20

YEARS OF EXCELLENCE

RMK GROUP OF EDUCATIONAL INSTITUTIONS

COMPUTER FUNDAMENTALS AND PROGRAMMING IN C

ANITA GOEL
AJAY MITTAL

Pearson

Computer Fundamentals and Programming in C

This page is intentionally left blank

Computer Fundamentals and Programming in C

Anita Goel
Department of Computer Science and Engineering
Dyal Singh College
University of Delhi
New Delhi

Ajay Mittal
Department of Computer Science and Engineering
PEC University of Technology
Chandigarh

Pearson

Published by Pearson India Education Services Pvt. Ltd, CIN: U72200TN2005PTC057128, formerly known as TutorVista Global Pvt. Ltd, licensee of Pearson Education in South Asia.

ISBN 978-93-325-7681-0
eISBN 978-93-325-7813-5

Head Office: A-8 (A), 7th Floor, Knowledge Boulevard, Sector 62, Noida 201 309, Uttar Pradesh, India.
Registered Office: 4th Floor, Software Block, Elnet Software City, TS-140, Block 2 & 9,
Rajiv Gandhi Salai, Taramani, Chennai 600 113, Tamil Nadu, India.
Fax: 080-30461003, Phone: 080-30461060
www.pearson.co.in, Email: companysecretary.india@pearson.com

Contributions from

Dr S. Radhika
Associate Professor
Department of Science and Humanities
R.M.K. Engineering College

Dr J. Faritha Banu
Associate Professor
Department of Computer Science and Engineering
R.M.K. College of Engineering and Technology

Ms G. Nirmala
Assistant Professor
Department of Science and Humanities
R.M.D. Engineering College

This page is intentionally left blank

Roadmap to the Syllabus

Computer Programming

(Common to all branches of B.E./B.Tech. Programmes)

UNIT I:

Introduction
Generation and Classification of Computers – Basic Organization of a Computer – Number System – Binary – Decimal – Conversion – Problems. Need for Logical Analysis and Thinking – Algorithm – Pseudo Code – Flow Chart.

Refer Chapters 1 and 2

UNIT II:

C Programming Basics
Problem Formulation – Problem Solving – Introduction to "C" Programming – Fundamentals – Structure of a "C" Program – Compilation and Linking Processes – Constants, Variables – Data Types – Expressions Using Operators in "C" – Managing Input and Output Operations – Decision Making and Branching – Looping Statements – Solving Simple Scientific and Statistical Problems.

Refer Chapters 2, 3, 4 and 5

UNIT III:

Arrays and Strings
Arrays – Initialization – Declaration – One Dimensional and Two Dimensional Arrays – String – String Operations – String Arrays. Simple Programs – Sorting – Searching – Matrix Operations.

Refer Chapters 6 and 7

UNIT IV:

Functions and Pointers
Function – Definition of Function – Declaration of Function – Pass by Value – Pass by Reference – Recursion – Pointers – Definition – Initialization – Pointers Arithmetic – Pointers and Arrays – Example Problems.

Refer Chapters 6 and 8

UNIT V:

Structures and Unions
Introduction – Need for Structure Data Type – Structure Definition – Structure Declaration – Structure within a Structure – Union – Programs Using Structures and Unions – Storage Classes, Pre-processor Directives.

Refer Chapters 9 and 10

Brief Contents

This page is intentionally left blank

Contents

5 Decision-Making and Looping Statements 5.1

Part-III Arrays, Pointers and Strings 6.1

6 Arrays and Pointers 6.3

This page is intentionally left blank

Preface

"Dreams transform into thoughts, thoughts into actions and actions into reality"
—A.P.J. Abdul Kalam

"Until you try, you don't know what you can't do"
—Henry James

Why and How I Wrote this Book

I ventured into the field of C programming as a young novice undergraduate like you about fifteen years back. At that time I had a little programming experience with BASIC, PASCAL and FORTRAN languages. I had heard about the enormous power of C programming language and was fascinated about it. I learnt and practiced it for about five years, and then fortunately had the opportunity to teach it to young engineering students at PEC University of Technology (formerly Punjab Engineering College), Chandigarh. This new assignment changed my perspective a bit; however, my learning and understanding about the language continued to evolve. Gradually, I developed a flair for solving problems faced by students in conceiving and understanding the intricacies of the language. Years of teaching have given me a clear idea about how a student perceives, conceives and understands the language. During these years, I have observed the deficiencies and the weaknesses in the literature available on C language. It adopts a unique and well-tested practical approach towards learning C language. I am sure that this book will help you in gaining proficiency in C programming. **Happy Learning and All the Best!**

C Programming Language

C is a general-purpose, block-structured, procedural, case-sensitive, free-flow, portable, powerful high-level programming language. The language is so powerful that UNIX, one of the most accepted operating systems, is written in it. It is said that programming languages are 'born', 'age' and eventually 'die'. However, C programming language has only matured from the time it was born. It holds the same relevance today, as it held when it was developed by Dennis Ritchie at the Bell Telephone Laboratories in 1972.

Programming in C is introduced into undergraduate professional courses as a first programming course. The course intends to make the students well conversant with the syntax of C programming language and also focuses on the development of logic and problem-solving abilities in the students. The importance of this course can be clearly fathomed from the fact that the knowledge of C programming language is maintained as a pre-requisite for placements in almost all reputed software companies. Good understanding of C language also creates a strong foundation for learning other programming languages like C++, Java, etc.

About the Book

The book *Computer Fundamentals and Programming in C* adopts a unique and well-tested practical approach towards learning C programming language. The book covers the concepts in a lucid manner for the benefit of novice as well as amateur programmers who are looking for a comprehensive source to increase their skill in C programming. Though the book does not assume prior knowledge in the subject; a basic awareness of the working of computers will make the going easier.

Structure of the Book

The book is structured in ten chapters that are divided into exclusive parts to enable facile understanding of the underlying ideas that enable programming in C.

Part I encapsulates the Fundamentals of Computer Programming in the first two chapters. Chapter 1 traces the history and evolution of the computer and discusses the concept of input–process–output and the characteristics of the computer. It also reviews the classification of digital computers and the application of computers in different domain areas. Chapter 2 deals at length with data representation and the fundamentals of programming. It explains how denary, binary, octal and hexadecimal numbers can be inter-converted from one system to another, and delineates the concept of logic gates and their application to computer programming.

Part II, spanning Chapters 3–5, expounds on the Basics of C Programming. Chapter 3 provides an introduction to the C language along with a chronological listing of its various standards. It starts with a presentation of the common programming vocabulary such as character set, identifiers, keywords, variables, constants and data types before proceeding to expound on the techniques of writing, compiling and executing simple C programs. Chapter 4 describes operators and the ways of creating expressions using them. A detailed classification of operators as arithmetic, relational, logical, and bitwise, is presented. It also reveals how expressions are evaluated and gives an insight into the intricacies involved in this evaluation process. Statements that form the smallest independent unit within a C program are discussed in Chapter 5. The classification of statements into executable and non-executable statements, simple statements and compound statements, branching statements and iteration statements is presented in detail.

Chapters 6 and 7 make up Part III of the book and delve into Arrays, Pointers and Strings. Chapter 6 is devoted to derived data type arrays and pointers. It talks about the inter-relationship between arrays and pointers. Chapter 7 introduces strings and character arrays. Various string operations using library functions and user-defined functions are presented.

Part IV of the book comprises of Chapter 8, dedicated exclusively on Functions. Functions help in modularizing the program and the code reuse. The chapter sheds light on the importance of functions and expounds on the concept of recursion in a unique manner.

Part V, the concluding part, focuses on Structures and Unions. Chapter 9 explicates the definition of new data types using structures, unions and enums. The chapter covers bit-fields and interrupt programming, the practical application of unions. Chapter 10 analyzes the translators and focuses on a translator known as the preprocessor. Various directives used to control preprocessors are described in detail.

Salient Features and Strengths of the Book

The salient features and strengths of the book are:

1. Comprehensive coverage of C programming language. The content of each chapter is clear, lucid and self-explanatory.

2. The theory is reiterated through conceptual questions and their elucidative explanatory answers. The book has an extensive collection of nearly 1000 unique, relevant and conceptual questions. These questions have either been asked by the students during the courses on C programming or have been developed to cover each and every concept of the C programming language.
3. The concepts are explained with the help of programming examples. One of the unique features of the book is the presentation of programming examples with the help of remarks.
4. Simultaneous discussion on the behavior of a program with Borland Turbo C 3.0, Borland Turbo C 4.5 and MS-VC++ 6.0 compilers.
5. Unique and in-depth discussion on structure padding and recursion.

Typographical Conventions

The book tries to keep a consistent style in the use of special or technical terms. The normal text is written in Palatino Linotype regular typeface, whereas the C syntactic terms like reserved words, etc. are written in Agency FB regular typeface. The conceptual questions presented at the end of each chapter are written in *Palatino Linotype italic typeface* for normal text and Agency FB regular typeface for C syntactic terms. The answers to these conceptual questions appear at the same place in Palatino Linotype regular typeface for normal text and Agency FB regular typeface for C syntactic terms. The outputs to the code snippets and answers to multiple-choice questions are present at the end of each chapter using the same typographical conventions. The first occurrence of each technical term is in **bold**. The references to the topics present in the same chapter are given by providing footnotes.

Web Resources

All the source codes, online chapters and resources are available at our website.

Acknowledgements

I am grateful to Dr S. C. Gupta, Prof. S. K. Wasan and Dr Mukul Sinha for encouraging me to write a book. I thank them for their valuable advice and for their encouragement to disseminate information. I also thank them for continually inspiring me to write a good book.

I thank Mr Neeraj Saxena, Ms Renu Saxena and Mr Rajendra for their extensive help in clicking the photographs in this book.

My special thanks go to all my students, Amit Jain in particular, who have always been eager to inform me about the expectations of the students about the book. Their suggestions and feedback have helped me to write the book in a student-friendly manner.

Thanks to the team of Pearson Education for their extensive support. The book in its present form is a result of the long discussions and the brainstorming sessions with Sachin. I am grateful to Sachin for his ideas, suggestions and excellent support provided to me during the writing of this book.

I express my regards and love to my mother Urmila, and my father Amar Chand. Being their child makes me feel special, and I thank them both for guiding me through life. Through this book, I carry forward the legacy of book writing from my maternal grandfather late Mr C. S. Jain. I express my respect and thanks to my father-in-law Gopal Krishan for his motivation and cooperation, and for taking care of matters while I was busy writing the book. My thanks are due, in memory of my mother-in-law Pushpa who has always loved and supported me.

Thanks to my loving husband, Ajay, for standing by me in the difficult times during the course of writing this book. He is a bagful of innovative ideas and has contributed creatively to the writing of this book. My special thanks to my lovely and beautiful children, Anirudh and Ashima, who brilliantly contributed towards the creation and editing of this book. They are my greatest source of inspiration and motivation.

Since this is not the first book on this topic, I thank the authors of other books on similar topic, whose books have been a source of ideas for me.

I thank the Almighty without whose grace it would have been impossible for me to accomplish this task.

Anita Goel

A dream is visualized by a pair of eyes; however, many pairs of hands join together and work hard towards its realization. Throughout the project, I received the much-needed support at all fronts from various people. The list is so exhaustive that I may not be able to enumerate all the names. I express my heartfelt thanks to all who helped me at any point of time during the writing of this book. I would like to specially thank the following persons who have helped me in different ways.

My sincere thanks to Dr Manoj Dutta, Director, PEC University of Technology; Dr Sanjeev Sofat, Professor and Head, Computer Science and Engineering Department; Dr Vijay Gupta, Vice-Chancellor, Lovely Professional University, Ex-Director, Punjab Engineering College; my colleagues Divya and Arvind Kakria and my friends Praveen Grewal and Naveen Aggarwal for their unabated support and inspiration.

My students provided helpful insights while working on the drafts of the manuscript: Mohit Virmani, Deepti Sabani, Akansha Bansal, Subhangi Harsha, Ankit Anand, Amandeep Jakhu and Shefali Saroha. I thank Mohit for his thoughtful comments and dedicated efforts in proof reading. His reviews have considerably improved this book.

I am obliged to Thomas Mathew Rajesh, M.E. Sethurajan, Jennifer Sargunar, C. Purushothaman, Munish Modi and other members of the editorial and production teams of Pearson Education for their hard work and vast patience. I am especially thankful to Jennifer, who has taken personal interest towards the betterment of the script.

Last but not the least, I express my heartfelt gratitude to my parents Sh. T.L. Mittal and Smt. Prem Lata, and my brother Hemraj Mittal and his wife Sabina for their moral support and patience throughout the period of writing the book. My little nephew Jai Mittal was my inspiration and played an important role in his own way towards the early completion of the book.

Ajay Mittal

PART – I

COMPUTER FUNDAMENTALS

This page is intentionally left blank

1

BASICS OF COMPUTER

Learning Objectives

In this chapter, you will learn about:

- Digital and analog computers
- Characteristics of computer
- History of computer
- Generations of computer
- Classification of computer
- The computer system
- Central processing unit
- Memory unit
- Instruction format
- Instruction set
- Instruction cycle
- Microprocessor
- Interconnecting the units of a computer
- Performance of a computer
- Inside a computer cabinet
- Application of computers

1.1 Introduction

Nowadays, computers are an integral part of our lives. They are used for the reservation of tickets for airplanes and railways, payment of telephone and electricity bills, deposit and withdrawal of money from banks, processing of business data, forecasting of weather conditions, diagnosis of diseases, searching for information on the Internet, etc. Computers are also used extensively in schools, universities, organizations, music industry, movie industry, scientific research, law firms, fashion industry, etc.

The term computer is derived from the word *compute*. The word *compute* means *to calculate*. A *computer* is an electronic machine that accepts data from the user, processes the data by performing calculations and operations on it, and generates the desired output results. Computer performs both simple and complex operations, with speed and accuracy.

This chapter discusses the history and evolution of computer, the concept of input-process-output and the characteristics of computer. This chapter also discusses the classification of digital computers based on their size and type, and the application of computer in different domain areas.

1.2 Digital and Analog Computers

A *digital computer* uses distinct values to represent the data internally. All information are represented using the digits 0s and 1s. The computers that we use at our homes and offices are digital computers.

Analog computer is another kind of a computer that represents data as variable across a continuous range of values. The earliest computers were analog computers. Analog computers are used for measuring of parameters that vary continuously in real time, such as temperature, pressure and voltage. Analog computers may be more flexible but generally less precise than digital computers. Slide rule is an example of an analog computer.

This book deals only with the *digital computer* and uses the term *computer* for them.

1.3 Characteristics of Computer

Speed, accuracy, diligence, storage capability and versatility are some of the key characteristics of a computer. A brief overview of these characteristics are:

1. **Speed:** The computer can process data very fast, at the rate of millions of instructions per second. Some calculations that would have taken hours and days to complete otherwise, can be completed in a few seconds using the computer. For example, calculation and generation of salary slips of thousands of employees of an organization, weather forecasting that requires analysis of a large amount of data related to temperature, pressure and humidity of various places, etc.
2. **Accuracy:** Computer provides a high degree of accuracy. For example, the computer can accurately give the result of division of any two numbers up to 10 decimal places.
3. **Diligence:** When used for a longer period of time, the computer does not get tired or fatigued. It can perform long and complex calculations with the same speed and accuracy from the start till the end.

4. **Storage Capability:** Large volumes of data and information can be stored in the computer and also retrieved whenever required. A limited amount of data can be stored, temporarily, in the primary memory. Secondary storage devices like floppy disk and compact disk can store a large amount of data permanently.
5. **Versatility:** Computer is versatile in nature. It can perform different types of tasks with the same ease. At one moment you can use the computer to prepare a letter document and in the next moment you may play music or print a document.

Computers have several limitations too. Computer can only perform tasks that it has been programmed to do. Computer cannot do any work without instructions from the user. It executes instructions as specified by the user and does not take its own decisions.

1.4 History of Computer

Until the development of the first generation computers based on vacuum tubes, there had been several developments in the computing technology related to the mechanical computing devices. The key developments that took place till the first computer was developed are as follows:

1. **Calculating Machines** ABACUS was the first mechanical calculating device for counting of large numbers. The word ABACUS means calculating board. It consists of bars in horizontal positions on which sets of beads are inserted. The horizontal bars have 10 beads each, representing units, tens, hundreds, etc. An abacus is shown in Figure 1.1
2. **Napier's Bones** was a mechanical device built for the purpose of multiplication in 1617 AD. by an English mathematician John Napier.

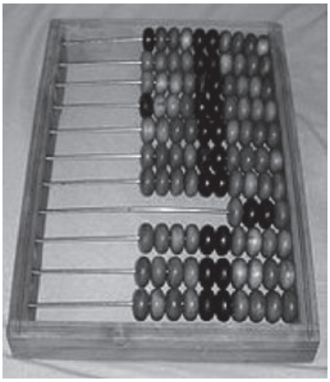


Figure 1.1 | Abacus

3. **Slide Rule** was developed by an English mathematician Edmund Gunter in the 16th century. Using the slide rule, one could perform operations like addition, subtraction, multiplication and division. It was used extensively till late 1970s. Figure 1.2 shows a slide rule.

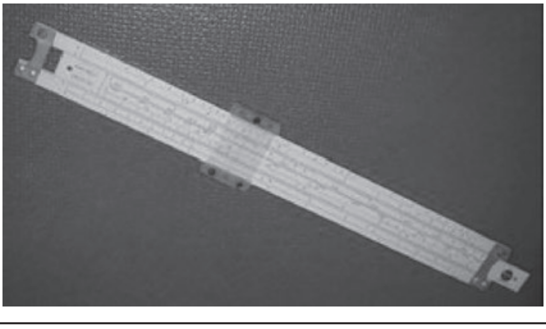


Figure 1.2 | Slide rule

4. **Pascal's Adding and Subtraction Machine** was developed by Blaise Pascal. It could add and subtract. The machine consisted of wheels, gears and cylinders.
5. **Leibniz's Multiplication and Dividing Machine** was a mechanical device that could both multiply and divide. The German philosopher and mathematician Gottfried Leibniz built it around 1673.
6. **Punch Card System** was developed by Jacquard to control the power loom in 1801. He invented the punched card reader that could recognize the presence of hole in the punched card as binary one and the absence of the hole as binary zero. The 0s and 1s are the basis of the modern digital computer. A punched card is shown in Figure 1.3.

Figure 1.3 | Punched card

7. **Babbage's Analytical Engine** An English man Charles Babbage built a mechanical machine to do complex mathematical calculations, in the year 1823. The machine was called as difference engine. Later, Charles Babbage and Lady Ada Lovelace developed a general-purpose calculating machine, the analytical engine. Charles Babbage is also called the father of computer.
8. **Hollerith's Punched Card Tabulating Machine** was invented by Herman Hollerith. The machine could read the information from a punched card and process it electronically.

The developments discussed above and several others not discussed here, resulted in the development of the first computer in the 1940s.

1.5 Generations of Computer

The computer has evolved from a large-sized simple calculating machine to a smaller but much more powerful machine. The evolution of computer to the current state is defined in terms of the generations of computer. Each generation of computer is designed based on a new technological development, resulting in better, cheaper and smaller computers that are more powerful, faster and efficient than their predecessors. Currently, there are five generations of computer. In the following subsections, we will discuss the generations of computer in terms of:

1. the technology used by them (hardware and software),
2. computing characteristics (speed, i.e., number of instructions executed per second),
3. physical appearance, and
4. their applications.

1.5.1 First Generation (1940 to 1956): Using Vacuum Tubes

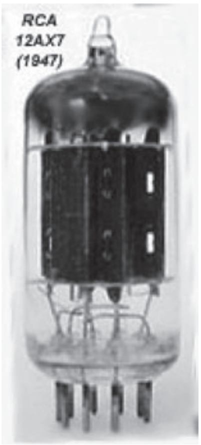


Figure 1.4 | Vacuum tube

1. **Hardware Technology:** The first generation of computers used vacuum tubes (Figure 1.4) for circuitry and magnetic drums for memory. The input to the computer was through punched cards and paper tapes. The output was displayed as printouts.
2. **Software Technology:** The instructions were written in machine language. Machine language uses 0s and 1s for coding of the instructions. The first generation computers could solve one problem at a time.
3. **Computing Characteristics:** The computation time was in milliseconds.
4. **Physical Appearance:** These computers were enormous in size and required a large room for installation.
5. **Application:** They were used for scientific applications as they were the fastest computing device of their time.
6. **Examples:** UNIVersal Automatic Computer (UNIVAC), Electronic Numerical Integrator And Calculator (ENIAC), and Electronic Discrete Variable Automatic Computer (EDVAC).

The first generation computers used a large number of vacuum tubes and thus generated a lot of heat. They consumed a great deal of electricity and were expensive to operate. The machines were prone to frequent malfunctioning and required constant maintenance. Since first generation computers used machine language, they were difficult to program.

1.5.2 Second Generation (1956 to 1963): Using Transistors

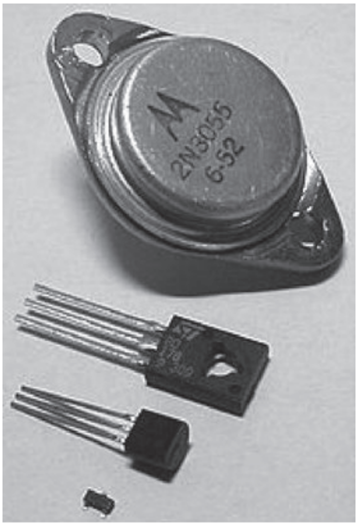


Figure 1.5 | Transistors

1. **Hardware Technology:** Transistors (Figure 1.5) replaced the vacuum tubes of the first generation of computers. Transistors allowed computers to become smaller, faster, cheaper, energy efficient and reliable. The second generation computers used *magnetic core technology* for primary memory. They used magnetic tapes and magnetic disks for secondary storage. The input was still through punched cards and the output using printouts. They used the concept of a stored program, where instructions were stored in the memory of computer.
2. **Software Technology:** The instructions were written using the *assembly language*. Assembly language uses mnemonics like ADD for addition and SUB for subtraction for coding of the instructions. It is easier to write instructions in assembly language, as compared to writing instructions in machine language. High-level programming languages, such as early versions of COBOL and FORTRAN were also developed during this period.

3. **Computing Characteristics:** The computation time was in microseconds.
4. **Physical Appearance:** Transistors are smaller in size compared to vacuum tubes, thus, the size of the computer was also reduced.
5. **Application:** The cost of commercial production of these computers was very high, though less than the first generation computers. The transistors had to be assembled manually in second generation computers.
6. **Examples:** PDP-8, IBM 1401 and CDC 1604.

Second generation computers generated a lot of heat but much less than the first generation computers. They required less maintenance than the first generation computers.

1.5.3 Third Generation (1964 to 1971): Using Integrated Circuits

1. **Hardware Technology:** The third generation computers used the *Integrated Circuit (IC)* chips. Figure 1.6 shows IC chips. In an IC chip, multiple transistors are placed on a silicon chip. Silicon is a type of semiconductor. The use of IC chip increased the speed and the efficiency of computer, manifold. The keyboard and monitor were used to interact with the third generation computer, instead of the punched card and printouts.
2. **Software Technology:** The keyboard and the monitor were interfaced through the *operating system*. Operating system allowed different applications to run at the same time. *High-level languages* were used extensively for programming, instead of machine language and assembly language.

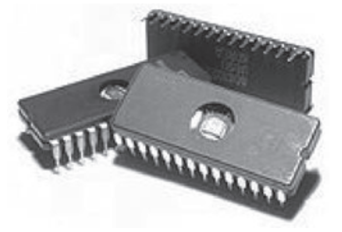


Figure 1.6 | IC chips

3. **Computing Characteristics:** The computation time was in nanoseconds.
4. **Physical Appearance:** The size of these computers was quite small compared to the second generation computers.
5. **Application:** Computers became accessible to mass audience. Computers were produced commercially, and were smaller and cheaper than their predecessors.
6. **Examples:** IBM 370, PDP 11.

The third generation computers used less power and generated less heat than the second generation computers. The cost of the computer reduced significantly, as individual components of the computer were not required to be assembled manually. The maintenance cost of the computers was also less compared to their predecessors.

1.5.4 Fourth Generation (1971 to present): Using Microprocessors

1. **Hardware Technology:** They use the *Large Scale Integration (LSI)* and the *Very Large Scale Integration (VLSI)* technology. Thousands of transistors are integrated on a small silicon chip using LSI technology. VLSI allows hundreds of thousands of components to be integrated in a small chip. This era is marked by the development of microprocessor. *Microprocessor* is a chip containing millions of transistors and components, and, designed using LSI and VLSI technology. A microprocessor chip is shown in Figure 1.7. This generation of computers gave rise to Personal Computer (PC). Semiconductor memory replaced the earlier magnetic core memory, resulting in fast random access to memory. Secondary storage device like magnetic disks became smaller in physical size and larger in capacity.

The *linking of computers* is another key development of this era. The computers were linked to form networks that led to the emergence of the Internet. This generation also saw the development of pointing devices like mouse, and handheld devices.

2. **Software Technology:** Several new operating systems like the MS-DOS and MS-Windows developed during this time. This generation of computers supported *Graphical User Interface* (*GUI*). GUI is a user-friendly interface that allows user to interact with the computer via menus and icons. High-level programming languages are used for the writing of programs.
3. **Computing Characteristics:** The computation time is in picoseconds.
4. **Physical Appearance:** They are smaller than the computers of the previous generation. Some can even fit into the palm of the hand.
5. **Application:** They became widely available for commercial purposes. Personal computers became available to the home user.
6. **Examples:** The Intel 4004 chip was the first microprocessor. The components of the computer like Central Processing Unit (CPU) and memory were located on a single chip. In 1981, IBM introduced the first computer for home use. In 1984, Apple introduced the Macintosh.

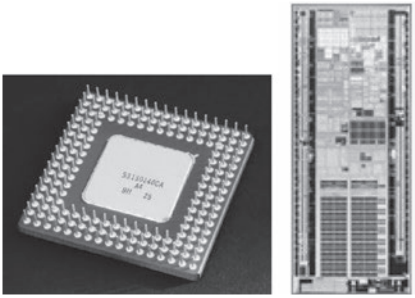


Figure 1.7 | Microprocessors

The microprocessor has resulted in the fourth generation computers being smaller and cheaper than their predecessors. The fourth generation computers are also portable and more reliable. They generate much lesser heat and require less maintenance compared to their predecessors. GUI and pointing devices facilitate easy use and learning on the computer. Networking has resulted in resource sharing and communication among different computers.

1.5.5 Fifth Generation (Present and Next): Using Artificial Intelligence

The goal of fifth generation computing is to develop computers that are capable of learning and self-organization. The fifth generation computers use *Super Large Scale Integrated (SLSI)* chips that are able to store millions of components on a single chip. These computers have large memory requirements.

This generation of computers uses *parallel processing* that allows several instructions to be executed in parallel, instead of serial execution. Parallel processing results in faster processing speed. The Intel dual-core microprocessor uses parallel processing.

The fifth generation computers are based on *Artificial Intelligence (AI)*. They try to simulate the human way of thinking and reasoning. Artificial Intelligence includes areas like Expert System (ES), Natural Language Processing (NLP), speech recognition, voice recognition, robotics, etc.

The various generations of the computer in terms of technology and other features is tabulated at the end of this chapter.

1.6 Classification of Computer

The digital computers that are available nowadays vary in their sizes and types. The computers are broadly classified into four categories (Figure 1.8) based on their size and type: (1) Microcomputers, (2) Minicomputers, (3) Mainframe computers, and (4) Supercomputer.

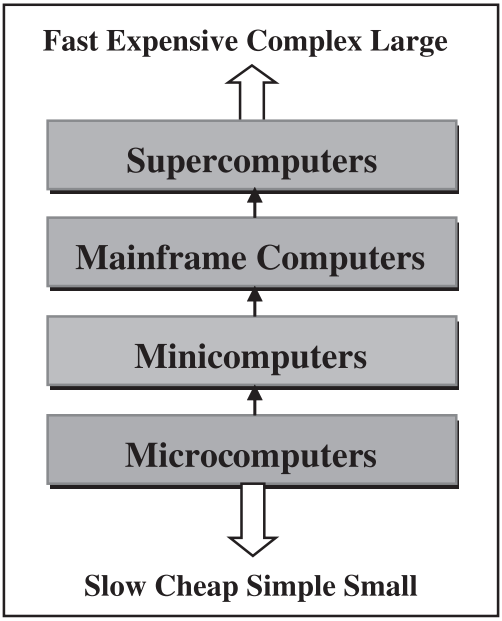
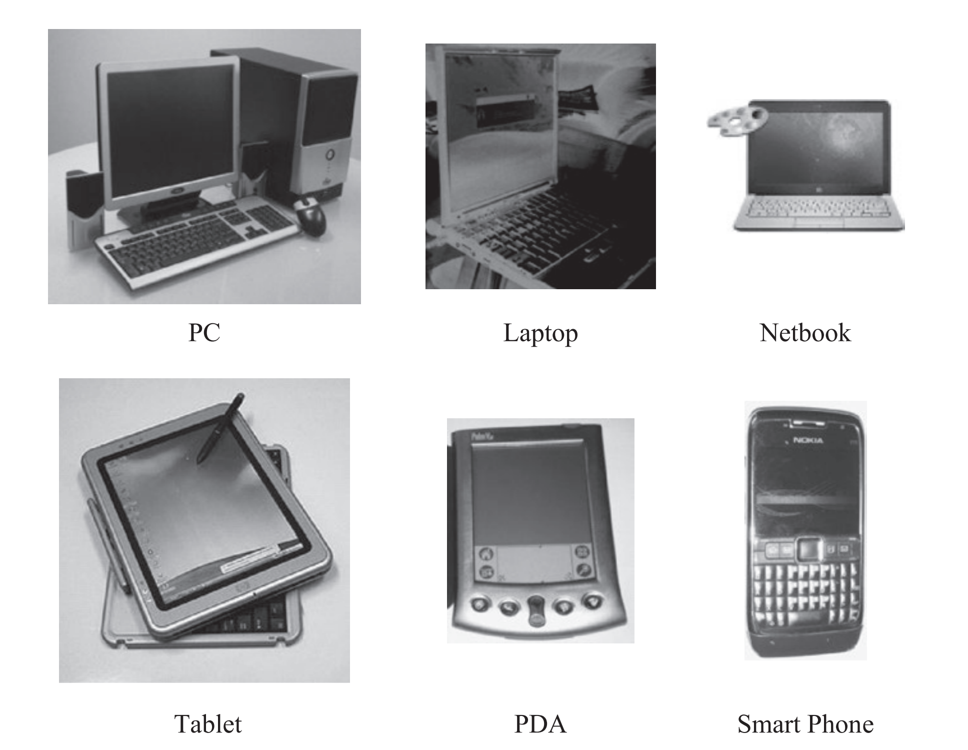


Figure 1.8 | Classification of computers based on size and type

1.6.1 Microcomputers

Microcomputers are small, low-cost and single-user digital computer. They consist of CPU, input unit, output unit, storage unit and the software. Although microcomputers are stand-alone machines, they can be connected together to create a network of computers that can serve more than one user. IBM PC based on Pentium microprocessor and Apple Macintosh are some examples of microcomputers. Microcomputers include desktop computers, notebook computers or laptop, tablet computer, handheld computer, smart phones and netbook, as shown in Figure 1.9.

1. **Desktop Computer or Personal Computer (PC)** is the most common type of microcomputer. It is a stand-alone machine that can be placed on the desk. Externally, it consists of three units—keyboard, monitor, and a system unit containing the CPU, memory, hard disk drive, etc. It is not very expensive and is suited to the needs of a single user at home, small business units, and organizations. Apple, Microsoft, HP, Dell and Lenovo are some of the PC manufacturers.
2. **Notebook Computers or Laptop** resemble a notebook. They are portable and have all the features of a desktop computer. The advantage of the laptop is that it is small in size (can be put inside a briefcase), can be carried anywhere, has a battery backup and has all the functionality of the desktop. Laptops can be placed on the lap while working (hence the name). Laptops are costlier than the desktop machines.
3. **Netbook** These are smaller notebooks optimized for low weight and low cost, and are designed for accessing web-based applications. Starting with the earliest netbook in



PC Laptop Netbook

Tablet PDA Smart Phone

Figure 1.9 | Microcomputers

late 2007, they have gained significant popularity now. Netbooks deliver the performance needed to enjoy popular activities like streaming videos or music, emailing, Web surfing or instant messaging. The word *netbook* was created as a blend of Inter*net* and note*book*.

4. **Tablet Computer** has features of the notebook computer but it can accept input from a stylus or a pen instead of the keyboard or mouse. It is a portable computer. Tablet computer are the new kind of PCs.
5. **Handheld Computer or Personal Digital Assistant (PDA)** is a small computer that can be held on the top of the palm. It is small in size. Instead of the keyboard, PDA uses a pen or a stylus for input. PDAs do not have a disk drive. They have a limited memory and are less powerful. PDAs can be connected to the Internet via a wireless connection. Casio and Apple are some of the manufacturers of PDA. Over the last few years, PDAs have merged into mobile phones to create smart phones.
6. **Smart Phones** are cellular phones that function both as a phone and as a small PC. They may use a stylus or a pen, or may have a small keyboard. They can be connected to the Internet wirelessly. They are used to access the electronic-mail, download music, play games, etc. Blackberry, Apple, HTC, Nokia and LG are some of the manufacturers of smart phones.

1.6.2 Minicomputers

Minicomputers (Figure 1.10) are digital computers, generally used in multi-user systems. They have high processing speed and high storage capacity than the microcomputers. Minicomputers can support 4–200 users simultaneously. The users can access the minicomputer through their PCs or terminal. They are used for real-time applications in industries, research centers, etc. PDP 11, IBM (8000 series) are some of the widely used minicomputers.

Figure 1.10 | Minicomputer

1.6.3 Mainframe Computers

Mainframe computers (Figure 1.11) are multi-user, multi-programming and high performance computers. They operate at a very high speed, have very large storage capacity and can handle the workload of many users. Mainframe computers are large and powerful systems generally used in centralized databases. The user accesses the mainframe computer via a terminal that may be a dumb terminal, an intelligent terminal or a PC. A *dumb terminal* cannot store data or do processing of its own. It has the input and output device only. An *intelligent terminal* has the input and output device, can do processing, but, cannot store data of its own. The dumb and the intelligent terminal use the processing power and the storage facility of the mainframe computer. Mainframe computers are used in organizations like banks or companies, where many people require frequent access to the same data. Some examples of mainframes are CDC 6600 and IBM ES000 series.

Figure 1.11 | Mainframe computer

1.6.4 Supercomputers

Supercomputers (Figure 1.12) are the fastest and the most expensive machines. They have high processing speed compared to other computers. The speed of a supercomputer is generally measured in FLOPS (FLoating point Operations Per Second). Some of the faster supercomputers can perform trillions of calculations per second. Supercomputers are built by interconnecting thousands of processors that can work in parallel.

Supercomputers are used for highly calculation-intensive tasks, such as, weather forecasting, climate research (global warming), molecular research, biological research, nuclear research and aircraft design. They are also used in major universities, military agencies and scientific research laboratories. Some examples of supercomputers are IBM Roadrunner, IBM Blue gene and Intel ASCI red. PARAM is a series of supercomputer assembled in India by C-DAC (Center for Development of Advanced Computing), in Pune. PARAM Padma is the latest machine in this series. The peak computing power of PARAM Padma is 1 Tera FLOP (TFLOP).

Figure 1.12 | Supercomputer

1.7 The Computer System

Computer is an electronic device that accepts data as input, processes the input data by performing mathematical and logical operations on it, and gives the desired output. The computer system consists of four parts: (1) Hardware, (2) Software, (3) Data, and (4) Users. The parts of computer system are shown in Figure 1.13.

1. **Hardware** consists of the mechanical parts that make up the computer as a machine. The hardware consists of physical devices of the computer. The devices are required for input, output, storage and processing of the data. Keyboard, monitor, hard disk drive, floppy disk drive, printer, processor and motherboard are some of the hardware devices.
2. **Software** is a set of instructions that tells the computer about the tasks to be performed and how these tasks are to be performed. *Program* is a set of instructions, written in a language understood by the computer, to perform a specific task. A set of programs and documents are collectively called software. The hardware of the computer system cannot perform any task on its own. The hardware needs to be instructed about the task

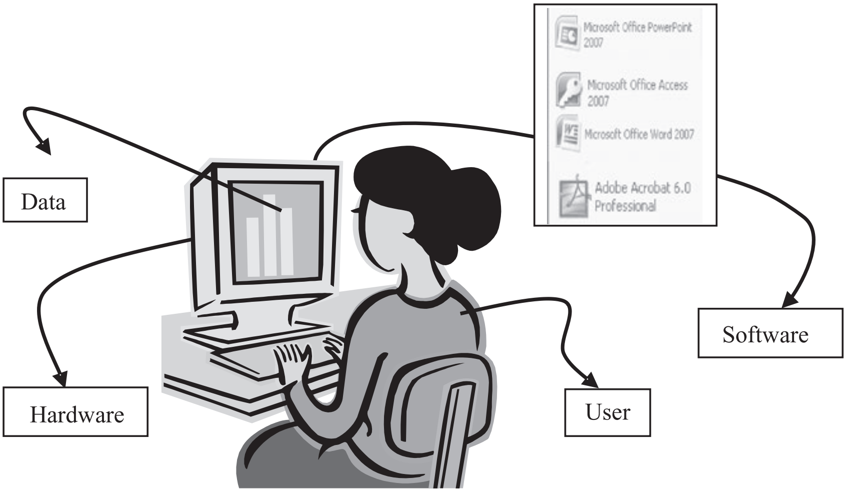

Microsoft Office PowerPoint 2007
Microsoft Office Access 2007
Microsoft Office Word 2007
Adobe Acrobat 6.0 Professional
Data
Software
Hardware
User

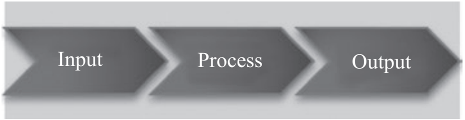
Figure 1.13 | Parts of computer system

to be performed. Software instructs the computer about the task to be performed. The hardware carries out these tasks. Different software can be loaded on the same hardware to perform different kinds of tasks.

3. **Data** are isolated values or raw facts, which by themselves have no much significance. For example, the data like *29, January*, and *1994* just represent values. The data is provided as input to the computer, which is processed to generate some meaningful information. For example, 29, January and 1994 are processed by the computer to give the date of birth of a person.
4. **Users** are people who write computer programs or interact with the computer. They are also known as *skinware, liveware, humanware or peopleware*. Programmers, data entry operators, system analyst and computer hardware engineers fall into this category.

1.7.1 The Input-Process-Output Concept

A computer is an electronic device that (1) accepts data, (2) processes data, (3) generates output, and (4) stores data. The concept of generating output information from the input data is also referred to as *input-process-output* concept.



The input-process-output concept of the computer is explained as follows:

1. **Input:** The computer accepts input data from the user via an input device like keyboard. The input data can be characters, word, text, sound, images, document, etc.
2. **Process:** The computer processes the input data. For this, it performs some actions on the data by using the instructions or program given by the user of the data. The action could be an arithmetic or logic calculation, editing, modifying a document, etc. During processing, the data, instructions and the output are stored temporarily in the computer's main memory.

3. **Output:** The output is the result generated after the processing of data. The output may be in the form of text, sound, image, document, etc. The computer may display the output on a monitor, send output to the printer for printing, play the output, etc.
4. **Storage:** The input data, instructions and output are stored permanently in the secondary storage devices like disk or tape. The stored data can be retrieved later, whenever needed.

1.7.2 Components of Computer Hardware

The computer system hardware comprises of three main components:

1. Input/Output (I/O) Unit,
2. Central Processing Unit (CPU), and
3. Memory Unit.

The I/O unit consists of the input unit and the output unit. CPU performs calculations and processing on the input data, to generate the output. The memory unit is used to store the data, the instructions and the output information. Figure 1.14 illustrates the typical interaction among the different components of the computer.

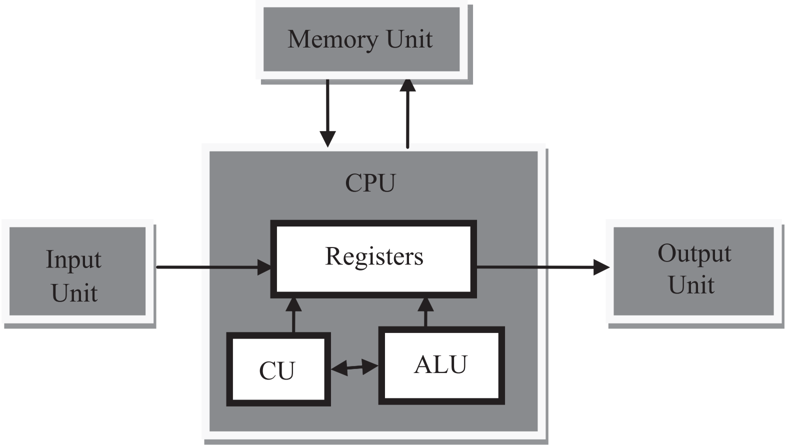


Figure 1.14 | The computer system interaction

1. **Input/Output Unit:** The user interacts with the computer via the I/O unit. The Input unit accepts data from the user and the Output unit provides the processed data i.e. the information to the user. The Input unit converts the data that it accepts from the user, into a form that is understandable by the computer. Similarly, the Output unit provides the output in a form that is understandable by the user. The input is provided to the computer using input devices like keyboard, trackball and mouse. Some of the commonly used output devices are monitor and printer.
2. **Central Processing Unit:** CPU controls, coordinates and supervises the operations of the computer. It is responsible for processing of the input data. CPU consists of Arithmetic Logic Unit (ALU) and Control Unit (CU).

a. ALU performs all the arithmetic and logic operations on the input data.
b. CU controls the overall operations of the computer i.e. it checks the sequence of execution of instructions, and, controls and coordinates the overall functioning of the units of computer.

Additionally, CPU also has a set of *registers* for temporary storage of data, instructions, addresses and intermediate results of calculation.

3. **Memory Unit:** Memory unit stores the data, instructions, intermediate results and output, *temporarily*, during the processing of data. This memory is also called the *main memory or primary memory* of the computer. The input data that is to be processed is brought into the main memory before processing. The instructions required for processing of data and any intermediate results are also stored in the main memory. The output is stored in memory before being transferred to the output device. CPU can work with the information stored in the main memory. Another kind of storage unit is also referred to as the *secondary memory* of the computer. The data, the programs and the output are stored *permanently* in the storage unit of the computer. Magnetic disks, optical disks and magnetic tapes are examples of secondary memory.

1.8 Central Processing Unit

Central Processing Unit (CPU) or the processor is also often called the *brain of computer*. CPU (Figure 1.15) consists of Arithmetic Logic Unit (ALU) and Control Unit (CU). In addition, CPU also has a set of registers which are temporary storage areas for holding data, and instructions. *ALU* performs the arithmetic and logic operations on the data that is made available to it. *CU* is responsible for organizing the processing of data and instructions. CU controls and coordinates the activity of the other units of computer. CPU uses the registers to store the data, instructions during processing.

CPU executes *the stored program instructions*, i.e. instructions and data are stored in memory before execution. For processing, CPU gets data and instructions from the memory. It interprets

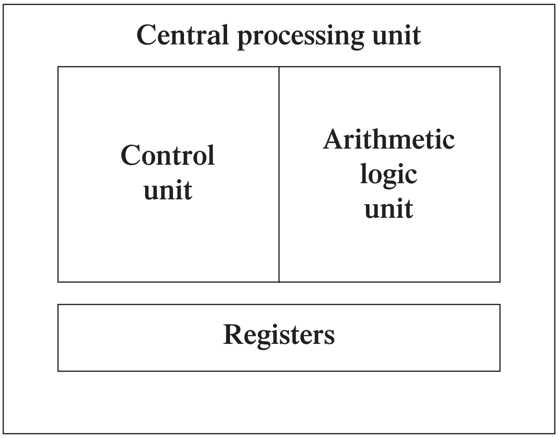


Figure 1.15 | CPU

the program instructions and performs the arithmetic and logic operations required for the processing of data. Then, it sends the processed data or result to the memory. CPU also acts as an administrator and is responsible for supervising operations of other parts of the computer.

The CPU is fabricated as a single Integrated Circuit (IC) chip, and is also known as the *microprocessor*. The microprocessor is plugged into the motherboard of the computer (*Motherboard* is a circuit board that has electronic circuit etched on it and connects the microprocessor with the other hardware components).

1.8.1 Arithmetic Logic Unit

1. ALU consists of two units—arithmetic unit and logic unit.
2. The arithmetic unit performs arithmetic operations on the data that is made available to it. Some of the arithmetic operations supported by the arithmetic unit are—addition, subtraction, multiplication and division.
3. The logic unit of ALU is responsible for performing logic operations. Logic unit performs comparisons of numbers, letters and special characters. Logic operations include testing for greater than, less than or equal to condition.
4. ALU performs arithmetic and logic operations, and uses *registers* to hold the data that is being processed.

1.8.2 Registers

1. Registers are high-speed storage areas within the CPU, but have the least storage capacity. Registers are not referenced by their address, but are directly accessed and manipulated by the CPU during instruction execution.
2. Registers store data, instructions, addresses and intermediate results of processing. Registers are often referred to as the CPU's *working memory*.
3. The data and instructions that require processing must be brought in the registers of CPU before they can be processed. For example, if two numbers are to be added, both numbers are brought in the registers, added and the result is also placed in a register.
4. Registers are used for different purposes, with each register serving a specific purpose. Some of the important registers in CPU (Figure 1.16) are as follows:

PC
IR
MAR
ACC
MBR
DR

Figure 1.16 | CPU registers

i. Accumulator (ACC) stores the result of arithmetic and logic operations.
ii. Instruction Register (IR) contains the current instruction most recently fetched.
iii. Program Counter (PC) contains the address of next instruction to be processed.
iv. Memory Address Register (MAR) contains the address of next location in the memory to be accessed.
v. Memory Buffer Register (MBR) temporarily stores data from memory or the data to be sent to memory.
vi. Data Register (DR) stores the operands and any other data.

5. The number of registers and the size of each (number of bits) register in a CPU helps to determine the power and the speed of a CPU.
6. The overall number of registers can vary from about ten to many hundreds, depending on the type and complexity of the processor.
7. The size of register, also called *word size*, indicates the amount of data with which the computer can work at any given time. The bigger the size, the more quickly it can process data. The size of a register may be 8, 16, 32 or 64 bits. For example, a 32-bit CPU is one in which each register is 32 bits wide and its CPU can manipulate 32 bits of data at a time. Nowadays, PCs have 32-bit or 64-bit registers.
10. 32-bit processor and 64-bit processor are the terms used to refer to the size of the registers. Other factors remaining the same, a 64-bit processor can process the data twice as fast as one with 32-bit processor.

1.8.3 Control Unit

1. The control unit of a computer does not do any actual processing of data. It organizes the processing of data and instructions. It acts as a supervisor and, controls and coordinates the activity of the other units of computer.
2. CU coordinates the input and output devices of a computer. It directs the computer to carry out stored program instructions by communicating with the ALU and the registers. CU uses the instructions in the Instruction Register (IR) to decide which circuit needs to be activated. It also instructs the ALU to perform the arithmetic or logic operations. When a program is run, the Program Counter (PC) register keeps track of the program instruction to be executed next.
3. CU tells when to fetch the data and instructions, what to do, where to store the results, the sequencing of events during processing etc.
4. CU also holds the CPU's Instruction Set, which is a list of all operations that the CPU can perform.

The function of a (CU) can be considered synonymous with that of a conductor of an orchestra. The conductor in an orchestra does not perform any work by itself but manages the orchestra and ensures that the members of orchestra work in proper coordination.

1.9 Memory Unit

The memory unit consists of cache memory and primary memory. *Primary memory or main memory* of the computer is used to store the data and instructions during execution of the instructions. Random Access Memory (RAM) and Read Only Memory (ROM) are the primary memory. In addition to the main memory, there is another kind of storage device known as the secondary memory. Secondary memory is non-volatile and is used for permanent storage of data and programs. A program or data that has to be executed is brought into the RAM from the secondary memory.

1.9.1 Cache Memory

1. The data and instructions that are required during the processing of data are brought from the secondary storage devices and stored in the RAM. For processing, it is required that the data and instructions are accessed from the RAM and stored in the registers. The time taken to move the data between RAM and CPU registers is large. This affects the speed of processing of computer, and results in decreasing the performance of CPU.
2. Cache memory is a very high speed memory placed in between RAM and CPU. Cache memory increases the speed of processing.
3. Cache memory is a storage buffer that stores the data that is used more often, temporarily, and makes them available to CPU at a fast rate. During processing, CPU first checks cache for the required data. If data is not found in cache, then it looks in the RAM for data.
4. To access the cache memory, CPU does not have to use the motherboard's system bus for data transfer. (The data transfer speed slows to the motherboard's capability, when data is passed through system bus. CPU can process data at a much faster rate by avoiding the system bus.)
5. Cache memory is built into the processor, and may also be located next to it on a separate chip between the CPU and RAM. Cache built into the CPU is faster than separate cache, running at the speed of the microprocessor itself. However, separate cache is roughly twice as fast as RAM.
6. The CPU has a built-in *Level 1 (L1)* cache and *Level 2 (L2)* cache, as shown in Figure 1.17. In addition to the built-in L1 and L2 cache, some CPUs have a separate cache chip on the motherboard. This cache on the motherboard is called *Level 3 (L3)* cache. Nowadays, high-end processor comes with built-in L3 cache, like in Intel core i7. The L1, L2 and L3 cache store the most recently run instructions, the next ones and the possible ones, respectively. Typically, CPUs have cache size varying from 256KB (L1), 6 MB (L2), to 12MB (L3) cache.
7. Cache memory is very expensive, so it is smaller in size. Generally, computers have cache memory of sizes 256 KB to 2 MB.

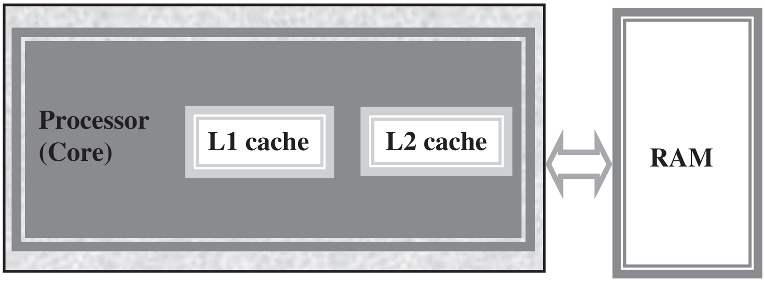


Figure 1.17 | Illustration of cache memory

1.9.2 Primary Memory

1. Primary memory is the main memory of computer. It is used to store data and instructions during the processing of data. Primary memory is semiconductor memory.
2. Primary memory is of two kinds—Random Access Memory (RAM) and Read Only Memory (ROM).
3. RAM is volatile. It stores data when the computer is on. The information stored in RAM gets erased when the computer is turned off. RAM provides *temporary storage* for data and instructions.
4. ROM is non-volatile memory, but is a read only memory. The storage in ROM is permanent in nature, and is used for storing standard processing programs that permanently reside in the computer. ROM comes programmed by the manufacturer.
5. RAM *stores data and instructions during the execution* of instructions. The data and instructions that require processing are brought into the RAM from the storage devices like hard disk. CPU accesses the data and the instructions from RAM, as it can access it at a *fast* speed than the storage devices connected to the input and output unit (Figure 1.18).
6. The input data that is entered using the input unit is stored in RAM, to be made available during the processing of data. Similarly, the output data generated after processing is stored in RAM before being sent to the output device. Any intermediate results generated during the processing of program are stored in RAM.
7. RAM provides a *limited storage capacity*, due to its *high cost*.

