) feof() (*d*) end()

13. A mode which is used to open an existing file for both reading and writing is __ .

(*a*) +r (*b*) r+ (*c*) w+ (*d*) w

14. Which of the following function can be used to re-read a file without closing & re-opening?

(*a*) fseek() (*b*) rewind()

(*c*) ftell() (*d*) Both (*a*) and (*b*)

15. Which of the following is not a file status function?

(*a*) ferror() (*b*) ftell() (*c*) clearerr() (*d*) feof()

ANSWERS															
1.	(*d*)	2.	(*a*)	3.	(*b*)	4.	(*d*)	5.	(*d*)	6.	(*b*)	7.	(*d*)	8.	(*d*)
9.	(*d*)	10.	(*b*)	11.	(*b*)	12.	(*c*)	13.	(*b*)	14.	(*d*)	15.	(*b*)		

Programming Problems

1. Given a text file containing some text. Write a program that prompts the user to input the name of a text file, reads this file, and outputs the number of vowels and number of words in the text.
2. Given a text file containing some text. Write a program that prompts the user to input the name of a text file, reads this file, and replaces each character 'x' with uppercase 'X'.
3. A beginner to the C language has typed the entire program in the uppercase letter; as you know, the usual convention in C language is to type the source code using lowercase letters only except for macros and defined constants. Write a program that converts the source code into the lowercase letter. Comments should be left unchanged.
4. Write a program to read numbers from a file and write even, odd and prime numbers to separate file.
5. Write a program to merge two files end-to-end into a third file.
6. Write a program to print source code of itself as output on the screen.
7. Write a program to count characters, words and lines in a text file

PRACTICALS

1. Write a program that reads some text from the keyboard and write it in a file named *sample.txt*. The program then reads this file and displays its contents on screen without closing and reopening the file.

   The file is opened in "w+" mode, and once the file is created, we rewind it using *rewind()* to reposition the file pointer at the beginning of the file.

Listing 10.13

```
/*
 Program that creates a file and then reads its contents without
 closing and re-opening, and display them on the computer screen
 */
 #include <stdio.h>
 int main()
 {
    char ch;
    FILE *fp;
    fp = fopen("sample.txt", "w+");
    if ( fp == NULL ) {
       printf("\nUnable to open file sample.txt\n");
       return 1;
    }
```

```
    printf("\nType some text...\n\n");
    while ( ( ch = getche() ) != '\r' )
       fputc(ch, fp);

    rewind(fp);
    printf("\n\n\nContents of file...\n\n");
    while ( !feof(fp) ) {
       ch = fgetc(fp);
       putchar(ch);
    }
    fclose(fp);
    return 0;
}
```

Test Run

```
Type some text...
This is a sample file.
Contents of file...
This is a sample file.
```

2. Write a C program to reverse the first *n* characters in a file. The *user provides the file name and value of n*.

Listing 10.14

```
/*
    Program to reverse the first n characters in a file
*/
#include <stdio.h>
int main()
{
   char str[80], filename[30], ch;
   int i, n;
   FILE *fp;
   printf("\nEnter file name : ");
   gets(filename);
   printf("\nEnter value for n : ");
   scanf("%d", &n);

   fp = fopen(filename, "r+");
   if ( fp == NULL ) {
        printf("\nUnable to open file: %s\n", filename);
        return 1;
   }
```

```
    i = 0;
    while ( ( !feof(fp) ) && ( i < n ) )
    {
       str[i] = fgetc(fp);
       i++;
    }
    rewind(fp); /* reposition the file pointer to the beginning */
    i = n-1;
    while ( i >= 0 )
    {
       fputc( str[i], fp);
       i--;
    }
    fclose(fp);
    return 0;
}
```

Test Run

```
Enter file name: testfile.txt
Enter the value for n: 10
Original contents of the file were
1234567890abcdefghijklmnopqrstuvwxyz
After the program execution, the contents of the file are
0987654321abcdefghijklmnopqrstuvwxyz
```

You can verify that the file's contents after program execution by using the type command on the command prompt.