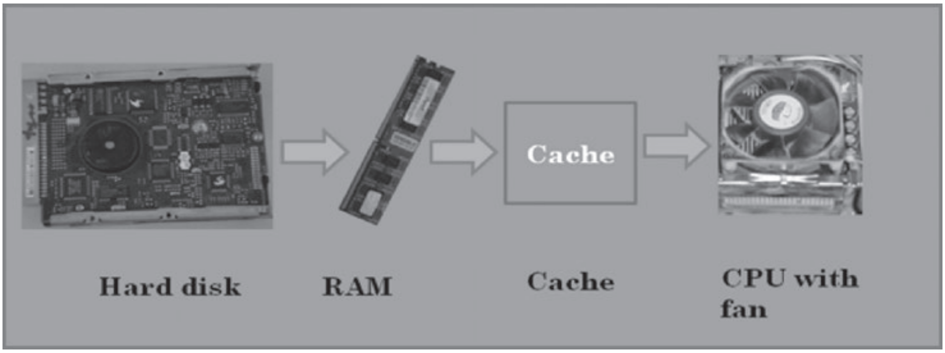


Figure 1.18 | Interaction of CPU with memory

1.9.3 Secondary Memory

1. The secondary memory stores data and instructions *permanently*. The information can be stored in secondary memory for a long time (years), and is generally permanent in nature unless erased by the user. It is a non-volatile memory.
2. It provides *back-up storage* for data and instructions. Hard disk drive, floppy drive and optical disk drives are some examples of storage devices.
3. The data and instructions that are currently not being used by CPU, but may be required later for processing, are stored in secondary memory.
4. Secondary memory has a *high storage capacity* than the primary memory.
5. Secondary memory is also *cheaper* than the primary memory.

6. It takes *longer time to access* the data and instructions stored in secondary memory than in primary memory.

Magnetic tape drives, disk drives and optical disk drives are the different types of storage devices.

1.10 Instruction Format

A computer program is a *set of instructions* that describe the steps to be performed for carrying out a computational task. The program and the data, on which the program operates, are stored in main memory, waiting to be processed by the processor. This is also called the *stored program concept*.

An instruction is designed to perform a task and is an elementary operation that the processor can accomplish. An instruction is divided into groups called fields. The common fields of an instruction are—Operation (op) code and Operand code (Figure 1.19). The remainder of the instruction fields differs from one computer type to other. The *operation code* represents action that the processor must execute. It tells the processor what basic operations to perform. The *operand code* defines the parameters of the action and depends on the operation. It specifies the locations of the data or the operand on which the operation is to be performed. It can be data or a memory address.

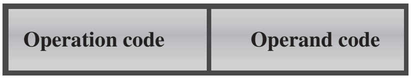


Figure 1.19 | Instruction format

The number of bits in an instruction varies according to the type of data (could be between 8 and 32 bits). Figure 1.20 shows the instruction format for ADD command.

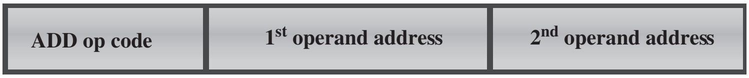


Figure 1.20 | Instruction format for ADD command

1.11 Instruction Set

A processor has a set of instructions that it understands, called as instruction set. An instruction set or an instruction set architecture is a part of the computer architecture. It relates to programming, instructions, registers, addressing modes, memory architecture, etc. An *Instruction Set* is the set of all the basic operations that a processor can accomplish. Examples of some instructions are shown in Figure 1.21. The instructions in the instruction set are the language that a processor understands. All programs have to communicate with the processor using these instructions. An instruction in the instruction set involves a series of logical operations (may be thousands) that are performed to complete each task. The instruction set is embedded in the processor (hardwired), which determines the machine language for the processor. All programs written in a high-level language are compiled and translated into machine code before execution, which is understood by the processor for which the program has been coded.

```
LOAD R1, A

ADD R1, B

STORE R1, X
```

Figure 1.21 | Examples of some instructions

Two processors are different if they have different instruction sets. A program run on one computer may not run on another computer having a different processor. *Two processors are compatible* if the same machine level program can run on both the processors. Therefore, the system software is developed within the processor's instruction set.

✍ **Microarchitecture** is the processor design technique used for implementing the *Instruction Set*. Computers having different microarchitecture can have a common Instruction Set. Pentium and Athlon CPU chips implement the x86 instruction set, but have different internal designs.

1.12 Instruction Cycle

The primary responsibility of a computer processor is to execute a sequential set of instructions that constitute a program. CPU executes each instruction in a series of steps, called *instruction cycle* (Figure 1.22).

1. A instruction cycle involves four steps (Figure 1.23):
   i. **Fetching:** The processor fetches the instruction from the memory. The fetched instruction is placed in the *Instruction Register*. *Program Counter* holds the address of next instruction to be fetched and is incremented after each fetch.

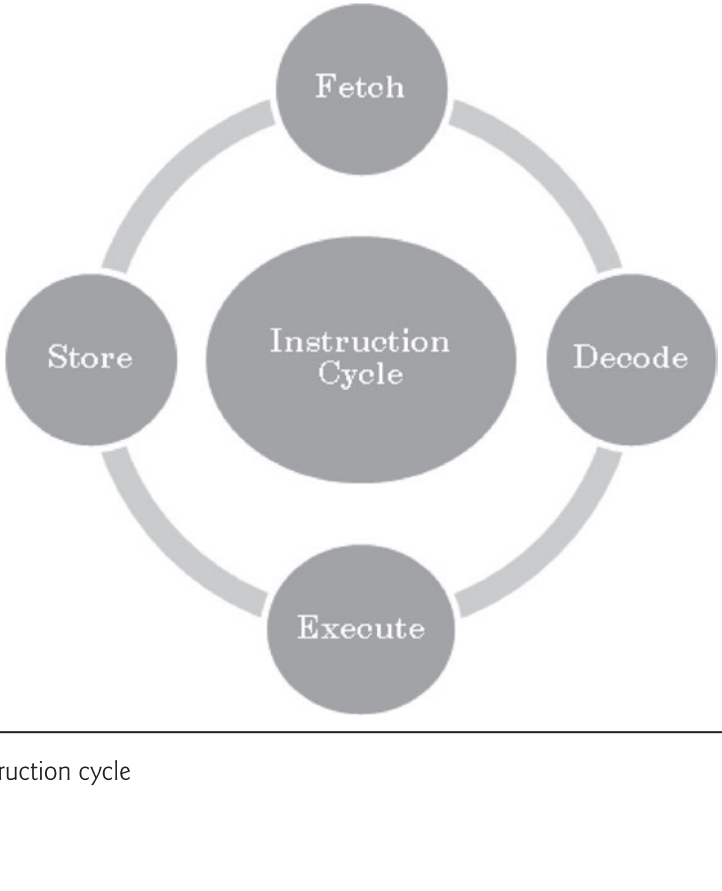


Figure 1.22 | Instruction cycle

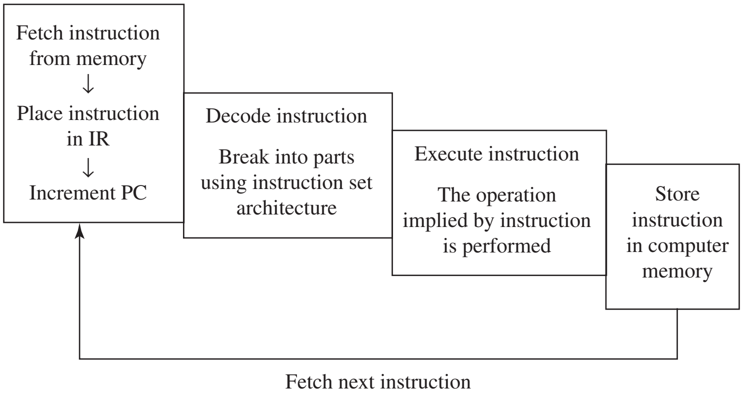

Fetch instruction from memory
Place instruction in IR
Increment PC
Decode instruction
Break into parts using instruction set architecture
Execute instruction
The operation implied by instruction is performed
Store instruction in computer memory
Fetch next instruction

Figure 1.23 | Steps in instruction cycle

ii. **Decoding:** The instruction that is fetched is broken down into parts or decoded. The instruction is translated into commands so that they correspond to those in the CPU's instruction set. The instruction set architecture of the CPU defines the way in which an instruction is decoded.
iii. **Executing:** The decoded instruction or the command is executed. CPU performs the operation implied by the program instruction. For example, if it is an ADD instruction, addition is performed.
iv. **Storing:** CPU writes back the results of execution, to the computer's memory.

2. Instructions are of different categories. Some categories of instructions are:

   i. Memory access or transfer of data between registers.
   ii. Arithmetic operations like addition and subtraction.
   iii. Logic operations such as AND, OR and NOT.
   iv. Control the sequence, conditional connections, etc.

A CPU performance is measured by the number of instructions it executes in a second, i.e., ***MIPS*** (million instructions per second), or ***BIPS*** (billion instructions per second).

1.13 Microprocessor

A processor's instruction set is a determining factor in its architecture. On the basis of the instruction set, microprocessors are classified as—Reduced Instruction Set Computer (RISC), and Complex Instruction Set Computer (CISC). The *x86* instruction set of the original Intel 8086 processor is of the CISC type. The PCs are based on the *x86* instruction set.

1. **CISC** architecture hardwires the processor with complex instructions, which are difficult to create otherwise using basic instructions. CISC combines the different instructions into one single CPU.

   i. CISC has a large instruction set that includes simple and fast instructions for performing basic tasks, as well as complex instructions that correspond to statements in the high level language.

ii. An increased number of instructions (200 to 300) results in a much more complex processor, requiring millions of transistors.

iii. Instructions are of variable lengths, using 8, 16 or 32 bits for storage. This results in the processor's time being spent in calculating where each instruction begins and ends.

iv. With large number of application software programs being written for the processor, a new processor has to be backwards compatible to the older version of processors.

v. AMD and Cyrix are based on CISC.

2. **RISC** has simple, single-cycle instructions, which performs only basic instructions. RISC architecture does not have hardwired advanced functions. All high-level language support is done in the software.

   i. RISC has fewer instructions and requires fewer transistors, which results in the reduced manufacturing cost of processor.

   ii. The instruction size is fixed (32 bits). The processor need not spend time in finding out where each instruction begins and ends.

   iii. RISC architecture has a reduced production cost compared to CISC processors.

   iv. The instructions, simple in nature, are executed in just one clock cycle, which speeds up the program execution when compared to CISC processors.

   v. RISC processors can handle multiple instructions simultaneously by processing them in parallel.

   vi. Apple Mac G3 and PowerPC are based on RISC.

Processors like Athlon XP and Pentium IV use a hybrid of both technologies.

Pipelining improves instruction execution speed by putting the execution steps into parallel. A CPU can receive a single instruction, begin executing it, and receive another instruction before it has completed the first. This allows for more instructions to be performed, about, one instruction per clock cycle.

Parallel Processing is the simultaneous execution of instructions from the same program on different processors. A program is divided into multiple processes that are handled in parallel in order to reduce execution time.

1.14 Interconnecting the Units of a Computer

CPU sends data, instructions and information to the components inside the computer as well as to the peripherals and devices attached to it. ***Bus*** is a set of electronic signal pathways that allows information and signals to travel between components inside or outside of a computer. The different components of computer, i.e., CPU, I/O unit, and memory unit are connected with each other by a bus. The data, instructions and the signals are carried between the different components via a bus. The features and functionality of a bus are as follows:

1. A bus is a set of wires used for interconnection, where each wire can carry one bit of data.
2. A bus width is defined by the number of wires in the bus.

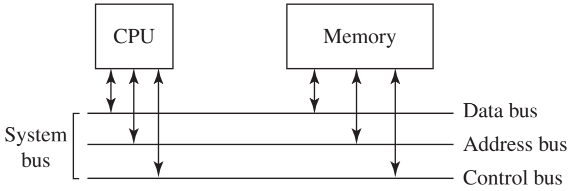


Figure 1.24 | Interaction between CPU and memory

3. A computer bus can be divided into two types—Internal Bus and External Bus.
4. The Internal Bus connects components inside the motherboard like, CPU and system memory. It is also called the *System Bus*. Figure 1.24 shows interaction between processor and memory.
5. The External Bus connects the different external devices, peripherals, expansion slots, I/O ports and drive connections to the rest of computer. The external bus allows various devices to be attached to the computer. It allows for the expansion of computer's capabilities. It is generally slower than the system bus. It is also referred to as the *Expansion Bus*.
6. A system bus or expansion bus comprise of three kinds of buses — data bus, address bus and control bus.
7. The interaction of CPU with memory and I/O devices involves all the three buses.
   i. The command to access the memory or the I/O device is carried by the ***control bus***.
   ii. The address of I/O device or memory is carried by the ***address bus.***
   iii. The data to be transferred is carried by the ***data bus***.

Figure 1.25 shows interaction between processor, memory and the peripheral devices.

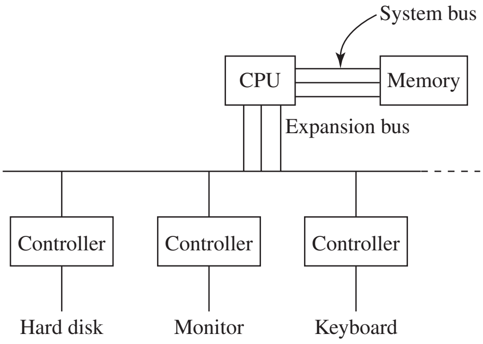


Figure 1.25 | Interaction between CPU, memory and peripheral devices

1.14.1 System Bus

The functions of data bus, address bus and control bus, in the system bus, are as follows:

1. **Data Bus** transfers data between the CPU and memory. The bus width of a data bus affects the *speed of computer*. The size of data bus defines the size of the processor. A processor can be 8, 16, 32 or 64-bit processor. An 8-bit processor has 8 wire data bus to carry 1 byte of data. In a 16-bit processor, 16-wire bus can carry 16 bits of data, i.e., transfer 2 bytes, etc.
2. **Address Bus** connects CPU and RAM with set of wires similar to data bus. The width of address bus determines the *maximum number of memory locations* the computer can address. Currently, Pentium Pro, II, III, IV have 36-bit address bus that can address 2^{36} bytes or 64 GB of memory.
3. **Control Bus** specifies whether data is to be read or written to the memory, etc.

1.14.2 Expansion Bus

The functions of data bus, address bus and control bus, in the expansion bus, are as follows:

1. The expansion bus connects external devices to the rest of computer. The external devices like monitor, keyboard and printer connect to ports on the back of computer. These ports are actually a part of the small circuit board or *expansion card* that fits into an *expansion slot* on the motherboard. Expansion slots are easy to recognize on the motherboard.
2. Expansion slots make up a row of long plastic connectors at the back of the computer with tiny copper 'finger slots' in a narrow channel that grab the connectors on the expansion cards. The slots are attached to tiny copper pathways on the motherboard (the expansion bus), which allows the device to communicate with the rest of computer.
3. **Data Bus** is used to transfer data between I/O devices and CPU. The exchange of data between CPU and I/O devices is according to the industry standard data buses. The most commonly used standard is Extended Industry Standard Architecture (EISA) which is a 32-bit bus architecture. Some of the common bus technologies are:
   - i. Peripheral Component Interconnect (PCI) bus for hard disks, sound cards, network cards and graphics cards,
   - ii. Accelerated Graphics Port (AGP) bus for 3-D and full motion video,
   - iii. Universal Serial Bus (USB) to connect and disconnect different devices.
4. **Address Bus** carries the addresses of different I/O devices to be accessed like the hard disk, CD ROM, etc.
5. **Control Bus** is used to carry read/write commands, status of I/O devices, etc.

1.14.3 External Ports

The peripheral devices interact with the CPU of the computer via the bus. The connections to the bus from the peripheral devices are made via the ports and sockets provided at the sides of the computer. The different ports and sockets facilitate the connection of different devices to the computer. Some of the standard port connections available on the outer sides of the computer are—port for mouse, keyboard, monitor, network, modem, and, audio port, serial port, parallel port and USB port. The different ports are physically identifiable by their different shapes, size of contact pins and number of pins. Figure 1.26 shows the interaction of serial and parallel port interfaces with the devices.

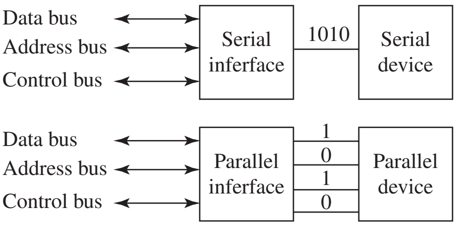


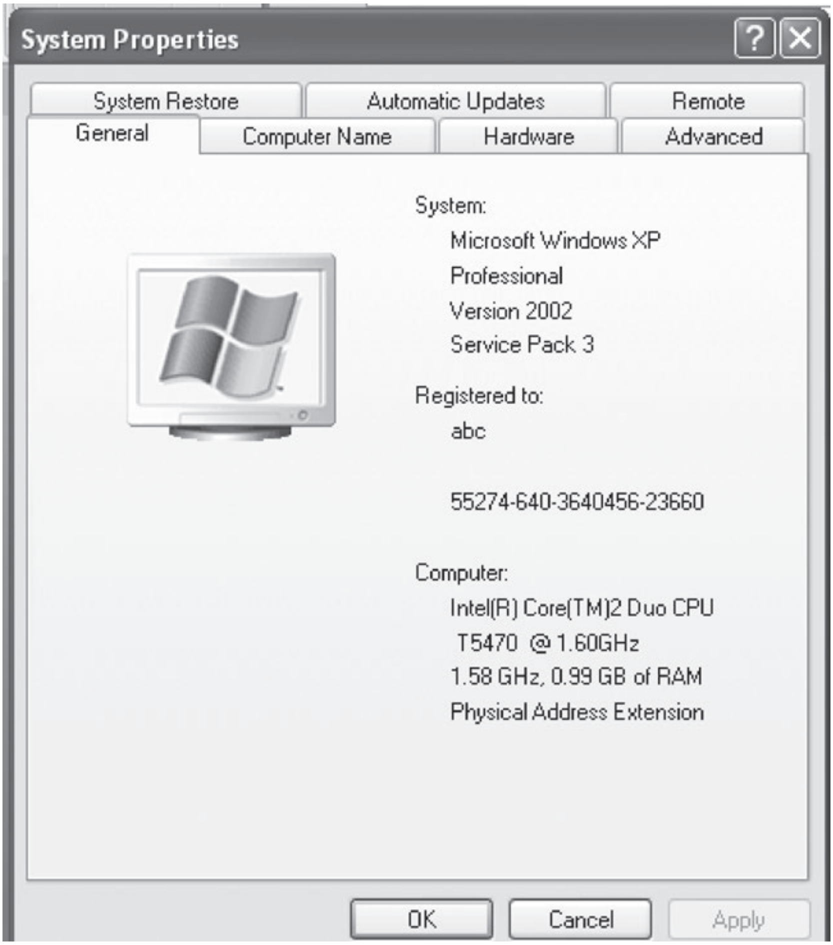
Figure 1.26 | Interaction of serial and parallel port interfaces

1.15 Performance of a Computer

There are a number of factors involved that are related to the CPU and have an effect on the overall speed and performance of the computer. Some of the factors that affect the performance of the computer include:

1. **Registers:** The size of the register (word size) indicates the amount of data with which the computer can work at any given time. The bigger the size, the more quickly it can process data. A 32-bit CPU is one in which each register is 32 bits wide.

2. **RAM:** It is used to store data and instructions during execution of the instructions. Anything you do on your computer requires RAM. When the computer is switched on, the operating system, device drivers, the active files and running programs are loaded into RAM. If RAM is less, then the CPU waits each time the new information is swapped into memory from the slower devices. Larger the RAM size, the better it is. PCs nowadays usually have 1 GB to 4 GB of RAM.
3. **System Clock:** The clock speed of a CPU is defined as the frequency with which a processor executes instructions or the data is processed. Higher clock frequencies mean more clock ticks per second. The computer's operating speed is linked to the speed of the system clock. The clock frequency is measured in millions of cycles per second or megahertz (MHz) or gigahertz (GHz) which is billions of cycles per second. A CPU's performance is measured by the number of instructions it executes in a second, i.e., *MIPS* or *BIPS*. PCs nowadays come with a clock speed of more than 1 GHz. *In Windows OS, you can select the System Properties dialog box to see the processor name and clock frequency.*



System:
Microsoft Windows XP
Professional
Version 2002
Service Pack 3

Registered to:
abc

55274-640-3640456-23660

Computer:
Intel(R) Core(TM)2 Duo CPU
T5470 @ 1.60GHz
1.58 GHz, 0.99 GB of RAM
Physical Address Extension

Figure 1.27 | System properties in Windows XP Professional

4. **Bus:** *Data bus* is used for transfering data between CPU and memory. The data bus width affects the speed of computer. In a 16-bit processor, 16-bit wire bus can carry 16 bits of data. The bus speed is measured in MHz. Higher the bus speed the better it is. *Address bus* connects CPU and RAM with a set of wires similar to data bus. The address bus width determines the *maximum number of memory locations* the computer can address. Pentium Pro, II, III, IV have 36-bit address bus that can address 2^{36} bytes or 64 GB of memory. PCs nowadays have a bus speed varying from 100 MHz to 400 MHz.
5. **Cache Memory:** Two of the main factors that affect a cache's performance are its size (amount of cache memory) and level L1, L2 and L3. Larger the size of cache, the better it is. PCs nowadays have a L1 cache of 256KB and L2 cache of 1MB.

Figure 1.27 shows the general information about a computer as displayed in the system properties window in Windows XP Professional.

1.16 Inside a Computer Cabinet

The computer cabinet encloses the components that are required for the running of the computer. The components inside a computer cabinet include the power supply, motherboard, memory chips, expansion slots, ports and interface, processor, cables and storage devices.

1.16.1 Motherboard

The computer is built up around a *motherboard*. The motherboard is the most important component in the PC. It is a large Printed Circuit Board (PCB), having many chips, connectors and other electronics mounted on it. The motherboard is the hub, which is used to connect all the essential components of a computer. The RAM, hard drive, disk drives and optical drives are all plugged into interfaces on the motherboard. The motherboard contains the processor, memory chips, interfaces and sockets, etc.

The motherboard may be characterized by the form factor, chipset and type of processor socket used. *Form factor* refers to the motherboard's geometry, dimensions, arrangement and electrical requirements. Different standards have been developed to build motherboards, which can be used in different brands of cases. Advanced Technology Extended (ATX) is the most common design of motherboard for desktop computers. *Chipset* is a circuit, which controls the majority of resources (including the bus interface with the processor, cache memory and RAM, expansion cards, etc.) Chipset's job is to coordinate data transfers between the various components of the computer (including the processor and memory). As the chipset is integrated into the motherboard, it is important to choose a motherboard, which includes a recent chipset, in order to maximize the computer's upgradeability. The *processor socket* may be a rectangular connector into which the processor is mounted vertically (slot), or a square-shaped connector with many small connectors into which the processor is directly inserted (socket). The Basic Input Output System (BIOS) and Complementary Metal-Oxide Semiconductor (CMOS) are present on the motherboard.

1. **BIOS:** It is the basic program used as an interface between the operating system and the motherboard. The BIOS (Figure 1.28) is stored in the ROM and cannot be rewritten. When the computer is switched on, it needs instructions to start. BIOS contain the instructions for the starting up of the computer. The BIOS runs when the computer is

switched on. It performs a Power On Self Test (POST) that checks that the hardware is functioning properly and the hardware devices are present. It checks whether the operating system is present on the hard drive. BIOS invokes the bootstrap loader to load the operating system into memory. BIOS can be configured using an interface named BIOS setup, which can be accessed when the computer is booting up (*by pressing the DEL key*).

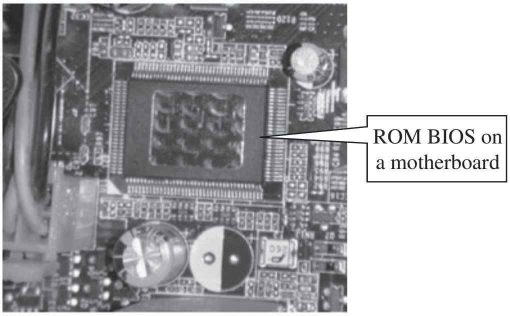


Figure 1.28 | ROM BIOS

2. **CMOS Chip:** BIOS ROMs are accompanied by a smaller CMOS (CMOS is a type of memory technology) memory chip. When the computer is turned off, the power supply stops providing electricity to the motherboard. When the computer is turned on again, the system still displays the correct clock time. This is because the CMOS chip saves some system information, such as time, system date and essential system settings. CMOS is kept powered by a button battery located on the motherboard (Figure 1.29). The CMOS chip is working even when the computer power is switched off. Information of the hardware installed in the computer (such as the number of tracks or sectors on each hard drive) is stored in the CMOS chip.

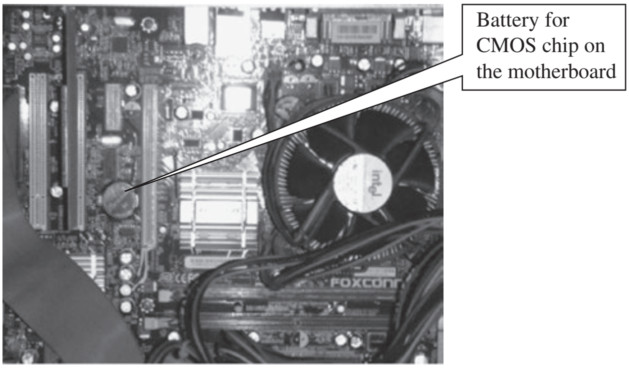


Figure 1.29 | Battery for CMOS chip

1.16.2 Ports and Interfaces

Motherboard has a certain number of I/O sockets that are connected to the ports and interfaces found on the rear side of a computer (Figure 1.30). You can connect external devices to the ports and interfaces, which get connected to the computer's motherboard.

1. Serial Port—to connect old peripherals.
2. Parallel Port—to connect old printers.
3. USB Ports—to connect newer peripherals like cameras, scanners and printers to the computer. It uses a thin wire to connect to the devices, and many devices can share that wire simultaneously.

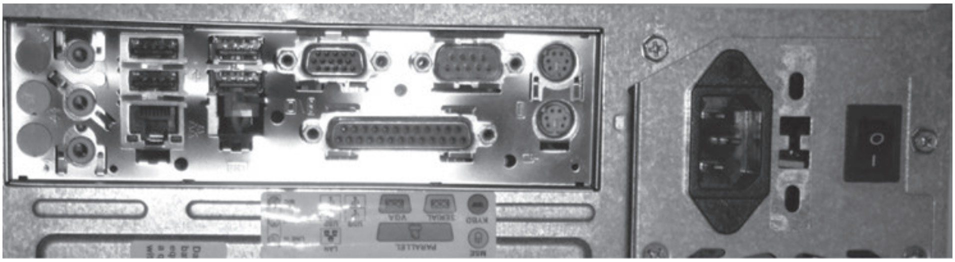


Figure 1.30 | Ports on the rear side of a PC

4. Firewire is another bus, used today mostly for video cameras and external hard drives.
5. RJ45 connector (called LAN or Ethernet port) is used to connect the computer to a network. It corresponds to a network card integrated into the motherboard.
6. VGA connector for connecting a monitor. This connector interfaces with the built-in graphics card.
7. Audio plugs (line-in, line-out and microphone), for connecting sound speakers and the microphone. This connector interfaces with the built-in sound card.
8. PS/2 port to connect mouse and keyboard into PC.
9. SCSI port for connecting the hard disk drives and network connectors.

1.16.3 Expansion Slots

The expansion slots (Figure 1.31) are located on the motherboard. The expansion cards are inserted in the expansion slots. These cards give the computer new features or increased performance. There are several types of slots:

1. ISA (Industry Standard Architecture) slot—To connect modem and input devices.
2. PCI (Peripheral Component InterConnect) slot—To connect audio, video and graphics. They are much faster than ISA cards.
3. AGP (Accelerated Graphic Port) slot—A fast port for a graphics card.

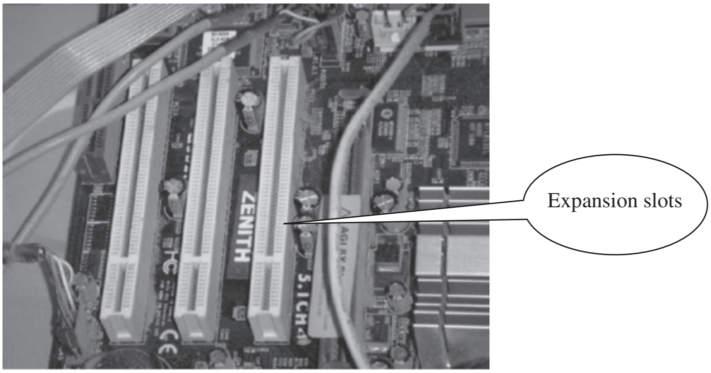


Figure 1.31 | Expansion slots

4. PCI (Peripheral Component InterConnect) Express slot—Faster bus architecture than AGP and PCI buses.
5. PC Card—It is used in laptop computers. It includes Wi-Fi card, network card and external modem.

1.16.4 Ribbon Cables

Ribbon cables (Figure 1.32) are flat, insulated and consist of several tiny wires moulded together that carry data to different components on the motherboard. There is a wire for each bit of the word or byte and additional wires to coordinate the activity of moving information. They also connect the floppy drives, disk drives and CD-ROM drives to the connectors in the motherboard. Nowadays, Serial Advanced Technology Attachment (SATA) cables have replaced the ribbon cables to connect the drives to the motherboard.

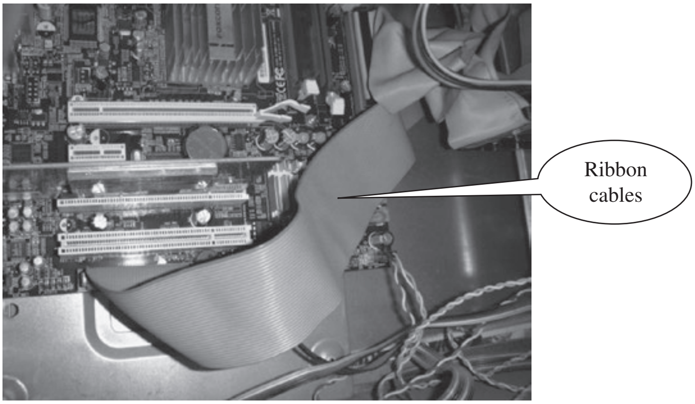


Figure 1.32 | Ribbon cables inside a PC

1.16.5 Memory Chips

The RAM consists of chips on a small circuit board (Figure 1.33). Two types of memory chips—Single In-line Memory Module (SIMM) and Dual In-line Memory Module (DIMM) are used in desktop computers. The CPU can retrieve information from DIMM chip at 64 bits compared to 32 bits or 16 bits transfer with SIMM chips. DIMM chips are used in Pentium 4 onwards to increase the access speed.

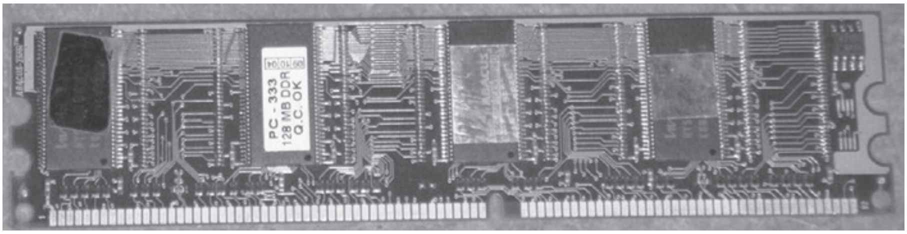


Figure 1.33 | RAM memory chip

1.16.6 Storage Devices

The disk drives are present inside the machine. The common disk drives in a machine are hard disk drive, floppy drive (Figure 1.34 (i & ii)) and CD drive or DVD drive. High-storage devices like hard disk, floppy disk and CDs (Figure 1.34 (iii) & (iv)) are inserted into the hard disk drive, floppy drive and CD drive, respectively. These storage devices can store large amounts of data, permanently.

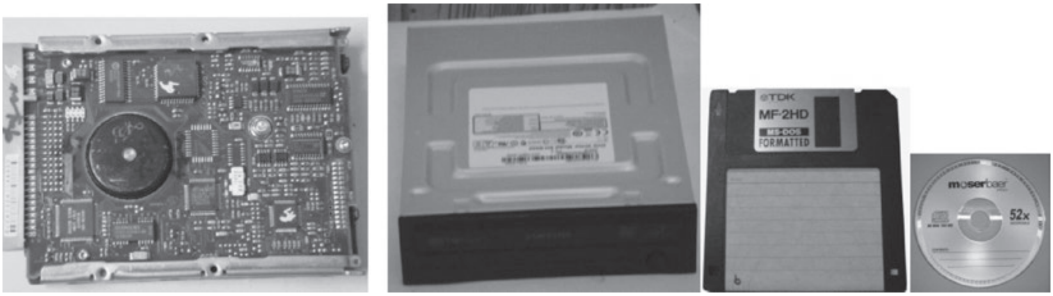


Figure 1.34 | Storage devices (i) Hard disk drive, (ii) DVD drive, (iii) Floppy disk, (iv) CD

1.16.7 Processor

The processor or the CPU is the main component of the computer. Select a processor based on factors like its speed, performance, reliability and motherboard support. Pentium Pro, Pentium 2 and Pentium 4 are some of the processors.

1.17 Application of Computers

Computers have proliferated into various areas of our lives. For a user, computer is a tool that provides the desired information, whenever needed. You may use computer to get information about the reservation of tickets (railways, airplanes and cinema halls), books in a library, medical history of a person, a place in a map, or the dictionary meaning of a word. The information may be presented to you in the form of text, images, video clips, etc.

Figure 1.35 shows some of the applications of computer. Some of the application areas of the computer are listed below:

1. **Education:** Computers are extensively used, as a tool and as an aid, for imparting education. Educators use computers to prepare notes and presentations of their lectures. Computers are used to develop computer-based training packages, to provide distance education using the e-learning software, and to conduct online examinations. Researchers use computers to get easy access to conference and journal details and to get global access to the research material.
2. **Entertainment:** Computers have had a major impact on the entertainment industry. The user can download and view movies, play games, chat, book tickets for cinema halls, use multimedia for making movies, incorporate visual and sound effects using computers, etc. The users can also listen to music, download and share music, create music using computers, etc.

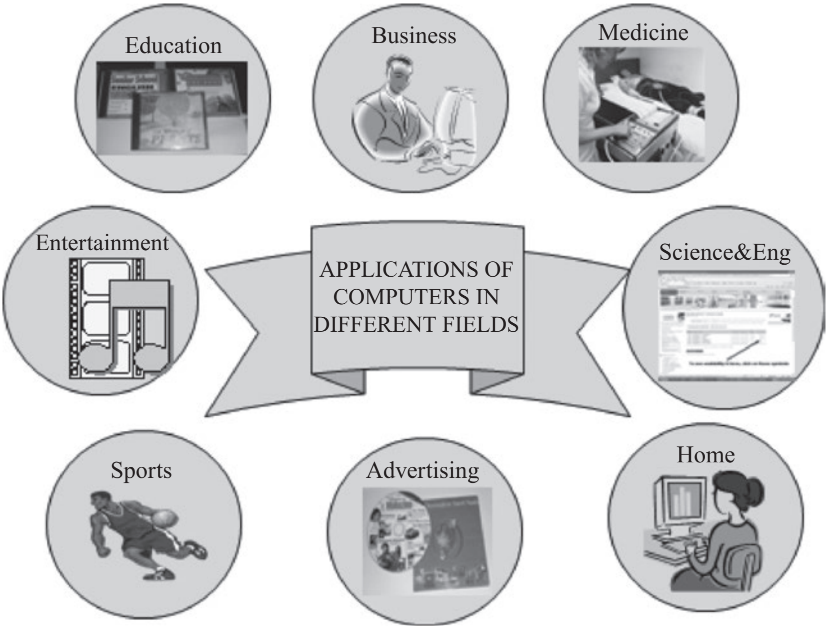


Figure 1.35 | Applications of computer

3. **Sports:** A computer can be used to watch a game, view the scores, improve the game, play games (like chess, etc.) and create games. They are also used for the purposes of training players.
4. **Advertising:** Computer is a powerful advertising media. Advertisement can be displayed on different websites, electronic-mails can be sent and reviews of a product by different customers can be posted. Computers are also used to create an advertisement using the visual and the sound effects. For the advertisers, computer is a medium via which the advertisements can be viewed globally. Web advertising has become a significant factor in the marketing plans of almost all companies. In fact, the business model of Google is mainly dependent on web advertising for generating revenues.
5. **Medicine:** Medical researchers and practitioners use computers to access information about the advances in medical research or to take opinion of doctors globally. The medical history of patients is stored in the computers. Computers are also an integral part of various kinds of sophisticated medical equipments like ultrasound machine, CAT scan machine, MRI scan machine, etc. Computers also provide assistance to the medical surgeons during critical surgery operations like laparoscopic operations, etc.
6. **Science and Engineering:** Scientists and engineers use computers for performing complex scientific calculations, for designing and making drawings (CAD/CAM applications) and also for simulating and testing the designs. Computers are used for storing the complex data, performing complex calculations and for visualizing 3-dimensional objects. Complex scientific applications like the launch of the rockets, space exploration, etc., are not possible without the computers.

7. **Government:** The government uses computers to manage its own operations and also for e-governance. The websites of the different government departments provide information to the users. Computers are used for the filing of income tax return, paying taxes, online submission of water and electricity bills, for the access of land record details, etc. The police department uses computers to search for criminals using fingerprint matching, etc.
8. **Home:** Computers have now become an integral part of home equipment. At home, people use computers to play games, to maintain the home accounts, for communicating with friends and relatives via Internet, for paying bills, for education and learning, etc. Microprocessors are embedded in house hold utilities like, washing machines, TVs, food processors, home theatres, security devices, etc.

The list of applications of computers is so long that it is not possible to discuss all of them here. In addition to the applications of the computers discussed above, computers have also proliferated into areas like banks, investments, stock trading, accounting, ticket reservation, military operations, meteorological predictions, social networking, business organizations, police department, video conferencing, telepresence, book publishing, web newspapers, and information sharing.

1.18 Summary

1. *Computer* is an electronic device which accepts data as input, performs processing on the data, and gives the desired output. A computer may be *analog or digital computer*.
2. Speed, accuracy, diligence, storage capability and versatility are the main *characteristics of computer.*
3. The *computing devices* have evolved from simple mechanical machines, like ABACUS, Napier's bones, Slide Rule, Pascal's Adding and Subtraction Machine, Leibniz's Multiplication and Dividing Machine, Jacquard Punched Card System, Babbage's Analytical Engine and Hollerith's Tabulating Machine, to the first electronic computer.
4. Charles Babbage is called the father of computer.
5. The evolution of computers to their present state is divided into *five generations of computers*, based on the hardware and software they use, their physical appearance and their computing characteristics.
6. *First generation computers* were vacuum tubes based machines. These were large in size, expensive to operate and instructions were written in machine language. Their computation time was in milliseconds.
7. *Second generation computers* were transistor based machines. They used the stored program concept. Programs were written in assembly language. They were smaller in size, less expensive and required less maintenance than the first generation computers. The computation time was in microseconds.
8. *Third generation computers* were characterized by the use of IC. They consumed less power and required low maintenance compared to their predecessors. High-level languages were used for programming. The computation time was in nanoseconds. These computers were produced commercially.
9. *Fourth generation computers* used microprocessors which were designed using the LSI and VLSI technology. The computers became small, portable, reliable and cheap. The

computation time is in picoseconds. They became available both to the home user and for commercial use.

10. *Fifth generation computers* are capable of learning and self organization. These computers use SLSI chips and have large memory requirements. They use parallel processing and are based on AI. The fifth generation computers are still being developed.
11. *Computers are broadly classified* as microcomputers, minicomputers, mainframe computers, and supercomputers, based on their sizes and types.
12. *Microcomputers* are small, low-cost stand-alone machines. Microcomputers include desktop computers, notebook computers or laptop, netbooks, tablet computer, hand-held computer and smart phones.
13. *Minicomputers* are high processing speed machines having more storage capacity than the microcomputers. Minicomputers can support 4-200 users simultaneously.
14. *Mainframe computers* are multi-user, multi-programming and high performance computers. They have very high speed, very large storage capacity and can handle large workloads. Mainframe computers are generally used in centralized databases.
15. *Supercomputers* are the most expensive machines, having high processing speed capable of performing trillions of calculations per second. The speed of a supercomputer is measured in FLOPS. Supercomputers find applications in computing-intensive tasks.
16. *Computer* is an electronic device based on the input-process-output concept. Input/ Output Unit, CPU and Memory unit are the three main components of computer.
17. *Input/Output Unit* consists of the Input unit which accepts data from the user and the Output unit that provides the processed data. CPU processes the input data, and, controls, coordinates and supervises the operations of the computer. CPU consists of ALU, CU and Registers. The memory unit stores programs, data and output, temporarily, during the processing. Additionally, storage unit or secondary memory is used for the storing of programs, data and output permanently.
18. Computers are used in various areas of our life. Education, entertainment, sports, advertising, medicine, science and engineering, government, office and home are some of the *application areas of computers*.
19. Different computers may have different organization, but the basic *organization of computer* remains the same.
20. I/O Unit, CPU and Memory Unit are the *main components of computer*.
21. *CPU or microprocessor* is called the brain of computer. It processes data and instructions. It also supervises the operations of the other parts of computer.
22. Registers, Arithmetic Logic Unit and Control Unit are the *parts of CPU*.
23. Cache memory, primary memory and secondary memory constitute the memory unit. Primary memory consists of RAM and ROM.
24. *Registers* are low-storage capacity, high-speed storage areas within the CPU. The data, instructions, addresses and intermediate results of processing are stored in the registers by the CPU.
25. *Cache memory* is a very high-speed memory placed in between RAM and CPU, to increase the processing speed. Cache memory is available in three levels - L1, L2 and L3.
26. *RAM* provides temporary storage, has a limited storage capacity and is volatile memory. The access speed of RAM is faster than access speed of the storage devices like hard disk. The data and the instructions stored in the hard disk are brought into the RAM so that the CPU can access the data and the instructions and process it.

27. *CU* organizes the processing of data and instructions. It acts as a supervisor and controls and coordinates the activity of other units of computer.
28. *ALU* performs arithmetic operations and logic operations on the data.
29. An *instruction* is an elementary operation that the processor can accomplish. The instructions in the *instruction set* are the language that a processor understands. The instruction set is embedded in the processor which determines the machine language for the processor.
30. A CPU *instruction cycle* involves four steps: (1) Fetching the instructions from the memory, (2) Decoding instructions so that they correspond to those in the CPU's instruction set, (3) Executing the decoded instructions, and (4) Storing the result to the computer memory.
31. *RISC and CISC* are the two kinds of microprocessors classified on the basis of instruction set. CISC has a large and complex instruction set. RISC has fewer instructions.
32. The different components of computer are connected with each other by a *bus*. A computer bus is of two types—system bus and expansion bus. A system bus or expansion bus comprise of three kinds of buses—data bus, address bus and control bus.
33. The *System Bus* connects the CPU, system memory, and all other components on the motherboard.
34. The *Expansion Bus* connects the different external devices, peripherals, expansion slots, I/O ports and drive connections to the rest of computer.
35. The *performance of computer* is affected by the size of registers, size of RAM, speed of system clock, width of bus, and size of cache memory.
36. Inside a *computer cabinet,* there is a motherboard, ports and interfaces, expansion slots, ribbon cables, RAM memory chips, high storage disk drives, and, processor.
37. The *motherboard* is characterized by the form factor, chipset and type of processor socket. Form factor is the motherboard's geometry, dimensions, arrangement and electrical requirements. Chipset controls the majority of resources of the computer.
38. *BIOS and CMOS* are present on the motherboard. BIOS is stored in ROM and is used as an interface between the operating system and the motherboard. The time, the system date, and essential system settings are saved in CMOS memory chip present on the motherboard. When the computer power is switched off, CMOS chip remains alive powered by a button battery.
39. *Ports and interfaces* are located on the sides of the computer case to which the external devices can be connected. Some of the ports and interfaces are— Serial port, Parallel port, USB port, Firewire, RJ45 connector, VGA connector, Audio plugs, PS/2 port, and SCSI port.

Exercise Questions

1. Define an analog computer and a digital computer.
2. Give an example each of analog computer and digital computer.
3. List the main characteristics of the computer.
4. Describe the characteristics of the computer.
5. List three significant limitations of the computer.
6. Explain briefly the developments in computer technology starting from a simple calculating machine to the first computer.

7. What is a calculating machine?
8. What is the key feature of the Jacquard's punch card?
9. Name the first calculating device for the counting of large numbers.
10. Who is called the Father of Computer?
11. The first generation computers used ____________ for circuitry.
12. Describe the first generation computer based on the
    a. Hardware
    b. Software
    c. Computing characteristics
    d. Physical appearance
    e. Their applications
13. Give two examples of first generation computers.
14. List the drawbacks of the first generation computers.
15. The second generation computers used ____________ for circuitry.
16. Describe the second generation computer based on the
    a. Hardware
    b. Software
    c. Computing characteristics
    d. Physical appearance
    e. Their applications
17. Give two examples of second generation computers.
18. List the drawbacks of the second generation computers.
19. The third generation computers used ____________ for circuitry.
20. Describe the third generation computer based on the
    a. Hardware
    b. Software
    c. Computing characteristics
    d. Physical appearance
    e. Their applications
21. Give two examples of third generation computers.
22. List the drawbacks of the third generation computers.
23. The fourth generation computers used ____________ for circuitry.
24. Describe the fourth generation computer based on the
    a. Hardware
    b. Software
    c. Computing characteristics
    d. Physical appearance
    e. Their applications
25. Give two examples of fourth generation computers.
26. List the drawbacks of the fourth generation computers.
27. The fifth generation computers used ____________ for circuitry.
28. Describe the fifth generation computer based on the
    a. Hardware
    b. Software
    c. Computing characteristics
    d. Physical appearance
    e. Their applications
29. Give two examples of fifth generation computers.
30. Compare in detail the five generations of computers based on the
    a. Hardware
    b. Software
    c. Computing characteristics
    d. Physical appearance
    e. Their applications
    Also give at least one example of each generation of computer.

31. Define microcomputer.
32. Give two examples of microcomputer.
33. List three categories of microcomputers.
34. Define minicomputers.
35. Give two examples of minicomputer.
36. Define mainframe computer.
37. Give two examples of mainframe computer.
38. Define a dumb terminal.
39. Define an intelligent terminal.
40. Define a supercomputer.
41. Give two examples of supercomputer.
42. The speed of supercomputer is generally measured in ____________.
43. List two uses of the supercomputer.
44. Name the supercomputer assembled in India.
45. Highlight the differences between microcomputer, minicomputer, mainframe computer and supercomputer.
46. Define a computer.
47. Define:
    a. Program
    b. Software
    c. Hardware
    d. ALU
    e. CU
    f. CPU
    g. Data
48. Differentiate between software, data and hardware.
49. List the components of computer hardware.
50. Explain in detail the components of computer hardware.
51. List the steps in the working of the computer.
52. Explain the working of the computer.
53. Explain the input-process-output cycle.
54. CPU is also often called the _________of computer.
55. Define a microprocessor.
56. Define a motherboard.
57. The different parts of the CPU are ______, ________ and _________.
58. _______ and ______ are the main memory.
59. What is the purpose of the main memory?
60. List the main functions of the CPU.
61. ALU consists of the ______ unit and ______ unit.
62. What are the functions of the ALU?
63. ______ is also called the working memory of the CPU.
64. List five important registers of the CPU. Also state the purpose of each register.
65. Why are Registers used in the CPU?
66. Define word size.
67. "This is a 64-bit processor". Explain its meaning.

68. The size of the register is also the ______ size.
69. Which is faster—a 32-bit processor or a 64-bitprocessor?
70. What are the functions of the control unit?
71. Explain the need of the cache memory?
72. The ______ memory is placed between the RAM and the CPU.
73. There are _______ levels of cache memory.
74. Explain the three levels of the cache memory.
75. State three important features of the cache memory.
76. The size of the cache memory is generally in the range ________.
77. What is the purpose of RAM?
78. List the features of the primary memory.
79. List the key features of the secondary memory.
80. Define the stored program concept.
81. Describe the format of an instruction.
82. The common fields of an instruction are ______ code and ______ code.
83. What is the function of the operand code and the operation code?
84. Define an Instruction set.
85. What is the significance of the Instruction set in the CPU?
86. "Two processors are compatible". How do you deduce this statement?
87. Define microarchitecture.
88. Define an instruction cycle.
89. Give a detailed working of the instruction cycle.
90. Name the four steps involved in an instruction cycle.
91. The number of instructions executed in a second by the CPU, is measured in _____.
92. The microprocessors are classified as _____ and _____ on the basis of the instruction set.
93. The x86 instruction set of the original Intel 8086 processor is of the _____ type.
94. Describe the features of the CISC architecture.
95. Give two examples of the CISC processor.
96. Describe the features of the RISC architecture.
97. Give two examples of the RISC processor.
98. What is the use of parallel processing and pipelining?
99. Define a bus.
100. Define a system bus.
101. Define an expansion bus.
102. Why is a bus used?
103. Define:
    a. Control bus
    b. Address bus
    c. Data bus
104. A system bus or expansion bus comprises of three kinds of buses ______, ______ and _____.
105. Name the bus connecting CPU with memory?
106. Name the bus connecting I/O devices with CPU?
107. In a system bus, what is the significance of the control bus, address bus and data bus?

108. The ______ of data bus affects the speed of computer.
109. Name the bus whose width affects the speed of computer?
110. The __________ of address bus determines the maximum number of memory locations the computer can address.
111. Name the bus whose width determines the maximum number of memory locations the computer can address?
112. What are the functions of data bus, address bus and control bus in the expansion bus?
113. Where is the expansion card fixed on the motherboard?
114. What is an expansion slot?
115. Name three common bus technologies.
116. What kind of devices is attached to the PCI bus, AGP bus and USB bus?
117. List the factors that affect the performance of the computer.
118. Explain in detail the factors that affect the performance of the computer.
119. What is the use of the system clock?
120. The clock frequency is measured in _____.
121. "The motherboard is characterized by the form factor, chipset and the type of processor socket used". Explain.
122. Define form factor.
123. Define chipset.
124. _____ is the most common design of the motherboard for desktop computers.
125. What is the significance of the chipset?
126. What is the function of the BIOS?
127. What is the function of the CMOS chip?
128. Explain the booting process when the computer is switched on.
129. What is POST?
130. List five ports and interfaces available on the backside of the computer to connect the devices.
131. What devices are attached to
    a. Serial Port
    b. Parallel Port
    c. USB Port
    d. Firewire
    e. RJ45 connector
    f. VGA connector
    g. Audio plugs (Line-In, Line-Out and microphone)
    h. PS/2 Port
    i. SCSI Port
132. List five expansion slots available in the computer.
133. What devices are attached to
    a. ISA slots
    b. PCI slot
    c. AGP slot
    d. PCI Express slot
    e. PC Card
134. What is the purpose of the Ribbon cables?

135. Two types of memory chips ______ and ______ are used in desktop computers.
136. List any three storage devices that are attached to the computer.
137. List some areas where the computers are used.
138. Explain briefly the use of computers in the following areas
   a. Education
   b. Advertising
   c. Government

Additional Questions

139. Give full form of the following abbreviations
   a. CPU
   b. I/O
   c. ALU
   d. CU
   e. LSI
   f. VLSI
   g. PC
   h. GUI
   i. SLSI
   j. ES
   k. NLP
   l. AI
   m. PDA
   n. FLOPS
   o. UNIVAC
   p. ENIAC
   q. EDVAC
140. Write short notes on:
   a. Components of Computer
   b. Input-Process-Output
   c. I/O Unit
   d. Central Processing Unit
   e. Storage Unit
   f. History of Computers
   g. First Generation Computer
   h. Second Generation Computer
   i. Third Generation Computer
   j. Fourth Generation Computer
   k. Fifth Generation Computer
   l. Microcomputers
   m. Minicomputers
   n. Mainframe Computers
   o. Supercomputer
   p. Personal Computer (PC)
   q. Notebook Computer
   r. Tablet Computer
   s. Netbook
   t. Personal Digital Assistant (PDA)
   u. Applications of Computer
141. Give differences between the following:
   a. Analog and Digital Computer
   b. Dumb Terminal and Intelligent Terminal
   c. Microcomputer and Minicomputer
   d. Minicomputer and Mainframe Computer
   e. Mainframe computer and Supercomputer
   f. First Generation Computers and Second Generation Computers
   g. Second Generation Computers and Third Generation Computers
   h. Third Generation Computers and Fourth Generation Computers
   i. Fourth Generation Computers and Fifth Generation Computers
   j. Desktop Computer and Notebook Computer

142. Give full form of the following abbreviations:

a. IC
b. MIPS
c. EISA
d. PCI
e. USB
f. AGP
g. BIPS
h. SIMM
i. DIMM
j. GHz
k. MHz
l. PCB
m. BIOS
n. CMOS
o. POST
p. ISA
q. ROM
r. ACC
s. IR
t. PC
u. MAR
v. MBR
w. DR
x. RISC
y. CISC
z. ATX
aa. SATA

143. Write short notes on:

a. Working of computer
b. Central processing unit
c. Registers
d. Cache memory
e. RAM
f. Control unit
g. ALU
h. Instruction format
i. Instruction set
j. Instruction Cycle
k. Microprocessor
l. System bus
m. Expansion bus
n. Performance of computer
o. System clock
p. Motherboard
q. BIOS
r. CMOS chip
s. Ports and interfaces in computer
t. Main components in a computer case
u. Expansion slots

144. Give differences between the following:

a. Registers and cache memory
b. Cache memory and RAM
c. RISC and CISC
d. System bus and expansion bus
e. Data bus, address bus and control bus

Generations of Computer

Features	First Generation	Second Generation	Third Generation	Fourth Generation	Fifth Generation
Year	1940 to 1956	1956 to 1964	1964 to 1971	1971 to present	Present and Next
Hardware Technology	Vacuum tubes for circuitry and magnetic drums for memory.	Transistors that are smaller, faster, cheaper, energy efficient and reliable.	*Integrated Circuit (IC)* chips. In an IC chip, multiple transistors are placed on a silicon chip. Silicon is a type of semiconductor.	*Large Scale Integration (LSI)* and the *Very Large Scale Integration (VLSI)* technologies.	*Super Large Scale Integrated (SLSI)* chips are able to store millions of components on a single chip. These computers have large memory requirements.
Input	Punched cards and paper tapes	*Magnetic core technology* for primary memory. They used magnetic tapes and magnetic disks for secondary storage. The input was still through punched cards.	Keyboard	Keyboard, mouse and other handheld devices	Keyboard, mouse and other handheld devices
Output	Displayed as printouts	Output using printouts	Monitor	Monitor	Monitor and other output devices
Software Technology	Instructions were written in machine language, i.e., 0s and 1s.	Assembly language programming, COBOL and FORTRAN	Operating system and high level languages	MS-Dos, Windows, and GUI	Artificial intelligence includes areas like Expert System (ES).
Processing speed	The computation time was in Milliseconds.	Microseconds	Nanoseconds	Picoseconds	Very fast

(*Continued*)

Features	First Generation	Second Generation	Third Generation	Fourth Generation	Fifth Generation
Application	They were used for scientific applications, as they were the fastest computing device of their time.	The transistors had to be assembled manually in second generation computers.	Computers became accessible to mass audience. Computers were produced commercially.	Personal computers and networking	Large parallel processing
Physical Appearance	These computers were enormous in size and required a large room for installation.	Smaller in size	Smaller and cheaper than their predecessors.	They are smaller and can even fit into the palm of the hand.	Smaller
Examples	UNIVersal Automatic Computer (UNIVAC), Electronic Numerical Integrator And Calculator (ENIAC) and Electronic Discrete Variable Automatic Computer (EDVAC)	PDP-8, IBM 1401 and CDC 1604	IBM 370 and PDP 11	Intel 4004 chip was the first microprocessor. In 1981, IBM home computer, and in 1984, Apple Macintosh	Natural Language Processing (NLP), speech recognition, voice recognition, robotics, etc.
Advantage/ disadvantage	Used a large number of vacuum tubes and thus generated a lot of heat. More electricity expensive. The machines were prone to frequent malfunctioning.	Second generation computers generated a lot of heat but much less than the first generation computers. They required less maintenance than the first generation computers.	Smaller and faster	Thousands of transistors are integrated on a small silicon chip using LSI technology. VLSI allows hundreds of thousands of components to be integrated in a small chip.	They are based on *Artificial Intelligence (AI)*. They try to simulate the human way of thinking and reasoning.

2

DATA REPRESENTATION AND PROGRAMMING FUNDAMENTALS

Learning Objectives

In this chapter, you will learn about:

- Data representation
- Number system
- Conversion from decimal to binary, octal, hexadecimal
- Conversion of binary, octal, hexadecimal to decimal
- Conversion of binary to octal, hexadecimal
- Conversion of octal, hexadecimal to binary
- Binary arithmetic
- Signed and unsigned numbers
- Binary data representation
- Binary coding schemes
- Logic gates
- Programming fundamentals
- Program development life cycle
- Algorithm
- Control structures
- Flowchart
- Pseudo code
- Programming paradigms
- Problem Formulation and Problem Solving

2.1 Data Representation

The data stored in the computer may be of different kinds, as follows:

1. Numeric data (0, 1, 2, …, 9)
2. Alphabetic data (A, B, C, …, Z)
3. Alphanumeric data—Combination of any of the symbols—(A, B, C… Z), (0, 1… 9), or special characters (+,–, Blank), etc.

All kinds of data, be it alphabets, numbers, symbols, sound data or video data, is represented in terms of 0s and 1s, in the computer. Each symbol is represented as a unique combination of 0s and 1s.

This chapter discusses the number systems that are commonly used in the computer. The number systems discussed in this chapter are—(1) Decimal number system, (2) Binary number system, (3) Octal number system, and (4) Hexadecimal number system. The number conversions described in this chapter are:

1. Decimal (Integer, Fraction, Integer.Fraction) to Binary, Octal, Hexadecimal
2. Binary, Octal, Hexadecimal (Integer, Fraction, Integer.Fraction) to Decimal
3. Binary to Octal, Hexadecimal
4. Octal, Hexadecimal to Binary

The chapter also discusses the binary arithmetic operations and the representation of signed and unsigned numbers in the computer. The representation of numbers using binary coding schemes and the logic gates used for the manipulation of data are also discussed.

2.2 Number System

A number system in *base r or radix r* uses unique symbols for *r* digits. One or more digits are combined to get a number. The *base* of the number decides the valid digits that are used to make a number. In a number, the *position* of digit starts from the right-hand side of the number. The rightmost digit has position 0, the next digit on its left has position 1, and so on. The digits of a number have two kinds of values:

1. Face value, and
2. Position value.

The **face value** of a digit is the digit located at that position. For example, in decimal number 52, face value at position 0 is 2 and face value at position 1 is 5.

The **position value** of a digit is ($\text{base}^{\text{position}}$). For example, in decimal number 52, the position value of digit 2 is 10^0 and the position value of digit 5 is 10^1. Decimal numbers have a base of 10.

The **number** is calculated as the sum of, face value * $\text{base}^{\text{position}}$, of each of the digits. For decimal number 52, the number is $5*10^1 + 2*10^0 = 50 + 2 = 52$

In computers, we are concerned with four kinds of number systems, as follows:

1. Decimal Number System —Base 10
2. Binary Number System —Base 2
3. Octal Number System —Base 8
4. Hexadecimal Number System—Base 16

The numbers given as input to computer and the numbers given as output from the computer, are generally in decimal number system, and are most easily understood by humans. However, computer understands the binary number system, i.e., numbers in terms of 0s and 1s. The binary data is also represented, internally, as octal numbers and hexadecimal numbers due to their ease of use.

A number in a particular base is written as $(\text{number})_{\text{base of number.}}$ For example, $(23)_{10}$ means that the number 23 is a decimal number, and $(345)_8$ shows that 345 is an octal number.

2.2.1 Decimal Number System

1. It consists of 10 digits—0, 1, 2, 3, 4, 5, 6, 7, 8 and 9.
2. All numbers in this number system are represented as combination of digits 0–9. For example, 34, 5965 and 867321.
3. The position value and quantity of a digit at different positions in a number are as follows:

Position:	3	2	1	0	.	–1	–2	–3
Position Value:	10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}
Quantity:	1000	100	10	1		1/10	1/100	1/1000

2.2.2 Binary Number System

1. The binary number system consists of two digits—0 and 1.
2. All binary numbers are formed using combination of 0 and 1. For example, 1001, 11000011 and 10110101.
3. The position value and quantity of a digit at different positions in a number are as follows:

Position:	3	2	1	0	.	–1	–2	–3
Position Value:	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}
Quantity:	8	4	2	1		1/2	1/4	1/8

2.2.3 Octal Number System

1. The octal number system consists of eight digits—0 to 7.
2. All octal numbers are represented using these eight digits. For example, 273, 103, 2375, etc.
3. The position value and quantity of a digit at different positions in a number are as follows:

Position:	3	2	1	0	.	–1	–2	–3
Position Value:	8^3	8^2	8^1	8^0		8^{-1}	8^{-2}	8^{-3}
Quantity:	512	64	8	1		1/8	1/64	1/512

2.2.4 Hexadecimal Number System

1. The hexadecimal number system consists of sixteen digits—0 to 9, A, B, C, D, E, F, where (A is for 10, B is for 11, C-12, D-13, E-14, F-15).
2. All hexadecimal numbers are represented using these 16 digits. For example, 3FA, 87B, 113, etc.

3. The position value and quantity of a digit at different positions in a number are as follows:

Position:	3	2	1	0	.	–1	–2	–3
Position Value:	16^3	16^2	16^1	16^0		16^{-1}	16^{-2}	16^{-3}
Quantity:	4096	256	16	1		1/16	1/256	1/4096

Table 2.1 summarizes the base, digits and largest digit for the above discussed number systems. Table 2.2 shows the binary, octal and hexadecimal equivalents of the decimal numbers 0–16.

Table 2.1 | Summary of number system

	Base	Digits	Largest Digit
Decimal	10	0–9	9
Binary	2	0,1	1
Octal	8	0–7	7
Hexadecimal	16	0–9, A, B, C, D, E, F	F (15)

Table 2.2 | Decimal, binary, octal and hexadecimal equivalents

Decimal	Binary	Octal	Hexadecimal
0	0000	000	0
1	0001	001	1
2	0010	002	2
3	0011	003	3
4	0100	004	4
5	0101	005	5
6	0110	006	6
7	0111	007	7
8	1000	010	8
9	1001	011	9
10	1010	012	A
11	1011	013	B
12	1100	014	C
13	1101	015	D
14	1110	016	E
15	1111	017	F
16	10000	020	10

2.3 Conversion from Decimal to Binary, Octal, Hexadecimal

A decimal number has two parts—integer part and fraction part. For example, in the decimal number 23.0786, 23 is the integer part and .0786 is the fraction part. The method used for the conversion of the integer part of a decimal number is different from the one used for the fraction part. In the following subsections, we shall discuss the conversion of decimal integer, decimal fraction and decimal integer.fraction number into binary, octal and hexadecimal number.

2.3.1 Converting Decimal *Integer* to Binary, Octal, Hexadecimal

A decimal integer is converted to any other base, by using the division operation.
To convert a decimal integer to:

1. binary-divide by 2,
2. octal-divide by 8, and,
3. hexadecimal-divide by 16.

Let us now understand this conversion with the help of some examples.

1. Make a table as shown below. Write the number in centre and *toBase* on the left side.

to Base	Number (Quotient)	Remainder
2	25	

2. Divide the number with *toBase*. After each division, write the remainder on right-side column and quotient in the next line in the middle column. Continue dividing till the quotient is 0.

to Base	Number (Quotient)	Remainder
2	25	
2	12	1
2	6	0
2	3	0
2	1	1
	0	1

Example 2.1 | Convert 25 from Base 10 to Base 2.

3. Write the digits in *remainder column* starting from *downwards to upwards*,

to Base	Number (Quotient)	Remainder
2	25	
2	12	1
2	6	0
2	3	0
2	1	1
	0	1

The binary equivalent of number $(25)_{10}$ is $(11001)_2$

The steps shown above are followed to convert a decimal integer to a number in any other base.

Example 2.1 | Continued

to Base	Number (Quotient)	Remainder
2	23	
2	11	1
2	5	1
2	2	1
2	1	0
	0	1

The binary equivalent of $(23)_{10}$ is $(10111)_2$

to Base	Number (Quotient)	Remainder
8	23	
8	2	7
	0	2

The octal equivalent of $(23)_{10}$ is $(27)_8$

to Base	Number (Quotient)	Remainder
16	23	
16	1	7
	0	1

The hexadecimal equivalent of $(23)_{10}$ is $(17)_{16}$

Example 2.2 | Convert 23 from Base 10 to Base 2, 8, 16.

to Base	Number (Quotient)	Remainder
2	147	
2	73	1
2	36	1
2	18	0
2	9	0
2	4	1
2	2	0
2	1	0
	0	1

The binary equivalent of $(147)_{10}$ is $(10010011)_2$

to Base	Number (Quotient)	Remainder
8	147	
8	18	3
8	2	2
	0	2

The octal equivalent of $(147)_{10}$ is $(223)_8$

to Base	Number (Quotient)	Remainder
16	147	
16	9	3
	0	9

The hexadecimal equivalent of $(147)_{10}$ is $(93)_{16}$

Example 2.3 | Convert 147 from Base 10 to Base 2, 8 and 16.

to Base	Number (Quotient)	Remainder
2	94	
2	47	0
2	23	1
2	11	1
2	5	1
2	2	1
2	1	0
	0	1

The binary equivalent of $(94)_{10}$ is $(1011110)_2$

to Base	Number (Quotient)	Remainder
8	94	
8	11	6
8	1	3
	0	1

The octal equivalent of $(94)_{10}$ is $(136)_8$

to Base	Number (Quotient)	Remainder
16	94	
16	5	14
	0	5

The number 14 in hexadecimal is E.
The hexadecimal equivalent of $(94)_{10}$ is $(5E)_{16}$

Example 2.4 | Convert 94 from Base 10 to Base 2, 8 and 16.

2.3.2 Converting Decimal *Fraction* to Binary, Octal, Hexadecimal

A fractional number is a number less than 1. It may be .5, .00453, .564, etc. We use the multiplication operation to convert decimal fraction to any other base.

$$\begin{array}{r} 0.2345 \\ \underline{\times 2} \\ \mathbf{0}.4690 \end{array}$$

$$\begin{array}{r} .4690 \\ \underline{\times 2} \\ \mathbf{0}.9380 \end{array}$$

$$\begin{array}{r} .9380 \\ \underline{\times 2} \\ \mathbf{1}.8760 \end{array}$$

$$\begin{array}{r} .8760 \\ \underline{\times 2} \\ \mathbf{1}.7520 \end{array}$$

$$\begin{array}{r} .7520 \\ \underline{\times 2} \\ \mathbf{1}.5040 \end{array}$$

$$\begin{array}{r} .5040 \\ \underline{\times 2} \\ \mathbf{1}.0080 \end{array}$$

The binary equivalent of $(0.2345)_{10}$ is $(0.001111)_2$

Example 2.5 | Convert 0.2345 from Base 10 to Base 2.

To convert a decimal fraction to:

1. binary-multiply by 2,
2. octal-multiply by 8, and,
3. hexadecimal-multiply by 16.

Steps for conversion of a decimal fraction to any other base are:

1. Multiply the fractional number with the *toBase*, to get a resulting number.
2. The resulting number has two parts, non-fractional part and fractional part.
3. Record the non-fractional part of the resulting number.
4. Repeat the above steps at least four times.
5. Write the digits in the non-fractional part starting from *upwards to downwards.*

0.865	0.865	0.865
x 2	x 8	x 16
1.730	**6**.920	5190
x 2	x 8	865 x
1.460	**7**.360	**13**.840
x 2	x 8	x 16
0.920	**2**.880	5040
x 2	x 8	840 x
1.840	**7**.040	**13**.440
x 2		x 16
1.680	The octal equivalent of	2640
x 2	$(0.865)_{10}$ is $(.6727)_8$	440 x
1.360		**7**.040

The binary equivalent of $(.865)_{10}$ is $(.110111)_2$

The number 13 in hexadecimal is D.

The hexadecimal equivalent of $(0.865)_{10}$ is $(.DD7)_{16}$

Example 2.5a | Convert 0.865 from Base 10 to Base 2, 8 and 16.

2.3.3 Converting Decimal *Integer.Fraction* to Binary, Octal, Hexadecimal

A decimal *integer.fraction* number has both integer part and fraction part. The steps for conversion of a decimal *integer.fraction* to any other base are:

1. Convert decimal integer part to the desired base following the steps shown in Section 2.3.1.
2. Convert decimal fraction part to the desired base following the steps shown in Section 2.3.2.
3. The integer and fraction part in the desired base is combined to get *integer.fraction.*

to Base	Number (Quotient)	Remainder
2	34	
2	17	**0**
2	8	**1**
2	4	**0**
2	2	**0**
2	1	**0**
	0	**1**

The binary equivalent of $(34)_{10}$ is $(100010)_2$

0.4674

x 2

0.9348

x 2

1.8696

x 2

1.7392

x 2

1.4784

x 2

0.9568

x 2

1.936

The binary equivalent of $(0.4674)_{10}$ is $(.011101)_2$

The binary equivalent of $(34.4674)_{10}$ is $(100010.011101)_2$

Example 2.6 | Convert 34.4674 from Base 10 to Base 2.

to Base	Number (Quotient)	Remainder
8	34	
8	4	**2**
	0	**4**

The octal equivalent of $(34)_{10}$ is $(42)_8$

0.4674

x 8

3.7392

x 8

5.9136

x 8

7.3088

x 8

2.4704

The octal equivalent of $(0.4674)_{10}$ is $(.3572)_8$

The octal equivalent of $(34.4674)_{10}$ is $(42.3572)_8$

Example 2.7 | Convert 34.4674 from Base 10 to Base 8.

2.4 Conversion of Binary, Octal, Hexadecimal to Decimal

A binary, octal or hexadecimal number has two parts—integer part and fraction part. For example, a binary number could be 10011, 0.011001 or 10011.0111. The numbers 45, .362 or 245.362 are octal numbers. A hexadecimal number could be A2, .4C2 or A1.34.

The method used for the conversion of integer part and fraction part of binary, octal or hexadecimal number to decimal number is the same; multiplication operation is used for the conversion. The conversion mechanism uses the face value and position value of digits. The steps for conversion are as follows:

1. Find the sum of the ***Face Value* * *(fromBase)*^{position}** for each digit in the number.

   a. *In a non-fractional number, the rightmost digit has position 0 and the position increases as we go towards the left.*

to Base	Number (Quotient)	Remainder
16	34	
16	2	**2**
	0	**2**

The hexadecimal equivalent of $(34)_{10}$ is $(22)_{16}$

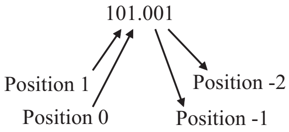
0.4674
x 16
28044
4674x
9.4784
x 16
28704
4784x
7.6544
x 16
39264
6544x
10.4704
x 16
28224
4904x
7.5264

The hexadecimal equivalent of $(0.4674)_{10}$ is $(.97A7)_{16}$

The hexadecimal equivalent of $(34.4674)_{10}$ is $(22.97A7)_{16}$

Example 2.8 | Convert 34.4674 from Base 10 to Base 16.

b. *In a fractional number, the first digit to the left of decimal point has position 0 and the position increases as we go towards the left. The first digit to the right of the decimal point has position –1 and it decreases as we go towards the right (–2, –3, etc.)*



1011 fromBase 2 toBase 10
$1011 = 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$
$= 1*8 + 0*4 + 1*2 + 1*1$
$= 8 + 0 + 2 + 1$
$= 11$
The decimal equivalent of $(1011)_2$ is 11.

62 fromBase 8 toBase 10
$62 = 6*8^1 + 2*8^0$
$= 6*8 + 2*1$
$= 48 + 2$
$= 50$
The decimal equivalent of $(62)_8$ is 50.

C15 fromBase 16 toBase 10
$C15 = C*16^2 + 1*16^1 + 5*16^0$
$= 12*256 + 1*16 + 5*1$
$= 3072 + 16 + 5$
$= 3093$
The decimal equivalent of $(C15)_{16}$ is 3093

Example 2.9 | Convert 1011 from Base 2 to Base 10.
Convert 62 from Base 8 to Base 10.
Convert C15 from Base 16 to Base 10.

.1101 fromBase 2 toBase 10

$.1101 = 1*2^{-1} + 1*2^{-2} + 0*2^{-3} + 1*2^{-4}$
$= 1/2 + 1/4 + 0 + 1/16$
$= 13/16$
$= .8125$

The decimal equivalent of $(.1101)_2$ is .8125

.345 fromBase 8 toBase 10

$.345 = 3*8^{-1} + 4*8^{-2} + 5*8^{-3}$
$= 3/8 + 4/64 + 5/512$
$= 229/512$
$= .447$

The decimal equivalent of $(.345)_8$ is .447

.15 fromBase 16 toBase 10

$.15 = 1*16^{-1} + 5*16^{-2}$
$= 1/16 + 5/256$
$= 21/256$
$= .082$

The decimal equivalent of $(.15)_{16}$ is .082

Example 2.10 | Convert .1101 from Base 2 to Base 10.
Convert .345 from Base 8 to Base 10.
Convert .15 from Base 16 to Base 10.

1011.1001 fromBase 2 toBase 10

$1011.1001 = 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 + 1*2^{-1} + 0*2^{-2} + 0*2^{-3} + 1*2^{-4}$
$= 8 + 0 + 2 + 1 + 1/2 + 0 + 0 + 1/16$
$= 11 + 9/16$
$= 11.5625$

The decimal equivalent of $(1011.1001)_2$ is 11.5625

24.36 fromBase 8 toBase 10

$24.36 = 2*8^1 + 4*8^0 + 3*8^{-1} + 6*8^{-2}$
$= 16 + 4 + 3/8 + 6/64$
$= 20 + 30/64$
$= 20.4687$

The decimal equivalent of $(24.36)_8$ is 20.4687

4D.21 fromBase 16 toBase 10

$4D.21 = 4*16^1 + D*16^0 + 2*16^{-1} + 1*16^{-2}$
$= 64 + 13 + 2/16 + 1/256$
$= 77 + 33/256$
$= 77.1289$

The decimal equivalent of $(4D.21)_{16}$ is 77.1289

Example 2.11 | Convert 1011.1001 from Base 2 to Base 10.
Convert 24.36 from Base 8 to Base 10.
Convert 4D.21 from Base 16 to Base 10.

2.5 Conversion of Binary to Octal, Hexadecimal

A binary number can be converted into octal or hexadecimal number using a shortcut method. The shortcut method is based on the following information:

1. An octal digit from 0 to 7 can be represented as a combination of 3 bits, since 23 = 8.
2. A hexadecimal digit from 0 to 15 can be represented as a combination of 4 bits, since $2^4 = 16$.

The Steps for Binary to Octal Conversion are:

1. Partition the binary number in groups of three bits, starting from the right-most side.
2. For each group of three bits, find its octal number.
3. The result is the number formed by the combination of the octal numbers.

The Steps for Binary to Hexadecimal Conversion are:

1. Partition the binary number in groups of four bits, starting from the right-most side.
2. For each group of four bits, find its hexadecimal number.
3. The result is the number formed by the combination of the hexadecimal numbers.

Given binary number 1110101100110

1. Partition binary number in groups of three bits, starting from the right-most side.

   1 110 101 100 110

2. For each group find its octal number.

   1 110 101 100 110
   1 6 5 4 6

3. The octal number is 16546.

Example 2.12 | Convert the binary number 1110101100110 to octal.

Given binary number 1110101100110

1. Partition binary number in groups of four bits, starting from the right-most side.

   1 1101 0110 0110

2. For each group find its hexadecimal number.

   1 1101 0110 0110
   1 D 6 6

3. The hexadecimal number is 1D66.

Example 2.13 | Convert the binary number 1110101100110 to hexadecimal

2.6 Conversion of Octal, Hexadecimal to Binary

The conversion of a number from octal and hexadecimal to binary uses the inverse of the steps defined for the conversion of binary to octal and hexadecimal.

The Steps for Octal to Binary Conversion are:

1. Convert each octal number into a three-digit binary number.
2. The result is the number formed by the combination of all the bits.

The Steps for Hexadecimal to Binary Conversion are:

1. Convert each hexadecimal number into a four-digit binary number.
2. The result is the number formed by the combination of all the bits.

1. Given number is 2BA3
2. Convert each hexadecimal digit into four digit binary number.

```
     2      B      A      3
   0010   1011   1010   0011
```

3. Combine all the bits to get the result 0010101110100011.

Example 2.14 | Convert the hexadecimal number 2BA3 to binary.

1. Given number is 473
2. Convert each octal digit into three digit binary number.

```
     4      7      3
    100    111    011
```

3. Combine all the bits to get the result 100111011.

Example 2.15 | Convert the octal number 473 to binary.

2.7 Binary Arithmetic

The arithmetic operations—addition, subtraction, multiplication and division, performed on the binary numbers is called *binary arithmetic*. In computer, the basic arithmetic operations performed on the binary numbers is:

1. Binary addition, and
2. Binary subtraction.

In the following subsections, we discuss the binary addition and the binary subtraction operations.

2.7.1 Binary Addition

Binary addition involves addition of two or more binary numbers. The *binary addition rules* are used while performing the binary addition. Table 2.3 shows the binary addition rules.

Table 2.3 | Binary addition rules

Input 1	Input 2		Sum	Carry
0	0	➔	0	No carry
0	1	➔	1	No carry
1	0	➔	1	No carry
1	1	➔	0	1

Binary addition of three inputs follows the rule shown in Table 2.4.

Table 2.4 | Binary addition of three inputs

Input 1	Input 2	Input 3		Sum	Carry
0	0	0	→	0	No Carry
0	0	1	→	1	No Carry
0	1	0	→	1	No Carry
0	1	1	→	0	1
1	0	0	→	1	No Carry
1	0	1	→	0	1
1	1	0	→	0	1
1	1	1	→	1	1

Addition of the binary numbers involves the following steps:

1. Start addition by adding the bits in unit column (the right-most column). Use the rules of binary addition.
2. The result of adding bits of a column is a *sum* with or without a *carry*.
3. Write the *sum* in the result of that column.
4. If a *carry* is present, the *carry* is carried-over to the addition of the next left column.
5. Repeat steps 2–4 for each column, i.e., the tens column, hundreds column and so on.

Let us now understand binary addition with the help of some examples.

When we add 0 and 1 in the unit column, sum is 1 and there is no carry. The sum 1 is written in the unit column of the result. In the tens column, we add 1 and 0 to get the sum 1. There is no carry. The sum 1 is written in the tens column of the result.

Binary Addition	**Decimal Addition**
1 0	2
+ 0 1	+ 1
Result 1 1	3
$11_2 = 3_{10}$	

Example 2.16 | Add 10 and 01. Verify the answer with the help of decimal addition.

When we add 1 and 1 in the unit column, sum is 0 and carry is 1. The sum 0 is written in the unit column of the result. The carry is carried-over to the next column, i.e., the tens column. In the tens column, we add 0, 1 and the carried-over 1, to get sum 0 and carry 1. The sum 0 is written in the tens column of the result. The carry 1 is carried-over to the hundreds column. In the hundreds column, the result is 1.

Binary Addition	**Decimal Addition**
1 1 ←Carry	
0 1	1
+ 1 1	+ 3
Result 1 0 0	4
$100_2 = 4_{10}$	

Example 2.17 | Add 01 and 11. Verify the answer with the help of decimal addition.

Binary Addition	Decimal Addition
1 1 ← Carry	
1 1	3
+ 1 1	+ 3
Result 1 1 0	6
$110_2 = 6_{10}$	

Example 2.18 | Add 11 and 11. Verify the answer with the help of decimal addition.

Binary Addition	Decimal Addition
1 1 1 1 ← Carry	
1 0 0 1	9
+ 1 1 1 1	+ 1 5
1 1 0 0 0	2 4
$11000_2 = 24_{10}$	

Example 2.19 | Add 1001 and 1111. Verify the answer with the help of decimal addition.

Binary Addition	Decimal Addition
1 1 1 1 1 ← Carry	
1 0 1 1 1	2 3
+ 1 1 0 0 0	+ 2 4
1 1 1	7
1 1 0 1 1 0	5 4
$110110_2 = 54_{10}$	

Example 2.20 | Add 10111, 11000 and 111. Verify the answer with the help of decimal addition.

2.7.2 Binary Subtraction

Binary subtraction involves subtracting of two binary numbers. The *binary subtraction rules* are used while performing the binary subtraction. The binary subtraction rules are shown in Table 2.5, where "Input 2" is subtracted from "Input 1."

Table 2.5 | Binary subtraction rules

Input 1	Input 2		Difference	Borrow
0	0	→	0	No borrow
0	1	→	1	1
1	0	→	1	No borrow
1	1	→	0	No borrow

The steps for performing subtraction of the binary numbers are as follows:

1. Start subtraction by subtracting the bit in the lower row from the upper row, in the unit column.

2. Use the binary subtraction rules. If the bit in the upper row is less than lower row, *borrow* 1 from the upper row of the next column (on the left side). The result of subtracting two bits is the *difference*.
3. Write the *difference* in the result of that column.
4. Repeat steps 2 and 3 for each column, i.e., the tens column, hundreds column and so on.

Let us now understand binary subtraction with the help of some examples.

When we subtract 1 from 1 in the unit column, the difference is 0. Write the difference in the unit column of the result. In the tens column, subtract 0 from 1 to get the difference 1. Write the difference in the tens column of the result.

Binary Subtraction	Decimal Subtraction
1 1 − 0 1 Result 1 0	3 − 1 2
$10_2 = 2_{10}$	

Example 2.21 | Subtract 01 from 11. Verify the answer with the help of decimal subtraction.

When we subtract 1 from 0 in the unit column, we have to borrow 1 from the left column since 0 is less than 1. After borrowing from the left column, 0 in the unit column becomes 10, and, 1 in the left column becomes 0. We perform 10–1 to get the difference 1. We write the difference in the unit column of the result. In the tens column, subtract 0 from 0 to get the difference 0. We write the difference 0 in the tens column of the result.

Binary Subtraction	Decimal Subtraction
$\overset{0}{\cancel{1}}\ \overset{10}{0}$ − 0 1 0 1	2 − 1 1
$01_2 = 1_{10}$	

Example 2.22 | Subtract 01 from 10. Verify the answer with the help of decimal subtraction.

When we do 0–1 in the unit column, we have to borrow 1 from the left column since 0 is less than 1. After borrowing from the left column, 0 in the unit column becomes 10, and, 1 in the left column becomes 0. We perform 10–1 to get the difference 1. We write the difference in the unit column of the result. In the tens column, when we do 0–1, we again borrow 1 from the left column. We perform 10–1 to get the difference 1. We write the difference in the tens column of the result. In the hundreds column, when we do 0–1, we again borrow 1 from the left column. We perform 10–1 to get the difference 1. We write the difference in the hundreds column of the result. In the thousands column, 0–0 is 0. We write the difference 0 in the thousands column of the result.

Example 2.23 | Subtract 0111 from 1110. Verify the answer with the help of decimal subtraction.

Binary Subtraction	**Decimal Subtraction**
0 10 0 10 0 10 } Borrow ~~1~~ ~~1~~ ~~1~~ 0 − 0 1 1 1 0 1 1 1	1 4 − 0 7 7
$0111_2 = 7_{10}$	

Example 2.23 | Continued

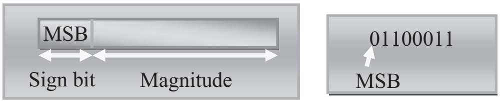
Binary Subtraction	**Decimal Subtraction**
1 1 ~~10~~ ~~10~~ 10 Borrow 1 ~~1~~ 0 0 0 1 1 0 0 1 1 0 0 0 1 0 1 1	4 9 − 3 8 1 1
$001011_2 = 11_{10}$	

Example 2.24 | Subtract 100110 from 110001. Verify the answer with the help of decimal subtraction.

2.8 Signed and Unsigned Numbers

A binary number may be positive or negative. Generally, we use the symbol "+" and "−" to represent positive and negative numbers, respectively. The sign of a binary number has to be represented using 0 and 1, in the computer. An *n-bit signed binary number* consists of two parts—sign bit and magnitude. The left most bit, also called the Most Significant Bit (MSB) is the *sign bit*. The remaining *n*−1 bits denote the *magnitude* of the number.

In signed binary numbers, the sign bit is 0 for a positive number and 1 for a negative number. For example, 01100011 is a positive number since its sign bit is 0, and, 11001011 is a negative number since its sign bit is 1. An 8-bit *signed number* can represent data in the range −128 to +127 (-2^7 to $+2^7-1$). The left-most bit is the sign bit.



In an *n-bit unsigned binary number*, the magnitude of the number *n* is stored in *n* bits. An 8-bit *unsigned number* can represent data in the range 0 to 255 ($2^8 = 256$).

2.8.1 Complement of Binary Numbers

Complements are used in computer for the simplification of the subtraction operation. For any number in base *r*, there exist two complements—(1) *r's* complement and (2) *r*−1*'s* complement.

Number System	Base	Complements possible
Binary	2	1′s complement and 2′s complement
Octal	8	7′s complement and 8′s complement
Decimal	10	9′s complement and 10′s complement
Hexadecimal	16	15′s complement and 16′s complement

Let us now see how to find the complement of a binary number. There are two types of complements for the binary number system—1′s complement and 2′s complement.

1. **1′s Complement of Binary Number** is computed by changing the bits 1 to 0 and the bits 0 to 1. For example,

   1′s complement of 101 is 010
   1′s complement of 1011 is 0100
   1′s complement of 1101100 is 0010011

2. **2′s Complement of Binary Number** is computed by adding 1 to the 1′s complement of the binary number. For example,

   2′s complement of 101 is 010 + 1 = 011
   2′s complement of 1011 is 0100 + 1 = 0101
   2′s complement of 1101100 is 0010011 + 1 = 0010100

The rule to find the complement of any number N in base r having n digits is
$(r - 1)$′s complement—$(r^n - 1) - N$
$(r$′s) complement—$(r^n - 1) - N + 1 = (r^n - N)$

2.9 Binary Data Representation

A binary number may also have a binary point, in addition to the sign. The binary point is used for representing fractions, integers and integer-fraction numbers. *Registers* are high-speed storage areas within the Central Processing Unit (CPU) of the computer. All data are brought into a register before it can be processed. For example, if two numbers are to be added, both the numbers are brought in registers, added, and the result is also placed in a register. There are two ways of representing the position of the binary point in the register—fixed point number representation and floating point number representation.

The *fixed point number representation* assumes that the binary point is fixed at one position either at the extreme left to make the number a fraction, or at the extreme right to make the number an integer. In both cases, the binary point is not stored in the register, but the number is treated as a fraction or integer. For example, if the binary point is assumed to be at extreme left, the number 1100 is actually treated as 0.1100.

The *floating point number representation* uses two registers. The first register stores the number without the binary point. The second register stores a number that indicates the position of the binary point in the first register.

We shall now discuss representation of data in the fixed point number representation and floating point number representation.

2.9.1 Fixed Point Number Representation

The integer binary signed number is represented as follows:

1. For a positive integer binary number, the sign bit is 0 and the magnitude is a positive binary number.
2. For a negative integer binary number, the sign bit is 1. The magnitude is represented in any one of the three ways:

   a. **Signed Magnitude Representation**—The magnitude is the positive binary number itself.

   b. **Signed 1's Complement Representation**—The magnitude is the 1's complement of the positive binary number.

   c. **Signed 2's Complement Representation**—The magnitude is the 2's complement of the positive binary number.

Table 2.6 shows the representation of the signed number 18.

Table 2.6 | Fixed point representation of the signed number 18

+18	**0 0010010**		**Sign bit is 0.** **0010010 is binary equivalent of +18**
–18	Signed magnitude representation	1 0010010	Sign bit is 1. 0010010 is binary equivalent of +18
	Signed 1′s complement representation	1 1101101	Sign bit is 1. 1101101 is 1′s complement of +18
	Signed 2′s complement representation	1 1101110	Sign bit is 1. 1101110 is 2′s complement of +18

Signed magnitude and signed 1′s complement representation are seldom used in computer arithmetic.

Let us now perform arithmetic operations on the signed binary numbers. We use the signed 2′s complement representation to represent the negative numbers.

1. **Addition of Signed Binary Numbers**—The addition of any two signed binary numbers is performed as follows:

   a. Represent the positive number in binary form.(For example, +5 is 0000 0101 and +10 is 0000 1010)

   b. Represent the negative number in 2′s complement form. (For example, –5 is 1111 1011 and –10 is 1111 0110)

   c. Add the bits of the two signed binary numbers.

   d. Ignore any carry out from the sign bit position.

Please note that the negative output is automatically in the 2′s complement form.

We get the decimal equivalent of the negative output number, by finding its 2′s complement, and attaching a negative sign to the obtained result.

Let us understand the addition of two signed binary numbers with the help of some examples.

+5 in binary form, i.e., 0000 0101. +10 in binary form, i.e., 0000 1010.

Binary Addition	Decimal Addition
0 0 0 0 0 1 0 1	+ 5
0 0 0 0 1 0 1 0	+1 0
0 0 0 0 1 1 1 1	+1 5
The result is 0000 1111_2 i.e, $+15_{10}$	

Example 2.25 | Add +5 and +10.

–5 in 2's complement form is 1111 1011. +10 in binary form is 0000 1010.

Binary Addition	Decimal Addition
1 1 1 1 1 0 1 1	– 5
0 0 0 0 1 0 1 0	+1 0
0 0 0 0 0 1 0 1	+ 5
The result is 0000 0101_2, i.e., $+5_{10}$	

Example 2.26 | Add –5 and +10.

+5 in binary form is 0000 0101. –10 in 2's complement form is 1111 0110.

Binary Addition	Decimal Addition
0 0 0 0 0 1 0 1	+ 5
1 1 1 1 0 1 1 0	– 1 0
1 1 1 1 1 0 1 1	– 5
The result is 1111 1011_2, i.e., $–5_{10}$	

The result is in 2's complement form. To find its decimal equivalent:

Find the 2's complement of 1111 1011, i.e., 0000 0100 + 1 = 0000 0101. This is binary equivalent of +5. Attaching a negative sign to the obtained result gives us –5.

Example 2.27 | Add +5 and –10.

–5 in 2's complement form is 1111 1011. –10 in 2's complement form is 1111 0110.

Binary Addition	Decimal Addition
1 1 1 1 1 0 1 1	– 5
1 1 1 1 0 1 1 0	– 1 0
1 1 1 1 0 0 0 1	– 1 5
The result is 1111 0001_2, i.e., $–15_{10}$	

The result is in 2's complement form. To find its decimal equivalent:

Find the 2's complement of 1111 0001, i.e., 0000 1110 + 1 = 0000 1111. This is binary equivalent of +15. Attaching a negative sign to the obtained result gives us –15.

Example 2.28 | Add –5 and –10.

2. **Subtraction of Signed Binary Numbers**—The subtraction of signed binary numbers is changed to the addition of two signed numbers. For this, the sign of the second number is changed before performing the addition operation.

(−A) − (+B) = (−A) + (−B)	(+B in subtraction is changed to −B in addition)
(+A) − (+B) = (+A) + (−B)	(+B in subtraction is changed to −B in addition)
(−A) − (−B) = (−A) + (+B)	(−B in subtraction is changed to +B in addition)
(+A) − (−B) = (+A) + (+B)	(−B in subtraction is changed to +B in addition)

We see that the subtraction of signed binary numbers is performed using the addition operation.

The hardware logic for the fixed point number representation is simple, when we use 2's complement for addition and subtraction of the signed binary numbers. When two large numbers having the same sign are added, then an overflow may occur, which has to be handled.

2.9.2 Floating Point Number Representation

The floating point representation of a number has two parts—mantissa and exponent. The mantissa is a signed fixed point number. The exponent shows the position of the binary point in the mantissa.

For example, the binary number +11001.11 with an 8-bit mantissa and 6-bit exponent is represented as follows:

1. Mantissa is 01100111. The left most 0 indicates that the number is positive.
2. Exponent is 000101. This is the binary equivalent of decimal number + 5.
3. The floating point number is Mantissa × 2^{exponent}, i.e., + (.1100111) × 2+5.

The arithmetic operation with the floating point numbers is complicated, and uses complex hardware as compared to the fixed point representation. However, floating point calculations are required in scientific calculations, so, computers have a built-in hardware for performing floating point arithmetic operations.

2.10 Binary Coding Schemes

The alphabetic data, numeric data, alphanumeric data, symbols, sound data and video data, are represented as combination of bits in the computer. The bits are grouped in a fixed size, such as 8 bits, 6 bits or 4 bits. A code is made by combining bits of definite size. *Binary Coding schemes* represent the data such as alphabets, digits 0–9, and symbols in a standard code. A combination of bits represents a unique symbol in the data. The standard code enables any programmer to use the same combination of bits to represent a symbol in the data.

The binary coding schemes that are most commonly used are:

1. Extended Binary Coded Decimal Interchange Code (EBCDIC),
2. American Standard Code for Information Interchange (ASCII), and
3. Unicode

In the following subsections, we discuss the EBCDIC, ASCII and Unicode coding schemes.

2.10.1 EBCDIC

1. The Extended Binary Coded Decimal Interchange Code (EBCDIC) uses 8 bits (4 bits for zone, 4 bits for digit) to represent a symbol in the data.
2. EBCDIC allows 2^8 = 256 combinations of bits.
3. 256 unique symbols are represented using EBCDIC code. It represents decimal numbers (0–9), lower case letters (a–z), uppercase letters (A–Z), Special characters, and Control characters (printable and non-printable, e.g., for cursor movement, printer vertical spacing, etc.).
4. EBCDIC codes are mainly used in the mainframe computers.

2.10.2 ASCII

1. The American Standard Code for Information Interchange (ASCII) is widely used in computers of all types.
2. ASCII codes are of two types—ASCII-7 and ASCII-8.
3. **SCII-7** is a 7-bit standard ASCII code. In ASCII-7, the first 3 bits are the zone bits and the next 4 bits are for the digits. ASCII-7 allows 27 = 128 combinations. 128 unique symbols are represented using ASCII-7. ASCII-7 has been modified by IBM to ASCII-8.
4. **ASCII-8** is an extended version of ASCII-7. ASCII-8 is an 8-bit code having 4 bits for zone and 4 bits for the digit. ASCII-8 allows 28 = 256 combinations. ASCII-8 represents 256 unique symbols. ASCII is used widely to represent data in computers.
5. The ASCII-8 code represents 256 symbols.

   a. Codes 0 to 31 represent control characters (non-printable), because they are used for actions like, Carriage return (CR), Bell (BEL), etc.

   b. Codes 48 to 57 stand for numeric 0–9.

   c. Codes 65 to 90 stand for uppercase letters A–Z.

   d. Codes 97 to 122 stand for lowercase letters a–z.

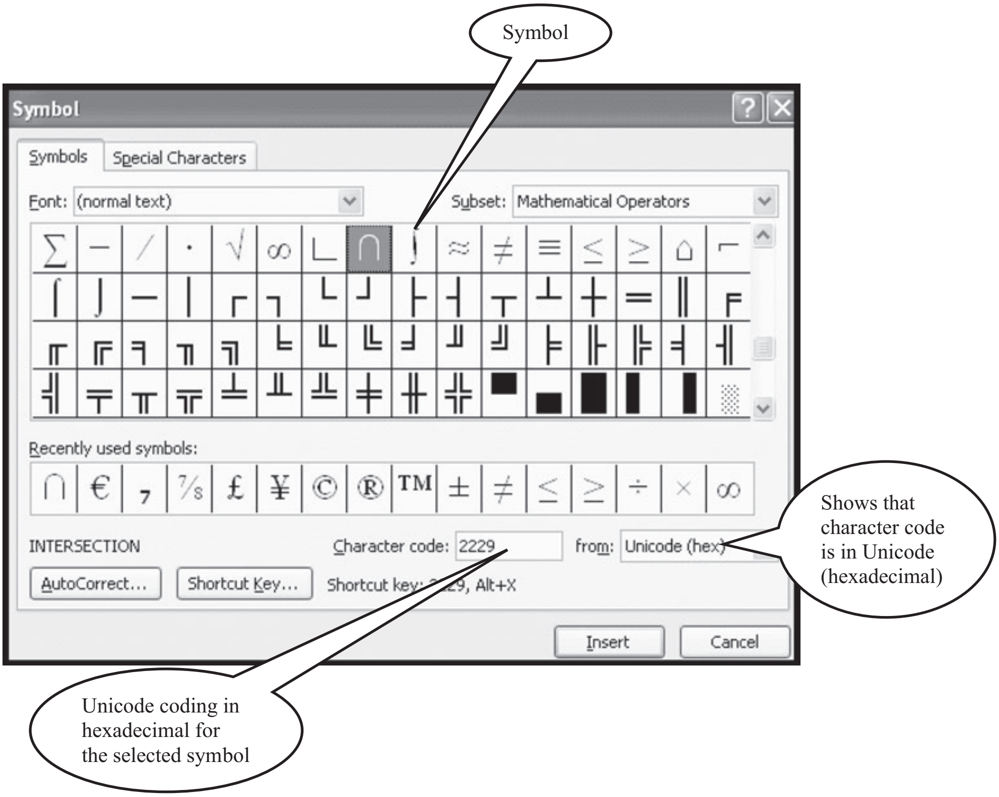
   e. Codes 128 to 255 are the extended ASCII codes.

2.10.3 Unicode

1. Unicode is a universal character encoding standard for the representation of text which includes letters, numbers and symbols in multi-lingual environments. The Unicode Consortium based in California developed the Unicode standard.
2. Unicode uses 32 bits to represent a symbol in the data.
3. Unicode allows 2^{32} = 4164895296 (~ 4 billion) combinations.
4. Unicode can uniquely represent any character or symbol present in any language like Chinese, Japanese, etc. In addition to the letters; mathematical and scientific symbols are also represented in Unicode codes.
5. An advantage of Unicode is that it is compatible with the ASCII-8 codes. The first 256 codes in Unicode are identical to the ASCII-8 codes.
6. Unicode is implemented by different *character encodings*. UTF-8 is the most commonly used encoding scheme. UTF stands for Unicode Transformation Format. UTF-8 uses 8 bits to 32 bits per code.

If you wish to see the Unicode character encoding in MS-Word 2007, do as follows:

<Insert> <Symbol>. A Symbol dialog box will appear which displays the symbols, and the character codes in a coding scheme, as shown in Figure 2.1.



Font: (normal text)

Subset: Mathematical Operators

INTERSECTION

Character code: 2229

from: Unicode (hex)

Shortcut key: 2229, Alt+X

Shows that character code is in Unicode (hexadecimal)

Unicode coding in hexadecimal for the selected symbol

Figure 2.1 | Unicode coding

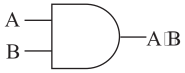
2.11 Logic Gates

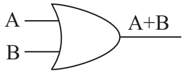
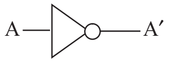
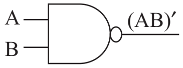
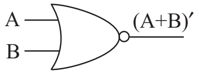
The information is represented in the computer in binary form. Binary information is represented using signals in two states *off* or *on* which correspond to 0 or 1, respectively. The manipulation of the binary information is done using logic gates. *Logic gates* are the hardware electronic circuits which operate on the input signals to produce the output signals. Each logic gate has a unique symbol and its operation is described using algebraic expression. For each gate, the *truth table* shows the output that will be outputted for the different possible combinations of the input signal. The AND, OR and NOT are the basic logic gates. Some of the basic combination of gates that are widely used are—NAND, NOR, XOR and XNOR.

Table 2.7 shows the different logic gates, their symbols, their algebraic function and the truth table for each logic gate. The comments list the features of each logic gate.

Table 2.7 | Logic gates

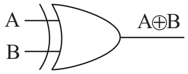
Operation	Symbol	Algebraic Function	Comments	Truth Table

AND		X = A.B or
		X = AB
			• Two or more binary inputs
			• The output is 1 if all the inputs are 1, otherwise the output is 0.
			• Represented using a multiplication symbol "."
				A | B | A.B
				0 | 0 | 0
				0 | 1 | 0
				1 | 0 | 0
				1 | 1 | 1

OR		X = A + B	• Two or more binary inputs • The output is 1 if at least one input is 1, otherwise the output is 0. • Represented using a "+"	A | B | A+B 0 | 0 | 0 0 | 1 | 1 1 | 0 | 1 1 | 1 | 1
NOT		A = A′	• One binary input • The output is complement (opposite) of input. If input is 1 output is 0 and if input is 0 output is 1. • Represented using a " ′ "	A | A' 0 | 1 1 | 0
NAND		X = (AB)′	• Two or more binary inputs • NAND is complement of AND	A | B | (A.B)' 0 | 0 | 1 0 | 1 | 1 1 | 0 | 1 1 | 1 | 0
NOR		X = (A + B)′	• Two or more binary inputs • NOR is complement of OR.	A | B | (A+B)' 0 | 0 | 1 0 | 1 | 0 1 | 0 | 0 1 | 1 | 0

(Continued)

Table 2.7 | Continued

<table>
<tr><th>Operation</th><th>Symbol</th><th>Algebraic Function</th><th>Comments</th><th>Truth Table</th></tr>
<tr><td>XOR</td><td>
</td><td>X = (A ⊕ B)</td><td>• Two or more binary inputs
• The output is 1 if the odd number of inputs is 1.
• Represented using a " ⊕ "</td><td><table><tr><th>A</th><th>B</th><th>(A⊕ B)</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table></td></tr>
<tr><td>XNOR</td><td></td><td>X = (A ⊕ B)′</td><td>• Two or more binary inputs
• XNOR is complement of XOR.</td><td><table><tr><th>A</th><th>B</th><th>(A⊕B)'</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table></td></tr>
</table>

2.12 Programming Fundamentals

Computer is an electronic device that accepts data, processes it, and generates the relevant output. It can perform both simple and complex tasks with very high speed and accuracy. However, a computer cannot perform any task—simple or complex, of its own. Computers need to be instructed about "how" the task is to be performed. The set of instructions that instruct the computer about the way the task is to be performed is called a *program*. A program is required for processing all kind of tasks—simple tasks like addition of two numbers, and complex tasks like gaming etc.

In this chapter, we will discuss the steps that are followed while writing a computer program. A brief description of different programming constructs is also presented. We will also discuss the characteristics of a good program.

2.13 Program Development Life Cycle

As stated earlier, a program is needed to instruct the computer about the way a task is to be performed. The instructions in a program have three essential parts:

1. Instructions to accept the input data that needs to be processed,
2. Instructions that will act upon the input data and process it, and
3. Instructions to provide the output to user

The instructions in a program are defined in a specific sequence. Writing a computer program is not a straightforward task. A person who writes the program (computer programmer) has to follow the Program Development Life Cycle.

Let's now discuss the steps that are followed by the programmer for writing a program:

1. **Problem Analysis**—The programmer first understands the problem to be solved. The programmer determines the various ways in which the problem can be solved, and decides upon a single solution which will be followed to solve the problem.

2. **Program Design**—The selected solution is represented in a form, so that it can be coded. This requires three steps:
    a. An *algorithm* is written, which is an English-like explanation of the solution.

    b. A *flowchart* is drawn, which is a diagrammatic representation of the solution. The solution is represented diagrammatically, for easy understanding and clarity.

    c. A *pseudo code* is written for the selected solution. Pseudo code uses the structured programming constructs. The pseudo code becomes an input to the next phase.

3. **Program Development**
    a. The computer programming languages are of different kinds—low-level languages, and high-level languages like C, C++ and Java. The pseudo code is coded using a *suitable* programming language.

    b. The coded pseudo code or program is compiled for any syntax errors. *Syntax errors* arise due to the incorrect use of programming language or due to the grammatical errors with respect to the programming language used. During compilation, the syntax errors, if any, are removed.

    c. The successfully compiled program is now ready for execution.

    d. The executed program generates the output result, which may be correct or incorrect. The program is tested with various inputs, to see that it generates the desired results. If incorrect results are displayed, then the program has *semantic error* (logical error). The semantic errors are removed from the program to get the correct results.

    e. The successfully tested program is ready for use and is installed on the user's machine.

4. **Program Documentation and Maintenance**—The program is properly documented, so that later on, anyone can use it and understand its working. Any changes made to the program, after installation, forms part of the maintenance of program. The program may require updating, fixing of errors etc. during the maintenance phase.

Table 2.8 summarises the steps of the program development cycle.

Table 2.8 | Program development life cycle

Program Analysis	• Understand the problem • Have multiple solutions • Select a solution
Program Design	• Write Algorithm • Write Flowchart • Write Pseudo code
Program Development	• Choose a programming language • Write the program by converting the pseudo code, and then using the programming language. • Compile the program and remove syntax errors, if any • Execute the program. • Test the program. Check the output results with different inputs. If the output is incorrect, modify the program to get correct results. • Install the tested program on the user's computer.
Program Documentation and maintenance	• Document the program, for later use. • Maintain the program for updating, removing errors, changing requirements etc.

2.14 Algorithm

Algorithm is an ordered sequence of finite, well defined, unambiguous instructions for completing a task. Algorithm is an English-like representation of the logic which is used to solve the problem. It is a step-by-step procedure for solving a task or a problem. The steps must be ordered, unambiguous and finite in number.