In the above program, first, the given file is opened in reading/write mode. Next, the first *n* bytes of the file are copied in a dynamically allocated character array. Next, we rewind the given file, *i.e.*, reposition the file pointer at the beginning of the file.

Finally, we write the contents of the character array in reverse order onto the file. Thus, we can accomplish the desired task.

3. Write a program that reads names and marks of *n* number of students from the keyboard and stores them in a file. If the file already exists, then add the information to the file.

Listing 10.15

```
/*
    Program to store names and marks of students in file
*/
#include <stdio.h>
int main()
{
   FILE *fp;
```

```
    char name[50], filename[30];
    int i, n, marks;
    printf("Enter name of file : ");
    gets(filename);
    printf("Enter number of students to add : ");
    scanf("%d", &n);
    fp = fopen(filename, "a");
    if (fp == NULL) {
         printf("\nUnable to open file: %s\n", filename);
            return 1;
    }
    for(i = 0; i < n; i++)
    {
       printf("\nEnter data for student %d\n", i+1);
       printf("\nEnter name : ");
       scanf("%s", name);
       printf("Enter marks: ");
       scanf("%d", &marks);
       fprintf(fp,"\nName: %s \nMarks=%d \n", name, marks);
   }
   fclose(fp);
   return 0;
}
```

KNOW MORE

The teacher is expected to understand the concepts of data files and the various aspects related to file handling in the C language.

The teacher should also demonstrate files by taking appropriate examples and creating C programs to process the files with student participation.

REFERENCES & SUGGESTED READINGS

1. R. S. Salaria, Problem Solving & Programming in C, Khanna Book Publishing Co(P) Ltd., New Delhi.
2. E. Balagurusamy, Programming in ANSI C, Tata McGraw Hill, New Delhi.
3. Yashavant Kanetkar, Let Us C, BPB Publications, New Delhi.
4. Byron Gottfried, Programming with C, Schaum's Outlines.
5. https://onlinecourses.nptel.ac.in/noc21_cs01/preview
6. https://ocw.mit.edu/courses/intro-programming/
7. https://www.programiz.com/c-programming
8. https://www.javatpoint.com/c-programming-language-tutorial

REFERENCES FOR FURTHER LEARNING

1. Brain W. Kernighan and Deniss M. Ritchie, The C Programming Language, 2nd Edition, Prentice Hall.
2. Herbert Schildt, C: The Complete Reference, 4th Edition, McGraw Hill.
3. R. S. Salaria, Test Your Skills in C, Khanna Book Publishing Co(P) Ltd.
4. R. S. Salaria, Cracking IT Interviews, Khanna Book Publishing Co(P) Ltd.
5. Stephen Prata, C Primer Plus, Addison-Wesley Professional.
6. David Griffiths and Dawn Griffiths, Head First C, Shroff Publishers & Distributors Pvt. Ltd.
7. Antti Laaksonen, Guide to Competitive Programming, 2nd Edition, Springer.
8. https://www.greatlearning.in/academy/learn-for-free/courses/c-programming
9. https://www.udemy.com/topic/c-programming/
10. https://www.edx.org/learn/c-programming
11. https://www.coursera.org/courses?query=c%20programming
12. https://www.learnconline.com/
13. https://www.codecademy.com/

CO AND PO ATTAINMENT TABLE

Course outcomes (COs) for this course can be mapped with the programme outcomes (POs) after the completion of the course and a correlation can be made for the attainment of POs to analyse the gap. After proper analysis of the gap in the attainment of POs, necessary measures can be taken to overcome the gaps.

Table for CO and PO Attainment

Course Outcomes	Expected Mapping with Programme Outcomes (1- Weak Correlation; 2- Medium correlation; 3- Strong Correlation)											
	PO-1	PO-2	PO-3	PO-4	PO-5	PO-6	PO-7	PO-8	PO-9	PO-10	PO-11	PO-12
CO-1												
CO-2												
CO-3												
CO-4												
CO-5												
CO-6												
CO-7												
CO-8												

The data filled in the above table can be used for gap analysis.

Index

N

O

P

Q

R

S