For accomplishing a particular task, different algorithms can be written. The different algorithms differ in their requirements of time and space. The programmer selects the best-suited algorithm for the given task to be solved.

Let's now look at two simple algorithms to find the greatest among three numbers, as follows:

Algorithm to find the greatest among three numbers:

ALGORITHM 1

```
Step 1:  Start
Step 2:  Read the three numbers A, B, C
Step 3:  Compare A and B. If A is greater perform step 4 else perform step 5.
Step 4:  Compare A and C. If A is greater, output "A is greatest" else output "C is greatest".
         Perform step 6.
Step 5:  Compare B and C. If B is greater, output "B is greatest" else output "C is greatest".
Step 6:  Stop
```

ALGORITHM 2

```
Step 7:   Start
Step 8:   Read the three numbers A, B, C
Step 9:   Compare A and B. If A is greater, store A in MAX, else store B in MAX.
Step 10:  Compare MAX and C. If MAX is greater, output "MAX is greatest" else output
          "C is greatest".
Step 11:  Stop
```

Both the algorithms accomplish the same goal, but in different ways. The programmer selects the algorithm based on the advantages and disadvantages of each algorithm. For example, the first algorithm has more number of comparisons, whereas in the second algorithm an additional variable MAX is required.

2.15 Control Structures

The logic of a program may not always be a linear sequence of statements to be executed in that order. The logic of the program may require execution of a statement based on a decision. It may repetitively execute a set of statements unless some condition is met. *Control structures* specify the statements to be executed and the order of execution of statements.

Flowchart and Pseudo code use control structures for representation. There are three kinds of control structures:

1. Sequential—instructions are executed in linear order
2. Selection (branch or conditional)—it asks a true/false question and then selects the next instruction based on the answer
3. Iterative (loop)—it repeats the execution of a block of instructions.

The flowchart and the pseudo code control structures are explained in their respective sections.

2.16 Flowchart

A *flowchart* is a diagrammatic representation of the logic for solving a task. A flowchart is drawn using boxes of different shapes with lines connecting them to show the flow of control. The purpose of drawing a flowchart is to make the logic of the program clearer in a visual form. There is a famous saying "A photograph is equivalent to thousand words". The same can be said of flowchart. The logic of the program is communicated in a much better way using a flowchart. Since flowchart is a diagrammatic representation, it forms a common medium of communication.

2.16.1 Flowchart Symbols

A flowchart is drawn using different kinds of symbols. A symbol used in a flowchart is for a specific purpose. Figure 2.2 shows the different symbols of the flowchart along with their names. The flowchart symbols are available in most word processors including MS-WORD, facilitating the programmer to draw a flowchart on the computer using the word processor.

A single line description of the flowchart symbols is given in Table 2.9.

2.16.2 Preparing a Flowchart

Aflowchartmaybesimpleorcomplex.Themostcommonsymbolsthatareusedtodrawaflowchart are—Process, Decision, Data, Terminator, Connector and Flow lines. While drawing a flowchart, some rules need to be followed—(1) A flowchart should have a start and end, (2) The

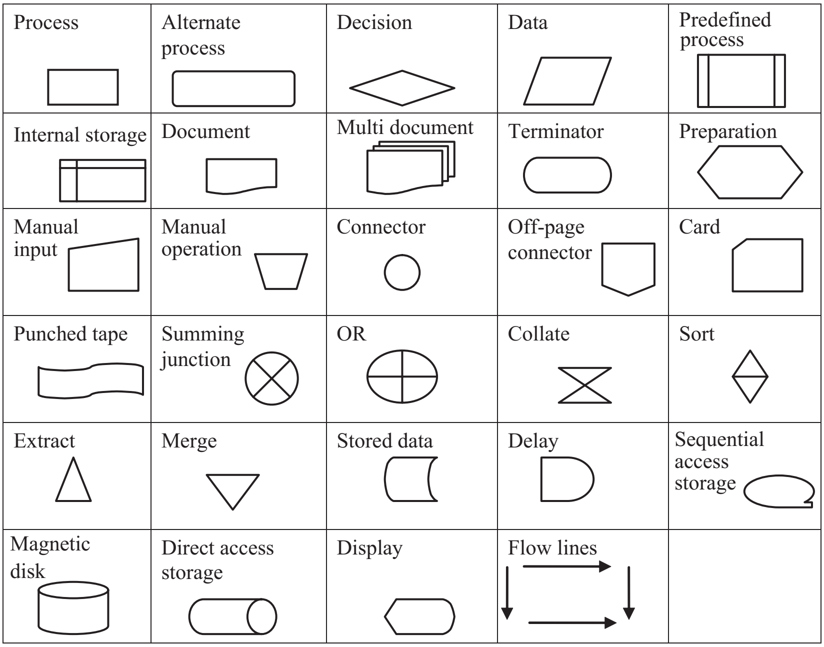


Figure 2.2 | Flowchart symbols (available for use in MS-WORD)

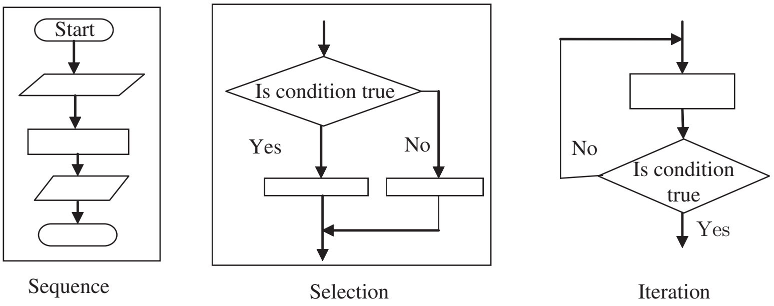
Table 2.9 | Description of flowchart symbols

Process—operation or action step
Alternate Process—alternate to normal process
Document—a document
Multi document—more than one document
Preparation—set-up process
Punched Tape—I/O from punched tape
Collate—organize in a format
Merge—merge in a predefined order
Sort—sort in some order
Display—display output
Predefined process—process previously specified
Internal Storage—stored in memory
Termination—start or stop point
Delay—wait
Decision—decision or a branch
Data—I/O to or from a process
Manual Input—Data entry from a form
Manual Operation—operation to be done manually
Connector—join flow lines
Off page connector—continue on another page
Summing Junction—Logical AND
OR—Logical OR
Sequenti al Access storage—stored on magnetic tape
Stored Data—general data storage
Magnetic Disk—I/O from magnetic disk
Direct access storage—storing on hard disk
Flow lines—indicates direction of flow
Extract—split process
Card—I/O from a punched card

direction of flow in a flowchart must be from top to bottom and left to right, and (3) The relevant symbols must be used while drawing a flowchart. While preparing the flowchart, the sequence, selection or iterative structures may be used wherever required. Figure 2.3 shows the sequence, selection and iteration structures.

We see that in a sequence, the steps are executed in linear order one after the other. In a selection operation, the step to be executed next is based on a decision taken. If the condition is true (yes) a different path is followed than if the condition evaluates to false (no). In case of iterative operation, a condition is checked. Based upon the result of this conditional check, true or false, different paths are followed. Either the next step in the sequence is executed or the control goes back to one of the already executed steps to make a loop.

Here, we will illustrate the method to draw flowchart, by discussing three different examples. To draw the flowcharts, relevant boxes are used and are connected via flow lines. The flowchart for the examples is shown in Figure 2.4.



Sequence Selection Iteration

Figure 2.3 | Control structures in flowchart

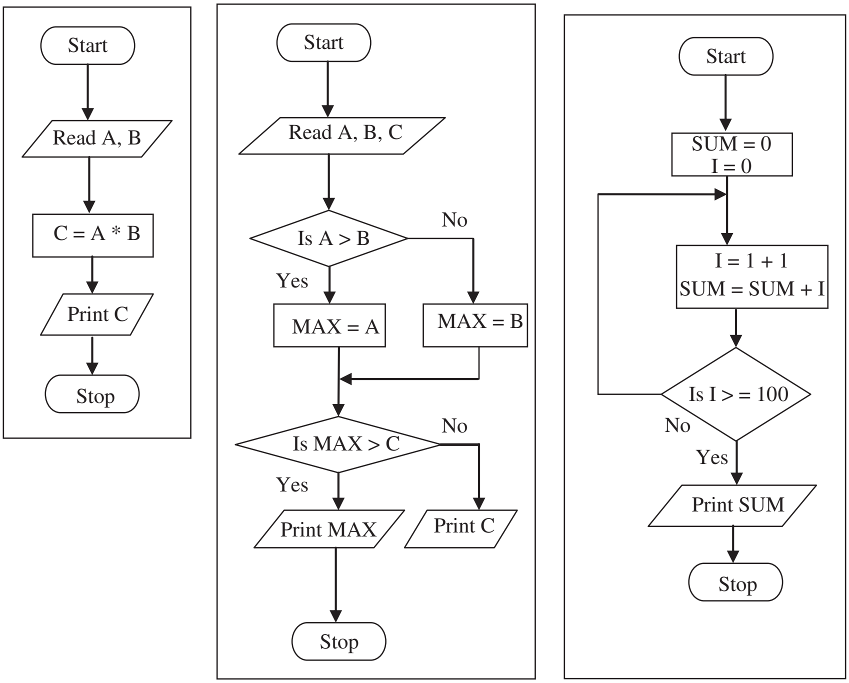


Figure 2.4 | Examples of flowchart

1. The first flowchart computes the product of any two numbers and gives the result. The flowchart is a simple sequence of steps to be performed in a sequential order.
2. The second flowchart compares three numbers and finds the maximum of the three numbers. This flowchart uses selection. In this flowchart, decision is taken based upon a condition, which decides the next path to be followed, i.e. If A is greater than B then the true (Yes) path is followed else the false (No) path is followed. Another decision is again made while comparing MAX with C.
3. The third flowchart finds the sum of first 100 integers. Here, iteration (loop) is performed so that some steps are executed repetitively until they fulfill some condition to exit from the repetition. In the decision box, the value of I is compared with 100. If it is false (No), a loop is created which breaks when the condition becomes true (Yes).

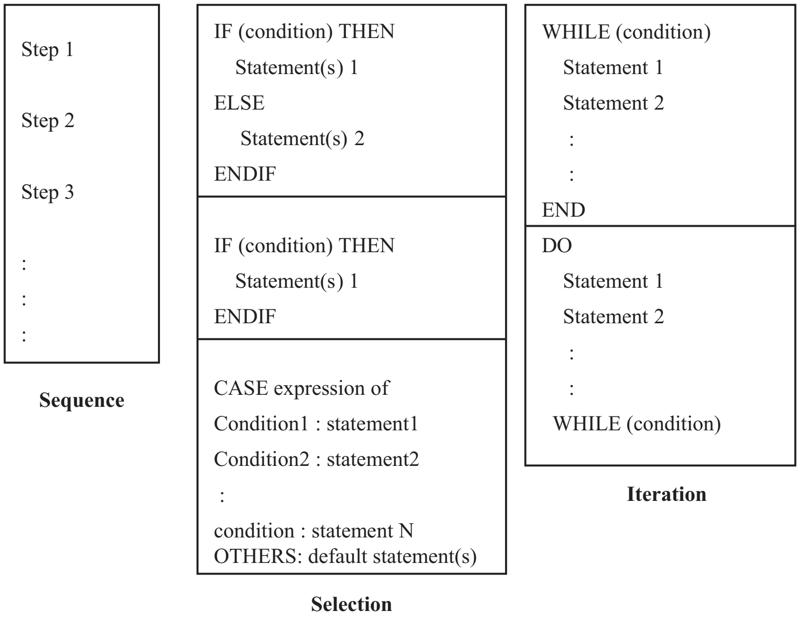
Flowcharts have their own benefits; however, they have some limitations too. A complex and long flowchart may run into multiple pages, which becomes difficult to understand and follow. Moreover, updating a flowchart with the changing requirements is a challenging job.

2.17 Pseudo Code

Pseudo code consists of short, readable and formally-styled English language used for explaining an algorithm. Pseudo code does not include details like variable declarations, subroutines etc. Pseudo code is a short-hand way of describing a computer program. Using pseudo code, it is easier for a programmer or a non-programmer to understand the general working of the program, since it is not based on any programming language. It is used to give a sketch of the structure of the program, before the actual coding. It uses the structured constructs of the programming language but is not machine-readable. Pseudo code cannot be compiled or executed. Thus, no standard for the syntax of pseudo code exists. For writing the pseudo code, the programmer is not required to know the programming language in which the pseudo code will be implemented later.

2.17.1 Preparing a Pseudo Code

1. Pseudo code is written using structured English.
2. In a pseudo code, some terms are commonly used to represent the various actions. For example, for inputting data the terms may be (INPUT, GET, READ), for outputting data (OUTPUT, PRINT, DISPLAY), for calculations (COMPUTE, CALCULATE), for incrementing (INCREMENT), in addition to words like ADD, SUBTRACT, INITIALIZE used for addition, subtraction, and initialization, respectively.
3. The control structures—sequence, selection, and iteration are also used while writing the pseudo code.
4. Figure 2.5 shows the different pseudo code structures. The *sequence structure* is simply a sequence of steps to be executed in linear order. There are two main *selection constructs*—if-statement and case statement. In the *if-statement*, if the condition is true then the THEN part is executed otherwise the ELSE part is executed. There can be variations of the if-statement also, like there may not be any ELSE part or there may be nested ifs. The *case statement* is used where there are a number of conditions to be checked. In a case statement, depending on the value of the expression, one of the conditions is true, for which the corresponding statements are executed. If no match for the expression occurs, then the OTHERS option which is also the default option, is executed.



```
Step 1

Step 2

Step 3

:
:
:
```

Sequence

```
IF (condition) THEN
   Statement(s) 1
ELSE
   Statement(s) 2
ENDIF
```

```
IF (condition) THEN
   Statement(s) 1
ENDIF
```

```
CASE expression of
Condition1 : statement1
Condition2 : statement2
 :
condition : statement N
OTHERS: default statement(s)
```

Selection

```
WHILE (condition)
   Statement 1
   Statement 2
    :
    :
END
```

```
DO
   Statement 1
   Statement 2
    :
    :
 WHILE (condition)
```

Iteration

Figure 2.5 | Control structures for pseudo code

```
READ values of A and B
COMPUTE C by multiplying A with B
PRINT the result C
STOP
```

(i) Find product of any two numbers

```
READ values of A, B, C
IF A is greater than B THEN
    ASSIGN A to MAX
ELSE
    ASSIGN B to MAX
IF MAX is greater than C THEN
    PRINT MAX is greatest
ELSE
    PRINT C is greatest
STOP
```

(ii) Find maximum of any three numbers

```
INITIALIZE SUM to zero
INIT IALIZE I to zero
DO WHILE (I less than 100)
    INCREMENT I
    ADD I to SUM and store in SUM
PRINT SUM
STOP
```

(iii) Find sum of first 100 integers

Figure 2.6 | Examples of pseudo code

5. WHILE and DO-WHILE are the two iterative statements. The WHILE loop and the DO-WHILE loop, both execute while the condition is true. However, in a WHILE loop the condition is checked at the start of the loop, whereas, in a DO-WHILE loop the condition is checked at the end of the loop. So the DO-WHILE loop executes at least once even if the condition is false when the loop is entered.

In Figure 2.6, the pseudo code is written for the same three tasks for which the flowchart was shown in the previous section. The three tasks are—(i) compute the product of any two numbers, (ii) find the maximum of any three numbers, and (iii) find the sum of first 100 integers.

A pseudo code is easily translated into a programming language. But, as there are no defined standards for writing a pseudo code, programmers may use their own style for writing the pseudo code, which can be easily understood. Generally, programmers prefer to write pseudo code instead of flowcharts.

Difference between Algorithm, Flowchart, and Pseudo Code: An algorithm is a sequence of instructions used to solve a particular problem. Flowchart and Pseudo code are tools to document and represent the algorithm. In other words, an algorithm can be represented using a flowchart or a pseudo code. Flowchart is a graphical representation of the algorithm. Pseudo code is a readable, formally styled English like language representation of the algorithm. Both flowchart and pseudo code use structured constructs of the programming language for representation. The user does not require the knowledge of a programming language to write or understand a flowchart or a pseudo code.

2.18 Programming Paradigms

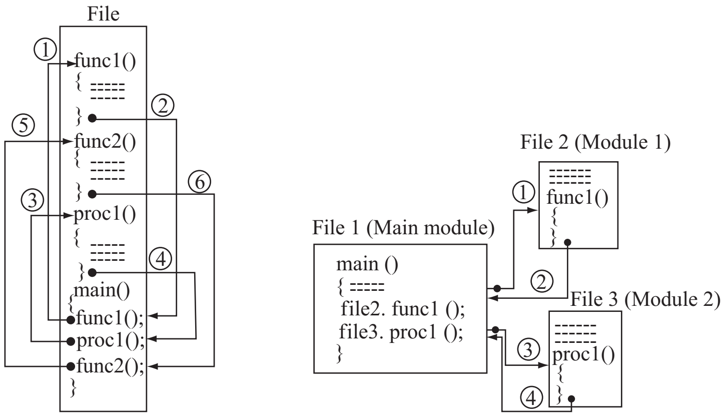
The word "paradigm" means an example that serves as a pattern or a model. Programming paradigms are the different patterns and models for writing a program. The programming paradigms may differ in terms of the basic idea which relates to the program computation. Broadly, programming paradigms can be classified as follows:

1. Structured Programming,
2. Object-Oriented Programming (OOP), and
3. Aspect-Oriented Programming (AOP). AOP is a new programming paradigm.

Earlier, the unstructured style of programming was used, where all actions of a small and simple program were defined within a single program only. It is difficult to write and understand a long and complex program using unstructured programming. The unstructured style of programming is not followed nowadays.

2.18.1 Structured Programming

1. Structured programming involves building of programs using small modules. The modules are easy to read and write.
2. In structured programming, the problem to be solved is broken down into small tasks that can be written independently. Once written, the small tasks are combined together to form the complete task.
3. Structured programming can be performed in two ways—*Procedural Programming* and *Modular Programming* (Figure 2.7).
4. **Procedural Programming** requires a given task to be divided into smaller procedures, functions or subroutines. A procedural program is largely a single file consisting of many procedures and functions and a function named *main ()*. A procedure or function performs a specific task. The function *main ()* integrates the procedures and functions



(i) Procedural programming

(ii) Modular programming

Figure 2.7 | Structured programming

by making calls to them, in an order that implements the functionality of the program. When a procedure or function is called, the execution control jumps to the called procedure or function, the procedure or function is executed, and after execution the control comes back to the calling procedure or function.

5. **Modular Programming** requires breaking down of a program into a group of files, where each file consists of a program that can be executed independently. In a modular program, the problem is divided into different independent but related tasks. For each identified task, a separate program (module) is written, which is a program file that can be executed independently. The different files of the program are integrated using a *main program file*. The main program file invokes the other files in an order that fulfills the functionality of the problem.
6. In structured programming, the approach to develop the software is process-centric or procedural. The software is divided into procedures or modules, based on the overall functionality of the software. As a result, the procedures and modules become tightly interwoven and interdependent. Thus, they are not re-usable.
7. *C, COBOL* and *Pascal* are examples of structured programming languages.

2.18.2 Object-Oriented Programming (OOP)

OOP focuses on developing the software based on their component objects. The components interact with each other to provide the functionality of the software. Object-oriented programming differs from procedural programming. In OOP the software is broken into components not based on their functionality, but based on the components or parts of the software. Each component consists of data and the methods that operate on the data. The components are complete by themselves and are re-usable. The terms that are commonly associated with object-oriented programming are as follows:

1. *Class* is the basic building block in object-oriented programming. A class consists of data attributes and methods that operate on the data defined in the class.
2. *Object* is a runtime instance of the class. An object has a state, defined behavior and a unique identity. The state of the object is represented by the data defined in the class. The methods defined in the class represent object behavior. A class is a template for a set of objects that share common data attributes and common behavior.
3. *Abstraction, Encapsulation, Inheritance and Polymorphism* are the unique features of object-oriented software.
4. *Abstraction* allows dealing with the complexity of the object. Abstraction allows picking out the relevant details of the object, and ignoring the non-essential details. Encapsulation is a way of implementing abstraction.
5. *Encapsulation* means information hiding. The encapsulation feature of object-oriented software hides the data defined in the class. Encapsulation separates implementation of the class from its interface. The interaction with the class is through the interface provided by the set of methods defined in the class. This separation of interface from its implementation allows changes to be made in the class without affecting its interface.
6. The *Inheritance* feature of object-oriented software allows a new class, called the derived class, to be derived from an already existing class known as the base class. The derived

class (subclass) inherits all data and methods of the base class (super class). It may override some or all of the data and methods of the base class or add its own new data and methods.

7. *Polymorphism* means, many forms. It refers to an entity changing its form depending on the circumstances. It allows different objects to respond to the same message in different ways. This feature increases the flexibility of the program by allowing the appropriate method to be invoked depending on the object executing the method invocation call.
8. *C++* and *Java* are object-oriented programming languages.

2.18.3 Aspect-Oriented Programming (AOP)

Aspect-oriented programming is a new programming paradigm that handles the crosscutting concerns of the software. The *crosscutting concerns* are the global concerns like logging, authentication, security, performance, etc., that do not fit into a single module or related modules. In OOP, the business logic or *core concern* is encapsulated in well-defined classes. However, the code for implementing crosscutting concerns is intertwined with a number of related classes or modules and gets scattered in the different classes of the software.

AOP is a new paradigm that focuses on the issue of handling crosscutting concerns at the programming language level. It helps the programmer in cleanly separating the core concerns and the crosscutting concerns of the software. AOP introduces a new modular unit called "aspect" that encapsulates the functionality of the crosscutting concerns. Aspects of a system are independent elements that can be changed, inserted or removed at compile time, and even reused without affecting the rest of system. Aspects are similar to the classes of object oriented programs; however, they implement the crosscutting concerns. At compilation time, the classes of object oriented programs and the aspects are combined into a final executable form using an "aspect weaver".

- *AspectJ and AspectC* are examples of aspect-oriented programming languages.

After having selected a suitable programming paradigm for the program to be written, the coding of the logic of a program has to be done in a computer programming language. For the purposes of coding, the programmer checks the requirements and suitability of the task, and selects from among the programming languages available for the selected programming paradigm.

Characteristics of a Good Program: A program written using any of the programming language must have certain characteristics, which makes it a good program. Some of the key characteristics of a good program are as follows:

1. The program should be well-written so that it is easily readable and structured.
2. The program should not have hard-coded input values. This implies that it should not be written to work for a particular input value, but must be a general program (also called generic program) that accepts input from the user.
3. The program should also be well-documented so that later the author or any other programmer can understand the program.
4. Since new and better operating systems keep coming up, a program must be designed to be portable, i.e. with minimum dependence on a particular operating system.

A program comprising of the above features is generally characterized as a good program.

2.19 Problem Formulation and Problem Solving

2.19.1 Problem Solving

Problem solving is an innovative process for finding solutions to problems. The process involves the following sequence of steps to be followed:

1. Identify the problem
2. Collect information to improve the understanding about the problem
3. Chart down the set of solutions
4. Select the best solution
5. Implement the best solution
6. Assess the results
7. If the results are satisfactory, stop else investigate the alternative solutions

The commonly used problem-solving tools are algorithms, flowcharts and pseudo code which have been discussed in this chapter.

2.19.2 Problem Formulation

Problem formulation is the methodology of describing the problem and the results by stating the requirements and objectives that are required to solve the problem using a computer program. The objectives are defined in terms of:

1. Input and its characteristics
2. Expected Output and its characteristics
3. Relationship between the input & the desired output

Example: ***Problem formulation to find the largest of three numbers***

Requirement: To find and print the largest of three numbers

Input: Three numbers

Characteristics of input: Numbers

Output: One of the three input that is large

Characteristics of output: Number

Relationship between Input and Output: Output will be one of the three numbers given as input.

2.20 Summary

1. *Face value* of a digit is the digit located at that place. The *position value* of digit is ($\text{base}^{\text{position}}$). The number is the sum of (face value * $\text{base}^{\text{position}}$) of all the digits.
2. In computer science, decimal *number system* (base 10), binary number system (base 2), octal number system (base 8), and hexadecimal number system (base 16) concern us.
3. *Decimal number system* has 10 digits—0 to 9, the maximum digit being 9.
4. *Binary number system* has two digits—0 and 1.
5. *Octal number system* consists of eight digits—0 to 7, the maximum digit being 7.
6. *Hexadecimal number system* has sixteen digits—0 to 9, A, B, C, D, E, F, where (A is for 10, B is for 11, C—12, D—13, E—14, F—15). The maximum digit is F, i.e., 15.

7. *Conversion of octal or hexadecimal number to binary* or vice-versa uses the shortcut method. Three and four bits of a binary number correspond to an octal digit and hexadecimal digit, respectively.
8. *Binary arithmetic operations* are the binary addition, subtraction, multiplication and division operations performed on the binary numbers.
9. For any number in base *r*, there is *r's complement and r–1's complement*. For example, binary numbers can be represented in 1's complement and 2's complement.
10. *Sign bit* is the most significant bit. The sign bit is 1 and 0 for a positive number and negative number, respectively.
11. *Position of binary point* in a binary number is represented using Fixed Point Number Representation and Floating Point Number Representation.
12. In *fixed point representation*, the positive integer binary number is represented with sign bit 0 and magnitude as positive binary number. The negative integer is represented in signed magnitude representation, signed 1's complement representation and signed 2's complement representation.
13. *Addition of two signed binary numbers* requires the positive number to be represented as binary number and negative number to be represented in 2's complement form.
14. *Floating point representation* has two parts—Mantissa and Exponent. Mantissa is a signed fixed point number and exponent shows the position of the binary point in the mantissa.
15. *Binary Coding schemes* represent data in a binary form in the computer. ASCII, EBCDIC, and Unicode are the most commonly used binary coding scheme.
16. *EBCDIC* is a 8-bit code with 256 different representations of characters. It is mainly used in mainframe computers.
17. *ASCII-8* is a 8-bit code and allows 256 characters to be represented. ASCII is widely to represent data in computers, internally.
18. *Unicode* is a universal character encoding standard for the representation of text in multi-lingual environments. UTF-8 is the most commonly used encoding.
19. *Logic gate* is the hardware electronic circuit that operates on input signals to produce output signal. AND, OR, NOT, NAND, NOR, XOR and XNOR are some of the logic gates.
20. *Program* is a set of instructions that instruct the computer about the way a task is to be performed.
21. *Program development life cycle* consists of—analyze problem to select a solution, write algorithm, draw flowchart and write pseudo code for the selected solution, write program code in a programming language, remove syntax and semantic errors, and install successfully tested program. Also, document the program to make program maintenance easy.
22. *Algorithm* is an ordered sequence of finite, well-defined, unambiguous instructions for completing a task.
23. *Control structures* specify the statements that are to be executed and the order of the statements that have to be executed. Sequential, selection, and iteration are three kinds of control structures.
24. *Flowchart* is a diagrammatic representation of the logic for solving a task. Flowchart is a tool to document and represent the algorithm. Flowchart is drawn using the flowchart symbols.

25. *Pseudo code* consists of short, readable and formally-styled English language which is used to explain and represent an algorithm. There is no standard syntax for writing the pseudo code, though some terms are commonly used in a pseudo code.
26. A pseudo code is easily translated into a *programming language*.
27. In *structured programming*, the given problem is broken down into smaller tasks based on their functionality. The individual tasks can be written independently, and later combined together to form the complete task. A structured program can be procedural or modular.
28. A *procedural program* is largely a single file consisting of many procedures and functions.
29. A *modular program* is a group of files, where each file consists of a program that can be executed independently.
30. In *OOP*, software is broken into components based on the components of the software. *Class* is the basic building block. *Object* is a runtime instance of class. Abstraction, encapsulation, inheritance, and polymorphism are the unique features of object-oriented software.
31. *AOP* is a new paradigm that focuses on the handling of crosscutting concerns like logging, authentication, security, and performance, at the programming language level. The crosscutting concerns are defined in a new modularization unit called aspect.
32. A *good program* is readable, structured, generic, well-documented and portable.

Exercise Questions

Conceptual Questions and Answers

1. *Convert the following decimal numbers into binary, octal and hexadecimal.*

a. *24*
b. *47*
c. *675*
d. *89*
e. *34.24*
f. *150.64*
g. *.98*
h. *.29*
i. *24.14*
j. *16.1*
k. *22.33*
l. *24.14*

a. $(24)_{10} = (11000)_2 = (30)_8 = (18)_{16}$
b. $(47)_{10} = (101111)_2 = (57)_8 = (2F)_{16}$
c. $(675)_{10} = (1010100011)_2 = (1243)_8 = (2A3)_{16}$
d. $(89)_{10} = (10110001)_2 = (131)_8 = (59)_{16}$
e. $(34.24)_{10} = (100010.00111)_2 = (42.1727)_8 = (22.3D70)_{16}$
f. $(150.64)_{10} = (10010110.1010)_2 = (226.5075)_8 = (96.A70A)_{16}$
g. $(.98)_{10} = (.1111)_2 = (.7656)_8 = (FAE1)_{16}$
h. $(.29)_{10} = (.0100)_2 = (.2243)_8 = (.4A3D)_{16}$
i. $(24.14)_{10} = (11000.0010)_2 = (30.1075)_8 = (18.23D)_{16}$
j. $(16.1)_{10} = (10000.0001)_2 = (20.063)_8 = (10.199)_{16}$
k. $(22.33)_{10} = (10110.0101)_2 = (26.250)_8 = (16.547)_{16}$
l. $(24.14)_{10} = (11000.0010)_2 = (30.1075)_8 = (18.23D)_{16}$

2. *Convert the following binary numbers into decimal numbers.*

a. *110000111*
b. *110011*
c. *1001111*
d. *11000001*

e. *1100110.1110*
f. *11110.0000*
g. 01001.0101
h. *1010.10101*
i. *11000011.111*
j. *11001.1101*
k. *100.111*
l. *101.0111*

a. $(110000111)_2 = (391)_{10}$
b. $(110011)_2 = (51)_{10}$
c. $(1001111)_2 = (79)_{10}$
d. $(11000001)_2 = (193)_{10}$
e. $(1100110.1110)_2 = (102.087)_{10}$
f. $(11110.0000)_2 = (30.0)_{10}$
g. $(01001.0101)_2 = (9.312)_{10}$
h. $(1010.10101)_2 = (10.65)_{10}$
i. $(11000011.111)_2 = (195.875)_{10}$
j. $(11001.1101)_2 = (25.8125)_{10}$
k. $(100.111)_2 = (4.875)_{10}$
l. $(101.0111)_2 = (5.4375)_{10}$

3. *Convert the following octal numbers into decimal numbers.*
a. *234*
b. *36*
c. *456*
d. *217*
e. *25.33*
f. *65.34*
g. *34.56*
h. *267.12*

a. $(234)_8 = (156)_{10}$
b. $(36)_8 = (30)_{10}$
c. $(456)_8 = (302)_{10}$
d. $(217)_8 = (143)_{10}$
e. $(25.33)_8 = (21.4218)_{10}$
f. $(65.34)_8 = (53.4375)_{10}$
g. $(34.56)_8 = (28.7187)_{10}$
h. $(267.12)_8 = (183.1562)_{10}$

4. *Convert the following hexadecimal numbers into decimal numbers.*
a. *E16*
b. *389*
c. *2AB*
d. *FF*
e. *E4.16*
f. *2A.1B*
g. *23.89*
h. *AC.BD*

a. $(E16)_{16} = (3606)_{10}$
b. $(389)_{16} = (905)_{10}$
c. $(2AB)_{16} = (683)_{10}$
d. $(FF)_{16} = (255)_{10}$
e. $(E4.16)_{16} = (228.0859)_{10}$
f. $(2A.1B)_{16} = (42.1054)_{10}$
g. $(23.89)_{16} = (35.5351)_{10}$
h. $(AC.BD)_{16} = (172.7382)_{10}$

5. *Convert the following binary into octal.*
a. *1100011*
b. *110011001100*
c. *100111100*
d. *110000011*
e. *110011011*
f. *1111000*
g. *0010101*
h. *101010101*

a. $(1100011)_2 = (143)_8$
b. $(110011001100)_2 = (6314)_8$
c. $(100111100)_2 = (474)_8$
d. $(110000011)_2 = (603)_8$
e. $(110011011)_2 = (633)_8$
f. $(1111000)_2 = (170)_8$
g. $(0010101)_2 = (025)_8$
h. $(101010101)_2 = (525)_8$

6. *Convert the following binary into hexadecimal.*
a. *11000011111*
b. *1100110011*
c. *100111100*
d. *1100000100*
e. *11001101110*
f. *111100000*
g. *010010101*
h. *101010101*

a. $(11000011111)_2 = (61F)_{16}$
b. $(1100110011)_2 = (333)_{16}$
c. $(100111100)_2 = (13C)_{16}$
d. $(1100000100)_2 = (304)_{16}$
e. $(11001101110)_2 = (66E)_{16}$
f. $(111100000)_2 = (1E0)_{16}$
g. $(010010101)_2 = (095)_{16}$
h. $(101010101)_2 = (155)_{16}$

7. *Convert the following octal into binary.*
a. *25*
b. *65*
c. *34*
d. *267*
e. *45*
f. *71*
g. *150*
h. *111*

a. $(25)_8 = (010101)_2$
b. $(65)_8 = (110101)_2$
c. $(34)_8 = (011100)_2$
d. $(267)_8 = (010110111)_2$
e. $(45)_8 = (100101)_2$
f. $(71)_8 = (111001)_2$
g. $(150)_8 = (001101000)_2$
h. $(111)_8 = (001001001)_2$

8. *Convert the following hexadecimal into binary.*
a. *A1*
b. *2AB*
c. *239*
d. *CCD*
e. *45C*
f. *71D*
g. *150*
h. *AAA*

a. $(A1)_{16} = (10100001)_2$
b. $(2AB)_{16} = (001010101011)_2$
c. $(239)_{16} = (001000111001)_2$
d. $(CCD)_{16} = (110011001101)_2$
e. $(45C)_{16} = (010001011100)_2$
f. $(71D)_{16} = (011100011101)_2$
g. $(150)_{16} = (000101010000)_2$
h. $(AAA)_{16} = (101010101010)_2$

9. *Perform binary addition on the following binary numbers.*
a. *111100, 011011*
b. *1001, 1111*
c. *0110, 1100*
d. *1100, 1010*

a. 1919111
b. 11000
c. 10010
d. 10110

10. *Perform binary subtraction on the following binary numbers.*
a. 111000, 011010
b. 1111, 1001
c. 0110, 0010
d. 1100, 1010

a. 11110
b. 0110
c. 0100
d. 0010

11. *Find 1's complement of the following binary numbers.*
a. *11000011111*
b. *1100110011*
c. *100111100*
d. *1100000100*

a. 00111100000
b. 0011001100
c. 011000011
d. 0011111011

12. *Find 2's complement of the following binary numbers.*
a. *11000011111*
b. *1100110011*
c. *100111100*
d. *1100000100*

a. 00111100001
b. 0011001101
c. 011000100
d. 0011111100

13. *Represent the following as 8-bit numbers in (a) Signed Magnitude representation, (b) Signed 1's complement representation, and (c) Signed 2's complement representation*
(i) *−22* (ii) *−55* (iii) *−34* (iv) *−67*

a. (i) 10010110 (ii) 10110111 (iii) 10100010 (iv) 11000011
b. (i) 01101001 (ii) 01001000 (iii) 01011101 (iv) 00111100
c. (i) 01101010 (ii) 01001001 (iii) 01011110 (iv) 00111101

14. *Represent the following as 8-bit numbers in Fixed Point number representation.*
    a. *+22*
    b. *+55*
    c. *+34*
    d. *+67*

    a. 00010110
    b. 00110111
    c. 00100010
    d. 01000011

15. *Perform binary addition of the following numbers.*
    a. *(+7) + (–9)*
    b. *(+3) + (+15)*
    c. *(–12) + (+15)*
    d. *(–14) + (+25)*
    e. *(–7) + (–7)*
    f. *(–9) + (–23)*
    g. *(–2) + (+4)*
    h. *(+34) + (–2)*

    a. $-2 = (11111110)_2$
    b. $+18 = (00010010)_2$
    c. $+3 = (00000011)_2$
    d. $+11 = (00001011)_2$
    e. $-14 = (11110010)_2$
    f. $-32 = (11100000)_2$
    g. $+2 = (00000010)_2$
    h. $+32 = (00100000)_2$

16. *Perform binary subtraction of the following numbers.*
    a. *(+7) – (–19)*
    b. *(+13) – (+15)*
    c. *(–12) – (+15)*
    d. *(–14) – (+25)*
    e. *(–7) – (–7)*
    f. *(–9) – (–23)*
    g. *(–2) – (+4)*
    h. *(+34) – (–2)*

    a. $+26 = (00011010)_2$
    b. $-2 = (11111110)_2$
    c. $-27 = (11100101)_2$
    d. $-39 = (11011001)_2$
    e. $0 = (00000000)_2$
    f. $+14 = (00001110)_2$
    g. $-6 = (11111010)_2$
    h. $+36 = (00100100)_2$

17. *Represent the following binary numbers in Floating Point number representation.*
    a. *1100.011*
    b. *110.001*
    c. *11.110*
    d. *1010.011*

    a. $.1100011 \times 2^{+4}$
    b. $.110001 \times 2^{+3}$
    c. $.11110 \times 2^{+2}$
    d. $.1010011 \times 2^{+4}$

Additional Questions

1. What is the significance of the base of number?
2. Explain the significance of the face value and position value of a number. Give an example.
3. What is the position value of a digit?
4. The decimal number system is in base _____.
5. The binary number system is in base _____.
6. The octal number system is in base _____.
7. The hexadecimal number system is in base _____.
8. Give the valid digits in the number systems.
    a. decimal
    b. binary
    c. octal
    d. hexadecimal

9. Write the largest digit in the number systems.
   a. decimal
   b. binary
   c. octal
   d. hexadecimal
10. How many valid digits are there in the number systems?
    a. decimal
    b. binary
    c. octal
    d. hexadecimal
11. Show the octal, binary and hexadecimal equivalent of the decimal number 11.
12. Why are binary coding schemes needed?
13. List any four commonly used binary coding schemes.
14. What number of bits is used to represent the following codes?
    a. EBCDIC
    b. ASCII-7
    c. ASCII-8
15. How many characters can be represented in the following codes?
    a. EBCDIC
    b. ASCII-7
    c. ASCII-8
16. How is Unicode different from the other Binary coding schemes? (Hint: multilingual, no. of characters)
17. What is UTF-8 character encoding?
18. Name the basic logic gates.
19. Draw the symbols of the following logic gates.
    a. AND
    b. OR
    c. NOT
    d. NAND
    e. NOR
    f. XOR
    g. XNOR
20. Write the truth table of the following logic gates.
    a. AND
    b. OR
    c. NOT
    d. NAND
    e. NOR
    f. XOR
    g. XNOR
21. Write the algebraic function of the following logic gates.
    a. AND
    b. OR
    c. NOT
    d. NAND
    e. NOR
    f. XOR
    g. XNOR
22. Define a program.
23. Explain the program development life cycle in detail.
24. What is the difference between syntax error and semantic error?
25. Define syntax error.
26. Define semantic error.
27. What is the purpose of program maintenance?
28. Define algorithm.
29. What are control structures?

30. Name the three kinds of control structures.
31. State the purpose of each of the control structures:
    a. Sequence
    b. Selection
    c. Iteration
32. Define flowchart.
33. Draw the flowchart symbol for the following.
    a. Process
    b. Decision
    c. Document
    d. Connector
    e. Magnetic Disk
34. State the meaning of the following flowchart symbols.
    a. Process
    b. Decision
    c. Document
    d. Connector
    e. Magnetic Disk
    f. Flow lines
35. Draw the control structures (Sequence, Selection and Iteration) for the flowchart.
36. Define pseudo code.
37. Write the pseudo code control (structures sequence, selection, and iteration).
38. What is the difference between WHILE and DO-WHILE statements?
39. Name the different programming paradigms.
40. What is modular programming?
41. What is procedural programming?
42. Name two procedural programming languages.
43. What are the key features of OOP?
44. Define:
    a. Class
    b. Object
    c. Abstraction
    d. Encapsulation
    e. Inheritance
    f. Polymorphism
45. How is static binding different from dynamic binding in OOP?
46. Name two object-oriented programming languages.
47. How is AOP different from OOP?
48. Define an aspect.
49. Name two aspect-oriented programming languages.
50. Explain the characteristics of a good program.
51. Give full form of the following abbreviations:
    a. EBCDIC
    b. MSB
    c. UTF
    d. ASCII
52. Write short notes on:
    a. Decimal Number System
    b. Binary Number System
    c. Octal Number System
    d. Hexadecimal Number System
    e. Binary arithmetic operations
    f. 1's complement of Binary number
    g. 2's complement of Binary number
    h. Fixed Point Number Representation

i. Floating Point Number Representation
j. Addition of signed binary numbers
k. Subtraction of signed binary numbers
l. Binary Coding schemes
m. Logic Gates
n. ASCII coding scheme
o. EBCDIC coding scheme
p. Unicode character encoding

53. Give differences between the following:
    a. 1′s complement and 2′s complement of Binary number
    b. ASCII coding scheme and EBCDIC coding scheme
    c. Decimal Number System and Binary Number System
    d. Octal Number System and Hexadecimal Number System
    e. Fixed Point Number Representation and Floating Point Number Representation
54. Give full form for the following abbreviations:
    a. AOP b. OOP
55. Write short notes on:
    a. Program Development Life Cycle
    b. Algorithm
    c. Control structures
    d. Flowchart
    e. Pseudo code
    f. Structured programming
    g. Object-Oriented Programming
    h. Aspect-Oriented Programming
    i. Characteristics of a good program
56. Give differences between the following:
    a. Flowchart and Pseudo code
    b. Algorithm, Flowchart and Pseudo code
    c. Modular Programming and Procedural Programming
    d. Selection and Iteration
    e. OOP and AOP
57. What is the relation between the 1′s complement and 2′s complement of a binary number?
58. In addition to the digits, a number may contain a ____ and ______.
59. What is a sign bit?
60. Which bit is considered as a sign bit when representing a number?
61. What is the value of sign bit for a positive number?
62. What is the value of sign bit for a negative number?
63. What is the range of data that can be represented using an 8-bit signed number?
64. What is the range of data that can be represented using an 8-bit unsigned number?
65. ______ representation and ________ representation are the two ways of representing the position of the binary point in the register.

Programming Exercise

1. Write the algorithm, draw a flowchart, and write pseudo code for the following:
   a. To find the sum of square root of any three numbers.
   b. To find the sum of first 100 integers.
   c. To find the sum of all even numbers till 100.
   d. To find the sum of all odd numbers till 100.
   e. To find the sum of any five integers.
   f. To find the factorial of a number n. Hint: $n! = n(n-1)(n-2)\ldots.3.2.1$
   g. To find the first n numbers in a Fibonacci series. Hint: $f(0) = 0$, $f(1) = 1$, $f(n) = f(n-1) + f(n-2)$
   h. To find the sum of digits of a number. (For example, for number 345 find 3+4+5)
   i. To check whether a number is prime or not.
   j. To convert the temperature from Fahrenheit to Celsius. Hint: C= (5/9)*(F-32)

This page is intentionally left blank

PART – II

BASICS OF C PROGRAMMING

This page is intentionally left blank

3

DATA TYPES, VARIABLES AND CONSTANTS

Learning Objectives

In this chapter, you will learn about:

- Various features of C language
- Various C's standards
- C's character set
- Identifiers and Keywords
- Rules to write identifier names in C
- Data types, type qualifiers and type modifiers
- Declaration statement
- Difference between declaration and definition
- Length and Range of various data types
- l-value and r-value concept
- Variables and constants
- Classification of constants
- Structure of a C program
- Process of compiling and executing a C program
- Writing simple C programs
- Using printf and scanf functions
- Use of sizeof operator

3.1 Introduction

C is a general-purpose, block-structured, procedural, case-sensitive, free-flow, portable and high-level programming language developed by Dennis Ritchie at the Bell Telephone Laboratories. The selection of 'C' as the name of a programming language seems to be an odd choice but it was named C because it evolved from earlier languages **B**asic **C**ombined **P**rogramming **L**anguage (**BCPL**) and **B**.

In 1967, Martin Richards developed BCPL for writing system software (i.e. operating systems and compilers). Ken Thompson in 1970 developed a stripped version of BCPL and named it B. The language B was used to create early versions of UNIX operating system. Both the languages BCPL and B were 'typeless', and every data object occupied one word in the memory. In 1972, Dennis Ritchie developed C programming language by retaining the important features of BCPL and B programming languages and adding data types and other powerful features to the retained feature set of BCPL and B. The language C was initially designed as a system implementation language for developing system software for the UNIX operating system. Thus, it was widely known as the development language of the UNIX operating system. However, after its popularity, it has spread over many other platforms and is used for creating many other applications in addition to the system software. Thus, nowadays, C is known as a general-purpose language and not only as a system implementation language.

3.2 C Standards

The rapid expansion of C to various platforms led to many variations that were similar but were often incompatible. This was a serious problem for programmers who wanted to develop code that could run on several platforms. This problem led to the realization of the need for a standard. This section lists the formulation of various C standards in the chronological order:

3.2.1 Kernighan & Ritchie (K&R) C Standard

The first edition of 'The C Programming Language' book by Brian Kernighan and Dennis Ritchie was published in 1978. This book was one of the most successful computer science books and has served as an informal standard for the C language for many years. This informal standard was known as '**K&R C**'.

3.2.2 ANSI C/Standard C/C89 Standard

In 1983, a technical committee was created under the American National Standards Institute (ANSI) committee to establish a standard specification of C. In 1989, the standard proposed by the committee was formally approved and is often referred to as **ANSI C**, **Standard C** or sometimes **C89**.

3.2.3 ISO C/C90 Standard

In 1990, the International Organization for Standardization (ISO) adopted the ANSI C standard after minor modifications. This version of the standard is called **ISO C** or sometimes **C90**.

3.2.4 C99 Standard

After the adoption of the ANSI standard, the C language specifications remained unchanged for sometime, whereas the language C++ continued to evolve. To accommodate this evolution of C++, a new standard of C language that corrected some details of ANSI C standard and added more extensive support to it was introduced in 1995. The standard was published in 1999 and is known as **C99**. The C99 standard has not been widely adopted and is not supported by many popular C compilers.

> The text and questions in this book are in accordance to ANSI/ISO standards and are tested on Borland **T**urbo **C** (TC) 3.0 compiler for DOS, Borland TC 4.5 compiler for Windows and Microsoft VC++ 6.0 compiler for Windows.

3.3 Learning Programming Language and Natural Language: An Analogy

Writing a C program is analogous to writing an essay. Recall all the stages through which you have undergone in the process of learning how to write an essay in English. Your teacher must have told you:

1. How to create words from letters.
2. How to form sentences using words and grammar.
3. How to organize sentences and create paragraphs.
4. How to arrange paragraphs and write an essay.

In this book, you will learn about:

1. How to create identifiers using the characters available in the character set of C language. This is analogous to creating words in a natural language.
2. How to use identifiers to form expressions, which can be further converted to statements, the smallest logical unit of a program. Forming a statement is analogous to forming a sentence.
3. How to use statements to write functions. Writing a function is analogous to writing a paragraph.
4. How to use functions to create a program. This is analogous to creating an essay from paragraphs.

The above learning objectives are organized in this book as follows:

1. Creating identifier names:	Chapter 1
2. Creating expressions and statements:	Chapters 4 and 5
3. Creating functions:	Chapter 8

Since, I do not want to restrain you from writing programs till Chapter 8, I will make some forward jumps in the flow of learning C programming language. I will introduce you to program writing in this chapter itself, but if something does not seem obvious, I advise you to be a bit patient. The concepts will be clearer when you go through the first few chapters and will be clear by the end of Chapter 8.

3.4 C Character Set

A **character set** defines the valid characters that can be used in a source program or interpreted when a program is running. The set of characters that can be used to write a source program is called a **source character set**, and the set of characters available when the program is being executed is called an **execution character set**. It is possible that the source character set is different from the execution character set, but in most of the implementations of C language, the two character sets are identical.

The basic source character set of C language includes:

1. Letters:
   a. Uppercase letters: A, B, C, …, Z
   b. Lowercase letters: a, b, c, …, z
2. Digits: 0, 1, 2, …, 9
3. Special characters: , . : ; ! ” ^ # % ^ & * ( ) { } [ ] < > | \ / _ ~ etc.
4. White space characters:
   a. Blank space character
   b. Horizontal tab space character
   c. Carriage return
   d. New line character
   e. Form feed character

3.5 Identifiers and Keywords

If you know C’s source character set, the next step is to write identifiers. This is analogous to writing words in a natural language.

3.5.1 Identifiers

An **identifier** refers to the name of an object. It can be a variable name, a label name, a function name, a typedef name, a macro name or a macro parameter, a tag or a member of a structure, a union or an enumeration.

The syntactic rules to write an identifier name in C are as follows:

1. Identifier name in C can have letters, digits or underscores.
2. The first character of an identifier name must be a letter (either uppercase or lowercase) or an underscore. The first character of an identifier name cannot be a digit.
3. No special character (except underscore), blank space and comma can be used in an identifier name.
4. Keywords or reserved words cannot form a valid identifier name.
5. The maximum number of characters allowed in an identifier name is compiler dependent, but the limit imposed by all the compilers provides enough flexibility to create meaningful identifier names.

The following identifier names are valid in C:

`Student_Name, StudentName, student_name, student1, _student`

The following identifier names are not valid in C:

Student Name (due to blank space), Name&Rollno (due to special character &), 1st_student (first character being a digit), for (`for` being a keyword).

> i It is always advisable to create meaningful identifier names. Meaningful identifier names are easier to read and increase the maintainability of a program. For example, it is better to create an identifier name as `student_name` instead of `snam`.

3.5.2 Keywords

Keyword is a reserved word that has a particular meaning in the programming language. The meaning of a keyword is predefined. A keyword cannot be used as an identifier name in C language. There are 32 keywords available in C. Table 3.1 gives a set of keywords present in C language.

Table 3.1 | List of keywords in C

S.No	Keyword	S.No	Keyword	S.No	Keyword	S.No	Keyword
1.	auto	9.	double	17.	int	25.	struct
2.	break	10.	else	18.	long	26.	switch
3.	case	11.	enum	19.	register	27.	typedef
4.	char	12.	extern	20.	return	28.	union
5.	const	13.	float	21.	short	29.	unsigned
6.	continue	14.	for	22.	signed	30.	void
7.	default	15.	goto	23.	sizeof	31.	volatile
8.	do	16.	if	24.	static	32.	while

3.6 Declaration Statement

If you have learnt how to create an identifier name, you should know that every identifier (except label name) needs to be declared before it is used.

An identifier can be declared by making use of the **declaration statement**. The **role** of a declaration statement is to introduce the name of an identifier along with its data type (or just type) to the compiler before its use. The general form of a declaration statement is:

[storage_class_specifier][type_qualifier[†]|type_modifier[‡]] **type[§] identifier** [=value[,....]]**;**

The terms enclosed within square brackets (i.e. []) are optional and might not be present in a declaration statement. The type, identifier and the terminating semicolon (shown in bold) are the mandatory parts of a declaration statement.

The following declaration statements[¶] are valid in C:

int variable;	(type int and identifier name variable present)
static int variable;	(Storage class specifier static, type int and identifier name variable present)
static unsigned int variable;	(Storage class specifier static, type modifier unsigned, type int and identifier name variable present)
static const unsigned int variable;	(Storage class specifier static, type qualifier const, type modifier unsigned, type int and identifier name variable present)
int variable=20;	(type int, identifier name variable and value 20 present)
int a=20, b=10;	(type int, identifier name a and its initial value 20 present, another identifier name b and its initial value 10 present✍)

[†] Refer Section 3.8.1 for a description on type qualifiers.

[‡] Refer Section 3.8.2 for a description on type modifiers.

[§] Refer Section 3.7 for a description on types.

[¶] These are actually definition statements. Refer Section 3.9 for a description on declaration and definition.

A declaration statement in which more than one identifier is declared is known as a **shorthand declaration statement**. For example, int a=20, b=10; is a shorthand declaration statement. The corresponding **longhand declaration statements** equivalent to this shorthand declaration statement are int a=20; int b=10;. It is important to note that shorthand declaration can only be used to declare identifiers of the same type. In no way can it be used to declare identifiers of different types, e.g. int a=10, float b=2.3; is an invalid statement.

3.7 Data Types

If you know how to write a declaration statement, you would probably know that the declaration statement is used to tell the data type (or just type) of an identifier to the compiler before its use.

Data type or just **type** is one of the most important attributes of an identifier. It determines the possible values that an identifier can have and the valid operations that can be applied on it.

In C language, data types are broadly classified as:

1. Basic data types (primitive data types)
2. Derived data types
3. User-defined data types

3.7.1 Basic/Primitive Data Types

The five basic data types and their corresponding keywords available in C are:

1. Character (char)
2. Integer (int)
3. Single-precision floating point (float)
4. Double-precision floating point (double)
5. No value available (void)

3.7.2 Derived Data Types

These data types are derived from the basic data types. Derived data types available in C are:

1. Array type e.g. char[ ], int[ ], etc.
2. Pointer type e.g. char*, int*, etc.
3. Function type e.g. int(int,int), float(int), etc.

3.7.3 User-defined Data Types

The C language provides flexibility to the user to create new data types. These newly created data types are called **user-defined data types**. The user-defined data types in C can be created by using:

1. Structure
2. Union
3. Enumeration

3.8 Type Qualifiers and Type Modifiers

The declaration statement can optionally have type qualifiers or type modifiers or both.

3.8.1 Type Qualifiers

A **type qualifier** neither affects the range of values nor the arithmetic properties of the declared object. They are used to indicate the special properties of data within an object. Two type qualifiers available in C are:

1. **const**[††] **qualifier:** Declaring an object const announces that its value will not be changed during the execution of a program.
2. **volatile qualifier:** volatile qualifier announces that the object has some special properties relevant to optimization.

3.8.2 Type Modifiers

A **type modifier** modifies the base type to yield a new type. It modifies the range[‡‡] and the arithmetic properties of the base type. The type modifiers and the corresponding keywords available in C are:

1. Signed (signed)
2. Unsigned (unsigned)
3. Short (short)
4. Long (long)

3.9 Difference Between Declaration and Definition

It is very important to know the difference between the terms **declaration** and **definition**. **Declaration** only introduces the name of an identifier along with its type to the compiler before it is used. During declaration, no memory space is allocated to an identifier. **Definition** of an identifier means the declaration of an identifier plus reservation of space for it in the memory. The amount of memory space reserved for an identifier depends upon the data type of the identifier. Identifiers of different data types take different amounts of memory space. The memory space required by an identifier also depends upon the compiler and the working environment used. Table 3.2 lists the length of various data types in DOS and Windows environment.

[††] Refer Section 3.11.2.2 for a description on const qualifier.

[‡‡] Refer Section 3.9 for a description on range modification by type modifiers.

Table 3.2 | Data types and their memory requirements

S.No	Data type	Base/Modified	TURBO C 3.0/DOS	MS VC++ 6.0/WINDOWS
1.	char	Base	1 Byte	1 Byte
2.	int	Base	2 Bytes	4 Bytes
3.	float	Base	4 Bytes	4 Bytes
4.	double	Base	8 Bytes	8 Bytes
5.	signed ⟨data type 1, 2⟩	Modified	⟨same as data type 1, 2⟩	⟨same as data type 1, 2⟩
6.	unsigned ⟨data type 1, 2⟩	Modified	⟨same as data type 1, 2⟩	⟨same as data type 1, 2⟩
7.	short int	Modified	2 Bytes	2 Bytes
8.	long int	Modified	4 Bytes	4 Bytes
9.	long float	Modified	8 Bytes	8 Bytes
10.	long double	Modified	10 Bytes	8 Bytes
11.	void	Base	Object of void type cannot be created	

The data type determines the possible values that an identifier can have. The range of a data type depends upon the length of the data type. Table 3.3 lists the range of various data types in DOS and Windows environment.

Table 3.3 | Range of various data types

S.No	Data type	TURBO C 3.0/DOS	MS VC++ 6.0/WINDOWS
1.	char	–128 to 127	–128 to 127
2.	int	–32768 to 32767	–2,147,483,648 to 2,147,483,647
3.	float	$3.4 * 10^{-38}$ to $3.4 * 10^{38}$	$3.4 * 10^{-38}$ to $3.4 * 10^{38}$
4.	double	$1.7 * 10^{-308}$ to $1.7 * 10^{308}$	$1.7 * 10^{-308}$ to $1.7 * 10^{308}$
5.	signed ⟨data type 1, 2⟩	Same as 1, 2 as by default data types are signed	Same as 1, 2 as by default data types are signed
6.	unsigned char	0 to 255	0 to 255
7.	unsigned int	0 to 65535	0 to 4,294,967,295
8.	unsigned long int	0 to 4,294,967,295	0 to 4,294,967,295
9.	short int	–32768 to 32767	–32768 to 32767
10.	long double	$3.4 * 10^{-4932}$ to $1.1 * 10^{4932}$	$1.7 * 10^{-308}$ to $1.7 * 10^{308}$

Despite the big difference between the terms declaration and definition, the word declaration is commonly used in place of definition. All the statements written in Section 3.6 are actually definition statements, but I have referred to them as declarations because at that point I just wanted to focus on the name and the type of an identifier.

The statement int variable=20; mentioned in Section 3.6 is actually a definition statement because it allocates 2 bytes (or 4 bytes) to variable somewhere in the memory (say, at memory location with address 2000) and initializes it with the value 20. The memory allocation is purely random (i.e. any free memory location will be randomly allocated). This is illustrated in Figure 3.1.

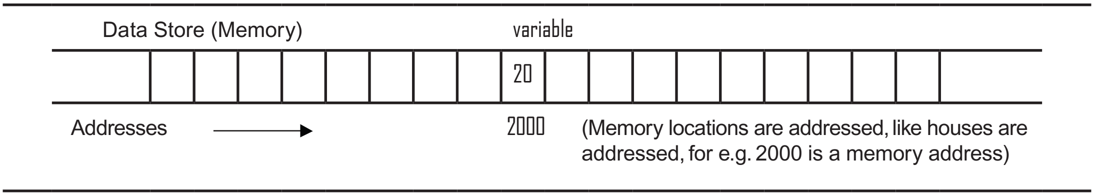

Data Store (Memory)
variable
20
Addresses
2000
(Memory locations are addressed, like houses are addressed, for e.g. 2000 is a memory address)

Figure 3.1 | Allocation of memory to variable

If int variable; is a definition statement, then how can I declare variable?

If you want to actually declare variable, write extern int variable;. extern is a storage class specifier. The keyword extern provides a method for declaring a variable without defining it. The extern declaration does not allocate the memory.

3.10 Data Object, L-value and R-value

You must have known by this time that upon definition, an identifier is allocated some space in memory depending upon its data type and the working environment. This memory allocation gives rise to two important concepts known as the **l-value concept** and the **r-value concept**. These concepts are described below.

3.10.1 Data Object

Data object is a term that is used to specify the region of data storage that is used to hold values. Once an identifier is allocated memory space, it will be known as a data object.

3.10.2 L-value

L-value is a data object locator. It is an expression that locates an object. In Figure 3.1, variable is a sort of name given to the memory location 2000. variable here refers to l-value,✍ an object locator. The term l-value can be further categorized as:

1. **Modifiable l-value:** A modifiable l-value is an expression that refers to an object that can be accessed and legally changed in the memory.
2. **Non-modifiable l-value:** A non-modifiable l-value refers to an object that can be accessed but cannot be changed in the memory. [¶¶]

✍ l in **l-value** stands for '**left**'; this means that the l-value could legally stand on the left side of an assignment operator.

3.10.3 R-value

R-value refers to 'read value'. In Figure 3.1, variable has an r-value✍ 20.

[¶¶] Refer Section 3.11.2.2 to learn how to make an l-value non-modifiable.

✍ **r** in **r-value** stands for '**right**' or '**read**'; this means that if an identifier name appears on the right side of an assignment operator it refers to the r-value.

Consider Figure 3.1 and the expression variable=**variable**+20. variable on the left side of the assignment operator refers to the l-value. **variable** on the right side of the assignment operator (in bold) refers to the r-value. **variable** appearing on the right side refers to 20. The number 20 is added to 20 and the value of expression is 40 (r-value). This outcome (40) is assigned to variable on the left side of the assignment operator, which signifies l-value.✍ The l-value variable locates the memory location where this value is to be placed, i.e. at 2000. After the evaluation of the expression variable=**variable**+20, the contents of the memory are shown in Figure 3.2.

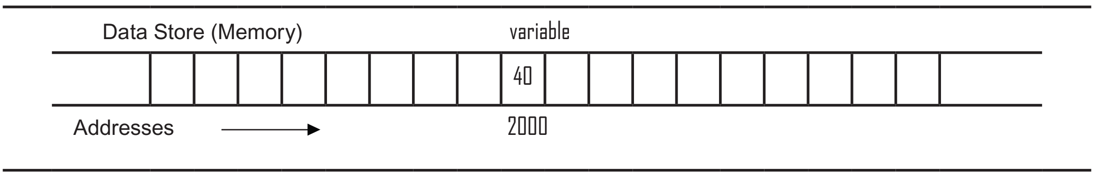


Figure 3.2 | Contents of memory location 2000 after the evaluation of expression variable=**variable**+20

✍ **Remember it as:**
The **l-value** refers to the location value, i.e. the location of the object, and the **r-value** refers to the read value, i.e. the value of the object.

3.11 Variables and Constants

Variables and **constants** are two most commonly used terms in a programming language.

3.11.1 Variables

A **variable** is an entity whose value can vary (i.e. change) during the execution of a program. The value of a variable can be changed because it has a modifiable l-value. Since it has a modifiable l-value, it can be placed on the left side of the assignment operator. Note that only the entities that have modifiable l-values can be placed on the left side of the assignment operator. The variable can also be placed on the right side of the assignment operator. Hence, it has an r-value too. Thus, a variable has both an l-value and an r-value.

3.11.2 Constants

A **constant** is an entity whose value remains the same throughout the execution of a program. It cannot be placed on the left side of the assignment operator because it does not have a modifiable l-value. It can only be placed on the right side of the assignment operator. Thus, a constant has an r-value only. Constants are classified as:

1. Literal constants
2. Qualified constants
3. Symbolic constants

3.11.2.1 Literal Constant

Literal constant or just **literal** denotes a fixed value, which may be an integer, floating point number, character or a string. The type of literal constant is determined by its value. Literal constants are of the following types:

1. Integer literal constant
2. Floating point literal constant
3. Character literal constant
4. String literal constant

3.11.2.1.1 Integer Literal Constant

Integer literal constants are integer values like -1, 2, 8, etc. The rules for writing integer literal constants are as follows:

1. An integer literal constant must have at least one digit.
2. It should not have any decimal point.
3. It can be either positive or negative. If no sign precedes an integer literal constant, then it is assumed to be positive.
4. No special characters (even underscore) and blank spaces are allowed within an integer literal constant.
5. If an integer literal constant starts with 0, then it is assumed to be in an octal number system, e.g. 023 is a valid integer literal constant, which means 23 is in an octal number system and is equivalent to 19 in the decimal number system.
6. If an integer literal constant starts with 0x or 0X, then it is assumed to be in a hexadecimal number system, e.g. 0x23 or 0X23 is a valid integer literal constant, which means 23 is in a hexadecimal number system and is equivalent to 35 in the decimal number system.
7. The size of the integer literal constant can be modified by using a length modifier. The length modifier can be a suffix character l, L, u, U, f or F. If the integer literal constant is terminated with l or L then it is assumed to be long. If it is terminated with u or U, then it is assumed to be an unsigned integer, e.g. 23l is a long integer and 23u is an unsigned integer. The length modifier f or F can only be used with a floating point literal constant and not with an integer literal constant.

3.11.2.1.2 Floating Point Literal Constant

Floating point literal constants are values like -23.1, 12.8, -1.8e12, etc. Floating point literal constants can be written in a **fractional form** or in an **exponential form**. The rules for writing floating point literal constants in a fractional form are as follows:

1. A fractional floating point literal constant must have at least one digit.
2. It should have a decimal point.
3. It can be either positive or negative. If no sign precedes a floating point literal constant, then it is assumed to be positive.

4. No special characters (even underscore) and blank spaces are allowed within a floating point literal constant.
5. A floating point literal constant by default is assumed to be of type double, e.g. the type of 23.45 is double.
6. The size of the floating point literal constant can be modified by using the length modifier f or F, i.e. if 23.45 is written as 23.45f or 23.45F, then it is considered to be of type float instead of double.

The following are valid floating point literal constants in a fractional form:

−2.5, 12.523, 2.5f, 12.5F

The rules for writing floating point literal constants in an exponential form are as follows:

1. A floating point literal constant in an exponential form has two parts: the mantissa part and the exponent part. Both parts are separated by e or E.
2. The mantissa can be either positive or negative. The default sign is positive.
3. The mantissa part should have at least one digit.
4. The mantissa part can have a decimal point but it is not mandatory.
5. The exponent part must have at least one digit. It can be either positive or negative. The default sign is positive.
6. The exponent part cannot have a decimal point.
7. No special characters (even underscore) and blank spaces are allowed within the mantissa part and the exponent part.

The following are valid floating point literal constants in the exponential form:

−2.5E12, −2.5e−12, 2e10 (i.e. equivalent to 2×10^{10})

3.11.2.1.3 Character Literal Constant

A **character literal constant** can have one or at most two characters enclosed within single quotes e.g. 'A', 'a', '\n', etc. Character literal constants are classified as:

1. Printable character literal constants
2. Non-printable character literal constants

3.11.2.1.3.1 Printable Character Literal Constant

All characters of source character set except quotation mark, backslash and new line character when enclosed within single quotes form a **printable character literal constant**. The following are examples of printable character literal constants: 'A', '#', '6'.

3.11.2.1.3.2 Non-printable Character Literal Constant

Non-printable character literal constants are represented with the help of **escape sequences**. An escape sequence consists of a backward slash (i.e. \) followed by a character and both enclosed within single quotes. An escape sequence is treated as a single character. It can be used[§§] in a string like any other printable character. A list of the escape sequences available in C is given in Table 3.4.

[§§] Refer Programs 3-7 and 3-9 for learning the usage of the escape sequences '\t' and '\n'.

Table 3.4 | List of escape sequences

S.No	Escape sequence	Character value	Action on output device
1.	\'	Single quotation mark	Prints '
2.	\"	Double quotation mark (")	Prints "
3.	\?	Question mark (?)	Prints ?
4.	\\	Backslash character (\)	Prints \
5.	\a	Alert	Alerts by generating a beep
6.	\b	Backspace	Moves the cursor one position to the left of its current position
7.	\f	Form feed	Moves the cursor to the beginning of next page
8.	\n	New line	Moves the cursor to the beginning of the next line
9.	\r	Carriage return	Moves the cursor to the beginning of the current line
10.	\t	Horizontal tab	Moves the cursor to the next horizontal tab stop
11.	\v	Vertical tab	Vertical tab
12.	\0	Null character	Prints nothing

3.11.2.1.4 String Literal Constant

A string literal constant consists of a sequence of characters (possibly an escape sequence) enclosed within double quotes. Each string literal constant is implicitly terminated by a null character (i.e. '\0'). Hence, the number of bytes occupied by a string literal constant is one more than the number of characters present in the string. The additional byte is occupied by the terminating null character. Thus, the empty string (i.e. "") occupies one byte in the memory due to the presence of the terminating null character. However, the terminating null character is not counted while determining the length of a string. Therefore, the length of string "ABC" is 3 although it occupies 4 bytes in the memory.

3.11.2.2 Qualified Constants

Qualified constants are created by using const qualifier. The following statement creates a qualified character constant named a:

```
const char a='A';
```

Consider a definition statement int a=10;. This statement allocates 2 bytes (or 4 bytes, in case of Windows environment) to a somewhere in the memory and initializes it with the value 10. The memory location can be thought of as a transparent box in which 10 has been placed. It is possible to modify the value of a. This means that it is possible to open the box and change the value placed in it. Now, consider the statement const int a=10;. The usage of the const qualifier places a lock on the box after placing the value 10 in it. Since the box is transparent, it is possible to see (i.e. read) the value placed within the box, but it is not possible to modify the value within the box as it is locked. This is depicted in Figure 3.3.

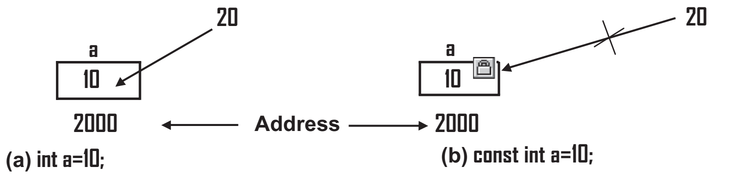


Figure 3.3 | Use of const qualifier

Since qualified constants are placed in the memory, they have l-value. However, as it is not possible to modify them, this means that they do not have a modifiable l-value, i.e. **they have a non-modifiable l-value.**

3.11.2.3 Symbolic Constants

Symbolic constants are created with the help of the define preprocessor directive. For example: #define PI 3.14124 defines PI as a symbolic constant with value 3.14124. Each symbolic constant is replaced by its actual value during the preprocessing stage.

3.12 Structure of a C Program

In general, a C program is composed of the following sections:

1. Section 1: **Preprocessor directives**
2. Section 2: **Global declarations**
3. Section 3: **Functions**

Sections 1 and 2 are optional, i.e. they may or may not be present in a C program but Section 3 is mandatory. Section 3 should always be present in a C program. Thus, it can be said that '**A C program is made up of functions**'. Look at the simple program in Program 3-1.

Prog 3-1.c

```
//Comment: First C program
#include<stdio.h>
main()
{
    printf("Hello Readers!!");
}
```

Output window

```
Hello Readers!!
```

Program 3-1 | A simple program that prints "Hello Readers!!"

Program 3-1 on execution[†††] outputs Hello Readers!!. The contents of Program 3-1 are described below.

3.12.1 Comments

Line 1: is a comment. **Comments** are used to convey a message and to increase the readability of a program. They are not processed by the compiler. There are two types of comments:

1. Single-line comment
2. Multi-line comment

3.12.1.1 Single-line Comment

A **single-line comment** starts with two forward slashes (i.e. //) and is automatically terminated with the end of line. Line 1 of Program 3-1 is a single-line comment.

3.12.1.2 Multi-line Comment

A **multi-line comment** starts with /* and terminates with */. A multi-line comment is used when multiple lines of text are to be commented.

3.12.2 Section1: Preprocessor Directive Section

Line 2: #include<stdio.h> is a preprocessor directive statement. The **preprocessor directive section** is optional but you will find it in most of the C programs. In the initial phase of learning, just remember that #include<stdio.h> is a preprocessor directive statement, which includes **st**an**d**ard **i**nput/**o**utput (i.e. **stdio**) **h**eader (.h) file. This file is to be included if standard input/output functions like printf or scanf are to be used in a program.

The following points must be remembered while writing preprocessor directives:

1. The preprocessor directive always starts with a pound symbol (i.e. #).
2. The pound symbol # should be the first non-white space character in a line.
3. The preprocessor directive is terminated with a new line character and not with a semi-colon.
4. Preprocessor directives are executed before the compiler compiles the source code. These will change the source code, usually to suit the operating environment (pragma directive) or to add the code (include directive) that will be required by the calls to library functions.

3.12.3 Section 2: Global Declaration Section

The **global declaration section** is optional. This section is not present in Program 3-1. In the initial phase of learning, I am not going to use global declarations.

[†††] Refer Section 3.13 to learn how to execute a C program.

3.12.4 Section 3: Functions Section

This section is mandatory and must be present in a C program. This section can have one or more functions. A function named main is always required. The functions section (Lines 3–6) in Program 3-1 consists of only one function, i.e. main function. Every function consists of two parts:

1. Header of the function
2. Body of the function

3.12.4.1 Header of a Function

The general form of the header of a function is

```
[return_type] function_name([argument_list])
```

The terms enclosed within square brackets are optional and might not be present in the function header. Since the name of a function is an identifier name, all the rules discussed in Section 3.5.1 for writing an identifier name are applicable for writing the function name. Line 3 in Program 3-1 specifies the header of the function main, in which the return_type and the argument_list are not present. The name of the function is main and it is a valid identifier name. In the initial phase of learning, I will write functions without specifying a return type and an argument list.

> *i* Writing a function without specifying a return type may lead to the generation of a warning message during the compilation but we can ignore it for the time being.

3.12.4.2 Body of a Function

The body of a function consists of a set of statements enclosed within curly brackets commonly known as **braces**. Lines 4–6 in Program 3-1 form the body of main function. The body of a function consists of a set of statements. Statements are of two types:

1. Non-executable statements: For example: declaration statement
2. Executable statements: For example: printf function call statement

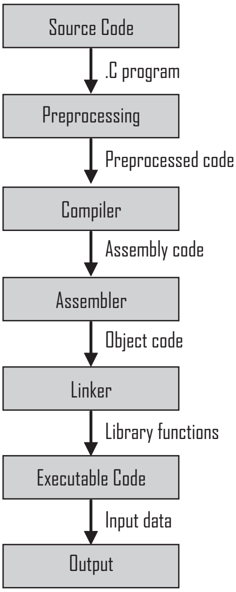
It is possible that no statement is present within the braces. In such a case, the program produces no output on execution. However, if there are statements written within the braces, remember that non-executable statements can only come prior to an executable statement, i.e. first non-executable statements are written and then executable statements are written. The body of main function in Program 3-1 has only one executable statement, i.e. printf function call statement.

3.13 Executing a C Program

If you have finished writing the code listed in Program 3-1, follow these steps to execute your program:

1. **Save program:** with .c extension. This will help you in retrieving the code in case the program crashes upon execution.
2. **Compile program:** Compilation can be done by going to the Compile Menu of Borland TC 3.0 and invoking the compile option available in that menu. The shortcut for this step is the Alt+F9 key. If working with Borland Turbo C 4.5, go to the Project Menu and invoke the compile option. It has the same shortcut key. In Microsoft Visual C++ 6.0, go to the Build Menu and invoke the compile option. The shortcut for this is the Ctrl+F7 key. After the compilation, look for errors and warnings. Warnings will not prevent you from executing the program and if there are any, just ignore them for the time being. If there are errors, check that you have written the code properly. There should be no typing mistake and all the characters listed in Program 3-1 should be present as such. If there is no error, Congrats!! you can now execute your program.
3. **Execute/run program:** Execution can be done by going to the Run Menu and invoking the run option in Borland Turbo C 3.0. The shortcut key is Ctrl+F9. In Borland Turbo C 4.5, the program can be executed by going to the Debug Menu and invoking the run option. It has the same shortcut key. In Microsoft Visual C++ 6.0, go to the Build Menu and invoke the run option. The shortcut key for this is Ctrl+F5.
4. **See the output:** If working with Borland Turbo C 3.0, to see the output go to the user screen. This can be done by going to the Window Menu and invoking the user screen option. The shortcut for this step is Alt+F5. In Borland TC 4.5 and Microsoft Visual C++ 6.0, the output screen will automatically pop-up.

3.14 Compilation and Linking process



Flowchart depicting the compilation and linking process

The steps in the execution of a C program are as follows:

1. Write the program (source code).
2. Preprocessing is the first stage of the compilation process. The preprocessor accepts source code as input and interprets preprocessor directives denoted by #. It removes comments and empty lines in the program.
3. The C compiler translates the source code into assembly code (machine understandable code).
4. The assembler creates the object code.
5. The linker combines the source code with the library functions referred within it or functions defined in other source files along with main(), to create an executable file. External variable references are resolved here.
6. Executes the program by giving data input.

3.15 More Programs for Startup

If you have successfully executed Program 3-1 and have gained some confidence, look at some more programs (Programs 3-2 to 3-11). Type the programs as such and compile them. If there are errors, find out the errors and rectify them. After rectification, recompile the programs and execute them to get a practical feel of all the concepts that we have discussed till now.

Line	Prog 3-2.c	Output window
1	`//Comment: Case Sensitivity`	Linker error
2	`#include<stdio.h>`	**Reasons:**
3	`Main()`	• C Language is case sensitive
4	`{`	• Main is not same as main
5	`    int valid_name=20;`	**What to do?**
6	`    printf("%d", valid_name);`	• Replace Main by main in line 3 and then recheck
7	`}`	

Program 3-2 | A program that emphasizes the case sensitivity of C language

Line	Prog 3-3.c	Output window
1	`//Comment: Identifier`	Compilation error
2	`#include<stdio.h>`	**Reason:**
3	`main()`	• 1st_student is not a valid identifier name
4	`{`	**What to do?**
5	`    int 1st_student=20;`	• Replace it everywhere by student1 and then recheck
6	`    printf("%d", 1st_student);`	
7	`}`	

Program 3-3 | A program that emphasizes the rules to write an identifier name

Line	Prog 3-4.c	Output window

1
2
3
4
5
6
7
	//Comment: Keyword
	#include<stdio.h>
	main()
	{
	    int if=20;
	    printf("%d", if);
	}
		Compilation error
		Reason:
		• if is a keyword. It cannot be used as an identifier name
		What to do?
		• Replace it everywhere by a valid identifier name and then recheck

Program 3-4 | A program that emphasizes the fact that keyword is not a valid identifier name

Line	Prog 3-5.c	Output window

1
2
3
4
5
6
7
	//Comment: Semicolon is Terminator
	#include<stdio.h>
	main()
	{
	    int valid_name=20
	    printf("%d", valid_name);
	}
		Compilation error
		Reasons:
		• A statement in C is terminated with a semicolon
		• In line 5, declaration (actually definition) statement is not terminated with a semicolon. This leads to the compilation error
		What to do?
		• Place semicolon at end of line 5 and then recheck

Program 3-5 | A program that emphasizes the fact that statements in C are terminated with a semicolon

Line	Prog 3-6.c	Output window

1
2
3
4
5
6
7
	//Comment: printf function use
	#include<stdio.h>
	main()
	{
	    int valid_name=20;
	    printf("The value is %d", valid_name);
	}
		The value is 20

Program 3-6 | A program that illustrates the use of printf function to print the value of an identifier

Program 3-6 upon execution outputs The value is 20. The definition statement in line 5 defines an identifier valid_name and initializes it with the value 20. This value is printed with the help of printf function in line 6. The rules for using printf function are as follows:

1. The name of printf function should be in lowercase.
2. The inputs (or arguments) to printf function are given within round or circular brackets, popularly called **parentheses**.
3. At least one input is required, and the first input to printf function should always be a string literal or an identifier of type char*.
4. The inputs are separated by commas.
5. If values of identifiers are to be printed with the help of printf function, the first input to printf function should be a **format string**. For example, in Program 3-6, in line 6, "The value is %d" is a format string. A **format string** consists of **format specifiers**. For example, line 6

in Program 3-6 consists of a format specifier %d. A **format specifier** specifies the format according to which the printing will be done. There is a different format specifier for each data type. Format specifier is written as %x, where x is a character code listed in Table 3.5.

Table 3.5 | Format specifiers in C language

S.No	Data type	x	Format specifier	Remark
1.	char	c	%c	Single character
2.	int	i	%i	Signed integer
3.	int	d	%d	Signed integer in decimal number system
4.	unsigned int	o	%o	Unsigned integer in octal number system
5.	unsigned int	u	%u	Unsigned integer in decimal number system
6.	unsigned int	x	%x	Unsigned integer in hexadecimal number system
7.	unsigned int	X	%X	Unsigned integer in hexadecimal number system
8.	long int	ld	%ld	Signed long
9.	short int	hd	%hd	Signed short
10.	unsigned long	lu	%lu	Unsigned long
11.	unsigned short	hu	%hu	Unsigned short
12.	float	f	%f	Signed single precision float in form of [-]dddd.dddd e.g. 22.25, −12.34
13.	float	e	%e	Singed single precision float in form of [-]d.dddde[+/-]ddd e.g. −2.3e4, 2.25e−2
14.	float	E	%E	Same as %e, with E for exponent
15.	float	g	%g	Singed value in either e or f form, based on given value and precision
16.	float	G	%G	Same as %g, with E for exponent if e format is used
17.	double	lf	%lf	Signed double-precision float
18.	String type	s	%s	String
19.	Pointer type	p	%p	Pointer

Line | **Prog 3-7.c**

```
1  //Comment: scanf function use
2  #include<stdio.h>
3  main()
4  {
5      int number;
6      printf("Enter number\t");
7      scanf("%d",&number);
8      printf("The number entered is %d",number);
9  }
```

Output window

```
Enter number    12
The number entered is 12
```

Remarks:

- '\t' present in line 6 is an escape sequence and is used to create tab-spacing
- Observe the tab-space between the string "Enter number" and the value 12 in the output window

Program 3-7 | A program that illustrates the use of scanf function

Program 3-7 upon execution prompts the user to enter a value of number. In response, the user enters the value 12. The entered value is then printed by the printf function. The scanf function is used to take the input just like the printf function is used to print the output. The rules for using scanf function are as follows:

1. The name of scanf function should be in lowercase.
2. The inputs (or arguments) to scanf function are given within parentheses.
3. The first input to scanf function should always be a format string or an identifier of type char*. Ideally, the format string of a scanf function should only consist of blank separated format specifiers.
4. The inputs are separated by commas.
5. The inputs following the first input should denote l-values. For example, in line 7 of Program 3-7, the second input is &number. The symbol & is address-of operator and is used to find the l-value of its operand. Thus, &number refers to the l-value.

The scanf function takes inputs from the user according to the available format specifiers in the specified format string and stores the entered values at the specified l-values. Thus, the scanf function specified in line 7 of Program 3-7 takes an integer value (due to %d format specifier) and stores it at the l-value (i.e. &number).

Line	Prog 3-8.c	Output window
1	//Comment: Add two numbers	Enter numbers 12 13
2	#include<stdio.h>	The sum is 25
3	main()	
4	{	
5	int number1, number2, number3;	
6	printf("Enter numbers\t");	
7	scanf("%d %d",&number1, &number2);	
8	number3 = number1+number2;	
9	printf("The sum is %d",number3);	
10	}	

Program 3-8 | A program to add two numbers entered by the user

Line	Prog 3-9.c	Output window
1	//Comment: Swap two numbers	Enter numbers 12 13
2	#include<stdio.h>	Numbers before swap 12 13
3	main()	Numbers after swap 13 12
4	{	
5	int number1, number2, number3;	

(*Contd...*)

```
 6      printf("Enter numbers\t");
 7      scanf("%d %d",&number1, &number2);
 8      printf("Numbers before swap %d %d\n",number1, number2);
 9      number3=number1;
10      number1=number2;
11      number2=number3;
12      printf("Numbers after swap %d %d\n",number1, number2);
13  }
```

Remark:

- '\n' present in line 8 is an escape sequence and is used to place a new line character in the output

Program 3-9 | A program to swap two numbers

Line | **Prog 3-10.c**

```
//Comment: Swap two numbers without using a third number
#include<stdio.h>
main()
{
    int number1, number2;
    printf("Enter numbers\t");
    scanf("%d %d",&number1, &number2);
    printf("Numbers before swap %d %d\n",number1, number2);
    number2=number1+number2;
    number1=number2-number1;
    number2=number2-number1;
    printf("Numbers after swap %d %d\n",number1, number2);
}
```

Output window

```
Enter numbers    12 13
Numbers before swap 12 13
Numbers after swap 13 12
```

Program 3-10 | A program to swap two numbers without using a third number

Line | **Prog 3-11.c**

```
//Comment: Usage of sizeof operator
#include<stdio.h>
main()
{
    printf("Character takes %d byte in memory\n", sizeof(char));
    printf("Integer takes %d bytes in memory\n", sizeof(int));
    printf("Float takes %d bytes in memory\n", sizeof(float));
    printf("Long takes %d bytes in memory\n", sizeof(long));
    printf("Double takes %d bytes in memory\n", sizeof(double));
}
```

Output window

```
Character takes 1 byte in memory
Integer takes 2 bytes in memory
Float takes 4 bytes in memory
Long takes 4 bytes in memory
Double takes 8 bytes in memory
```

Remark:

- The output of the program may vary with the compiler and the working environment

Program 3-11 | A program to find the size of various data types

Program 3-11 makes the use of sizeof operator to find the size of data types. The specified output is the result of execution using Borland Turbo C 3.0/4.5. If it is executed using MS VC++ 6.0, the size of integer would be 4 bytes.

3.16 Summary

1. C is a general-purpose, block-structured, procedural, case-sensitive, free-flow, portable, high-level language.
2. There are various C standards: Kernighan & Ritchie (K&R) C standard; ANSI C/Standard C/C89 standard; ISO C/C90 standard; C99 standard.
3. ANSI C and ISO C are the most popular C standards. Most popular compilers nowadays are ANSI compliant.
4. C character set consists of letters, digits, special characters and white space characters.
5. Identifier refers to the name of an object. It can be a variable name, a label name, a typedef name, a macro name, name of a structure, a union or an enumeration.
6. Keyword cannot form a valid identifier name. The meaning of keyword is predefined and cannot be changed.
7. Every identifier (except label name) needs to be declared before its use. They can be declared by using a declaration statement.
8. The declaration statement introduces the name of an identifier along with its data type to the compiler before its use.
9. Data types are categorized as: basic data types, derived data types and user-defined data types.
10. The declaration statement can optionally have type qualifiers or type modifiers or both.
11. A type qualifier does not modify the type.
12. A type modifier modifies the base type to yield a new type.
13. Declaration is different from definition in the sense that definition in addition to declaration allocates the memory to an identifier.
14. Variables have both l-value and r-value.
15. Constants do not have a modifiable l-value. They have an r-value only.
16. C program is made up of functions.
17. C program should have at least one function. A function named `main` is always required.

Exercise Questions

Conceptual Questions and Answers

1. *What method is adopted for locating includable source files in ANSI specifications?*

   For including source files, `include` directive is used. The `include` directive can be used in two forms:
   `#include<name-of-file>`
   or
   `#include"name-of-file"`
   `#include<name-of-file>` searches the prespecified list of directories (names of include directories can be specified in IDE✍ settings) for the source file (whose name is given within angular brackets), and text embeds the entire content of the source file in place of itself. If the file is not found there, it will show an error ‘Unable to include ‘name-of-file’’.

#include“name-of-file” searches the file first in the current working directory. If this search is not supported or if the search fails, this directive is reprocessed as if it reads #include<name-of-file>, i.e. search will be carried out in the prespecified list of directories. If the search still fails, it will show the error ‘Unable to include ‘name-of-file’’.

IDE stands for **I**ntegrated **D**evelopment **E**nvironment. All the tools (like text editor, preprocessor, compiler and linker) required for developing programs are integrated into one package, known as IDE.

2. *Is there any difference that arises if double quotes, instead of angular brackets are used for including standard header files?*

   If double quotes instead of angular brackets are used for the inclusion of standard header files, the search space unnecessarily increases (because in addition to the prespecified list of directories, the search will unnecessarily be carried out first in the current working directory) and thus, the time required for the inclusion will be more.

3. *Under what circumstances should the use of quotes be preferred over the use of angular brackets for the inclusion of header files, and under what circumstances is the use of angular brackets beneficial?*

   Self-created or user-defined header files should be included with double quotes because inclusion with double quotes makes files to be searched first in the current working directory (where the user has kept self-created header files) and then in the prespecified list of directories. If standard header files are to be included, angular brackets should be used because the standard header files are present in the prespecified list of directories and there is no use of searching them in the current working directory. Usage of double quotes for including standard header files will also work, but will take more time.

4. *‘C is a case-sensitive language’. Therefore, does it create any difference if instead of* #include<stdio.h>, #include<STDIO.H> *is used? If no, why?*

   ‘C is a case-sensitive language’ means that the C constructs are case sensitive (i.e. depends upon whether uppercase (like A) or lowercase (like a) is used). The name of the source file specified for inclusion is not a C construct. Whether it will be case sensitive or not depends upon the working environment. In case of DOS and Windows environment, file names are case insensitive. In Unix and Linux environment, file names are case sensitive. So, if working in DOS or Windows environment, <STDIO.H> can be used instead of <stdio.h>, it does not create any difference. But, in case of Unix or Linux environment, it does create a difference.

5. *A program file contains the following five lines of the source code:*

   ```
   #include<stdio.h>
   main()
   {
       printf(“Hello World”);
   }
   ```

   When the program is compiled, the compiler shows the number of lines compiled to be greater than 5, why it is happening so?

   During the preprocessing stage, include preprocessor directive (the first line of source code) searches the file stdio.h in the prespecified list of directories and if the header file is found, it (the include directive) is replaced by the entire content of the header file. If the included header file contains another include directive, it will also be processed. This processing is carried out recursively till either no include directive remains or till maximum translation limit is achieved (ISO specifies the nesting level of include files to be at most 15). Hence, one line of source code gets replaced by multiple lines of the header file. During the compilation stage, these added lines will also be compiled; hence, the compiler shows the number of lines compiled to be greater than five.

6. *Is `int a;` actually a declaration or a definition?*

   The role of the declaration statement is to introduce the name of an identifier along with its data type (or just type) to the compiler before its use. During the declaration, no memory space is allocated to an identifier. Since `int a;` statement in addition to introducing the name and the type of identifier `a`, allocates memory to `a`, it actually becomes a definition.

7. *How are negative integral numbers stored in C?*

   Internally, numbers are stored in the form of bits (i.e. **b**inary dig**its**) and are represented in the binary number system. In the binary number system, negative numbers are not stored directly. To store both the sign and magnitude of a number, some convention for storage has to be used. In C language, the convention used for storing an integral✍ number is **sign-two's complement representation.**

   What is sign-two's complement representation?

   1. For every integral number, the **M**ost **S**ignificant **B**it (MSB) contains the sign, and the rest of the bits contain the magnitude.
   2. If the sign is positive, the MSB is 0 and if the sign is negative, the MSB is 1.
   3. If the MSB contains bit 0 (i.e. a positive number), the magnitude is in the direct binary representation.
   4. If the MSB contains bit 1 (i.e. a negative number), the magnitude is not in the direct binary representation. The magnitude is stored in two's complement form. To get the value of the magnitude, take two's complement of the stored magnitude.

      How to find two's complement of a binary number?

      Two's complement of a binary number is its one's complement plus one.

      One's complement of a binary number can be determined by negating every bit (i.e. by converting 0's to 1's and 1's to 0's). For e.g. One's complement of 100101 is 011010 (i.e. every bit is negated). Two's complement of 100101 is its one's complement plus one (i.e. 011010 + 1 = 011011). The following tables show how 200 and −200 are stored in memory:

Storage representation of 200:

Sign Bit 16 MSB	Magnitude (MSB is 0, so direct binary representation of 200)														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	1	1	0	0	1	0	0	0

Storage representation of −200:

Sign Bit 16 MSB	Magnitude (MSB is 1, so magnitude is two's complement representation of 200)														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
1	1	1	1	1	1	1	1	0	0	1	1	1	0	0	0

✍ **Integral type** consists of integer type and character type.

8. *How does the maximum value that an integral data type supports depends upon its size?*

Consider integer data type, taking 2 bytes, i.e. 16 bits in memory. The maximum value it can have is as follows:

Sign Bit 16 MSB	Magnitude (MSB is 0, so direct binary representation)														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Sign Bit = 0 (means number is positive), magnitude is maximum (as all the magnitude bits have maximum value, i.e. 1). The stored number is 32767 (i.e. 2^{15}–1).

Now, consider character data type (taking 1 byte, i.e. 8 bits in memory). The maximum value it can have is 2^7–1 = 127. This can be shown as follows:

Sign Bit 8 MSB	Magnitude (MSB is 0, so direct binary representation)						
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	1	1	1	1	1	1	1

This shows that the maximum value that an integral type can take is directly in relation to its size. That is why, if an integer variable is not able to store a value (e.g. 70000), we switch to long integer because long integer takes 32 bits in memory. Thirty-one bits will be used for storage of magnitude. Hence, the maximum value (2147483647, i.e. 2^{31}–1) of long integer is far greater than the maximum value of integer (32767, i.e. 2^{15}–1), which has only 15 bits for the storage of magnitude.

Data type as such does not take any space in memory. Objects associated with the defined identifiers take memory space according to their data types. Wherever it is referred in the text that data type takes some space in memory, it implies that the object of the specified data type takes that much memory space.

9. *What will the output of the following program segment be? (Assume that integer data type takes 2 bytes of memory.)*

```
#include<stdio.h>
main()
{
   int a=32768;
   printf("%d",a);
}
```

The output that this program snippet prints is -32768. This can be well understood if one knows how integers are stored in the memory.

If integer type takes 2 bytes in the memory, 32767 is stored as follows:

Sign Bit 16 MSB	Magnitude (MSB is 0, so direct binary representation)														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Now, 32768 is 32767+1. If 1 is added in the above representation:

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
															1
1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

The value that comes in the memory is given in bold. The carry generated from Bit 15 has moved into Bit 16 (i.e. sign bit). Now, the sign bit becomes 1 (i.e. number becomes negative). If sign bit is 1, the magnitude of number is stored in two's complement form. The magnitude of number, i.e.

Magnitude (MSB is 1, so magnitude is in two's complement representation)														
Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

is in two's complement form. To get the value of magnitude, take two's complement of two's complemented representation of the magnitude. The magnitude can be found as follows:

		Magnitude														
		Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
Magnitude in two's complement form (Row 1)		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
One's complement		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Two's complement of value in row 1	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Decimal equivalent of the value obtained is 2^{15} = 32768. The sign was negative, so the number becomes −32768. Hence, whenever the value of an integral data type exceeds the range, the value **wraps around** to the other side of the range.

10. *If a value assigned to an integral variable exceeds the range, the assigned value wraps around to the other side of range. Why?*

A value greater than the maximum value that the magnitude field can hold makes the sign bit 1, i.e. makes the number negative and it seems like that value has wrapped around to the other side of range; e.g. for character data type, 127 (the maximum value) can be stored as follows:

Sign Bit 8 MSB	Magnitude						
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	1	1	1	1	1	1	1

If the value is further increased by 1, it becomes as follows:

Sign Bit 8 MSB	Magnitude						
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
1	0	0	0	0	0	0	0

The sign bit turns out to be 1. Hence, the number is negative and the magnitude is in two's complement form. To get the value of magnitude, take two's complement of 0000000. It comes out to be 10000000. This is equivalent to 128 and because the sign bit was 1, the value becomes −128 (seems as the value has wrapped around to the other side of the range).

11. *What are l-value and r-value?*

12. *Are nested multi-line comments by default allowed in C? If no, how can nested comments be allowed?*

No, by default nested multi-line comments✍ are not allowed in C. Multi-line comments do not nest, i.e. we cannot have a multi-line comment within another multi-line comment. This happens because after finding /*, which marks the beginning of the multi-line comment, the contents of comments are examined only to find the characters */, which terminates the comment.

In the following example:

```
/* comment starts here
....../*nested comment starts here
.....this terminator gets associated with marker of the first line*/
...............this line will not become comment*/
```

In the first line /* is encountered and the multi-line comment starts. Now only */ will be searched. It appears in line 3. This occurrence of */ gets associated with /* of the first line, and the comment is assumed to be finished but some part of the outer comment still persists and this leads to an error.

So in the above example, the portion that gets commented out is given in bold:

/* comment starts here

....../*nested comment starts here

.....this terminator gets associated with marker of the first line*/

...............this line will not become comment*/

Nested comments can be allowed by making changes in IDE settings or by using `pragma` directive. Use `#pragma option –C` to allow nested multi-line comments.

✍ **Comment** is a feature provided by almost all the programming languages. It is used to increase the readability of the program.

13. *How are real floating-type numbers treated in C?*

    Real floating-type numbers in C, by default, are treated as that of type double (i.e. using double precision), so that there should be lesser loss in precision. The following piece of code on execution (using Turbo C 3.0):

```
#include<stdio.h>
main()
{
    printf("%d",sizeof(7.0));
}
```

    prints 8 instead of 4. This is because 7.0 is treated as double (double precision) and not as float (single precision). To make it float, write it as 7.0f.

14. *The following piece of code is written to get a value from the user:*

```
main()
{
   int number;
   scanf("Enter a number %d",&number);
   printf("The number entered is %d",number);
}
```

    Irrespective of the number that I enter, I get a garbage value. Why?

    This problem is because of the string present inside scanf function. The scanf function cannot print a string on to the screen. Therefore, **Enter a number** will not be printed. In addition, the entered input should exactly match the format string✍ present inside the scanf function. Therefore, if a number say 10 is entered, it does not match with the format string and the output will be garbage. However, if **Enter a number 10** is given in the input, the string in the input exactly matches the format string. The number will take the value 10, and the output will be The number entered is 10.

✍ The **format specifiers** in a format string are generic terms and get matched with any value of the corresponding type. For example, %d gets matched with 10, 20, −23 or any other integer value.

15. *I have written the following piece of code keeping in mind the fact that the format string of* scanf *function should only consist of format specifiers. Still, I get a garbage value. Why?*

```
main()
{
      int number;
      printf("Enter a number\t");
      scanf("%d",number);
      printf("The number entered is %d",number);
}
```

    The given piece of code gives a garbage value due to the erroneous use of scanf function. Since, the second argument to the scanf function is not an l-value of the variable number, it will not be able to place the entered value at the designated memory position. The rectified statement can be written as scanf("%d",&number);.

16.
```
main()
{
      int a,b;
      printf("Enter two numbers");
```

```
        scanf("%d %d",&a,&b);
        printf("%d + %d = %d\n",a,b,a+b);
        printf("%d / %d= %d\n",a,b,a/b);
        printf("%d % %d=%d\n",a,b,a%b);
 }
```

The above piece of code on giving inputs 3 and 4 prints

```
3 + 4 = 7
3 / 4 = 0
3 % %d= 4
```

The last line is not printed correctly. How can I rectify the problem?

This problem can be rectified by using character stuffing. Instead of writing printf("%d % %d=%d\n",a,b,a%b); use printf("%d %% %d=%d\n",a,b,a%b);.

17. *What will the output of the following code snippet be and why?*

```
main()
{
        char *p="Hello\n";
        printf(p);
        printf("Hello  ""Readers!..");
}
```

The output of the code snippet is as follows:

```
Hello
Hello Readers!..
```

The printf function requires the first argument to be of char* type (i.e. a string); hence, printf(p) is perfectly valid and on execution prints Hello.
Adjacent string literals get concatenated; hence, "Hello ""Readers!.." will get concatenated to form "Hello Readers!..". It will be printed by the next printf statement.

18. *What will the output of the following code snippet be and why?*

```
main()
{
        char *p="Hello\n";
        printf(p"Readers!..");
}
```

There is a compilation error in this code snippet. This error is due to the fact that only adjacent string literals are concatenated. p is a variable and is not a string literal. It will not concatenate with the string literal "Readers!..". Hence the error.

19. *What will the output of the following piece of code be?*

```
main()
{
        int a=10,b=5,c;
        c=a/**//b;
        printf("%d",c);
}
```

The output of the code snippet will be 2. /**/ is a comment and will be neglected. Hence, the expression becomes c=a/b. Its output is 2.

20. *How are floating point numbers stored in C?*

Institute of Electrical & Electronics Engineers (IEEE) has produced a standard **(IEEE 754)** for floating point numbers. The standard specifies how single precision (4 bytes, i.e. 32 bits) and double precision (8 bytes, i.e. 64 bit) floating point numbers are represented.
An IEEE single-precision floating point number is stored in 4 bytes (32 bits). The MSB is Sign-bit, the next 8 bits are the Exponent bits 'E' and the final 23 bits are the fraction bits 'F'.

S	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	8-bits for exponent								**23-bits for mantissa (or fraction)**																						

How is a floating point number stored?
Look at the following example to understand the concept. To store 5.75:

1. **Convert 5.75 from the decimal number system (DNS) to the binary number system (BNS).**
   The integer part 5 in DNS is equivalent to 101 in BNS.
   The fractional part 0.75 in DNS is equivalent to 0.110 in BNS.
   Therefore, 5.75 in DNS is equivalent to 101.110 in BNS.
2. **The straight binary representation of a floating point number is normalized to make it IEEE 754 compliant**. Normalized numbers are represented in the form of 1.ffffff........ffff (f is binary digit) * 2^p, where *p* is the exponent. In a normalized number, the integer part is always 1. The decimal point is adjusted by selecting a suitable value of exponent, i.e. *p*. 101.110 in the normalized form is expressed as 1.01110 * 2^2.

   The value after the decimal point is stored in 23 fraction bits and the integer value is not stored (as it is always 1 in all normalized numbers, so there is no need to store it). So, in 1.01110 * 2^2, only 01110 is stored in 23 bits as fraction.
3. **The exponent is biased with a magic number 127_{10},** i.e. 127 is added to the exponent to make it 129. The binary equivalent of 129 (i.e. 10000001) is stored in 8 bits reserved for the storage of the exponent.

Thus, 5.75 is stored as follows:

0	1	0	0	0	0	0	0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	8-bits for exponent								**23-bits for mantissa**																						

Why are exponents biased with magic number 127_{10}?
Exponents are biased with magic number 127_{10}, so that floating point numbers can be compared for equality, greater than or less than.
Suppose exponents are not biased with magic number 127_{10}. Instead, sign-two's complement representation is used to store the value of the exponent. If such a representation is used:
2.0, i.e. 1.0 * 2^1 will be stored as follows:

0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	8-bits for exponent								**23-bits for mantissa**																						

0.5, i.e. 1.0 * 2^{-1} will be stored as follows:

0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	8-bits for exponent								**23-bits for mantissa**																						

Now, if it is checked that whether 2.0 > 0.5, it turns out to be false as 0.5 is stored as a greater value than the value of 2.0.

Now, consider that exponents are biased with the magic number 127_{10}.
2.0 is stored as: Sign Bit = 0, Exponent = 1000 0000 (128 = 1+127), Fraction = 00......0000
0.5 is stored as: Sign Bit = 0, Exponent = 0111 1110 (126 = −1+127), Fraction = 00......0000

It can be shown as follows:

2.0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0.5	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Value	S	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
		8-bits for exponent								**23-bits for mantissa**																						

2.0 is stored as a greater value than 0.5. Hence, greater than operator will give the correct result.

Conclusion with another example: To find storage representation of 0.4:

1. Convert 0.4 to binary.
   0.4 * 2 = 0.8
   0.8 * 2 = 1.6
   0.6   2 = 1.2
   0.2 * 2 = 0.4
   0.4 * 2 = 0.8; this sequence repeats.
   Therefore, 0.4 = **0.01100110011001100**...
2. Normalize **0.0110011001100**... After normalization it can be written as $1.10011001100...*2^{-2}$.
3. Exponent is biased with the magic number 127. Therefore, the exponent becomes −2 + 127 = 125. Its binary equivalent is 0111 1101.
   Hence, 0.4 will be stored as follows:

0	0	1	1	1	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
S	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	8-bits for exponent								**23-bits for mantissa**																						

Code Snippets

Determine the output of the following code snippets. Assume that the inclusion of the required header files has been made and there is no prototyping error due to them.

21.
```
main()
{
    printf(“%d %d %d %d”,72,072,0x72,0X72);
}
```

22.
```
main()
{
    printf(“%d %o %x”,72,72,72);
}
```

23.
```
main()
{
    printf(“%i %i %i %i”,72,072,0x72,0X72);
}
```

```
24. main()
    {
        printf(“%05d,%5d,%-5d”,32,32,32);
    }

25. main()
    {
        printf(“%6.3f,%06.3f,%09.3f,%-09.3f,%6.0f,%6.0f”,45.6,45.6,45.6,45.6,45.4,45.6);
    }

26. main()
    {
        int a=32768;
        unsigned int b=65536;
        printf(“%d %d”,a,b);
    }

27. main()
    {
        char a=128;
        unsigned char b=256;
        printf(“%d %d\n”,a,b);
    }

28. main()
    {
        float a=3.5e38;
        double b=3.5e309;
        printf(“%f %lf”,a,b);
    }

29. main()
    {
        printf(“%d %c”,‘A’,‘A’);
    }

30. main()
    {
        printf(“char occupies %d byte\n”, sizeof(char));
        printf(“int occupies %d bytes\n”, sizeof(int));
        printf(“float occupies %d bytes”, sizeof(float));
    }

31. main()
    {
        printf(“bytes occupied by ‘7’=%d\n”,sizeof(‘7’));
        printf(“bytes occupied by 7=%d\n”,sizeof(7));
        printf(“bytes occupied by 7.0=%d”,sizeof(7.0));
    }

32. main()
    {
        printf(“%d”,sizeof(‘\n’));
    }
```

```
33. main()
    {
        printf(“%d %c”);
    }
```

```
34. main()
    {
        printf(“%d %d %d %d %d\n”,sizeof(032),sizeof(0x32),sizeof(32),sizeof(32U),sizeof(32L));
        printf(“%d %d %d”,sizeof(32.4),sizeof(32.4f),sizeof(32.4F));
    }
```

```
35. main()
    {
        printf(“\nab”);
        printf(“\bsi”);
        printf(“\rha”);
    }
```

```
36. main()
    {
        printf(“c:\tc\bin”);
    }
```

```
37. main()
    {
        printf(“c:\\tc\\bin”);
    }
```

```
38. main()
    {
        printf(“hello,world
        ”);
    }
```

```
39. main()
    {
        printf(“hello,world\
        ”);
     }
```

```
40. main()
    {
        char *p=”Welcome!..””to C programming”;
        printf(p);
    }
```

Multiple-choice Questions

41. The primary use of C language was intended for

a. System programming
b. General-purpose use
c. Data processing
d. None of these

42. C is a/an

a. Assembly-level language
b. Machine-level language
c. High-level language
d. None of these

43. C is a

a. General-purpose language
b. Case-sensitive language
c. Procedural language
d. All of these

44. Which of the following cannot be the first character of the C identifier?

a. A digit
b. A letter
c. An underscore
d. None of these

45. Which of the following cannot be used as an identifier?

a. Variable name
b. Constant name
c. Function name
d. Keyword

46. Which of the following is not a basic data type?

a. `char`
b. `float`
c. `long`
d. `double`

47. Which of the following is not a type modifier?

a. `long`
b. `unsigned`
c. `signed`
d. `double`

48. Which of the following is a type qualifier?

a. `const`
b. `signed`
c. `long`
d. `short`

49. Which of the following is used to make an identifier a constant?

a. `const`
b. `signed`
c. `volatile`
d. None of these

50. Which of the following have both l-value and r-value?

a. Variables
b. Constants
c. Both variables and constants
d. None of these

51. Which of the following is not a C keyword?

a. `typedef`
b. `enum`
c. `volatile`
d. `type`

52. Qualified constant can be

a. Initialized with a value
b. Assigned a value
c. Both initialized and assigned
d. Neither initialized nor assigned

53. Which of the following is not a valid literal constant?

a. `'A'`
b. `1.234`
c. `"ABC"`
d. None of these

54. Which of the following is not a valid floating point literal constant?

a. `+3.2e−5`
b. `4.1e8`
c. `−2.8e2.3`
d. `+325.34`

55. By default, any real constant in C is treated as
    a. A float
    b. A double
    c. A long double
    d. Depends upon the memory model

56. Which of the following is not a valid escape sequence?
    a. \r
    b. \a
    c. \v
    d. \m

57. Escape sequence begins with
    a. /
    b. \
    c. %
    d. -

58. Single-line comment is terminated by
    a. //
    b. End of line
    c. */
    d. None of these

59. The maximum number of characters in a character literal constant can be
    a. 0
    b. 1
    c. 2
    d. Any number

60. Which of the following character is not a printable character?
    a. New line character
    b. Backslash character
    c. Quotation mark
    d. All of these

61. Attributes that characterize variables in C language are
    a. Its name and location in the memory
    b. Its value and its type
    c. Its storage class
    d. All of these

62. In the assignment statement x=x+l; the meaning of the occurrence of the variable x to the left of the assignment symbol is its
    a. Location (l-value)
    b. Value (r-value)
    c. Type
    d. None of these

63. Which one is an example of derived data type?
    a. Array type
    b. Pointer type
    c. Function type
    d. All of these.

64. In C language, which method is used for determining the type equivalence?
    a. Structural equivalence
    b. Name equivalence
    c. Both of these
    d. None of these

65. In the assignment statement x=x+l; the meaning of the occurrence of the variable x to the right of the assignment symbol is its:
    a. Location (l-value)
    b. Value (r-value)
    c. Type
    d. None of these

66. If specific implementation of C language uses 2 bytes for the storage of integer data type, what is the maximum value that an integer variable can take?
    a. 32767
    b. 32768
    c. –32768
    d. 65535

67. If specific implementation of C language uses 2 bytes for the storage of integer data type, then what is the minimum value that an integer variable can take?

    a. –32767
    b. –32768
    c. 0
    d. None of these

68. Which of the following format specifier is used for printing an integer value in octal format?

    a. %x
    b. %X
    c. %o
    d. %i

69. How many bytes are occupied by the string literal constant "xyz" in the memory?

    a. 1
    b. 2
    c. 3
    d. 4

70. The variables and constants of which of the following type cannot be declared?

    a. int**
    b. int(*)[]
    c. void
    d. float

Outputs and Explanations to Code Snippets

21. 72 58 114 114

    Explanation:

    All the outputs are desired in the decimal number system because of %d specifier. Now, 72 is a decimal number, 072 is an octal number equivalent to 58 in the decimal number system, 0x72 and 0X72 are hexadecimal numbers equivalent to 114 in the decimal number system. Hence, the output is 72 58 114 114.

22. 72 110 48

    Explanation:

    72 is to be printed in the decimal (%d specifier), the octal (%o specifier) and the hexadecimal number system (%x specifier). The octal equivalent of 72 is 110 and the hexadecimal equivalent of 72 is 48. Hence, the output is 72 110 48.

23. 72 58 114 114

    Explanation:

    %i specifier is used for integers. By default, it will output integer in the decimal number system as it is the most commonly used number system.

24. 00032,   32,32

    Explanation:

    In the given format string, width specifiers are used along with the format specifiers. Width specifier sets the minimum width for an output value.
    %5d means output will be minimum 5 columns wide and will be right justified.
    %-5d means output will be minimum 5 columns wide and will be left justified.
    %05d means output will be minimum 5 columns wide, right justified, and the blank columns will be padded by zeros.

<table>
<tr><td>0</td><td>0</td><td>0</td><td>3</td><td>2</td><td>,</td><td></td><td></td><td></td><td>3</td><td>2</td><td>,</td><td>3</td><td>2</td><td></td><td></td><td></td></tr>
<tr><td colspan="5">**%05d**
(*cw = 5, rj, padding of 0's)</td><td></td><td colspan="5">**%5d**
(*cw = 5, rj)</td><td colspan="6">**%–5d**
(*cw = 5, - is used for lj)</td></tr>
</table>

***cw is column width, rj is right justified and lj is left justified.**
Hence, the output is 00032,   32,32.

25. 45.600,45.600,00045.600,45.600   ,   45,   46

Explanation:

In the given format string, width specifiers and precision specifiers are used along with the format specifiers. Precision specification always begins with a period to separate it from the preceding width specifier.
%6.3f means output is 6 columns wide. 3 is the number of digits after decimal. It is shown as

4	5	.	6	0	0

%06.3f means output is 6 columns wide. 3 is the number of digits after decimal. 0 means blank spaces are to be padded by zeros. It is shown as

4	5	.	6	0	0

%09.3f means output is 9 columns wide. 3 is the number of digits after decimal. 0 means blanks spaces are to be padded by zeros. By default, the output is right justified. It is shown as

0	0	0	4	5	.	6	0	0

%–09.3f means output is 9 columns wide. 3 is the number of digits after decimal. Since – is used, the output will be left justified. Here the output shows blank spaces, and padding by zeros has not been done because only 3 digits can be printed after the decimal. It is shown as

4	5	.	6	0	0			

%6.0f means output is 6 columns wide. 0 is the number of digits after the decimal. Rounding off will take place and 45.4 will be rounded to 45. The output will be

				4	5

%6.0f means output is 6 columns wide. 0 is the number of digits after the decimal. Rounding off will take place and 45.6 will be rounded to 46. It is shown as

				4	6

26. –32768 0

Explanation:

Since the assigned values exceed the maximum value of integer type and unsigned integer type, the values wrap around to the other side of the range. Hence, the outputs are –32768 (minimum value of signed integer) and 0 (minimum value of unsigned integer).

27. –128 0

Explanation:

Since the assigned values exceed the maximum value of character type and unsigned character type, the values wrap around to the other side of the range. Hence, the outputs are –128 (minimum value of signed character) and 0 (minimum value of unsigned character).

28.
```
+INF +INF
```
Explanation:

Range wraps around only in case of integral data type. Wrap around does not occur in case of float and double data types. In case of float and double data types, if the value falls outside the range +INF✍ or −INF✍ is the output.

> ✍ **+INF** refers to +Infinity and **–INF** refers to –Infinity.

29.
```
65 A
```
Explanation:

Integers and characters together form integral data type and are not separated internally. If characters are printed using %d specifier, it gives the ASCII equivalent of the character. Hence, the output is 65, ASCII code of 'A'. If %c specifier is used, it prints the character, i.e. 'A'.

30.
```
char occupies 1 byte
int occupies 2 bytes
float occupies 4 bytes
```
Explanation:

sizeof operator outputs the size of the given data type.

31.
```
bytes occupied by '7'=1
bytes occupied by 7=2
bytes occupied by 7.0=8
```
Explanation:

sizeof operator can also take constant as input and returns the number of bytes required by the data type of that constant as output. 7.0 is a real floating number and will be treated as double type. Hence, sizeof(7.0) gives 8.

32.
```
1
```
Explanation:

'\n' is a character, more specifically a new line character. Hence, sizeof operator returns 1, i.e. the size of a character.

33.
```
Garbage
```
Explanation:

Since format specifiers %d and %c are not linked to any value, they will output garbage. This is only applicable for %d and %c specifiers. If %f specifier is not linked, it leads to abnormal program termination.

34.
```
2 2 2 2 4
8 4 4
```
Explanation:

032, 0x32, 32 all are integers in different number systems. 32U is an unsigned integer. Hence, their size is 2. 32L is a long integer of size 4. 32.4 is a real floating-type number and is treated as a double of size 8. 32.4f and 32.4F are float and their size is 4. Hence, the output.

35.
```
                    ←Blank line
hai
```

Explanation:

‘\n’ is a new line character. Due to ‘\n’, cursor appears in a new line and “ab” gets printed. ‘\b’ is a backspace character. It places the cursor below the character ‘b’ and “si” gets printed. Therefore, the output becomes “asi”. ‘\r’ is a carriage return character. It will make the cursor return to the starting of the same line. The cursor will be placed below ‘a’. “ha” gets printed and overwrites “as”. Hence, the output becomes “hai”.

36.
```
c:    in
```

Explanation:

‘\t’ is a tab character and ‘\b’ is a backspace character. Due to ‘\t’ character ‘c’ gets tab separated from “c:”. The output becomes “c: c”. ‘\b’ makes character ‘c’ to erase and “in” gets printed. Hence, the output becomes “c: in”.

37.
```
c:\tc\bin
```

Explanation:

The usage of an extra backslash is known as character stuffing. Now ‘\t’ will not be treated as a tab character and will actually get printed. Similarly ‘\b’ will not be treated as a backspace character.

38.
```
Compilation error
```

Explanation:

String cannot span multiple lines in this way. Hence, the error.

39.
```
hello,world
```

Explanation:

Each instance of the backslash character (\) immediately followed by a new line character is deleted. This process is known as **line splicing**. Physical source lines are spliced to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice.

Physical source lines

```
main()
{
   printf("hello,world\
   ");
}
```

after splicing will form the following logical source lines:

```
main()
{
   printf("hello,world");
}
```

Logical source lines are processed by the compiler. Hence, on execution, hello,world is the output.

40. Welcome!..to C programming

Explanation:

Adjacent string literals get concatenated. Hence, “Welcome!..””to C programming” gets concatenated and becomes “Welcome!..to C programming”. printf needs first argument to be of char* type. In printf(p) this constraint is satisfied as p is the only argument and is of char* type. Hence, the value of p, i.e. Welcome!..to C programming gets printed.

Answers to Multiple-choice Questions

41. a	42. c	43. d	44. a	45. d	46. c	47. d	48. a	49. a	50. a	51. d	52. a	53. d	54. c
55. b	56. d	57. b	58. b	59. c	60. d	61. d	62. a	63. d	64. a	65. b	66. a	67. b	68. c
69. d	70. c												

Programming Exercises

Program 1 | Convert the temperature given in Fahrenheit to Celsius

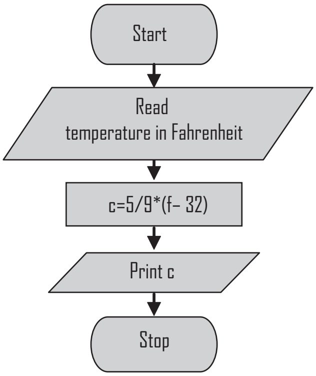
Algorithm:

```
Step 1: Start
Step 2: Read the temperature given in Fahrenheit (f)
Step 3: Temperature in Celsius (c) = 5/9*(f–32)
Step 4: Print temperature in Celsius
Step 5: Stop
```

PE 3-1.c

```c
//Convert temperature in Fahrenheit to
 //Celsius
#include<stdio.h>
main()
{
    float f,c;
    printf(“Enter temperature in Fahrenheit\t”);
    scanf(“%f”,&f);
    c=5.0/9.0*(f-32);
    printf(“Temperature in Celsius is %6.2f”,c);
}
```

Flowchart depicting the flow of control in program



Output window

```
Enter temperature in Fahrenheit    106
Temperature in Celsius is 41.11
```

Flowchart is a graphical representation that depicts the flow of program control.

Program 2 | Find the area and circumference of a circle with radius r

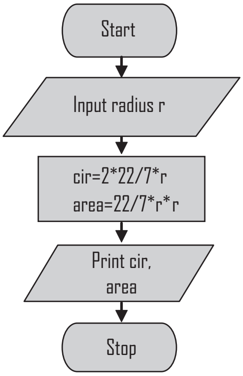
Algorithm:

```
Step 1: Start
Step 2: Read the radius of circle (r)
Step 3: Circumference cir = 2*22/7*r
Step 4: Area area = 22/7*r*r
Step 5: Print circumference and area
Step 6: Stop
```

PE 3-2.c

```c
//Circumference and area of circle
#include<stdio.h>
main()
{
    float r, cir, area;
    printf(“Enter the radius of circle\t”);
    scanf(“%f”,&r);
    cir=2*22.0/7*r;
    area=22.0/7*r*r;
    printf(“Circumference of circle is %6.2f\n”,cir);
    printf(“Area of circle is %6.2f\n”,area);
}
```

Flow chart depicting the flow of control in program



Output window

```
Enter the radius of circle     5
Circumference of circle is 31.43
Area of circle is 78.57
```

Program 3 | Find the average of three numbers

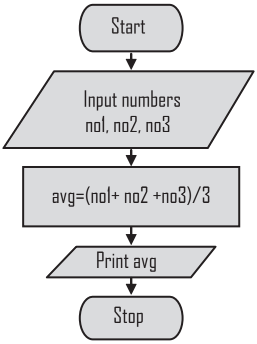
Algorithm:

```
Step 1: Start
Step 2: Read numbers no1, no2, no3
Step 3: Average avg = (no1+no2+no3)/3
Step 4: Print avg
Step 5: Stop
```

PE 3-3.c

```c
//Average of three numbers
#include<stdio.h>
main()
{
    float no1, no2, no3,avg;
    printf(“Enter three numbers\t”);
    scanf(“%f %f %f”,&no1, &no2, &no3);
    avg=(no1+no2+no3)/3;
    printf(“Average of numbers is %6.2f\n”,avg);
}
```

Flow chart depicting the flow of control in program



Output window

```
Enter three numbers     12 11 14
Average of numbers is 12.33
```

Program 4 | Simple Interest and the Maturity Amount

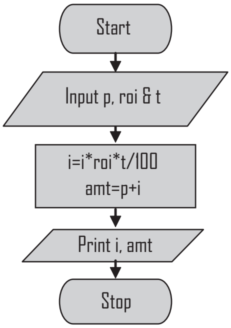
Algorithm:

```
Step 1: Start
Step 2: Read principle (p), rate of interest (roi), time period (t)
Step 3: Interest i = p*roi*t/100
Step 4: Amount amt = p+i
Step 5: Print i, amt
Step 6: Stop
```

PE 3-4.c

```
//Simple Interest
#include<stdio.h>
main()
{
    float p, roi, t, i, amt;
    printf(“Enter principle, rate and time\t”);
    scanf(“%f %f %f”,&p, &roi, &t);
    i=p*roi*t/100;
    amt=p+i;
    printf(“Interest is %6.2f\n”,i);
    printf(“Amount is %6.2f\n”,amt);
}
```

Flow chart depicting the flow of control in program



Output window

```
Enter principle, rate and time    1000 7 2
Interest is 140.00
Amount is 1140.00
```

Program 5 | Find area of a triangle whose sides are a, b and c

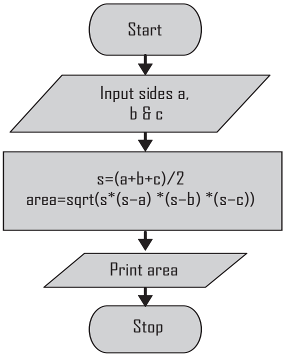
Algorithm:

```
Step 1: Start
Step 2: Read sides a, b and c of triangle
Step 3: s = (a+b+c)/2
Step 4: area = sqrt(s*(s–a)*(s–b)*(s–c))
Step 5: Print area
Step 6: Stop
```

PE 3-5.c

```
//Area of a triangle
#include<stdio.h>
#include<math.h>
main()
{
    float a, b, c, s, area;
    printf(“Enter the sides of a triangle\t”);
    scanf(“%f %f %f”,&a, &b, &c);
    s=(a+b+c)/2;
    area=sqrt(s*(s–a) *(s–b) *(s–c));
    printf(“Area of triangle is %6.2f sq. units”,area);
}
```

Flow chart depicting the flow of control in program



Output window

```
Enter the sides of a triangle    12 5 14
Area of triangle is 29.23 sq. units
```

Program 6 | The velocity of an object is given in km/hr. Write a C program to convert the given velocity from km/hr to m/sec

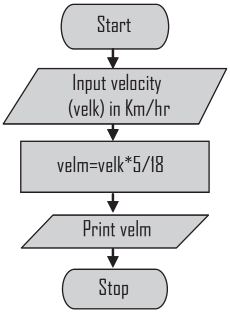
Algorithm:

```
Step 1: Start
Step 2: Input the velocity (velk) given in km/hr
Step 3: velocity in m/sec (velm) = velk*5/18
Step 4: Print velocity in m/sec (velm)
Step 5: Stop
```

PE 3-6.c

```c
//Convert units of velocity
#include<stdio.h>
main()
{
    float velk, velm;
    printf(“Enter velocity in Km/hr\t”);
    scanf(“%f”,&velk);
    velm=velk*5/18;
    printf(“Equivalent velocity is %f m/sec”,velm);
}
```

Flow chart depicting the flow of control in program



Output window

```
Enter velocity in km/hr     12
Equivalent velocity is 3.333333 m/sec
```

Program 7 | An object undergoes uniformly accelerated motion. The initial velocity (u) of the object and the acceleration (a) are known. Write a C program to find the velocity (v) of the object after time t

Algorithm:

```
Step 1: Start
Step 2: Input the initial velocity (u) and acceleration (a) of the object in SI units
Step 3: Input the time (t) after which velocity is to be computed
Step 4: Velocity v = u+a*t
Step 5: Print value of velocity (v)
Step 6: Stop
```

PE 3-7.c

```c
//Compute velocity after time t
#include<stdio.h>
main()
{
    float u, v,a, t;
    printf(“Enter the value of initial velocity in m/s\t”);
    scanf(“%f”,&u);
    printf(“Enter the amount of acceleration\t”);
    scanf(“%f”,&a);
    printf(“Enter the time in sec.\t”);
    scanf(“%f”,&t);
    v=u+a*t;
    printf(“Velocity after %4.2f sec is %4.2f m/s”,t,v);
}
```

Output window

```
Enter the value of initial velocity in m/s     2.4
Enter the amount of acceleration     4
Enter the time in sec.     2
Velocity after 2.00 sec is 10.40 m/s
```

Program 8 | A year approximately consists of 3.156 × 10^7 seconds. Write a C program that accepts your age in years and convert it into equivalent number of seconds

Algorithm:

```
Step 1: Start
Step 2: Enter age (age) in years
Step 3: Age in seconds (age_in_sec) = $3.156 \times 10^7$*age
Step 4: Print equivalent age in seconds (age_in_sec)
Step 5: Stop
```

PE 3-8.c

```c
//Equivalent age in seconds
#include<stdio.h>
main()
{
    int age;
    float age_in_sec;
    printf(“How old are you (years)?\t”);
    scanf(“%d”,&age);
    age_in_sec=3.156E7*age;
    printf(“Your age in seconds is %5.2E”,age_in_sec);
}
```

Output window

```
How old are you (years)?    18
Your age in seconds is 5.68E+08
```

Remark:

- %E specifier is used to print floating point value in exponent form

Test Yourself

1. Fill in the blanks in each of the following:
   a. The C language was developed by ____________.
   b. An identifier name in C starts with a ____________ .
   c. One of the most important attributes of an identifier is its ____________.
   d. `int var;` is a ____________ statement.
   e. ____________ is a data object locator.
   f. Constants do not have ____________ value.
   g. ____________ qualifier is used to create a qualified constant.
   h. Non-printable character constants are represented with the help of ____________ .
   i. The first argument of `printf` function should always be a ____________ .
   j. Floating point literal constant by default is assumed to be of type ____________ .
   k. A C program is made up of ____________ .
   l. Every statement in C is terminated with a ____________ .
   m. The `printf` function prints the value according to the ____________ specified in the ____________.
   n. The amount of memory that an object of a data type would take can be found by using ____________ operator.
   o. The arguments following the first argument in a `scanf` function should denote ____________.

2. State whether each of the following is true or false. If false, explain why.
   a. C is a case-sensitive language, which means that it distinguishes between uppercase characters and lowercase characters.
   b. An identifier name in C cannot start with a digit.
   c. All the variable names must be declared before they are used in a C program.
   d. Comments play an important role in a C program and are processed by the C compiler to produce an executable code.
   e. Keyword or a reserved word cannot be used as a valid identifier name.
   f. `int a=20, b=30, c;` is an example of a longhand declaration statement.
   g. A type qualifier modifies the base type to yield a new type.
   h. Constants have both l-value and r-value.
   i. A character literal constant can have one or at most two characters enclosed within single quotes.
   j. The `scanf` function can be used to read only one value at a time.

3. Determine which of the following are valid identifier names in C:
   a. main
   b. MAIN
   c. NewStudent
   d. New_Student
   e. a+b
   f. for_while
   g. 123abc
   h. abc123
   i. name&number
   j. _classnumber
   k. _number_

4. Determine which of the following are valid constants:
    a. "ABC"
    b. '#'
    c. Abc
    d. 1,234
    e. –22.124
    f. 1.23E-2.0
    g. 0x2AG
    h. '\r'
    i. 0x23
    j. 23L
    k. –7.0f
5. Identify and correct the errors in each of the following statements:
    a. int a=10, int b=20;
    b. int a=10, float b=2.5;
    c. int a=23u, b=2f;
    d. const int number=100;
       number=500;
    e. printf(1,2,3);
    f. Printf("To err is human");
    g. printf("%d %d" no1, no2);
    h. printf("Humans learn by making mistakes")
    i. scanf("%d %d", no1, no2);
    j. first_value+second_value=sum_of_values

This page is intentionally left blank

4

OPERATORS AND EXPRESSIONS

Learning Objectives

In this chapter, you will learn about:

- Operands and operators
- Expressions
- Simple expressions and compound expressions
- How compound expressions are evaluated
- Precedence and associativity of operators
- How operators are classified
- Classification based on number of operands
- Unary, binary and ternary operators
- Classification based on role of operator
- Arithmetic, relational, logical, bitwise, assignment and miscellaneous operators
- Rules for evaluation of arithmetic expressions
- Implicit and explicit-type conversions
- Promotions and demotions
- Conditional, comma, sizeof and address-of operator
- Combined precedence of all operators
- Reading strings from the keyboard
- Printing strings on the screen
- Unformatted functions

4.1 Introduction

In Chapter 3, you have learnt about identifiers (i.e. variables and functions specifically printf and scanf functions), constants and data types. In this chapter, I will take you a step forward and tell you how to create expressions from identifiers, constants and operators. Finally, we will look at how expressions are evaluated and the intricacies involved in this evaluation process.

4.2 Expressions

An **expression** in C is made up of one or more operands. The simplest form of an expression consists of a single operand. For example, 3 is an expression that consists of a single operand, i.e. 3. Such an expression does not specify any operation to be performed and is not meaningful. In general, a meaningful expression consists of one or more operands and operators that specify the operations to be performed on operands. For example, a=2+3 is a meaningful expression, which involves three operands, namely a, 2 and 3 and two operators, i.e. = (assignment operator) and + (arithmetic addition operator). Thus, an expression is a sequence of operands and operators that specifies the computation of a value. Let us look at the fundamental constituents of an expression, i.e. operands and operators.

4.2.1 Operands

An **operand** specifies an entity on which an operation is to be performed. An operand can be a variable name, a constant, a function call or a macro name. For example, a=printf("Hello")+2 is a valid expression involving three operands, namely a variable name, i.e. a, a function call, i.e. printf("Hello") and a constant, i.e. 2.

4.2.2 Operators

An **operator** specifies the operation to be applied to its operands. For example, the expression a=printf("Hello")+2 involves three operators, namely function call operator, i.e. (), arithmetic addition operator, i.e. + and assignment operator, i.e. =.

Based on the number of operators present in an expression, expressions are classified as simple expressions and compound expressions.

4.3 Simple Expressions and Compound Expressions

An expression that has only one operator is known as a **simple expression** while an expression that involves more than one operator is called a **compound expression**. For example, a+2 is a simple expression and b=2+3*5 is a compound expression. The evaluation of a simple expression is easier as compared to the evaluation of a compound expression. Since, there is more than one operator in a compound expression, while evaluating compound expressions one must determine the order in which operators will operate. For example, to determine the result of evaluation of the expression b=2+3*5, one must determine the order in which =, + and * will operate. This order determination becomes trivial in the case of evaluation of simple expressions like a+2, as there is only one operator and it has to operate in any case. The order in which the operators will operate depends upon the **precedence** and the **associativity** of operators.

4.3.1 Precedence of Operators

Each operator in C has a **precedence** associated with it. In a compound expression, if the operators involved are of different precedence, the operator of higher precedence operates first. For example, in an expression b=2+3*5, the sub-expression 3*5 involving multiplication operator (i.e. *) is evaluated first as the multiplication operator has the highest precedence among =, + and *. The result of evaluation of an expression is an r-value. The sub-expression 3*5 evaluates to an r-value 15. This r-value will act as a second operand for an addition operator and the expression becomes b=2+15. In the resultant expression, the sub-expression 2+15 will be evaluated next as the addition operator (i.e. +) has a higher precedence than the assignment operator (i.e. =). The expression after the evaluation of the addition operator reduces to b=17. Now, there is only one operator in the expression. The assignment operator will operate and the value 17 is assigned to b.

The knowledge of precedence of operators alone is not sufficient to evaluate a compound expression in case two or more operators involved are of the same precedence. For example, in the expression b=2*3/5, the multiplication operator (i.e. *) and the division operator (i.e. /) have the same precedence. The sub-expression 2*3/5 will evaluate to 1 if the multiplication operator operates before the division operator and to 0 if the division operator operates prior to the multiplication operator. To determine which of these operators will operate first, the associativity of these operators is to be considered.

4.3.2 Associativity of Operators

In a compound expression, when several operators of the same precedence appear together, the operators are evaluated according to their **associativity**. An operator can be either left-to-right associative or right-to-left associative. The operators with the same precedence always have the same associativity. If operators are left-to-right associative, they are applied in a left-to-right order, i.e. the operator that appears towards the left will be evaluated first. If they are right-to-left associative, they will be applied in the right-to-left order. The multiplication and the division operators are left-to-right associative. Hence, in expression 2*3/5, the multiplication operator is evaluated prior to the division operator as it appears before the division operator in the left-to-right order.

Now, let us look at various operators, their classification, precedence and associativity.

4.4 Classification of Operators

The operators in C are classified on the basis of the following criteria:

1. The number of operands on which an operator operates.
2. The role of an operator.

4.4.1 Classification Based on Number of Operands

Based upon the number of operands on which an operator operates, the operators are classified as:

1. Unary operators	A **unary** operator operates on only one operand. For example, in the expression -3, - is a unary minus operator as it operates on only one operand, i.e. 3. The operand can be present towards the right of the unary operator, as in -3 or towards the left of the unary operator, as in the expression a++. Examples of unary operators are: & (address-of operator), sizeof operator, ! (logical negation), ~ (bitwise negation), ++ (increment operator), -- (decrement operator), etc.
2. Binary operators	A **binary** operator operates on two operands. It requires an operand towards its left and right. For example, in expression 2-3, - acts as a binary minus operator as it operates on two operands, i.e. 2 and 3. Examples of binary operators are: * (multiplication operator), / (division operator), << (left shift operator), == (equality operator), && (logical AND), & (bitwise AND), etc.
3. Ternary operator	A **ternary** operator operates on three operands. Conditional operator (i.e. ?:) is the only ternary operator available in C.

4.4.2 Classification Based on Role of Operator

Based upon their role, operators are classified as:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Bitwise operators
5. Assignment operators
6. Miscellaneous operators

4.4.2.1 Arithmetic Operators

Arithmetic operations like addition, subtraction, multiplication, division, etc. can be performed by using arithmetic operators. The arithmetic operators available in C are given in Table 4.1.

Table 4.1 | Arithmetic operators

S.No	Operator	Name of operator	Category	-ary of operator	Precedence among arithmetic class	Associativity
1.	+ - ++ --	Unary plus Unary minus Increment Decrement	Unary operators	Unary	Level-I (Highest)	R→L (Right-to-left)
2.	* / %	Multiplication Division Modulus	Multiplicative operators	Binary	Level-II (Intermediate)	L→R (Left-to-right)
3.	+ -	Addition Subtraction	Additive operators	Binary	Level-III (Lowest)	L→R

The operators within a row have the same precedence, and the order in which they are written does not matter.

The following rules are observed while evaluating arithmetic expressions:

1. The parenthesized sub-expressions are evaluated first.
2. If the parentheses are nested, the innermost sub-expression is evaluated first.
3. The precedence rules are applied to determine the order of application of operators while evaluating sub-expressions.
4. The associativity rule is applied when two or more operators of the same precedence appear in the sub-expression.
5. If the operands of a binary arithmetic operator are of different but compatible types, C automatically applies **arithmetic-type conversion** to bring the operands to a common type. This automatic-type conversion is known as **implicit-type conversion**.The result of the evaluation of an operator will be of the **common type**. The basic principle behind the implicit arithmetic-type conversion is that if operands are of different types, the lower type (i.e. smaller in size) should be converted to a higher type (i.e. bigger in size) so that there is no loss in value or precision. Since a lower type is converted to a higher type, it is said that the lower type is **promoted** to a higher type and the conversion is known as **promotion**. The following are common **arithmetic-type conversions**:
   a. If one operand is long double, the other will be converted to long double, and the result will be long double.
   b. If one operand is double, the other will be converted to double, and the result will be double.
   c. If one operand is float, the other will be converted to float, and the result will be float.
   d. If one of the operands is unsigned long int, the other will be converted to unsigned long int, and the result will be unsigned long int.
   e. If one operand is long int and the other is unsigned int, then
      i. If unsigned int can be converted to long int, then unsigned int operand will be converted as such, and the result will be long int.
      ii. Else, both operands will be converted to unsigned long int, and the result will be unsigned long int.
   f. If one of the operands is long int, the other will be converted to long int, and the result will be long int.
   g. If one operand is unsigned int, the other will be converted to unsigned int, and the result will be unsigned int.
   h. If none of the above is carried out, both the operands are converted to int.

The above-mentioned rules can be **summarized** as:
Binary arithmetic operators can be used in one of the following three different modes:

1. **Integer mode:** If both the operands of a binary arithmetic operator are of integer type, the mode of operation is said to be **integer mode** and the result will be of integer type. For example: the result of 4/3 will be 1 instead of 1.3333, as integer mode operation results in the value of integer type.

2. **Floating point mode:** If both the operands of a binary arithmetic operator are of floating point type, the mode of operation is said to be **floating point mode** and the result will be of floating point type. For example: the result of 4.0/3.0 will be 1.333333, as the result of floating point mode operation is of floating point type.
3. **Mixed mode:** If one of the operands of a binary arithmetic operator is of integer type and another operand is of floating point type, the mode of operation is said to be **mixed mode**. The operand of integer type is promoted to floating point type and the result will be of floating point type. For example: the result of 4/3.0 will be 1.333333.

Consider Program 4-1.

Prog 4-1.c

```c
//Arithmetic expression
#include<stdio.h>
main()
{
    int a;
    a=2*3.25+((3+6)/2);
    printf("The result of evaluation is %d",a);
}
```

Output window

```
The result of evaluation is 10
```

Program 4-1 | A program that illustrates the evaluation of an arithmetic expression

Let us look at how the expression a=2*3.25+((3+6)/2) as specified in Program 4-1 gets evaluated. The innermost parenthesized sub-expression (3+6) gets evaluated first. This sub-expression evaluates to an r-value, i.e. 9. This r-value acts as an operand for the division operator. Now, the expression reduces to a=2*3.25+(9/2). The sub-expression (9/2) gets evaluated next. Since both the operands of the division operator are of integer type, the arithmetic involved is integer arithmetic and thus, the result is an r-value of integer type, i.e. 4 instead of 4.5. The expression becomes a=2*3.25+4. Since the multiplication operator (i.e. *) has a higher precedence than the addition operator (i.e. +) and the assignment operator (i.e. =), the sub-expression 2*3.25 gets evaluated next. In this sub-expression, the arithmetic involved is mixed mode arithmetic as one of the operands is of integer type and the other is of floating point type. The operand 2 is promoted to 2.0. The result of sub-expression 2.0*3.25 turns out to be 6.50. After the evaluation of this sub-expression, the expression gets reduced to a=6.50+4. The sub-expression 6.50+4 involves mixed mode operation and is evaluated to 10.50. Finally, the expression becomes a=10.50. In this expression, the value of floating point type is assigned to a variable of integer type. The operand of floating point type (i.e. 10.50) is automatically converted to an integer type so that it can be assigned to the integer variable a. Since a higher type (i.e. float, bigger in size) is converted to a lower type (i.e. int, smaller in size), it is said that the higher type is **demoted** to the lower type and this conversion is called **demotion**. The method followed during demotion is **truncation**. Thus, 10.50 is demoted (i.e. truncated) to 10 and is assigned to a. This value of a is printed by the printf function.

The important points about the arithmetic operators are as follows:

1. The **unary plus operator** can appear only towards the left side of its operand.
2. The **unary minus operator** can appear only towards the left side of its operand.

3. **Increment operator**

   a. The increment operator can appear towards the left side or towards the right side of its operand. If it appears towards the left side of its operand (e.g. ++a), it is known as the **pre-increment operator**. If it appears towards the right side of its operand (e.g. a++), it is known as the **post-increment operator**.

   b. The increment operator can only be applied to an operand that has a modifiable l-value. If it is applied to an operand that does not have a modifiable l-value, there will be 'L-value required' error. Try executing the code listed in Program 4-2.

Line	Prog 4-2.c	Output window

1
2
3
4
5
6
7
8
	//Increment/ Decrement operator's operand
	#include<stdio.h>
	main()
	{
	    int a;
	    a=++2;
	    printf("The result of application of pre-increment operator is %d",a);
	}
		Compilation error "L-value required"
		Reasons:
		• Operand of increment/decrement operator should have a modifiable l-value
		• 2 is a constant and does not have modifiable l-value
		What to do?
		• Create a variable b, place value 2 in it and instead of ++2 write ++b

Program 4-2 | A program to illustrate that operand of increment/decrement operator should have a modifiable l-value

   c. ++a or a++ is equivalent to a=a+1.

   d. The **difference between pre-increment and post-increment** lies in the point at which the value of their operand is incremented.

      i. In case of the pre-increment operator, first the value of its operand is incremented and then it is used for the evaluation of expression.

      ii. In case of the post-increment operator, the value of operand is used first for the evaluation of the expression and after its use, the value of the operand is incremented.

The difference between two versions of increment operator is shown in the code listed in Program 4-3.

Line	Prog 4-3.c	Output window

1
2
3
4
5
6
7
8
9
	//Difference between Pre-increment and Post-increment
	#include<stdio.h>
	main()
	{
	    int a=2, b=2,c,d;
	    c=++a;
	    d=b++;
	    printf("a=%d, b=%d, c=%d, d=%d",a,b,c,d);
	}
		a=3, b=3, c=3, d=2
		Reasons:
		• The value of a is incremented and then it is assigned to c as a is pre-incremented
		• The value of b is assigned to d before it is incremented as b is post-incremented

Program 4-3 | A program that illustrates the difference between pre-increment and post-increment

e. Increment operator is a **token**,✍ i.e. one unit. There should be no white-space character between two '+' symbols. If white space is placed between two '+' symbols, they become two unary plus (+) operators. Execute the code listed in Program 4-4 to understand the significance of white-space character.

Prog 4-4.c

```c
//++ is a token. Don't place white space in between + symbols
#include<stdio.h>
main()
{
    int a;
    a=+ +2;
    printf("The result of evaluation is %d",a);
}
```

Output window

```
The result of evaluation is 2
```

Remark:

- There will be no compilation error as in Program 4-2 because the expression a=+ +2 does not have an increment operator. Instead it has two unary plus operators, which can be applied on an operand that does not have a modifiable l-value

Program 4-4 | A program that illustrates the significance of white-space character in increment operator

✍ **Tokens** are the basic building blocks of a source code. Characters are combined into tokens according to the rules of the programming language. There are five classes of tokens: identifiers, reserved words, operators, separators and constants.

4. **Decrement operator**

   a. The decrement operator can appear towards the left side or towards the right side of its operand. If it appears towards the left side of its operand (e.g. --a), it is known as the **pre-decrement operator**. If it appears towards the right side of its operand (e.g. a--), it is known as the **post-decrement operator**.

   b. The decrement operator can only be applied to an operand that has a modifiable l-value. If it is applied on an operand that does not have a modifiable l-value, there will be a compilation error 'L-value required'.

   c. --a or a-- is equivalent to a=a-1.

   d. The **difference between pre-decrement and post-decrement** lies in the point at which the value of their operand is decremented.

      i. In case of the pre-decrement operator, first the value of its operand is decremented and then used for the evaluation of the expression in which it appears.

      ii. In case of the post-decrement operator, first the value of the operand is used for the evaluation of the expression in which it appears and then its value is decremented.

      The difference between two versions of the decrement operator is shown in the code listed in Program 4-5.

   e. Decrement operator is a token, i.e. one unit. There should be no white-space character between two '-' symbols. If white space is placed between two '-' symbols, they become two unary minus (-) operators.

5. **Division operator**

   a. The division operator is used to find the quotient.

Prog 4-5.c

```c
//Diff. between Pre-decrement & Post-decrement operator
#include<stdio.h>
main()
{
    int a=2, b=2,c,d;
    c=--a;
    d=b--;
    printf("a=%d, b=%d, c=%d, d=%d",a,b,c,d);
}
```

Output window

```
a=1, b=1, c=1, d=2
```

Reasons:

- The value of a is decremented and then it is assigned to c as a is pre-decremented
- The value of b is assigned to d before it is decremented as b is post-decremented

Program 4-5 | A program that illustrates the difference between pre-decrement and post-decrement

b. The sign of the result of evaluation of the division operator depends upon the sign of both the numerator as well as the denominator. If both are positive, the result will be positive. If both are negative, the result will be positive. If either of the two is negative, the result will be negative. For example: 4/3=1, -4/3=-1, 4/-3=-1 and -4/-3=1. This can be observed by executing the code listed in Program 4-6.

Prog 4-6.c

```c
//Sign of the result of division operator
#include<stdio.h>
main()
{
    printf("Sign of the result of division operator:\n");
    printf("4/3=%d, -4/3=%d\n",4/3,-4/3);
    printf("4/-3=%d, -4/-3=%d",4/-3,-4/-3);
}
```

Output window

```
Sign of the result of division operator:
4/3=1, -4/3=-1
4/-3=-1, -4/-3=1
```

Remark:

- The sign of the result of evaluation of the division operator depends upon the sign of the numerator as well as the denominator

Program 4-6 | A program that illustrates the sign of result of division operator

6. **Modulus operator**
   a. The modulus operator is used to find the remainder.
   b. The operands of modulus operator (i.e. %) must be of integer type. Modulus operator cannot have operands of floating point type. Try executing the code listed in Program 4-7.

Prog 4-7.c

```c
//Operands of the modulus operator must be of integer type
#include<stdio.h>
main()
{
    int a;
    a=2%3.0;
    printf("The value of a is %d",a);
}
```

Output window

```
Compilation error "Illegal use of floating point in the function main"
```

Reason:

- Operands of modulus operator should be of integer type

What to do?

- Write 3 instead of 3.0 or type cast 3.0 to int by writing (int)3.0

Program 4-7 | A program to illustrate that the operands of modulus operator must be of integer type

c. The sign of the result of evaluation of modulus operator depends only upon the sign of the numerator. If the sign of the numerator is positive, the sign of the result will be positive else negative. For example: 4%3=1, -4%3=-1, 4%-3=1 and -4%-3=-1. This can be observed by executing the code listed in Program 4-8.

Prog 4-8.c

```
//Sign of the result of modulus operator
#include<stdio.h>
main()
{
    printf("Sign of the result of modulus operator:\n");
    printf("4%%3=%d, -4%%3=%d\n",4%3,-4%3);
    printf("4%%-3=%d, -4%%-3=%d",4%-3,-4%-3);
}
```

Output window

```
Sign of the result of modulus operator:
4%3=1, -4%3=-1
4%-3=1, -4%-3=-1
```

Remarks:

- The sign of the result of evaluation of the modulus operator depends only upon the sign of the numerator
- The % sign marks the beginning of format specifier. If it is to be actually printed, use it twice. Refer Question number 16, Chapter 3

Program 4-8 | A program that illustrates the sign of result of modulus operator

4.4.2.2 Relational Operators

Relational operators are used to compare two quantities (i.e. their operands). There are six relational operators in C, which are given in Table 4.2.

Table 4.2 | Relational operators

S.No	Operator	Name of operator	Category	-ary of operator	Precedence among relational class	Associativity
1.	< > <= >=	Less than Greater than Less than or equal to Greater than or equal to	Relational operators	Binary	Level-I	L→R
2.	== !=	Equal to Not equal to	Equality operators	Binary	Level-II	L→R

The important points about the relational operators are as follows:

1. There should be no white-space character between two symbols of a relational operator.
2. The result of evaluation of a relational expression (i.e. involving relational operator) is a boolean constant, i.e. 0 or 1.
3. Each of the relational operators yields 1 if the specified relation is true and 0 if it is false. The result has type int.
4. The expression a<b<c is valid and is not interpreted as in ordinary mathematics. Since the less than operator (i.e. <) is left-to-right associative, the expression is interpreted as (a<b)<c. This means that 'if a is less than b, compare 1 with c, otherwise, compare 0 with c'.
5. An expression that involves a relational operator forms a **condition**. For example, a<b is a condition.

Consider Program 4-9 that illustrates the evaluation of a relational expression.

Prog 4-9.c

```
//Relational operators
#include<stdio.h>
main()
{
    int a;
    a=2<3!=2;
    printf("The value of a is %d",a);
}
```

Output window

```
The value of a is 1
```

Remark:

- The expression a=2<3!=2 is interpreted as a=(2<3)!=2. The sub-expression 2<3 is true (i.e. 1). 1!=2 is true (i.e. 1). So, 1 is assigned to a

Program 4-9 | A program that illustrates the use of relational operators

4.4.2.3 Logical Operators

Logical operators are used to logically relate the sub-expressions. The logical operators available in C are given in Table 4.3.

Table 4.3 | Logical operators

S.No	Operator	Name of operator	Category	-ary of operator	Precedence among logical class	Associativity
1.	!	Logical NOT	Unary	Unary	Level-I	R→L
2.	&&	Logical AND	Logical operator	Binary	Level-II	L→R
3.	||	Logical OR	Logical operator	Binary	Level-III	L→R

In C language, there is no operator available for logical e**X**clusive-**OR** (XOR) operation.

The important points about the logical operators are as follows:

1. Logical operators consider operand as an entity, a unit.
2. Logical operators operate according to the truth tables given in Table 4.4.

Table 4.4 | Truth tables of logical operations

AND Operation		
Operand1	**Operand2**	**Result**
False	False	False
False	True	False
True	False	False
True	True	True

(a)

OR Operation		
Operand1	**Operand2**	**Result**
False	False	False
False	True	True
True	False	True
True	True	True

(b)

NOT Operation	
Operand	**Result**
False	True
True	False

(c)

3. If an operand of a logical operator is a non-zero value, the operand is considered as true. If the operand is zero, it is considered as false.
4. Each of the logical operators yields 1 if the specified relation evaluates to true and 0 if it evaluates to false. The evaluation is done according to the truth tables mentioned in Table 4.4. The result has type int.
5. The logical AND (i.e. &&) operator and the logical OR (i.e. ||) operator guarantee left-to-right evaluation.
6. Expressions connected by the logical AND (&&) or the logical OR (||) operator are evaluated left to right and the evaluation stops as soon as truthfulness or falsehood of the expression is determined. Thus, in an expression:
   a. E1&&E2, where E1 and E2 are sub-expressions, E1 is evaluated first. If E1 evaluates to 0 (i.e. false), E2 will not be evaluated and the result of the overall expression will be 0 (i.e. false). If E1 evaluates to a non-zero value (i.e. true) then E2 will be evaluated to determine the truth value of the overall expression. The fragment of code in Program 4-10 illustrates the mentioned fact.

Line	Prog 4-10.c	Output window

1
2
3
4
5
6
7
8
9
	//Logical AND operator
	#include<stdio.h>
	main()
	{
	    int i=0,j=1,k=2,l;
	    l=i&&j++&&k++;
	    printf("Resultant values after evaluation are:\n");
	    printf("%d %d %d %d",i ,j,k,l);
	}
		Resultant values after evaluation are:
		0 1 2 0
		Remark:
		• The expression l=i&&j++&&k++ is interpreted as l=(i&&j++)&&k++. Since i is false, j++ will not be evaluated and (i&&j++) evaluates to 0 (i.e. false). Since (i&&j++) is false, k++ will not be evaluated and the expression i&&j++&&k++ evaluates to 0, i.e. false. So, 0 is assigned to l

Program 4-10 | A program that illustrates logical AND operation

   b. E1||E2, where E1 and E2 are sub-expressions, E1 is evaluated first. If E1 evaluates to a non-zero value (i.e. true), E2 will not be evaluated and the result of the overall expression will be 1 (i.e. true). If E1 evaluates to 0 (i.e. false) then E2 will be evaluated to determine the truth value of the overall expression. The fragment of code in Program 4-11 illustrates the mentioned fact.

Line	Prog 4-11.c	Output window

1
2
3
4
5
6
7
8
9
	//Logical OR operator
	#include<stdio.h>
	main()
	{
	    int i=0,j=1,k=2,l;
	    l=i&&j++||k++;
	    printf("Resultant values after evaluation are:\n");
	    printf("%d %d %d %d",i ,j,k,l);
	}
		Resultant values after evaluation are:
		0 1 3 1
		Remark:
		• The expression l=i&&j++||k++ is interpreted as l=(i&&j++)||k++. Since i is false, j++ will not be evaluated and (i&&j++) evaluates to 0 (i.e. false). Since (i&&j++) is false, k++ needs to be evaluated. k++ evaluates to 2 (i.e. true) and k becomes 3. The overall expression l=i&&j++||k++ evaluates to 1 (i.e. true). So, l=1

Program 4-11 | A program that illustrates logical OR operation

4.4.2.4 Bitwise Operators

The C language provides six operators for bit manipulation. These operators do not consider the operand as one entity and operate on the individual bits of the operands. The bitwise operators available in C are given in Table 4.5.

Table 4.5 | Bitwise operators

S.No	Operator	Name of operator	Category	-ary of operator	Precedence among bitwise class	Associativity
1.	~	Bitwise NOT	Unary	Unary	Level-I	R→L
2.	<< >>	Left Shift Right Shift	Shift operators	Binary	Level-II	L→R
3.	&	Bitwise AND	Bitwise operator	Binary	Level-III	L→R
4.	^	Bitwise X-OR	Bitwise operator	Binary	Level-IV	L→R
5.	|	Bitwise OR	Bitwise operator	Binary	Level-V	L→R

The important points about the bitwise operators are as follows:

1. Bitwise operators operate on the individual bits of the operands and are used for bit manipulation.
2. They can only be applied on operands of type `char`, `short`, `int`, `long`, whether `signed` or `unsigned`.
3. The bitwise-AND and the bitwise-OR operators operate on the individual bits of the operands according to the truth tables specified in Table 4.4.
4. The expression `2&3` evaluates to `2` and `2|3` evaluates to `3`. The operations on individual bits of operands (i.e. `2` and `3`) are shown in Figure 4.1.

Value, operator and result	Sign bit	Magnitude														
	Bit 16	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
`2`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
`3`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
`2&3=2`	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
`2|3=3`	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**

Figure 4.1 | Bitwise-AND and bitwise-OR operator operating on the individual bits of the operands

5. X-OR operator operates according to the truth table given in Table 4.6.

Table 4.6 | Truth table of X-OR operation

X-OR OPERATION		
Operand1	**Operand2**	**Result**
False	False	False
False	True	True
True	False	True
True	True	False

6. The bitwise NOT operator results in 1's complement of its operand.
7. Left shift by 1 bit is equivalent to multiplication by 2. Left shift by n bits is equivalent to multiplication by 2^n, provided the magnitude does not overflow.
8. Right shift by 1 bit is equivalent to an integer division by 2. Right shift by n bits is equivalent to integer division by 2^n.
9. The expression 4<<1 evaluates to 8 and 4>>1 evaluates to 2. This is shown in Figure 4.2.

Value, operator and result	Sign bit	Magnitude														
	Bit 16	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
4<<1=8	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**
4>>1=2	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**

Figure 4.2 | Left-shift and right-shift operations

4.4.2.5 Assignment Operators

A variable can be assigned a value by using an assignment operator. The assignment operators available in C language are given in Table 4.7.

Table 4.7 | Assignment operators

S.No	**Operator**	**Name of operator**	**Category**	**-ary of operator**	**Precedence**	**Associativity**
1.	= *= /= %= += -= &= |= ^= <<= >>=	Simple assignment Assign product Assign quotient Assign modulus Assign sum Assign difference Assign bitwise AND Assign bitwise OR Assign bitwise XOR Assign left shift Assign right shift	Assignment & Shorthand assignment operators	Binary	Level-I	R→L

The important points about the assignment operators are as follows:

1. The operand that appears towards the left side of an assignment operator should have a modifiable l-value. If the operand appearing towards the left side of the assignment operator does not have a modifiable l-value, there will be a compilation error 'L-value required'.
2. The shorthand assignment is of the form `op1 op=op2`, where `op1` and `op2` are operands and `op=` is a shorthand assignment operator. It is a shorter way of writing `op1 = op1 op op2`. For example, `a/=2` is equivalent to `a=a/2`.
3. There should be no white-space character between two symbols of shorthand assignment operators.
4. If two operands of an assignment operator are of different types, the type of operand on the right side of the assignment operator is automatically converted to the type of operand present on its left side. To carry out this conversion, either promotion or demotion is applied.
5. The result of evaluation of an assignment expression is the value that is assigned. For example, in the expression `a=10;`, the value `10` is assigned to `a` and the overall expression evaluates to `10` (i.e. the value that is assigned).
6. The terms assignment and initialization are related but it is important to note the differences between them. They are listed in Table 4.8.

Table 4.8 | Differences between initialization and assignment

S.No	Initialization	Assignment
1.	First time assignment at the time of definition is called initialization. For example: `int a=10;` is initialization of `a`	Value of a data object after initialization can be changed by the means of assignment. For example: Consider the following statements `int a=10; a=20;`. The value of `a` is changed to `20` by the assignment statement
2.	Initialization can be done only once	Assignment can be done any number of times
3.	Qualified constant can be initialized with a value. For example, `const int a=10;` is valid	Qualified constant cannot be assigned a value. It is erroneous to write `a=10;` if `a` is a qualified constant

4.4.2.6 Miscellaneous Operators

Other operators available in C are:

1. Function call operator (i.e. `()`)
2. Array subscript operator (i.e. `[]`)
3. Member select operator
   a. Direct member access operator (i.e. . (dot operator or period))
   b. Indirect member access operator (i.e. -> (arrow operator))
4. Indirection operator (i.e. *)
5. Conditional operator

6. Comma operator
7. sizeof operator
8. Address-of operator (i.e. &)

4.4.2.6.1 Conditional Operator

Conditional operator is the only ternary operator available in C (Table 4.9).

Table 4.9 | Conditional operator

S.No	Operator	Name of operator	Category	-ary of operator	Precedence	Associativity
1.	?:	Conditional operator	Conditional	Ternary	Level-I	R→L

The important points about the conditional operator are as follows:

1. The general form of conditional operator is E1?E2:E3, where E1, E2 and E3 are sub-expressions.
2. The sub-expression E1 must be of scalar type.✍
3. The sub-expression E1 is evaluated first. If it evaluates to a non-zero value (i.e. true), then E2 is evaluated and E3 is ignored. If E1 evaluates to zero (i.e. false), then E3 is evaluated and E2 is ignored.

> ✍ Integer and floating types are collectively called **arithmetic types**. Arithmetic types and pointer types are collectively called **scalar types**.

4.4.2.6.2 Comma Operator

The comma operator is used to join multiple expressions together (Table 4.10).

Table 4.10 | Comma operator

S.No	Operator	Name of operator	Category	-ary of operator	Precedence	Associativity
1.	,	Comma operator	Comma	Binary	Level-I	L→R

The important points about the comma operator are as follows:

1. Every instance of a comma symbol is not a comma operator. The commas separating arguments in a function call are not comma operators. If commas separating arguments in a function call are considered as comma operators, then no function could have more than one argument. The commas used in the declaration/definition statement are not considered as comma operators. The commas appearing between the arguments in a function call or commas appearing in a declaration/definition statement are separators.

2. The comma operator guarantees left-to-right evaluation.
3. In expression E1, E2, E3…En, the sub-expressions E1, E2, E3…En are evaluated in left-to-right order. The result and type of evaluation of the overall expression is the value and type of the evaluation of the rightmost sub-expression, i.e. En.
4. The comma operator has least precedence.

The piece of code in Program 4-12 illustrates the use of a comma operator.

Prog 4-12.c

```c
//Use of comma operator
#include<stdio.h>
main()
{
    int a,b;
    a=1, 2, 3, 4, 5;
    b=(1, 2, 3, 4, 5);
    printf(“Resultant values of a and b are:\n”);
    printf(“%d %d”,a ,b);
}
```

Output window

```
Resultant values of a and b are:
1 5
```

Remarks:

- The precedence of assignment operator is greater than comma operator
- Thus, in the expression a=1,2,3,4,5, the sub-expression a=1 gets evaluated first. Hence, the value assigned to a is 1
- In the expression b=(1,2,3,4,5), the sub-expression 1,2,3,4,5 is parenthesized and will be evaluated first. The result of evaluation of comma operator is the result of evaluation of the rightmost sub-expression, i.e. 5. Thus, (1,2,3,4,5) evaluates to 5 and is assigned to b

Program 4-12 | A program to illustrate the use of comma operator

4.4.2.6.3 sizeof Operator

The sizeof operator is used to determine the size in bytes, which a value or a data object will take in memory (Table 4.11).

Table 4.11 | sizeof operator

S.No	Operator	Name of operator	Category	-ary of operator	Precedence	Associativity
1.	sizeof	Size-of operator	Unary	Unary	Level-I	R→L

The important points about the sizeof operator are as follows:

1. The general form of a sizeof operator is:
   a. **sizeof expression** or **sizeof (expression)** (For example: sizeof 2, sizeof(a), sizeof(2+3))
   b. **sizeof (type-name)** (For example: sizeof(int), sizeof(int*), sizeof(char))
2. Parentheses should be used if the sizeof operator is applied on a type-name, as indicated in point 1 b) above.
3. The type of result of evaluation of the sizeof operator is int.
4. The operand of the sizeof operator is not evaluated. This fact can be seen by executing the code listed in Program 4-13.

Prog 4-13.c

```
//sizeof operator
#include<stdio.h>
main()
{
    int a=1,b;
    b=sizeof(++a);
    printf("Resultant values of a and b are:\n");
    printf("%d %d",a ,b);
}
```

Output window

```
Resultant values of a and b are:
1 2
```

Remark:

- The operand of sizeof operator is not evaluated. Hence, ++a is not evaluated and thus, the value of a remains unchanged, i.e. 1. The value of a takes 2 bytes in memory (in case of MS-VC++ 6.0, it takes 4 bytes). Thus, the value of b is 2

Program 4-13 | A program to illustrate that operand of sizeof operator is not evaluated

5. The sizeof operator cannot be applied on operands of incomplete type or function type.

4.4.2.6.4 Address-of Operator

The address-of operator is used to find the address, i.e. l-value of a data object (Table 4.12).

Table 4.12 | Address-of Operator

S.No	Operator	Name of operator	Category	-ary of operator	Precedence	Associativity
1.	&	Address-of operator	Unary	Unary	Level-I	R→L

The important points about the address-of operator are as follows:

1. The address-of operator must appear towards the left side of its operand.
2. The syntax of using the address-of operator is &operand.
3. The operand of the address-of operator should be a variable or a function designator. The address-of operator cannot be applied to constants, expressions, bit-fields and to the variables declared with register storage class.

4.5 Combined Precedence of All Operators

Till now, I have described different operators according to their role and have categorized them into various classes like arithmetic operators, relational operators, etc. I have described the precedence of operators within a class (i.e. intra-class precedence). Now, it is the time to consider the precedence of an operator with respect to the operators in other classes (i.e. inter-class precedence). Table 4.13 provides a combined table of precedence.

Table 4.13 | Combined precedence chart

S.No	Operator	Name of operator	Category	-ary of operator	Precedence	Associativity
1.	() [] -> .	Function call Array subscript Indirect member access Direct member access			Level-I (Highest)	
2.	! ~ + - ++ -- & * sizeof	Logical NOT Bitwise NOT Unary plus Unary minus Increment Decrement Address-of Deference Sizeof	Unary operators	Unary	Level-II	R→L
3.	* / %	Multiplication Division Modulus	Multiplicative operators	Binary	Level-III	L→R
4.	+ -	Addition Subtraction	Additive operators	Binary	Level-IV	L→R
5.	<< >>	Left Shift Right Shift	Shift operators	Binary	Level-V	L→R
6.	< > <= >=	Less than Greater than Less than or equal to Greater than or equal to	Relational operators	Binary	Level-VI	L→R
7.	== !=	Equal to Not equal to	Equality operators	Binary	Level-VII	L→R
8.	&	Bitwise AND	Bitwise operator	Binary	Level-VIII	L→R
9.	^	Bitwise X-OR	Bitwise operator	Binary	Level-IX	L→R
10.	|	Bitwise OR	Bitwise operator	Binary	Level-X	L→R
11.	&&	Logical AND	Logical operator	Binary	Level-XI	L→R
12.	||	Logical OR	Logical operator	Binary	Level-XII	L→R
13.	?:	Conditional operator	Conditional	Ternary	Level-XIII	R→L

(Contd...)

S.No	Operator	Name of operator	Category	-ary of operator	Precedence	Associativity
14.	= *= /= %= += -= &= |= ^= <<= >>=	Simple assignment Assign product Assign quotient Assign modulus Assign sum Assign difference Assign bitwise AND Assign bitwise OR Assign bitwise XOR Assign left shift Assign right shift	Assignment & Shorthaınd assignment operators	Binary	Level-XIV	R→L
15.	,	Comma operator	Comma	Binary	Level-XV (Least)	L→R

4.6 Reading Strings from the Keyboard

The user can enter strings and store them in character arrays at the run time in a similar manner as the string literal constants can be stored in the character arrays at the compile time. The methods that can be used to read strings from the user at the run time are as follows:

1. **Using scanf function:** The scanf function with %s format specification can be used to read a string from the user and store it in a character array. The code snippet in Program 4-14 illustrates the use of the scanf function to read a string from the user.

Prog 4-14.c

```
//Reading strings using the scanf function
#include<stdio.h>
main()
{
    char name[20];
    printf("Enter your name\t");
    scanf("%s",name);
    printf("Your name is %s",name);
}
```

Output window

```
Enter your name     Sam
Your name is Sam
```

Remark:

- The scanf function automatically terminates the input string with a null character, and therefore the character array should be large enough to hold the input string plus the terminating null character

Program 4-14 | A program to illustrate the use of scanf function to read a string from the user at the run time

The important points about the use of scanf function for reading strings are as follows:

a. The scanf function with %s specifier reads all the characters up to, but not including, the white-space character. For example, in Program 4-14, instead of entering the first name, enter the full name, e.g. "Sam Mine". Even on entering the full name, the output of the program would be "Your name is Sam". This happens because the scanf function reads the characters only up to the first white-space character.

Thus, scanf function with %s specifier can be used to read single word strings like "Sam" but cannot be used to read multi-word strings like "Sam Mine".

b. The scanf function can be used to read a specific number of characters by specifying the field width. The code snippet in Program 4-15 illustrates the use of a field width specifier.

Prog 4-15.c

```
//Field width specifier and scanf function
#include<stdio.h>
main()
{
    char name[20];
    printf("Enter your name\t");
    scanf("%3s",name);
    printf("Your name is %s",name);
}
```

Output window

```
Enter your name     Samuel
Your name is Sam
```

Remarks:

- If the length of the entered string is more than the specified field width, the number of characters read will be at most equal to the field width
- The scanf function reads all characters up to, but not including, the white-space character even if the value of field width specification is more than the position of first white-space character

Program 4-15 | A program to illustrate the use of a field width specifier and the scanf function

c. The scanf function can also be used to read selected characters by making use of search sets. A **search set** defines a set of possible characters that can make up the string. The rules to write search sets are as follows:

i. The possible set of characters making up the search set is enclosed within square brackets, e.g. [abcd]. The scanf function reads all the characters up to but not including the one that does not appear in a search set. If a search set [abcd] is used, the scanf function reads the input characters and stops when a character except a, b, c or d is encountered. The code snippet in Program 4-16 illustrates this fact.

Prog 4-16.c

```
//Search set and scanf function
#include<stdio.h>
main()
{
    char name[20];
    printf("Enter your name\t");
    scanf("%[abcd]",name);
    printf("Your name is %s",name);
}
```

Output window

```
Enter your name     daman
Your name is da
```

Remarks:

- Search sets are case sensitive
- If the specified search set is [abcd] and the entered string is Daman, no character will be read, as the character D does not belong to the search set

Program 4-16 | A program to illustrate the use of a search set and the scanf function

ii. If the first character in the bracket is a caret (i.e. ^), the search set is inverted to include all the characters (even white-space characters) except those between the brackets. For example, the search set [^abcd] searches the input for any character except a, b, c and d. The scanf function reads the input characters and stops when the characters a, b, c or d are encountered. The code snippet in Program 4-17 illustrates this fact.

Prog 4-17.c

```
//Inverted search set and scanf function
#include<stdio.h>
main()
{
    char name[20];
    printf(“Enter your name\t”);
    scanf(“%[^abcd]”,name);
    printf(“Your name is %s”,name);
}
```

Output window

```
Enter your name     Neha
Your name is Neh
```

Caution:

- The input will only terminate when any character specified within the brackets is encountered
- Matching process is case sensitive
- Re-execute the code and enter the name in uppercase, i.e. NEHA. The input will not terminate even on pressing enter. Enter character 'a' and then press enter. The input will terminate

Program 4-17 | A program to illustrate the use of inverted search set and the scanf function

The inverted search set can be used with the scanf function to **read a line of text**. The code snippet in Program 4-18 illustrates the use of an inverted search set to read a line of text.

Prog 4-18.c

```
//Reading a line of text using inverted search set
#include<stdio.h>
main()
{
    char line[50];
    printf(“Enter a line of text:\n”);
    scanf(“%[^\n]”,line);
    printf(“The text you entered is:\n%s”,line);
}
```

Output window

```
Enter a line of text:
We can change our destiny!!
The text you entered is:
We can change our destiny!!
```

Remark:

- The inverted search set [^\n] can be used to read the characters till the new line character is encountered

Program 4-18 | A program to illustrate the use of an inverted search set to read a line of text

iii. The search set can be used for including the characters that lie within a particular range. For example, the search set %[d-f] searches the input for any character that lies in the range d to f, i.e. d, e and f.

d. The scanf function automatically terminates the input string with a null character and therefore the character array should be large enough to hold the input string plus the terminating null character.

e. It is not mandatory to use ampersand, i.e. address-of operator (&) with string variable names while reading strings using the scanf function. The reason behind this relaxation is that the scanf function requires an l-value as an argument where it can store the input. Since the string variable is a character array and the name of an array refers to the address of the first element of the array, the string variable name itself refers to the l-value. However, if an address-of operator is used with the string variable name, there will be no problem since it also refers to the same address. The code snippet in Program 4-19 illustrates this fact.

Prog 4-19.c

```c
//Usage of address-of operator with
//string variable is not mandatory
#include<stdio.h>
main()
{
    char name[5];
    printf("Enter your name\t");
    scanf("%s",name);
    printf("Your name is %s\n",name);
    printf("Enter your name again\t");
    scanf("%s",&name);
    printf("Your name is %s\n",name);
}
```

Memory contents

name

A	j	a	y	\0
4000	4001	4002	4003	4004

Output window

```
Enter your name      Ajay
Your name is Ajay
Enter your name again      Ajay
Your name is Ajay
```

Remarks:

- Usage of address-of operator while using a string variable with the scanf function is not mandatory
- Both name and &name refer to the same memory address, i.e. 4000

Remember:

- The difference between name and &name is that the type of name is char* while that of &name is char(*)[5]

Program 4-19 | A program to illustrate that the usage of address-of operator with a string variable is not mandatory

2. **Using getchar function:** The getchar function is used to read a character from the terminal, i.e. keyboard. The prototype of the getchar function is int getchar(void); and is available in the stdio.h header file. The getchar function reads a character from the keyboard and returns the ASCII code of the read character. Since a string is a sequence of characters, the getchar function can be called repeatedly to read a string. The code snippet in Program 4-20 illustrates the use of the getchar function to read a string.
3. **Using gets function:** Another convenient way to accept a string from the user at the run time is by using the gets library function. The prototype of the gets function is char* gets(char*); and is available in the stdio.h header file. The gets function accepts a character array or a character pointer as an argument, reads characters from the keyboard until a new line character is encountered, stores them in a character array or in the memory location pointed by the character pointer, appends a null character to the string and returns the starting address of the location where the string is stored. The code snippet in Program 4-21 illustrates the use of the gets function.

Prog 4-20.c

```c
//Iterative use of getchar function to read a string
#include<stdio.h>
main()
{
    char ch, line[50];
    int loc=0;
    printf("Enter a line of text:\n");
    while((ch=getchar())!='\n')
        line[loc++]=ch;
    line[loc]='\0';
    printf("The text you entered is:\n%s",line);
}
```

Output window

```
Enter a line of text:
We can change our destiny!!
The text you entered is:
We can change our destiny!!
```

Program 4-20 | A program to illustrate the use of the getchar function to read a string

Line	Prog 4-21.c	Output window

1
2
3
4
5
6
7
8
9
10
11
	//Use of gets function to read a string
	#include<stdio.h>
	main()
	{
	    char plang[50];
	    printf("Enter name of a programming language\n");
	    gets(plang);
	    printf("First programming language is %s\n",plang);
	    printf("Enter name of another programming language\n");
	    printf("Second programming language is %s\n",gets(plang));
	}
		Enter name of a programming language
		Visual Basic
		First programming language is Visual Basic
		Enter name of another programming language
		Visual C#
		Second programming language is Visual C#
		Remarks:
		• The gets function can be used to read multi-word strings.
		• Since the gets function returns the pointer to the input string, it can be used as an argument within the printf function (as done in line number 10)

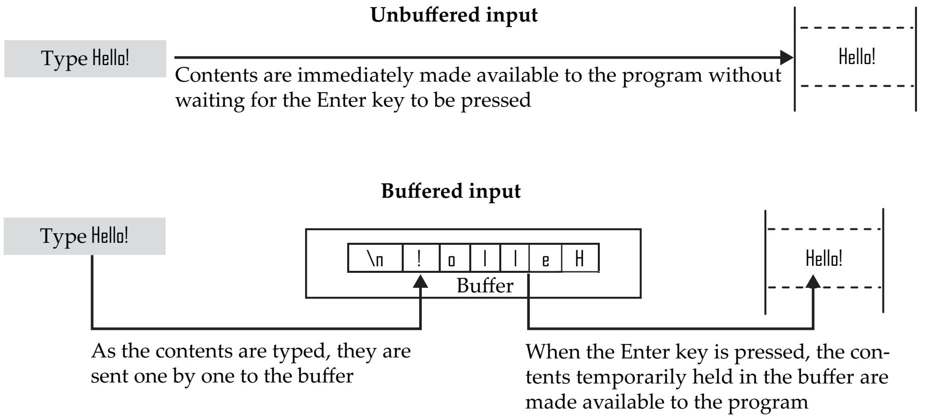
Program 4-21 | A program to illustrate the use of the gets function to read a string

The important points about the gets function are as follows:

a. Unlike the scanf function, the gets function reads the entire line of text until a new line character is encountered and does not stop upon encountering any other white-space character.
b. Thus, the gets function is suited for reading multi-word strings.

The important points about the input functions mentioned above are as follows:

a. The input functions are categorized into **buffered input functions** and **unbuffered input functions.**
b. In buffered input, the input given is kept in a temporary memory area known as the **buffer** and is transmitted to the program when the Enter key is pressed. The pressed Enter key is also transmitted to the program in the form of a new line character, which the program must handle.
c. In unbuffered input, the given input is immediately transferred to the program without waiting for the Enter key to be pressed.
d. The difference between buffered and unbuffered input is depicted in Figure 4.3.



Contents are immediately made available to the program without waiting for the Enter key to be pressed

As the contents are typed, they are sent one by one to the buffer

When the Enter key is pressed, the contents temporarily held in the buffer are made available to the program

Figure 4.3 | Unbuffered and buffered input

e. The examples of buffered input functions are scanf, getchar and gets function.
f. The examples of unbuffered input functions are getch and getche function.
g. Program 4-20 can be rewritten using the unbuffered input function getche as in Program 4-22.

Line	Prog 4-22.c	Output window

1
2
3
4
5
6
7
8
9
10
11
12
13
	//Iterative use of getche function to read a string
	#include<stdio.h>
	#include<conio.h>
	main()
	{
	    char ch, line[50];
	    int loc=0;
	    printf("Enter a line of text:\n");
	    while((ch=getche())!='\r')
	        line[loc++]=ch;
	    line[loc]='\0';
	    printf("\nThe text you entered is:\n%s",line);
	}
		Enter a line of text:
		We can change our destiny!!
		The text you entered is:
		We can change our destiny!!
		Remarks:
		• The prototype of the getche function is present in the header file conio.h
		• The sentinel value to be used for unbuffered input functions like getche is '\r', i.e. carriage return character instead of '\n', i.e. new line character that is used for buffered input functions

Program 4-22 | A program to illustrate the use of the getche function to read a string

4.7 Printing Strings on the Screen

The methods that can be used to print strings on the screen are as follows:

1. **Using printf function:** The printf function can be used to print a string literal constant, the contents of a character array and the contents of the memory locations pointed by a character pointer on the screen in two different ways:
   a. **Without using format specifier:** The printf function can print strings onto the screen without using any format specifier. The code snippet in Program 4-23 illustrates this use.

Line	Prog 4-23.c	Output window

1
2
3
4
5
6
7
8
9
10
11
	//Printing a string with the help of printf function without using any
	//format specifier
	#include<stdio.h>
	main()
	{
	    char str[20]="Readers!!"; //Array holding string
	    char* ptr="Dear"; //Character pointer pointing to a string
	    printf("Hello"); // Printing string literal constant
	    printf(ptr) // Printing string pointed to by a character pointer
	    printf(str); // Printing contents of character array
	 }
		HelloDearReaders!!
		Remarks:
		• The first argument of the printf function must be of type const char*
		• A string literal constant and a string variable name refer to const char*
		• Hence, the usage of the printf function as done in line numbers 8, 9 and 10 is perfectly valid

Program 4-23 | A program to illustrate the use of the printf function without a format specifier to print strings

The important points about this type of usage are as follows:

i. The first argument of the printf function must be of const char* type. Since the string variable name and the string literal constant implicitly decompose into const char*, this type of usage is perfectly valid.

ii. This type of usage however has a limitation that the contents of only one character array or the contents pointed by only one character pointer can be printed at a time.

b. **Using %s format specifier:** The second way to print the strings on the screen is by using the printf function along with the %s format specifier. The code snippet in Program 4-24 illustrates this use.

Prog 4-24.c

```c
//Printing strings by using printf function along with %s
//format specifier
#include<stdio.h>
main()
{
    char str[20]="Readers!!";
    char* ptr="Dear";
    printf("%s%s%s","Hello", ptr,str);
}
```

Output window

```
HelloDearReaders!!
```

Remarks:

- %s specifier is used to print a string literal, the contents of a character array and a string literal pointed to by a character pointer
- Two or more strings can be printed by a single call to the printf function having multiple %s specifiers

Program 4-24 | A program to illustrate the use of the printf function the along with %s specifier to print strings

This type of usage has an advantage that two or more strings can be printed by a single call to the printf function having multiple %s specifiers.

2. **Iteratively printing a string's constituent characters:** A string can be printed by iteratively printing its constituent characters. They can be printed either by using the putchar function or by using the putch function. The prototype of the putchar function is int putchar(int c); and is present in the header file stdio.h. The prototype of the putch function is int putch(int c); and is present in the header file conio.h. The code snippet in Program 4-25 prints the strings by using these functions.

Line | Prog 4-25.c

```c
 1  //Printing string by iteratively printing its constituent characters
 2  #include<stdio.h>
 3  #include<conio.h>
 4  main()
 5  {
 6      char str[20]="Hello";
 7      char *ptr="Dear";
 8      int i=0, j=0 ;
 9      while(str[i]!='\0')
10          printf("%c",str[i++]);
11      while(*ptr!='\0')
12          putch(*ptr++);
13      while(*("Readers!!"+j)!='\0')
14      {
15          putchar(*("Readers!!"+j));
16          j++;
17      }
18  }
```

Output window

```
HelloDearReaders!!
```

Remark:

- A character can also be printed by using the printf function and the %c format specifier as shown in line number 10

Program 4-25 | A program to illustrate the printing of a string by printing its constituent characters

3. **Using puts function:** Another convenient way to print the strings on the screen is by using the puts function. The prototype of the puts function is int puts(const char*); and is available in the stdio.h header file. The puts function prints the string on the screen and returns the number of characters printed. The code snippet in Program 4-26 illustrates the use of the puts function to print strings.

Prog 4-26.c

```
//Use of puts function to print a string
#include<stdio.h>
main()
{
    char str[20]="Readers!!";
    char* ptr="Dear";
    puts("Hello");
    puts(ptr);
    puts(str);
 }
```

Output window

```
Hello
Dear
Readers!!
```

Remarks:

- The argument to the puts function can be a string literal constant or a character array or a character pointer pointing to a string
- The puts function prints the string and places a new line character after the string

Program 4-26 | A program to illustrate the use of the puts function to print a string

The important points about the usage of the puts function are as follows:

i. It has a limitation that only one string can be printed at one time.

ii. The difference between the puts function and the printf function is that the puts function places a new line character after printing the string, whereas the printf function does not. Compare the outputs of Programs 4-23 and 4-26.

4.8 Unformatted Functions

C has three types of I/O functions.

a) Character I/O
b) String I/O
c) File I/O

a) Character I/O

I. **getchar()** This function reads character type data from the standard input. It reads one character at a time till the user presses the enter key.

The syntax of getchar() is as follows:

```
Variable name=getchar();
Example
char c;
c=getchar();
```

A program using getchar() function.

```
void main()
{
  char c;
  clrscr();
  printf("\nEnter a char :");
  c=getchar();
  printf("a=%c",c);
}
OUTPUT:
  Enter values :g
  a=g
```

Explanation

In the above program, a character variable c is declared. The getchar() reads a character through the keyboard. The same is displayed by printf() statement.

Example 4-1 | A program to accept characters through keyboard using getchar() function

```
void main()
{
  char ch[20];
  int c=0;
  clrscr();
  while((ch[c]=getchar())!='\n')
  c++;
  ch[c]='\0';
  printf("\n%s",ch);
}
OUTPUT:
  COMPILER
  COMPILER
```

Explanation

In the above program, a character array is declared. Using the while loop continuously, characters are read through the keyboard using getchar() function till the user presses enter. Using printf() statement, the entered string is displayed.

Example 4-2 | A program to accept characters through keyboard using getchar() function

II. **putchar()** This function prints one character on the screen at a time, which is read by the standard input.

The syntax is as follows:

```
putchar(variable name);
Example
char c='C';
putchar(c);
```

A program is provided on putchar().

```
void main()
{
  char c=‘C’;
  clrscr();
  putchar(c);
}

OUTPUT
  C
```

Explanation

In this program, the character variable c assigned char ‘C’ and the same is displayed by the putchar() statement. The argument c is used with putchar() statement.

Example 4-3 | A program to use putchar() in work

```
void main()
{
  char ch[20];
  int c=0;
  clrscr();
  printf(“\n Enter Text Here:”);
  scanf(“%s”,ch);
  printf(“\n The Entered Text:”);
  while(ch[c]!=‘\0’)
  {
   putchar(ch[c]); c++;
  }
}

OUTPUT:
  Enter Text Here : Characters
  The Entered Text : Characters
```

Explanation

In the above program, the scanf() statement reads the string from the terminal. The putchar() function displays one character at a time on the console. The while loop causes repetitive execution of putchar() function and increases the counter c. Counter c is used to display the successive characters. When ‘\0’ is encountered, the program is terminated.

Example 4-4 | A program to print the characters using putchar() functions

III. **getch()** and **getche()** These functions read the alphanumeric characters from the standard input device. The character entered is not displayed by getch() function.

```
void main()
{
  clrscr();
  printf("Enter any two alphabets");
  getche();
  getch();
}
OUTPUT:
  Enter any two alphabets A
```

Explanation

In the above program, even though the two characters are entered, the user can see only one character on the screen. The second character is accepted but not displayed on the console. The **getche()** accepts and displays the character, whereas **getch()** accepts but does not display the character.

Example 4-5 | A program to show the effect of getche() and getch()

IV. **putch()** This function prints any alphanumeric character taken by the standard input device.

```
void main()
{
  char ch;
  clrscr();
  printf("Press any key to continue");
  ch=getch();
  printf("\n You Pressed :");
  putch(ch);
}
OUTPUT:
Press any key to continue You Pressed: 9
```

Explanation

The function getch() reads a key stroke and assigns to the variable ch. putch() displays the character pressed.

Example 4-6 | A program to read and display the character using getch() and putch()

b) String I/O

I. **gets()** This function is used for accepting any string through stdin (keyboard) until enter key is pressed.

```
void main()
{
  char ch[30];
  clrscr();
  printf("Enter the String :");
  gets(ch);
  printf("\n Entered String: %s", ch);
}
```

```
OUTPUT:
  Enter the String : USE OF GETS()
  Entered String : USE OF GETS()
```

Explanation

In the above program, gets() reads string through the keyboard and stores it in character array ch[30]. The printf() function displays the string on the console.

Example 4-7 | A program to accept string through the keyboard using gets() function

II. **puts()** This function prints the string or character array.

```
void main()
{
  char ch[30];
  clrscr();
  printf("Enter the String :");
  gets(ch);
  puts("Entered String :");
  puts(ch);
}
```

```
OUTPUT:
  Enter the String: puts is in use.
  Entered String: puts is in use.
```

Explanation

This program is same as the previous one. Here, to display the string puts() function is used.

Example 4-8 | A program to print the accepted character using puts() function

III. **cgets()** This function reads string from the console. The syntax is given below.

Syntax

```
cgets(char *st);
```

It requires character pointer as an argument. The string begins from st[2].

IV. **cputs()** This function displays string on the console. The syntax is given below.

Syntax

```
cputs(char *st);
```

```
void main()
{
  static char *t;
  clrscr()
  printf("\n Enter Text Here:");
  cgets(t);
  t+=2;
  printf("\n Your Entered Text:");
  cputs(t);
  getche();
}
```

```
OUTPUT:
  Enter Text Here: How are you?
  Your Entered Text: How are you?
```

Explanation

In this example, character pointer t is declared. The cgets() function reads string through the keyboard and the cputs() function displays the string on the console.

Example 4-9 | A program to read string using cgets() and display it using cputs()

4.9 Summary

1. Operand is an entity on which an operation is performed.
2. Operator specifies the operation to be performed on an operand.
3. Expression is made up of operands and operators.
4. Operands constituting an expression can be identifiers, constants or expressions themselves. The identifiers allowed to constitute an expression are variables, functions and macros. However, label names, typedef name, tags of structure, union or enumeration cannot be a part of an expression. The expressions forming an expression are called **sub-expressions**. An expression that is not a part of another expression is called **full expression**.
5. Based upon the number of operators in an expression, the expressions are classified as simple expressions and compound expressions.
6. Simple expressions have only one operator.
7. There is more than one operator present in a compound expression. To evaluate a compound expression, the order in which the operators will operate is to be determined.
8. The order in which operators operate depends upon the precedence and the associativity of the operators.
9. In a compound expression, if operators of different precedence appear together, the operator of the higher precedence operates first.

10. In a compound expression, if operators of the same precedence appear together, then precedence is not sufficient to determine the order in which operators will operate. The order of evaluation can be determined by looking at the associativity of the operators.
11. If operators are left-to-right associative, the operator that appears first in the left-to-right traversal will operate first.
12. If operators are right-to-left associative, the operator that appears first in the right-to-left traversal will operate first.
13. The operators with the same precedence have the same associativity but vice versa is not true.
14. In an arithmetic expression, if the operands of a binary operator are of a different type, C automatically applies arithmetic-type conversion to bring the operands to a common type. The type of result of the binary operator will also be the common type.
15. Automatic-type conversion is called implicit-type conversion.
16. Type can also be changed by applying explicit-type conversion.
17. Explicit-type conversion is done with the help of a type cast operator, i.e. (). The syntax of using the type cast operator is **(target-type-name) expression**.

Exercise Questions

Conceptual Questions and Answers

1. *I have heard that white-space characters are ignored in C. If I write the statement* `a+ =2;` *in a C program, there is a compilation error. However, if I write it as* `a+=2;` *it works. Why is the blank space (i.e. a white-space character) between* + *and* = *not getting ignored?*

   Every white-space character is not ignored in C. White-space characters separating tokens are not significant and are ignored in C. Here, '+=' is a token (i.e. a single unit, one operator). We cannot have white space in between + and =. The occurrence of a white-space character between them makes '+' and '=' two different tokens (i.e. two different operators and both are binary operators). Two binary operators cannot come next to each other without having any operand in between. This leads to an error.
   The following are allowed:

   ```
   1. a        +=        2;
   2. a+=                2;
   3. a                +=2;
   ```

   because white-space characters come in between tokens and not within a token. Similarly, **printf ("Hello");** can be written and will work but **pri ntf("Hello");** will not work because the white-space character does not separate different tokens but comes within a token (i.e. **printf**). Thus, the statement 'White-space characters are ignored in C' can be corrected and refined as 'Non-significant white-space characters are ignored in C'.

2. *I want to check whether a number* b *lies in between numbers* a *and* c*. I have written the following segment of code:*

   ```
   if(a<b<c)
       printf("b lies between a and c");
   else
      printf("b is an outlier");
   ```

The above segment of code does not work for all test cases. Why? Correct the code so that it starts working as intended.

The answer to why this code does not work for all test cases lies in understanding how expression a<b<c gets evaluated. In the expression a<b<c, two less than operators (<) are involved. The less than operator is left-to-right associative, thus the expression a<b<c is interpreted as (a<b)<c. a<b is evaluated first. Less than is a relational operator and the outcome of a relational operator is a boolean constant, i.e. 1 (true) or 0 (false). Therefore, a<b can be 1 or 0, depending upon whether a is less than b or not. Then, the result of comparison of a and b gets compared with c. Therefore, instead of b getting compared with c, 0 or 1 gets compared with c. Here lies the flaw.

Suppose a=2, b=1 and c=5, in a<b<c (i.e. 2<1<5), 2<1 is false, i.e. 0. Therefore, the expression becomes 0<5. 0<5 is true, i.e. 1; hence, the output will be 'b lies between a and c', which is wrong.

Instead of writing a<b<c, the expression should be written as a<b&&b<c. The correct code is:

```
if(a<b&&b<c)
    printf("b lies between a and c");
else
    printf("b is an outlier");
```

In the expression a<b&&b<c (i.e. 2<1&&1<5), 2<1 is false. Therefore, the entire expression evaluates to false and the output is 'b is an outlier'.

3. *A programmer wants to find the average of three numbers. He has written the following piece of code in C:*

```
main()
{
    int a=10,b=12,c=13, average;
    average=a+b+c/3;
    printf("Average is %d",average);
}
```

Does the mentioned piece of code produce the correct result as intended? If no, why?

No, the code does not produce the intended result due to the following reasons:

1. The division operator has a higher precedence than the addition operator. Hence, the expression average=a+b+c/3 is interpreted as average=a+b+(c/3) instead of being interpreted as average=(a+b+c)/3.
2. The type of the variable average is taken as int instead of float.

4. *If the code in the previous question is rectified and rewritten as*

```
main()
{
    int a=10,b=12,c=13;
    float average;
    average=(a+b+c)/3;
    printf("Average is %f",average);
}
```

does this code produce the correct result? If no, why? Rewrite the code, so that it produces the correct result.

Still the code will not produce the correct result. This is due to the fact that in the expression average=(a+b+c)/3, the sub-expression a+b+c will be evaluated first and then it is divided by 3. 10+12+13 turns out to be 35. 35/3 gives 11. (As both 35 and 3 are integers, integer mode arithmetic is applicable. In this mode, the result of evaluation of binary arithmetic operator is an integer.) Now, 11 is assigned to a float variable. Before assigning an integer value to a float variable, the integer value gets promoted (i.e. converted into float). Thus, 11 get promoted to 11.0. Therefore, the average value that gets printed is 11.000000 instead of 11.666667.

The reason behind this problem is the application of integer arithmetic instead of floating point arithmetic. We must do something so that floating point arithmetic or mixed mode arithmetic is applied. To make this happen, any one of the below-mentioned ways can be adopted:

```
1. average=(a+b+c)/3.0;          //←Implicit-type conversion
2. average=(float)(a+b+c)/3;     //←Explicit-type conversion
3. average=(a+b+c)/(float)3;     //←Explicit-type conversion
```

In all the three cases, division is carried out between an int value and a float value. Thus, mixed mode arithmetic is applicable instead of integer arithmetic and the result of computation turns out to be a float value. By using any one of the above three ways, 'Average is 11.666667' gets printed.

5. *The output of the following piece of code turns out to be* 81 *instead of the expected output* 300. *Why does this happen? Suggest possible ways to rectify this problem.*

```
main()
{
    int a=100,b=900,c;
    c=a*b/300;
    printf("The value that c gets is %d",c);
}
```

The expression c=a*b/300 contains three operators, namely assignment operator, multiplication operator and division operator. Multiplication and division operators have the same precedence. The assignment operator has a lesser precedence than these operators. Therefore, multiplication and division operators will be evaluated prior to the assignment operator. Being left-to-right associative, multiplication will be carried out first as the multiplication operator appears towards the left. When 100 and 900 (i.e. both integers) get multiplied, the result turns out to be 90000, which exceeds the range of integer data type. Since the value exceeds the range, wrap around will occur and 90000 will be mapped to 90000–65536 = 24464. Now, this number is divided by 300 to give 81 as the result. Therefore, this problem occurs due to overflow and wrap-around effect.

In order to avoid this problem, we should prevent this overflow and wrap around. This can be done by using range of long integer type instead of integer type. The following alternatives will solve the problem:

```
1. c=(long)a*b/300;
2. c=a*(long)b/300;
```

Now, long integer and integer gets multiplied and the result turns out to be a long integer. 90000 is well within the range of long integer type; hence no overflow occurs.

It is very important to note that the following ways do not solve the problem:

```
1. c=(long)(a*b)/300;
2. c=a*b/(long)300;
3. c=a*b/300L;
```

This happens because type casting does not prevent overflow in the above-mentioned statements. In 1, first a and b are multiplied. At this stage, overflow occurs and the value becomes 24464. Therefore, there is no benefit now in type-casting it to a long integer. A similar reason applies for 2 and 3.

6. *Why does an assignment operator fail on constants, i.e. why cannot constants be placed on the left side of an assignment operator?*

Assignment operator fails on constants because the assignment operator on its left side expects an operand that has a modifiable l-value. Constants do not have a modifiable l-value and thus cannot be placed on the left side of the assignment operator. If a constant is placed on the left side of the assignment operator, the compiler shows 'L-value required' error.

7. *A programmer wants to find the exponent of a number. He has written the following piece of code:*

```
main()
{
    int x=10,y=2,result;
    result=x^y;
    printf("The result of exponent operation is %d",result);
}
```

Does the above-mentioned piece of code produce the intended result?

No, the mentioned piece of code does not produce the correct result. There is no operator in C to find the exponent of a number. The ^ operator is a bitwise XOR operator. Hence, the mentioned piece of code finds 'x bitwise-XOR y' instead of 'x exponent y'.

8. *I have read that 'Every statement in C is terminated with a semicolon'. The line number 1 in the given piece of code is terminated with a comma instead of a semicolon. Will this piece of code work? If yes, what would its output be?*

```
main()
{
    printf("Hello"),                //←line ❶
    printf("Readers!!..") ;         //←line ❷
}
```

As new line characters and comments are ignored during the translation phase by the compiler, the given piece of code:

```
main()
{
    printf("Hello"),                //←line ❶
    printf("Readers!!..");          //←line ❷
}
```

will be interpreted as

```
main()
{
    printf("Hello"), printf("Readers!!..");
}
```

The interpreted code has only one statement that consists of two comma-separated expressions, i.e. printf("Hello") and printf("Readers!!.."). As the operands of the comma operator are evaluated in left-to-right order, printf("Hello") is evaluated first followed by printf("Readers!.."). Hence, the output of the code would be HelloReaders!!..

9. *From the previous question, I have inferred that semicolons separating two* printf *functions can be replaced by commas. Is my inference correct?*

No. Consider the following piece of code:

```
main()
{
    printf("Hello"); ;printf("Readers!!..");
}
```

The given piece of code on execution prints HelloReaders!!... If the semicolons appearing between the printf functions are replaced by commas, the given code becomes

```
main()
{
    printf("Hello"), ,printf("Readers!!..");
}
```

The resultant code on compilation gives 'Expression syntax error'. This error is due to the fact that two comma operators cannot appear consecutively. There must be an operand in between them. Hence, the drawn inference is not correct.

10. *What will the output of the following code segment be?*

```
main()
{
    int a=10,b=20;
    printf("%d %d\n",a,b);
    a=a*b;
    b=a/b;
    a=a/b;
    printf("%d %d\n",a,b);
    a=a^b;
    b=a^b;
    a=a^b;
    printf("%d %d\n",a,b);
    a=a+b;
    b=a-b;
    a=a-b;
    printf("%d %d\n",a,b);
}
```

The code provides three different ways to swap the contents of two variables without using a temporary variable.

Initially, a=10, b=20. On execution of statements:
a=a*b; //←a will become 200
b=a/b; //←b will become 10
a=a/b; //←a will become 20
Values are swapped.
a=a^b; //←a will become 30
b=a^b; //←b will become 20
a=a^b; //←a will become 10
Values are swapped again.
a=a+b; //←a will become 30
b=a-b; //←b will become 10
a=a-b; //←a will become 20
Values are swapped again.
Hence, the output of the code would be:

```
10 20
20 10
10 20
20 10
```

Code Snippets

Determine the output of the following code snippets. Assume that the inclusion of the required header files has been made and there is no prototyping error due to them.

```
11. main()
    {
        int a;
        a=2*3+4%5–3/2+6;
        printf("%d",a);
    }
12. main()
    {
        printf("%d %d %d %d",6/5,-6/5,6/-5,-6/-5);
    }
13. main()
    {
        printf("%d %d %d %d",6%5,-6%5,6%-5,–6%–5);
    }
14. main()
    {
         int a=12,b;
         printf("%d %d",b,b=a);
    }
15. main()
     {
        int a=23,b=12,c=10,d;
        d=c=b=a;
        printf("%d %d %d %d",a,b,c,d);
    }
16. main()
    {
        int a=23,b=12,c=10,d;
        d=c+2=b+1=a;
        printf("%d %d %d %d",a,b,c,d);
    }
17. main()
    {
        int a=2,b=3,c=1,d;
        d=a<b>c;
        printf("%d",d);
    }
18. main()
    {
        int a=3,b=2,c=1,d;
        d=a<b<c-1;
        printf("%d",d);
    }
```

```
19. main()
    {
        int a=10,b=20,c=30;
        c==a=b;
        printf(“%d %d %d”,a,b,c);
    }
20. main()
    {
         int a=10,b=20,c=30;
         c=a==b;
         printf(“%d %d %d”,a,b,c);
    }
21. main()
    {
        int a=10,b=20,c=30;
        c==a==b;
        printf(“%d %d %d”,a,b,c);
    }

22. main ()
    {
        int a=012,b=034;
        int x=0x12,y=0x34;
        int c,d,u,v;
        c=a&b;
        d=a|b;
        u=x&y;
        v=x|y;
        printf(“%d %d %d %d”,c,d,u,v);
    }
23. main ()
    {
         int a=012,b=034;
        int x=0x12,y=0x34;
        int c,d,u,v;
        c=a&&b;
        d=a||b;
        u=x&&y;
        v=x||y;
        printf(“%d %d %d %d”,c,d,u,v);
    }
24. main()
    {
        int c=10,d,e;
        d=!c;
        e=~c;
        printf(“%d %d”,d,e);
    }
```

25.
```
main()
{
     int c=-4,d=4;
     printf(“%d %d %d %d”,~c,~d,c^d,~c^~d);
}
```

26.
```
main()
{
     int i=10;
     printf(“%d”,i++*i++);
}
```

27.
```
main()
{
     int i=10,j;
     j=++i++;
     printf(“%d %d”,i,j);
}
```

28.
```
main()
{
     int i=10,j=11,k,l;
     k=i+++j;
     l=i+++++j;
     printf(“%d %d”,l,k);
}
```

29.
```
main()
{
     int i=10,j=11,k,l;
     k=i+++j;
     l=i+++ ++j;
     printf(“%d %d”,l,k);
}
```

30.
```
main()
{
     int i=10,j=11,k,l;
     k=i+++j;
     l=i++ +++j;
     printf(“%d %d”,l,k);
}
```

31.
```
main()
{
     int x=20,y=35;
     x=y++ + x++;
     y=++y + ++x;
     printf(“%d %d”,x,y);
}
```

32.
```
main()
{
     int i=100,j=20;
```

```
        i++=j;
        printf(“%d %d”,i,j);
    }
```

```
33. main()
    {
        int a=10,b;
        a>=5?b=100:b=200;
        printf(“%d”,b);
    }
```

```
34. main()
    {
        int i=0,j=1,k=2,l;
        l=i||j++&&++k;
        printf(“%d %d %d %d”,i,j,k,l);
    }
```

```
35. main()
    {
        int i=0,j=1,k=2,l;
        l=i&&j++&&++k;
        printf(“%d %d %d %d”,i,j,k,l);
    }
```

```
36. main()
    {
        int i=0,j=1,k=2,l;
        l=++i&&j++&&++k;
        printf(“%d %d %d %d”,i,j,k,l);
    }
```

```
37. main()
    {
        int i=0,j=1,k=2,l;
        l=++i||j++&&++k;
        printf(“%d %d %d %d”,i,j,k,l);
    }
```

```
38. main()
    {
        int i=0,j=1,k=2,l;
        l=++i&&j++||++k;
        printf(“%d %d %d %d”,i,j,k,l);
    }
```

```
39. main()
    {
        int i=0,j=1,k=2,l;
        l=++i&&--j||++k;
        printf(“%d %d %d %d”,i,j,k,l);
    }
```

```
40. main()
    {
          int i=0,j=1,k=2,l;
          l=++i&&j--||++k;
          printf(“%d %d %d %d”,i,j,k,l);
     }
41.  main()
     {
          int x=4;
          printf(“%d %d %d”,x,x<<2,x>>2);
     }
42.  main()
     {
         int  x=32767;
         printf(“%d”,x<<1);
     }
43.  main()
     {
         int num=3;
         printf(“%d”,num<<2<<2);
     }
44.  main()
     {
         int num=3;
         printf(“%d”,num<<(2<<2));
     }
45.  main()
     {
         int num=5,i=1;
         printf(“%d”,(num<<i&1<<15)?1:0);
     }
46.  main()
     {
         int num=5,i=1;
         printf(“%d”,(num<<i&&1<<15)?1:0);
     }
47.  main()
     {
         float a=0.9;
         int c;
         c=a<0.9;
         printf(“%d”,c);
     }
48.  main()
     {
         float a=0.5;
         int c;
         c=a<0.5;
```

```
        printf(“%d”,c);
    }
49. main()
    {
        float a=0.9;
        int c;
        c=a<0.9f;
        printf(“%d”,c);
    }
50. main()
    {
        int a=0,b=0;
        ++a==0||++b==11;
        printf(“%d %d”,a,b);
    }
51. main()
    {
        int x=4+2%-8;
        printf(“%d”,x);
    }
52. main()
    {
        int i=5;
        i=!i>3;
        printf(“%d”,i);
    }
53. main()
    {
        int a=10,b=70,c;
        c=b=a*=2;
        printf(“%d %d %d”,a,b,c);
    }
54. main()
    {
          printf(“%x”,-1<<4);
    }
55. main()
    {
        int c=- -2;
        printf(“%d”,c);
    }
56. main()
    {
        int c=--2;
        printf(“%d”,c);
    }
```

57.
```
main()
{
    int i=5;
    printf(“%d %d %d %d %d”,i++,i--,++i,--i,i);
}
```

58.
```
main()
{
    200;
    printf(“%d”,200);
}
```

59.
```
main()
{
    int i=-1;
    +i;
    printf(“%d %d”,i,+i);
}
```

60.
```
main()
{
    char not;
    not=!2;
    printf(“%d”,not);
}
```

61.
```
main()
{
    int k=1;
    printf(“%d==1 is “”%s”,k,k==1?”True”:”False”);
}
```

62.
```
main()
{
    const int i=4;
    float j;
    j=++i;
    printf(“%d %d”,i,++j);
}
```

63.
```
main()
{
    int i=5;
    printf(“%d”,i=++i==6);
}
```

64.
```
main()
{
    int i=5,j=10;
    j=i&=j&&10;
    printf(“%d %d”,i,j);
}
```

65.
```
main()
{
    float x,y;
    x=7; y=10;
    x*=y*=y+28.5;
    printf(“%f %f”,x,y);
}
```

66.
```
main()
{
    unsigned int a=0xffff;
    ~a;
    printf(“%x”,a);
}
```

67.
```
main()
{
    unsigned char i=0x80;
    printf(“%d”,i<<1);
}
```

68.
```
main()
{
    unsigned a=–1;
    int b;
    printf(“%u ”,a);
    printf(“%u ”,++a);
}
```

69.
```
main()
{
    float u=3.5;
    int v,w,x,y;
    v=(int)(u+0.5);
    w=(int)u+0.5;
    x=(int)((int)u+0.5);
    y=(u+(int)0.5);
    printf(“%d %d %d %d”,v,w,x,y);
}
```

70.
```
main()
{
    int u=3.5,v,w,x,y;
    v=(int)(u+0.5);
    w=(int)u+0.5;
    x=(int)((int)u+0.5);
    y=(u+(int)0.5);
    printf(“%d %d %d %d”,v,w,x,y);
}
```

Multiple-choice Questions

71. The location of a global variable is bound at

a. Load time

b. Procedure entry time

c. Run time

d. None of these

72. Which of the following is not an arithmetic operator?

a. *
b. +
c. &
d %

73. Which of the following is not a bitwise operator?

a. &&
b. |
c. ^
d. >>

74. Which of the following operators in arithmetic class has the lowest precedence?

a. %
b. /
c. *
d. +

75. What is the correct way to round off a `float` variable `z` into an integer?

a. `x=(int)(z+0.5)`
b. `x=(z+(int)z+0.5`
c. `x=(int)z+0.5`
d. `x=(int)((int)z+0.5)`

76. Comma operator is a/an

a. Unary operator
b. Binary operator
c. Ternary operator
d. None of these

77. The location of local variables and reference parameter is typically bound at

a. Load time
b. Procedure entry time
c. Run time
d. None of these

78. Evaluation of the expression involving || operator
I. Takes place from left to right
II. Takes place from right to left
III. Stops when one of the operand evaluates to true
IV. Stops when one of the operand evaluates to false

a. I and III
b. III only
c. II only
d. IV

79. Evaluation of the expression involving && operator
I. Takes place from left to right
II. Takes place from right to left
III. Stops when one of the operand evaluates to true
IV. Stops when one of the operand evaluates to false

a. I and III
b. I and IV
c. II only
d. IV

80. Expressions in C can be made from
I. Operands alone
II. Operators alone
III. Operators and operands
IV. None of these

a. I and III
b. III only
c. II only
d. IV

81. What is the fundamental unit of execution in C?
a. Expression
b. Sub-expression
c. Statement
d. Function

82. What is the minimum number of temporary variables required to swap the content of two variables?

a. 1 c. 0
b. 2 d. None of these

83. `int a;` is actually a

a. Declaration c. Neither a definition nor a declaration
b. Definition d. None of these

84. The output of the following C code will be as follows:

```
main()
{
    int a=10,b=20;
    printf("%d %d",a,b);
    a ^=b ^=a ^=b;
    printf("%d %d",a,b);
}
```

a. 10 20 10 20 c. 10 10 10 10
b. 10 20 20 10 d. None of these

85.
```
main()
{
    if(~0 == -1)
    printf("Perfect");
}
```

a. Perfect c. Compilation error
b. No output d. None of these

Outputs and Explanations to Code Snippets

11. 15

Explanation:

The expression `a=2*3+4%5-3/2+6` gets evaluated as

```
a=6+4%5-3/2+6
a=6+4-3/2+6
a=6+4-1+6
a=10-1+6
a=9+6
a=15
```

12. 1 -1 -1 1

Explanation:

The sign of the result of evaluation of division operator depends upon the sign of both the numerator as well as the denominator. If both are positive, the result will be positive. If either is negative, the result will be negative and if both are negative, the result will be positive.

13. 1 -1 1 -1

Explanation:

The sign of the result of evaluation of modulus operator depends upon the sign of numerator only. If the numerator is positive, the result will be positive. If the numerator is negative, the result will be negative.

14. 12 12

Explanation:

The comma operator guarantees left-to-right evaluation, but the commas separating the arguments in a function call are not comma operators. They are considered as separators.✍ If commas separating arguments in a function call are considered as comma operators, then no function could have more than one argument. Hence, these arguments are not guaranteed to be evaluated from left to right. The order of evaluation of arguments in a function call is compiler dependent. In Borland TC 3.0 & Borland TC 4.5, evaluation takes place from right to left. Thus, if the code in the given question is executed using the specified compilers, b=a gets evaluated first and b gets the value 12. The result of the evaluation of the expression b=a turns out to be 12, i.e. the value that is assigned. Therefore, 12 12 gets printed.

✍ **Separators** are used to separate two tokens. Unlike other programming languages:

- Semicolon in C language is a terminator and not a separator. **Terminator** terminates a statement. Statements in C are terminated with semicolon.
- In C language, the white-space character acts as separator.

15. 23 23 23 23

Explanation:

d=c=b=a is a valid expression with no compilation error. The assignment operator✍ is right-to-left associative. Thus, d=c=b=a is interpreted as (d=(c=(b=a))). Thus, first the value of a will be placed in b, then the value of b will be placed in c and then the value of c will be placed in d. Hence, all b, c and d will have a value of a, i.e. 23.

✍ The result of evaluation of an expression is an r-value. Assignment expression is no exception to this rule. The result of evaluation of an assignment expression is the value that is assigned. For example, a=10; assigns 10 to a and the overall expression evaluates to 10 (i.e. the assigned value). As described in the explanation above, the value of b is not assigned to c. Actually, the result of evaluation of expression b=a is assigned to c. However, since the result of evaluation of expression b=a is the same as the value of variable b after assignment, the above explanation is also correct.

16. Compilation error (l-value required error)

Explanation:

c+2 and b+1 are expressions. The result of the evaluation of an expression is an r-value, and the assignment operator cannot have an r-value on its left side. Hence, the placement of c+2 and b+1 on the left side of the assignment operator is erroneous and leads to 'L-value required' error.

17. 0

Explanation:

In expression d=a<b>c, three operators namely, assignment operator (=), less than operator (<) and greater than operator (>) are involved. The precedence of the assignment operator is least and less than operator and greater than operator have the same precedence. < and > operators are left-to-right associative. Thus, less than operator (<) will be evaluated first. a<b, i.e. 2<3 turns out to be true, i.e. 1. Now 1>c, i.e. 1>1 is checked and it turns out to be false, i.e. 0. This is assigned to d. Hence, d got the value 0.

18. 0

Explanation:

Out of =, < and - operators, - operator has the highest precedence. Thus, c-1 will be evaluated first and turns out to be 0. a<b is evaluated then and turns out to be 0 (as 3<2 is false). Then 0<0 is evaluated and turns out to be 0. This outcome is assigned to d. Therefore, d will have the value 0.

19. Compilation error (l-value required)

Explanation:

Out of == and = operator, the equality (==) operator has a higher precedence than the assignment operator. Remember that the assignment operator has a lower precedence than every other operator except the comma operator. The equality operator will be evaluated first. c==a, i.e. 10==30 evaluates to false, i.e. 0. Now, the expression becomes 0=b (i.e. trying to assign a value of b to a constant). It is not allowed, as the assignment operator cannot have a constant (i.e. r-value) on its left side and if it happens (as in this case), there will be 'L-value required' error.

20. 10 20 0

Explanation:

a==b is evaluated first and turns out to be 0. 0 is assigned to c. The values of a and b are not manipulated and remain the same. Hence, the result is 10 20 0.

21. 10 20 30

Explanation:

The equality operator is left-to-right associative. Hence, the expression c==a==b is interpreted as (c==a)==b. The sub-expression c==a is evaluated first and turn out to be 0. Then, 0==b, i.e. 0==20 is evaluated and results in 0. This outcome is not assigned to any variable and will be ignored. Values of a, b and c are not modified anywhere in the function. Hence, the output is 10 20 30.

22. 8 30 16 54

Explanation:

a will be stored in memory as follows:

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0

b will be stored in memory as follows:

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0

Result of & (Bitwise AND) and |(Bitwise OR) operators is shown in the figure below:

Operator and result	Sign	Magnitude														
	Bit 16	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**

b	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**0**	**0**
c=a&b=8	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
d=a|b=30	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0

x will be stored in memory as follows:

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0

y will be stored in memory as follows:

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0

Result of & (Bitwise AND) and |(Bitwise OR) operators is shown in the figure below:

Operator and result	Sign	Magnitude (Magnitude is in two's complement representation)													
	Bit 16	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
x	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**
y	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**0**	**0**
u=x&y=16	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
v=x|y=54	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0

23. 1 1 1 1

Explanation:

In C language, a non-zero value is treated as true and zero value is treated as false. Therefore, 012,034, 0x12 and 0x34 are treated as true as all are non-zero values. True && True evaluates to true, i.e. 1. True || True evaluates to true, i.e. 1. Hence, 1 is assigned to c, d, u and v.

24. 0 -11

Explanation:

! is logical NOT operator and ~ is bitwise NOT operator. Logical NOT operator, i.e. ! operates on its operand considering it as a single entity while bitwise NOT operator, i.e. ~ operates on the individual bits of its operand. In d=!c, c is 10, i.e. true. The result of logical negation of true turns out to be false, i.e. 0. Hence, d will have a value of 0.

The value of c (i.e. 10) will be stored in memory as follows:

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0

Bitwise operator (~) negates every bit of c. Hence, the result of bitwise negation will be as follows:

Operator and result	Sign Bit 16 MSB	Magnitude of: c is in normal binary representation e is in two's complement representation														
		Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
c	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
e=~c	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1

Sign bit of e is 1. Therefore, the number e will be negative. Its value can be determined by taking two's complement of the two's complemented representation of its magnitude as shown in the figure below:

	Magnitude (MSB is 1, so magnitude is in two's complement representation)														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
e in two's complement form	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1
Its two's complement	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

Since sign bit was 1, the value of e will be -11.

25. 3 -5 -8 -8

Explanation:

Operator and result	Sign Bit 16 MSB	Magnitude														
		Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
c=-4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
d=4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
~c=3	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**
~d=-5	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**
c^d=-8 (XOR)	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**
~c^~d=-8	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**

26. 110

Explanation:

Actually the result of this program snippet is compiler dependent. In the case of a post-increment operator, the value of the operand is used first for the evaluation of expression and **after** its use the value of the operand is incremented. The precise meaning of words '**after**' and '**expression**' is left undefined. Two possible interpretations are as follows:

1. According to the first interpretation, the value of the operand (i.e. i) is incremented after the evaluation of the sub-expression✍ (i.e. i++).
2. According to the second interpretation, the value of the operand (i.e. i) is incremented after the evaluation of full expression✍ (i.e. i++*i++).

If the value is incremented after the evaluation of the sub-expression (i.e. interpretation 1), the expression i++*i++ will be evaluated to 10*11=110. If the value is incremented after the evaluation of full expression (i.e. interpretation 2), the expression i++*i++ will be evaluated to 10*10=100. Different compilers use different interpretations; hence, the result is compiler dependent. In Borland TC 3.0 and Borland TC 4.5 compilers, increment takes place after the evaluation of the sub-expression. Hence, the result is 110.

An expression that is part of another expression is called **sub-expression**. An expression that is not part of another expression is called **full expression**.

27. Compilation error (l-value required)

    Explanation:

    As the increment operator is right-to-left associative, the expression j=++i++ will be interpreted as j=++(i++). The result of evaluation of sub-expression i++ will be an r-value. This r-value will act as an operand for other increment operator, i.e. pre-increment operator. Thus, the expression reduces to ++(r-value). This reduced expression is erroneous, as increment and decrement operators can only work on operands that have a modifiable l-value. Hence, there will be 'L-value required' error.

28. Compilation error (l-value required)

    Explanation:

    The tokenizer✍ of C language is greedy in nature. It always tries to create the biggest possible token. Thus, the expression k=i+++j will be treated as k=i++ +j and it is a well-formed expression. The operator ++ will operate first and then operator + will operate. The outcome is assigned to k. In expression l=i+++++j, the tokenizer will divide the operator sequence +++++ into ++ ++ +. Thus, the expression l=i+++++j will be treated as l=i++ ++ +j. The sub-expression i++ will evaluate to an r-value. The second ++ operator cannot operate on an r-value and hence, will lead to 'L-value required' compilation error.

The first phase of a compiler that divides the sequence of input characters into tokens is known as a **tokenizer** or a **lexical analyzer**. C language has **'Greedy Tokenizer'**. It always tries to create the biggest possible token. For example, the sequence of input characters i++-+j will be divided into token sequences i, ++, -, + and j. Consider another example, the sequence of input characters i+++ ++j will be divided into token sequences i, ++, +, ++ and j. Note that the white-space character between two characters is not ignored while tokenizing.

29. 23 21

    Explanation:

    The expression k=i+++j will be treated as k=i++ +j. First, the value of i is used for the evaluation of sub-expression i++ and then it is incremented by 1. The value of j (i.e. 11) is added to the result of evaluation of i++ (i.e. 10) and the outcome is assigned to k. Therefore, k will be 10+11=21. i will become 11 and j remains 11.

    The expression l=i+++ ++j will be treated as l=i++ + ++j. First, the value of i is used for the evaluation of sub-expression i++ and then it is incremented by 1. The value of j will be incremented first and then its value is used for the evaluation of full expression. The result of evaluation of two sub-expressions (i.e. i++ and ++j) is added and is assigned to l. Thus, the value of l becomes 11+12=23. Both i and j become 12 after the evaluation of full expression l=i+++ ++j.

30. Compilation error (l-value required error)

    Explanation:

    The expression l=i++ +++j will be treated as i++ ++ +j and will give an error due to the reason mentioned in Answer 28.

31. 57 94

Explanation:

In the expression x=y++ + x++, the values of x (i.e. 20) and y (i.e. 35) are used for the evaluation of sub-expressions: y++ and x++. The outcomes of evaluation of these sub-expressions are added, and the result is assigned to variable x (i.e. 20+35=55 and 55 is assigned to x). Then the values of y and x are incremented (i.e. y becomes 36 and x becomes 56). In the next expression, the values of y and x get incremented first (i.e. x becomes 57 and y becomes 37) and then they are used for the evaluation of full expression y=++y + ++x (i.e. y=37+57=94). Hence, x and y become 57 and 94, respectively.

32. Compilation error (l-value required)

Explanation:

In the expression i++=j, the increment operator and the assignment operator are involved. The increment operator ++ has a higher precedence than the assignment operator and will get evaluated first. The result of evaluation of increment operator is an r-value. This r-value lies on the left side of the assignment operator and thus, leads to 'L-value required' error.

33. 100

Explanation:

The expression a>=5 evaluates to true. Hence, the expression✍ b=100 gets evaluated. Value 100 is assigned to variable b and is printed by the next printf statement.

✍ In conditional expression E1?E2:E3, the sub-expression E1 is evaluated first. If it evaluates to a non-zero value (i.e. true), then E2 is evaluated and E3 is ignored. If E1 evaluates to zero (i.e. false), then E3 is evaluated and E2 is ignored.

34. 0 2 3 1

Explanation:

Rules to be followed:

1. The precedence of the logical AND operator (&&) is higher than the precedence of the logical OR (||) operator. The precedence of the logical AND operator and the logical OR operator is only used to parenthesize the expression involving them.
2. The logical AND operator (&&) and the logical OR operator (||) always guarantee left-to-right evaluation irrespective of their precedence.
3. If the first operand of the logical OR operator (||) evaluates to true, the second operand will not be evaluated, as TRUE || anything (true or false) is TRUE.
4. If the first operand of the logical AND operator (&&) evaluates to false, the second operand will not be evaluated, as FALSE && anything (true or false) is FALSE.

Expression l=i||j++&&++k will be treated as l=i||(j++&&++k), as the logical AND operator has a higher precedence than the logical OR operator. The logical AND and logical OR operator guarantee left-to-right execution. Hence, the expression l=i||(j++&&++k) is executed from left to right. The first operand of the logical OR operator (||), i.e. i is 0, i.e. false; hence, the second operand needs to be evaluated to determine the truth value of full expression. The sub-expression j++&&++k starts evaluation. In sub-expression j++, j is post-incremented. The sub-expression j++ evaluates to 1 and the value of j is incremented to 2. Since the first operand of the logical AND operator, i.e. j++ evaluates to 1 (i.e. true), the second operand (i.e. ++k) needs to be evaluated. In sub-expression ++k, k is pre-incremented. The value of k is incremented first and then its value is used for the evaluation of expression. Thus, the value of k used for the evaluation of expression is 3. Therefore, 1&&3 turns

out to be 1. Thus, the second operand of the logical OR operator evaluates to 1. Hence, 0||1 will be evaluated and turns out to be 1. The outcome is assigned to l.
Therefore, the values are i=0, j=2, k=3, l=1.

35. 0 1 2 0

Explanation:

Since the logical AND operator is left-to-right associative, the expression l=i&&j++&&++k will be interpreted as l=(i&&j++)&&++k. Recall Rule 4 mentioned in the previous answer. In the sub-expression i&&j++, as the first operand of && operator, i.e. i is 0 (i.e. false), j++ will not be evaluated and the sub-expression i&&j++ evaluates to 0. Due to the same reason, the sub-expression ++k will not be evaluated and the full expression evaluates to 0. 0 is assigned to l. Hence, i=0, j=1, k=2 and l=0.

36. 1 2 3 1

Explanation:

The expression l=++i&&j++&&++k will be interpreted as l=(++i&&j++)&&++k. In the sub-expression ++i&&j++, i is pre-incremented. i becomes 1 and the sub-expression ++i, evaluates to 1. Since the first operand of the && operator evaluates to 1, i.e. true, the sub-expression j++ needs to be evaluated. The sub-expression j++ evaluates to 1 and the value of j becomes 2. As 1&&1 evaluates to 1, the sub-expression ++k will be evaluated. k will become 3. 1&&3 evaluates to 1. Hence, l will get value 1. Therefore, the values are i=1, j=2, k=3, l=1.

37. 1 1 2 1

Explanation:

Since the precedence of the logical AND operator is higher than the logical OR operator, the expression l=++i||j++&&++k will be interpreted as l=++i||(j++&&++k). In the sub-expression ++i, i is pre-incremented. i becomes 1 and the sub-expression ++i evaluates to 1. 1|| anything (0 or 1) is 1. Hence, the sub-expression (j++&&++k) will not be evaluated. The values of j and k remain 1 and 2, respectively. Therefore, the values are i=1, j=1, k=2, l=1.

38. 1 2 2 1

Explanation:

The expression l=++i&&j++||++k will be interpreted as l=(++i&&j++)||++k. In the sub-expression ++i&&j++, i is pre-incremented. i becomes 1 and the sub-expression ++i evaluates to 1. Since the first operand of the logical AND operator evaluates to true, the second operand needs to be evaluated. The sub-expression j++ evaluates to 1 and j is incremented to 2. 1&&1 evaluates to 1. 1 || anything (0 or 1) is 1. Therefore, the sub-expression ++k will not be evaluated and the value of k remains 2. Thus, the expression ++i&&j++||++k evaluates to 1 and is assigned to l. Hence, i=1, j=2, k=2 and l=1.

39. 1 0 3 1

Explanation:

The expression l=++i&&--j||++k will be interpreted as l=(++i&&--j)||++k. In the sub-expression ++i&&--j, i is pre-incremented. i becomes 1 and the sub-expression ++i evaluates to 1. Since the first operand of the logical AND operator evaluates to true, the second operand (i.e. --j) needs to be evaluated. j is decremented to 0 and the sub-expression --j evaluates to 0. Thus, 1&&0 evaluates to 0. As the first operand of the logical OR operator is 0, the sub-expression ++k needs to be evaluated. k becomes 3. 0||3 evaluates to 1 and is assigned to l. Hence, the values are i=1, j=0, k=3 and l=1.

40. 1 0 2 1

Explanation:

The expression l=++i&&j--||++k will be interpreted as l=(++i&&j--)||++k. In the sub-expression ++i&&j--, i is pre-incremented. i becomes 1 and the sub-expression ++i evaluates to 1. Since the first operand of the logical AND operator evaluates to true, the second operand (i.e. j--) needs to be evaluated. The sub-expression j-- evaluates to 1 and j is decremented to 0. 1&&1 evaluates to 1. The first operand of the logical OR operator evaluates to 1, so the second operand need not be evaluated. Hence, ++k will not be evaluated and the value of k remains 2. The expression ++i&&j--||++k evaluates to 1 and is assigned to l. Hence, i=1, j=0, k=2, l=1.

41. 4 16 1

Explanation:

<< is the left shift operator. A shift by 1 bit in the left direction is equivalent to multiplication✍ by 2. A shift by n bits is equivalent to multiplication by 2^n, provided the magnitude does not overflow.
>> is the right shift operator. A shift by 1 bit in the right direction is equivalent to integer division by 2. A shift by n bits is equivalent to integer division by 2^n.
4<<2 is equivalent to $4*2^2 = 4*4 = 16$
4>>2 is equivalent to $4/2^2 = 4/4 = 1$.

✍ The statement ‘A shift by 1 bit in the left direction is equivalent to multiplication by 2’ holds true till there is no overflow in the magnitude field of the number. For example, if an integer is stored in 2 bytes, 32767<<2 will not be 65534 because the magnitude field has overflowed.

42. -2

Explanation:

x=32767 will be stored in memory as follows:

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Shift by 1 bit in left direction will lead to

Sign Bit 16 MSB	Magnitude is in two’s complement representation														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

Sign bit becomes 1. Hence, the number will be negative and its value can be determined by taking two’s complement✍ of two’s complemented representation of its magnitude. Two’s complement of 111 1111 1111 1110 is 000 0000 0000 0010, i.e. 2. Sign bit was 1, i.e. negative. Hence, the result will be -2.

A good method to **find two's complement** of a number is:

1. Look from the right side in the bits sequence.
2. Till 1 is encountered keep the bits sequence same.
3. After 1 has been encountered, negate every bit, i.e. 0 to 1 and 1 to 0.

For example, consider number 111 1111 1111 1110 ←
two's complement will be 000 0000 0000 0010

43. 48

Explanation:

Since, the shift operator is left-to-right associative, the expression num<<2<<2 will be interpreted as (num<<2)<<2. The sub-expression num<<2, i.e. 3<<2 evaluates to an r-value 12. This r-value acts as an operand for the second shift operator and the sub-expression 12<<2 evaluates to 48.

44. 768

Explanation:

In expression num<<(2<<2), the sub-expression 2<<2 will be evaluated first.✍ The result of its evaluation will be an r-value, i.e. 8. Then, num<<8 (i.e. 3<<8) will be evaluated and results in 768.

Parenthesized sub-expressions are evaluated first.

45. 0

Explanation:

Since the shift operator has a higher precedence than the bitwise AND (&) operator, the expression (num<<i&1<<15)?1:0) will be interpreted as ((num<<i)&(1<<15))?1:0). First, the operand1 of the conditional operator (i.e. sub-expression num<<i&1<<15) will be evaluated as follows:

Operator and result	Sign bit 16 MSB	Magnitude														
		Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
num=5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
num<<i=num<<1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1<<15	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
num<<i&1<<15=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Since the sub-expression num<<i&&1<<15 evaluates to 0, i.e. false, the outcome of the conditional operator will be the result of evaluation of operand3, i.e. 0.

46. 1

Explanation:

The expression (num<<i&&1<<15)?1:0) will be interpreted as ((num<<i)&&(1<<15))?1:0). The sub-expression num<<i&&1<<15 will be evaluated first. The sub-expression num<<i (i.e. 5<<1) evaluates to 10 and the sub-expression 1<<15 evaluates to -32768. Both are non-zero values and non-zero values are considered as true. Also, true&&true is true. Hence, num<<i&&1<<15 evaluates to true, i.e. 1. Since, operand1 of the conditional operator evaluates to true, operand2 (i.e. 1) will be evaluated and results in 1.

47. I

Explanation:

This question can only be answered after looking at some of the technicalities and intricacies involved in storing floating point numbers. The following facts must be remembered:

1. **Each real floating-type number cannot be represented exactly in memory** (i.e. with infinite precision**).**

   During their storage, some round-off errors occur. Some real floating-type numbers are stored as a greater value and some are stored as a lesser value.
   Execute the given code and have a look at the output:

```
main()
{
  float a=0.4, b=0.9;
  printf("0.4 is stored as %.20f\n",a);
  printf("0.9 is stored as %.20f",b);
}
```

   The output of this code turns to be
   0.4 is stored as 0.40000000596046447800 (i.e greater value)
   0.9 is stored as 0.89999997615814209000 (i.e smaller value)

2. **Floats are stored in 32 bits (1 bit for Sign, 8 bits for Exponents and 23 bits for Fraction)**.
   0.9 as a float will be stored in memory as follows:

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
3				**F**				**6**				**6**				**6**				**6**				**6**				**6**			
S	**E**	**E**	**E**	**E**	**E**	**E**	**E**	**E**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**	**F**
	8-bits for exponent								**23-bits for mantissa**																						

3. **Doubles are stored in 64 bits (1 bit for Sign, 11 bits for Exponents and 52 bits for Fraction).**
   To store 0.9 (i.e. 0.111001100110011001100110011001100...) as double:

   I. Normalize it. Value becomes 1.1100110011001100...* 2^{-1}
   II. Bias the double exponent with value 1023 like float exponent is biased with 127. Therefore, exponent after biasing becomes -1+1023=1022 i.e. 01111111110 (in binary)
   III. Fractional part is 110011001100110011001100110011...

0.9 as double will be stored in memory as follows:

64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
0	0	1	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
3				**F**				**E**				**C**				**C**				**C**				**C**				**C**			
S	E	E	E	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	11-bits for exponent											**52-bits for mantissa (Continued in the next table)**																			

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
C				**C**				**C**				**C**				**C**				**C**				**C**				**D**			
F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
(Continued from the previous table)																															

4. **Last nibble gets rounded off**
   Why is the last nibble (i.e. 4 bits) in double represented by D instead of C?
   This is because of rounding. **C** gets rounded to **D**. This can be confirmed by running the following piece of code:

```
main()
{
  float a=0.9;
  double b=0.9;
  char *p;
  int i;
  p=(char*)&a;
  printf("Float is stored in memory as:\t");
  for(i=0;i<=3;i++)
  printf("%02X ",(unsigned char)p[i]);
  p=(char*)&b;
  printf("\n Double is stored in memory as:\t");
  for(i=0;i<=7;i++)
  printf("%02X ",(unsigned char)p[i]);
}
```

The above code gives as output

```
Float is stored in memory as:    66 66 66 3F
Double is stored in memory as:   CD CC CC CC CC CC EC 3F
```

Why is the output like CD CC CC CC CC CC EC 3F instead of 3F EC CC CC CC CC CC CD?
The output is like this because the Intel family of micro-processors stores numbers in **little-endian format**.✍ Therefore, the least significant byte, i.e. CD gets stored in the lowest memory location and hence gets printed first. The most significant byte, i.e. 3F is stored in the highest memory location and will get printed last.

✍ In **little-endian format** of storing numbers, the least significant byte is always stored in the lowest numbered memory location, and the most significant byte is stored in the highest.

5. **When float and double are compared, float gets converted into double first. This type of conversion is called promotion. We say that float gets promoted to double.**
   The float value 3F 66 66 66 is promoted to double and becomes:

64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
0	0	1	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
3				**F**				**E**				**C**				**C**				**C**				**C**				**C**			
S	E	E	E	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	11-bits for exponent											**52-bits for mantissa (Continued in the next table)**																			

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C				**0**				**0**				**0**				**0**				**0**				**0**				**0**			
F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
(Continued from the previous table)																															

Only 23 fraction bits are available in float. Therefore, when float is promoted to double, the rest of the fraction bits (shown in gray) will be taken as zero. Hence, when 0.9 as float is promoted to double, it becomes 3F EC CC CC C0 00 00 00.

This can be confirmed by running the below-mentioned piece of code:

```
main()
{
  float a=0.9;
  double c;
  int i;
  char *p;
  p=(char*)&a;
  printf(“Float value is stored as:\t”);
  for(i=0;i<=3;i++)
      printf(“%02X ”,(unsigned char)p[i]);
  printf(“\n”);
  printf(“Now float is converted to double\n”);
  c=a;
  p=(char*)&c;
  printf(“Promoted value is getting stored as:\t”);
  for(i=0;i<=7;i++)
      printf(“%02X ”,(unsigned char)p[i]);
}
```

6. **Comparison of double value and promoted float value**
   Therefore, when this promoted float value (Step No. 5) is compared with the actual double value (Step No. 3) with a less than operator, it results in 1 (i.e. true) because 3F EC CC CC C0 00 00 00 is lesser than 3F EC CC CC CC CC CC CD.

48. 0

Explanation:

0.5 as float will be stored as:

0	0	1	1	1	1	0	1	0	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
3				**F**				**0**				**0**				**0**				**0**				**0**				**0**			
S	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	8-bits for exponent								**23-bits for mantissa**																						

0.5 as double will be stored as follows:

64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3				**F**				**E**				**0**				**0**				**0**				**0**				**0**			
S	E	E	E	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	11-bits for exponent											**52-bits for mantissa (Continued in the next table)**																			

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C				**0**				**0**				**0**				**0**				**0**				**0**				**0**			
F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
(Continued from the previous table)																															

The fractional part in both the cases is zero. Therefore, when 0.5 as float is promoted to double, it becomes 3F E0 00 00 00 00 00 00. This promoted value is equal to double value (3F E0 00 00 00 00 00 00). Hence, the less than operator on comparison gives zero.

49. 0

Explanation:

0.9, a double value,✍ is demoted to float and is assigned to a float variable a. 0.9f is also stored as a float. In the expression c=a<0.9f, float is compared with float. Loss of precision is the same in both the demotions. Hence, a<0.9f evaluates to 0 and is assigned to c.

✍ **Floating-point literal constant** by default is of type double.

50. 1 1

Explanation:

The logical OR operator || guarantees left-to-right evaluation. Thus, in the expression ++a==0||++b==11, the sub-expression ++a==0 will be evaluated first. a will be incremented by 1 and the sub-expression ++a evaluates to 1. The sub-expression ++a==0 (i.e. 1==0) evaluates to 0 (i.e. false). As the first operand of the logical OR operator is false, the second operand needs to be evaluated to determine the truth value of the full expression. Thus, the sub-expression ++b==11 will be evaluated. b is incremented by 1 and the expression ++b evaluates to 1. The sub-expression ++b==11 (i.e. 1==11) evaluates to 0, i.e. false. Both the operands of the logical OR operator have evaluated to 0. Thus, the full expression evaluates to zero. This outcome is not assigned to any variable and will be ignored. Hence, the values of a and b that get printed are 1 and 1.

51. 6

Explanation:

In expression 4+2%-8, the modulus operator has the highest precedence. The result of the modulus operator depends only upon the sign of the numerator. Thus, the sub-expression 2%-8 evaluates to 2. This outcome is added to 4 and is assigned to x. Therefore, x will have value 6.

52. 0

Explanation:

As the logical NOT operator (i.e. !) has a higher precedence than the greater than operator (>), it gets evaluated first. The sub-expression !i (i.e. !5) evaluates to 0. This outcome is compared with 3 and the sub-expression 0>3 evaluates to 0 (i.e. false). This outcome is assigned to i.

53. 20 20 20

Explanation:

The assignment operator is right-to-left associative. The sub-expression a*=2 (i.e. a=a*2) is evaluated first. It evaluates to 20 and is assigned to a. The value of a is then assigned to b and b will become 20. The value of b is assigned to c and c will also become 20. Hence, all a, b and c are 20.

54. fff0

Explanation:

-1 will be stored in memory as follows:
MSB will be 1 as the sign is negative. Magnitude will be two's complemented representation of 1, i.e. 111 1111 1111 1111. This value is shifted in the left direction by 4 bits and the outcome of the shift operation is as follows:

Operator and result	Sign bit 16 MSB	Magnitude														
		Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
-1<<4	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**
In Hexadecimal	**f**				**f**				**f**				**0**			

55. 2

Explanation:

In the expression - -2, both the occurrences of - are instances of unary minus operator. It is right-to-left associative. The rightmost unary minus will first make 2 as -2. Then the second unary minus makes this -2 as 2. Therefore, the result will be 2.

56. Compilation error (l-value required)

Explanation:

-- is not the same as - -. It is one token (i.e. one operator, namely pre-decrement operator). The pre-decrement operator cannot operate on constants and requires an operand that has a modifiable l-value. In the given question, since -- is applied on constant, it shows 'L-value required' error.

57. 4 5 5 4 5

Explanation:

In Borland TC 3.0 and 4.5, arguments of printf function are evaluated from the right. The value of i is 5, so the sub-expression i evaluates to 5. In the sub-expression --i, the value of i is decremented to 4 and the sub-expression evaluates to 4. The sub-expression ++i increments the value of i to 5 and evaluates to 5. In the sub-expression i--, i is post-decremented. The sub-expression evaluates to 5 and then i is decremented to 4. In the sub-expression i++, i is post-incremented, so first the value of i (i.e. 4) will be used and then it is incremented to 5. After the evaluation of values, the printf function prints the values in a left-to-right order according to the given format specifiers. Therefore, the values that get printed are 4 5 5 4 5.

58. 200

Explanation:

200; is a valid statement but does nothing. In the next statement 200 is printed by the printf function.

59. -1 -1

Explanation:

In expression +i, + is unary plus✍ and will not have any effect on the value of i. It is not the same as ++i. In the next statement, the unmodified value of i gets printed. Hence, -1 -1 is the result.

Unary plus does nothing and is known as the **Dummy operator.**

60. 0

    Explanation:

    2 is considered as true as it is a non-zero value. !TRUE evaluates to false, i.e. 0. The outcome is assigned to the identifier not. This value of the identifier not is printed in the next statement.

61. 1==1 is True

    Explanation:

    In Borland TC 3.0 and 4.5, arguments of the printf function are evaluated from the right. Thus, expression k==1?"True":"False" is evaluated first. The sub-expression k==1 evaluates to true, hence the result of the conditional operator turns out to be "True". Also, adjacent string literals get concatenated. Hence, "%d==1 is""%s" will get concatenated to form "%d==1 is %s". The integer specifier is matched with k, which has value 1, and the string specifier %s is matched with string "True". Hence, the result that gets printed is "1==1 is True".

62. Compilation error (Cannot modify a constant object)

    Explanation:

    The expression ++i is erroneous as i is defined as a qualified constant.

Qualified constants do not have a modifiable l-value. Hence, it cannot be used as the operand of an increment/decrement operator.

63. 1

    Explanation:

    First, the value of i is incremented by 1 and it becomes 6. 6 is compared for equality with 6 and evaluates to true, i.e. 1. This outcome is then assigned to i and gets printed.

64. 1 1

    Explanation:

    The logical AND operator && has a higher precedence than the assign-bitwise AND operator &=. The sub-expression j&&10 is evaluated first (i.e. 10&&10) and turns out to be true, i.e. 1. The sub-expression i&=1 is equivalent to i=i&1 (i.e. i=5&1). On evaluation it gives 1. Therefore, i will take value 1. This value of i is assigned to j. Hence, both i and j will have value 1.

65. 2695.000000 385.000000

    Explanation:

    y+28.5 is computed first and turns out to be 38.5. y*=38.5 is computed then and the value of y becomes 385.0. Then, x*=385.0 is computed and the value of x becomes 2695.0. Therefore, the values that get printed are 2695.000000 and 385.000000.

66. ffff

    Explanation:

    ~a does not change the value of a. The value of a remains the same and gets printed as ffff.

67. 256

    Explanation:

    0x80 is 1000 0000, shifting by 1 bit in the left direction gives 1 0000 0000 and this is equivalent to 256 in decimal.

68. 65535 0

Explanation:

-1 will be stored in memory as follows:

Bit 16	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

i.e. all sixteen 1's. -1 is assigned to a. Therefore, a becomes

	Sign bit 16	Magnitude														
		Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
a=-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
++a 1 (carry gets overflowed)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

a is declared as unsigned. Therefore, the 16th bit is not considered as a sign bit but it is considered as a magnitude bit. Therefore, the value of a that gets printed is 65535. If one is added to a, carry overflows and the result turns out to be 0.

69. 4 3 3 3

Explanation:

In v=(int)(u+0.5), first 3.5+0.5 is evaluated and turns out to be 4.0. This is then type casted✍ to the integer and becomes 4. 4 is then assigned to v.

In w=(int)u+0.5, first u is type casted to the integer, i.e. it becomes 3. Then 0.5 is added to make it 3.5. This value is then assigned to an integer variable. Before assignment, demotion will be carried out. 3.5 will be demoted to 3 and then assigned to w.

In x=(int)((int)u+0.5), x will get a value 3. Instead of implicitly demoting 3.5 to 3 as in the previous case, it is now explicitly type casted to 3.

In y=(u+(int)0.5), first 0.5 is type casted to 0. 0 is added to 3.5 and it comes out to be 3.5. 3.5 after implicit demotion is assigned to an integer variable y. Hence, the value assigned to y will be 3.

✍ Type casting can be done explicitly by using a type cast operator. The syntax of using a type cast operator is (target-type-name) expression.

70. 3 3 3 3

Explanation:

The identifier u is declared as int. Therefore, 3.5 will be demoted to 3 and will then be assigned to u. Hence, u will have the value 3 instead of 3.5. All the remaining computations are carried out in the same way as in the previous answer.

Answers to Multiple-choice Questions

71. a 72. c 73. a 74. d 75. a 76. b 77. b 78. a 79. b 80. a 81. c 82. c 83. b 84. b 85. a

Programming Exercises

Program 1 | Find one's and two's complement of a number

Algorithm:
Step 1: Start
Step 2: Read the number (num)
Step 3: One's complement (oc) = ~num i.e. negate every bit using bitwise NOT operator
Step 4: Two's complement (tc) = oc+1 i.e. two's complement is one's complement plus 1
Step 5: Print values of oc and tc
Step 6: Stop

Line | PE 4-1.c

```
//One's and Two's complement
#include<stdio.h>
main()
{
    int num, oc, tc;
    printf("Enter number\t");
    scanf("%d",&num);
    oc=~num;
    tc=oc+1;
    printf("One's complement is %d\n",oc);
    printf("Two's complement is %d\n",tc);
}
```

Memory content

num=2

B 16	B 15	B 14	B 13	B 12	B 11	B 10	B 9	B 8	B 7	B 6	B 5	B 4	B 3	B 2	B 1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

oc = -3 (Two's complement representation)

B 16	B 15	B 14	B 13	B 12	B 11	B 10	B 9	B 8	B 7	B 6	B 5	B 4	B 3	B 2	B 1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1

tc = -2 (Two's complement representation)

B 16	B 15	B 14	B 13	B 12	B 11	B 10	B 9	B 8	B 7	B 6	B 5	B 4	B 3	B 2	B 1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

Output window

```
Enter number    2
One's complement is -3
Two's complement is -2
```

Remarks:

- ~ is bitwise NOT operator
- The sign bits of oc and tc are 1. Hence, they are negative and are stored in two's-complement representation

Program 2 | Assuming that bit numbering starts from 1. Write a C program to set a particular bit in a given number

Algorithm:
Step 1: Start
Step 2: Read the number (num)
Step 3: Read the bit number (bit) that is to be set (i.e. to be made 1) in the given number
Step 4: Construct a temporary number such that it has 1 at the bit position that is to be set in the given number and zero elsewhere. Temporary number can be constructed by using left-shift operator as temp=1<<(bit-1)
Step 5: To set the bit in the given number, perform bitwise OR of the number with the constructed temporary number and save result in the number i.e. num=num|temp
Step 6: Print number (num)
Step 7: Stop

Line | PE 4-2.c

```
//Set particular bit in a given number
#include<stdio.h>
main()
{
int num, bit, temp;
```

Memory content

num=5

B 16	B 15	B 14	B 13	B 12	B 11	B 10	B 9	B 8	B 7	B 6	B 5	B 4	B 3	B 2	B 1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

Output window

```
Enter number    5
Enter the bit number to be set    2
Value after setting bit is 7
```

(*Contd...*)

<table>
<tr><th>Line</th><th>PE 4-2.c</th><th>Memory content</th><th>Output window</th></tr>
<tr><td>6
7
8
9
10
11
12
13</td><td>printf("Enter number\t");
scanf("%d",&num);
printf("Enter the bit number to be set\t");
scanf("%d",&bit);
temp=1<<(bit-1);
num=num|temp;
printf("Value after setting bit is %d", num);
}</td><td>After setting bit 2, the value of num becomes<table><tr><td>B16</td><td>B15</td><td>B14</td><td>B13</td><td>B12</td><td>B11</td><td>B10</td><td>B9</td><td>B8</td><td>B7</td><td>B6</td><td>B5</td><td>B4</td><td>B3</td><td>B2</td><td>B1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr></table>i.e.7</td><td></td></tr>
</table>

<table>
<tr><th colspan="4">Program 3 | Assuming that bit numbering starts from 1. Write a C program to negate a particular bit in a given number</th></tr>
<tr><td colspan="4"><b>Algorithm:</b>
Step 1: Start
Step 2: Read the number (num)
Step 3: Read the bit number (bit) that is to be negated (i.e. to be made 1 if it is 0 and vice-versa) in the given number
Step 4: Construct a temporary number such that it has 1 at the bit position that is to be negated in the given number and zero elsewhere. Temporary number can be constructed by using left-shiftoperator as temp=1<<(bit-1)
Step 5: To negate the bit in the given number, perform bitwise XOR of the number with the constructed temporary number and save result in the number i.e. num=num^temp
Step 6: Print number (num)
Step 7: Stop</td></tr>
<tr><th>Line</th><th>PE 4-3.c</th><th>Memory content</th><th>Output window</th></tr>
<tr><td>1
2
3
4
5
6
7
8
9
10
11
12
13</td><td>//Negate a particular bit in a given number
#include<stdio.h>
main()
{
int num, bit, temp;
printf("Enter number\t");
scanf("%d",&num);
printf("Enter the bit number to be negated\t");
scanf("%d",&bit);
temp=1<<(bit-1);
num=num^temp;
printf("Value after negating bit is %d", num);
}</td><td><b>num=5</b><table><tr><td>B16</td><td>B15</td><td>B14</td><td>B13</td><td>B12</td><td>B11</td><td>B10</td><td>B9</td><td>B8</td><td>B7</td><td>B6</td><td>B5</td><td>B4</td><td>B3</td><td>B2</td><td>B1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr></table>After negating bit 2, the value of num becomes<table><tr><td>B16</td><td>B15</td><td>B14</td><td>B13</td><td>B12</td><td>B11</td><td>B10</td><td>B9</td><td>B8</td><td>B7</td><td>B6</td><td>B5</td><td>B4</td><td>B3</td><td>B2</td><td>B1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr></table>i.e.7
<b>num=5</b><table><tr><td>B16</td><td>B15</td><td>B14</td><td>B13</td><td>B12</td><td>B11</td><td>B10</td><td>B9</td><td>B8</td><td>B7</td><td>B6</td><td>B5</td><td>B4</td><td>B3</td><td>B2</td><td>B1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr></table>After negating bit 3, the value of num becomes<table><tr><td>B16</td><td>B15</td><td>B14</td><td>B13</td><td>B12</td><td>B11</td><td>B10</td><td>B9</td><td>B8</td><td>B7</td><td>B6</td><td>B5</td><td>B4</td><td>B3</td><td>B2</td><td>B1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>i.e.1</td><td>Enter number 5
Enter the bit number to be negated 2
Value after negating bit is 7
<b>Output window (second execution)</b>
Enter number 5
Enter the bit number to be negated 3
Value after negating bit is 1</td></tr>
</table>

Program 4 | Given two numbers, say val and key. Wherever the bits of number key are 1, set the corresponding bits of number val. Leave all other bits of number val unchanged

Algorithm:

```
Step 1: Start
Step 2: Read the numbers, val and key
Step 3: val=val|key
Step 4: Print number (val)
Step 5: Stop
```

PE 4-4.c

```
//Set the corresponding bits
#include<stdio.h>
main()
{
int val, key;
printf(“Enter two numbers\t”);
scanf(“%d %d”,&val, &key);
val=val|key;
printf(“After setting bits, result is %d”,val);
}
```

Memory content

val=4

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

key = 10

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0

After setting the corresponding bits, val becomes 14

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0

Output window

```
Enter two numbers    4 10
After setting bits, result is 14
```

Output window (second execution)

```
Enter two numbers    4 5
After setting bits, result is 5
```

Program 5 | Given two numbers, say val and key. Wherever the bits of number key are 1, negate the corresponding bits of number val. Leave all other bits of number val unchanged

Algorithm:

```
Step 1: Start
Step 2: Read the numbers, val and key
Step 3: val=val^key
Step 4: Print number (val)
Step 5: Stop
```

PE 4-5.c

```
//Negate the corresponding bits
#include<stdio.h>
main()
{
int val, key;
printf(“Enter two numbers\t”);
scanf(“%d %d”,&val, &key);
val=val^key;
printf(“After negating bits, result is %d”,val);
}
```

Memory content

val = 4

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

key = 5

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

After negating the corresponding bits, val becomes 1

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Output window

```
Enter two numbers    2 5
After negating bits, result is 7
```

Output window (second execution)

```
Enter two numbers    4 5
After negating bits, result is 1
```

Program 6 | Given two numbers, say val and key. Wherever the bits of number key are 1, reset (i.e. make 0) the corresponding bits of number val. Leave all other bits of number val unchanged

Algorithm:

```
Step 1: Start
Step 2: Read the numbers, val and key
Step 3: Construct a temporary which is one's complement of the key i.e. temp=˜key.
Step 4: val=val&temp
Step 5: Print number (val)
Step 6: Stop
```

PE 4-6.c

```c
//Reset the corresponding bits
#include<stdio.h>
main()
{
int val, key, temp;
printf("Enter two numbers\t");
scanf("%d %d",&val, &key);
temp=~key;
val=val&temp;
printf("After resetting bits, result is %d",val);
}
```

Memory content

val = 4

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

key = 5

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

After resetting the corresponding bits, val becomes 0

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Output window

```
Enter two numbers    4 5
After resetting bits, result is 0
```

Output window (second execution)

```
Enter two numbers    2 5
After resetting bits, result is 2
```

Program 7 | Write a C program to multiply a given number with 2^n, without using a multiplication operator. The value of n will be entered by the user

Algorithm:

```
Step 1: Start
Step 2: Read a number (num).
Step 3: Input the value of n.
Step 4: To multiply number with 2^n, shift the bits of number in left direction n times i.e. res=num<<n
Step 5: Print number (res)
Step 6: Stop
```

PE 4-7.c

```c
//Multiply by 2 raise to the power n
#include<stdio.h>
main()
{
int num, n, res;
printf("Enter number to be multiplied\t");
scanf("%d",&num);
printf("Enter value of n\t");
scanf("%d",&n);
res=num<<n;
printf("Result of multiplication is %d",res);
}
```

Memory content

val = 4

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

res = 16

B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0

Output window

```
Enter number to be multiplied    2
Enter value of n    3
Result of multiplication is    16
```

Remark:

- Left shift by n bits is equivalent to multiplication by 2^n, provided the magnitude does not overflow

Test Yourself

1. Fill in the blanks in each of the following:

   a. An ____________ specifies an entity on which operation is to be performed.
   b. An expression that has only one operator is known as ____________.
   c. Assignment operator is ____________ associative.
   d. The operands of a modulus operator must be of ____________ type.
   e. The result of evaluation of a relational expression is a ____________ .
   f. In a compound expression, if operators of different precedence appear together, the operator of ____________ precedence operates first.
   g. The order in which operators operate depends upon the ____________ and the ____________ of the operators.
   h. If the operands of an operator are of different types, C automatically applies ____________to bring the operands to a common type.
   i. The ____________ operator returns the number of bytes the operand occupies.
   j. ____________operator has least precedence.

2. State whether each of the following is true or false. If false, explain why.

   a. The operators with the same precedence always have the same associativity.
   b. The multiplication and division operators are left-to-right associative.
   c. The sign of the result of evaluation of the modulus operator depends upon the sign of both the numerator as well as the denominator.
   d. The knowledge of precedence alone is sufficient to evaluate a compound expression.
   e. Conditional operator is a binary operator.
   f. The increment operator can only be applied to an operand that has a modifiable l-value.
   g. The expression ++a is equivalent to a+=1.
   h. In C language, there is no operator available for logical eXclusive-OR (XOR) operation.
   i. Qualified constant cannot be initialized with a value.
   j. The expression !(x>=y) is equivalent to the expression x<y.

3. Find the result of evaluation for the following expressions:

   a. 5*3/4-2
   b. ~5+3&&2
   c. 4-5&&!2
   d. 2<<2>>2
   e. 2<<2>2
   f. 2<3,4,5*3+2
   g. 5 != 10 && 2 | 3&5
   h. 2?2^2:2|5
   i. 3?~2?~5:4:3
   j. +2.25+-3.85

4. Which of the following expressions are valid? If valid, find the result of evaluation of expressions, assuming identifiers a and b are defined and their values are a=10 and b=15. If invalid, identify the errors.

   a. a+++b=20
   b. a=b==12==b
   c. ++a=23*5-4

d. b=7.5%2.5
e. 2==3+5+=6
f. 2*3/2.0&3
g. a+++++b
h. ~2~3^4
i. a^=b^=10
j. a&&=10

This page is intentionally left blank

5

DECISION-MAKING AND LOOPING STATEMENTS

Learning Objectives

In this chapter, you will learn about:

- Statements
- How statements are classified
- Non-executable statements and executable statements
- Simple statements and compound statements
- Declaration statement and definition statement
- Null statement and expression statements
- Labeled statements
- Flow control statements
- How to implement decision making
- Selection statements and jump statements
- How to perform iteration
- Iteration statements
- Role of `break` and `continue` statements
- Graphical representation of flow of control

5.1 Introduction

In the last chapter, you have learnt how to form and evaluate expressions. In C language, the expressions do not have any independent existence. To make them exist, they must be converted into statements. A **statement** is the smallest logical entity that can independently exist in a C program. In this chapter, I will tell you how to convert expressions into statements. I will also describe how statements are executed, how to make decisions with the help of branching statements and how to make a set of statements execute a number of times by using iteration statements. Finally, we will look at how various statements can be used in conjunction to perform meaningful tasks.

5.2 Statements

A **statement** is the smallest logical entity that can independently exist in a C program. No entity smaller than a statement, i.e. expressions, variables, constants, etc. can independently exist in a C program unless and until they are converted into statements. The code snippet in Program 5-1 proves the above-mentioned fact.

Line	Prog 5-1.c	Output window
1	`//Expressions cannot exist independently`	Compilation error "Statement missing ; in function main"
2	`#include<stdio.h>`	**Reason:**
3	`main()`	• Expression in line 6 cannot exist indepen-
4	`{`	dently. It should be a part of some statement
5	`    int a;`	**What to do?**
6	`    a=2+3`	• Convert expression a=2+3 into a statement
7	`    printf("Value of a is %d",a);`	by terminating it with a semicolon and re-
8	`}`	execute the code

Program 5-1 | A program to illustrate that an expression cannot exist independently

A statement in a programming language is analogous to a sentence in a natural language. Just as sentences are terminated with a period (i.e. full stop) in the English language, statements in C language are terminated with a semicolon. When an expression is terminated with a semicolon, it forms an **expression statement**. For example, a=2+3 is an expression. When it is terminated with a semicolon, it forms an expression statement, i.e. a=2+3;. Expression statements are classified according to the type of operator involved in the expression. Since an assignment operator is involved in the expression statement a=2+3;, it can be called an **assignment statement**. Moreover, as an arithmetic operator (+) is also involved in the expression statement a=2+3;, it can also be called an **arithmetic statement**.

5.3 Classification of Statements

Statements in C are classified according to the following criteria:

1. Based upon the type of action they perform
2. Based upon the number of constituent statements
3. Based upon their role

5.3.1 Based Upon the Type of Action they Perform

A statement specifies an action to be performed. Based upon the type of action it performs, the statements in C are classified into the following:

1. Non-executable statements
2. Executable statements

5.3.1.1 Non-executable Statements

Non-executable statements tell the compiler how to build a program. The important points about non-executable statements are listed as follows:

1. These statements help the compiler to determine how to allocate memory, interpret and compile other statements in a program.
2. These statements are intended mainly for the compiler, and no machine code is generated for them. Only executable[†] statements play a role during the execution of a program.
3. The order in which non-executable statements appear in a program is important. When a compiler compiles a program, it scans all the statements from top to bottom. A non-executable statement can only affect the statements that appear below it. Thus, a non-executable statement should appear only before executable statements within a block**.** [‡]
4. Only non-executable statements can appear outside the body of a function.
5. Examples of non-executable statements are function prototypes, global variable declarations, constant declarations and preprocessor directive statements.

> Although the separation between executable and non-executable statements is simple and effective, it was rather rigid earlier. This rigidity was relaxed in the C99 standard, and flexibility in terms of freely mixing executable and non-executable statements was provided.

5.3.1.2 Executable Statements

Executable statements represent the instructions that are to be performed when the program is executed. The important points about executable statements are listed as follows:

1. For an executable statement, some machine code is generated by the compiler.
2. An executable statement can appear only inside the body of a function.
3. The examples of executable statements are assignment statements, branching statements, looping statements, function call statements, etc.
4. A global definition like `const int obj=10;` appears to be an executable statement, but it is a non-executable statement.

The code segment in Program 5-2, if compiled with compilers conforming to pre-C99 standards, illustrates the fact that within a block, non-executable statements can appear only before an executable statement.

[†] Refer Section 5.3.1.2 for a description on executable statements.

[‡] Refer Section 5.3.2.2 for a description on blocks.

Line	Prog 5-2.c	Output window

1
2
3
4
5
6
7
8
9
	//Executable and Non-executable statements
	#include<stdio.h>
	#include<conio.h>
	main()
	{
	    clrscr();
	    int a=10;
	    printf("Value of a is %d",a);
	}
		Compilation error "Declaration is not allowed here"
		Remarks:
		• Borland TC 3.0 generates this error but some compilers (like Borland TC 4.5 and other latest compilers) do not enforce this constraint and does not produce an error
		• File must be saved with .C extension and not with .CPP extension
		Reason:
		• Line 6 is an executable statement but line 7 is a non-executable statement. If a compiler conforming to pre-C99 standards is used, non-executable statements can appear only before executable statements
		What to do?
		• Interchange lines 6 and 7 and re-execute the code

Program 5-2 | A program that emphasizes on the order of occurrence of executable and non-executable statements

The code snippet in Program 5-3 illustrates the fact that executable statements can appear only inside the body of a function while non-executable statements can even appear outside the body of a function, i.e. in global scope.

Line	Prog 5-3.c	Output window

1
2
3
4
5
6
7
8
9
	//Executable and Non-executable statements
	#include<stdio.h>
	#include<conio.h>
	int a=10;
	a=a*2;
	main()
	{
	    printf("Value of a is %d",a);
	}
		Compilation error "Type name expected"
		Reason:
		• Line 5 is an executable statement. Executable statements can appear only inside the body of a function, i.e. in local scope. Hence, line 5 leads to the compilation error
		What to do?
		• Place content of lines 5 after line 7 and re-execute the code

Program 5-3 | A program to show that executable statements can appear only inside the body of a function

5.3.2 Based Upon the Number of Constituent Statements

Based upon the number of constituent statements, statements in C language are classified as follows:

1. Simple statements
2. Compound statements

5.3.2.1 Simple Statements

A **simple statement** consists of a single statement. It is terminated with a semicolon. Examples of simple statements are as follows:

```
1. int variable=10;               //←definition statement
2. variable+5;                    //←expression statement
3. variable=variable+10;          //←assignment statement
```

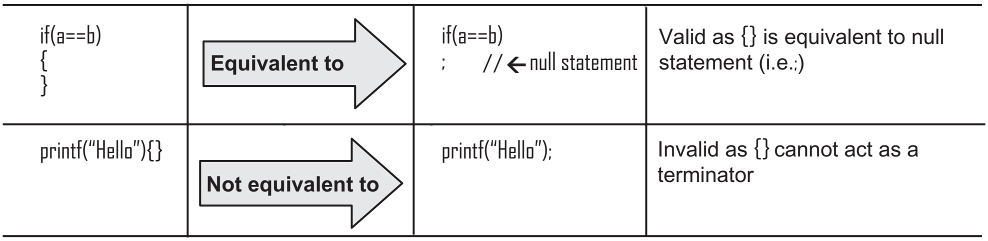
5.3.2.2 Compound Statements

A **compound statement** consists of a sequence of simple statements enclosed within a pair of braces.✍ An example of a compound statement is as follows:

```
{                       //← a compound statement consisting of three simple statements
int variable=10;
variable=variable*2;
variable+=5;
}
```

The important points about compound statements are listed below:

1. A compound statement is also known as a **block**.
2. A compound statement need not be terminated with a semicolon. However, if it is terminated with a semicolon, there will be no compilation error but it will be interpreted in a different way.[§]
3. A compound statement can be empty, i.e. there is no simple statement present inside the pair of braces, like {}. An empty compound statement is equivalent to a null[¶] statement, but it cannot act as a terminator for a statement. Figure 5.1 illustrates the interpretation of this fact.



if(a==b)
{
}
	Equivalent to	if(a==b)
		; //← null statement
			Valid as {} is equivalent to null statement (i.e.;)

printf("Hello"){}	Not equivalent to	printf("Hello");	Invalid as {} cannot act as a terminator

Figure 5.1 | Empty compound statement acts as a null statement but not as a terminator

4. A compound statement is treated as a single unit. If there is no jump[††] statement present inside the block, all the constituent simple statements will be executed in a sequence if the program control enters the block.
5. A compound statement can appear at any point in a program wherever a simple statement can appear.
6. In a block, non-executable statements (e.g. declaration statements) should come before executable statements.

[§] Refer Section 5.3.3.2 for a description on how a compound statement terminated with a semicolon is interpreted.

[¶] Refer Section 5.3.3.2 for a description on null statement.

[††] Refer Section 5.4.7 for a description on jump statements.

✍ Curly brackets, i.e. {}, are known as **braces**.

5.3.3 Based Upon their Role

Based upon their role, statements are classified as follows:

1. Declaration statement and definition statement
2. Null statement and expression statement
3. Labeled statements
4. Flow control statements
   a. Branching statements
      i. Selection statements
      ii. Jump statements
   b. Iteration statements

5.3.3.1 Declaration Statement and Definition Statement

The role of a **declaration statement** is to introduce the name of an identifier along with its data type to the compiler before its use. All identifier names (except label names) need to be declared before they are used. During declaration, no memory is allocated to an identifier. The memory space for an identifier can be reserved by using a **definition statement**. The definition statement declares an identifier and also reserves the memory space for it depending upon its data type. For example, `int a;` is a definition statement, which reserves 2 bytes (or 4 bytes) for `a` in the memory. To declare `a`, write `extern int a;`.

5.3.3.2 Null Statement and Expression Statements

A **null statement** just consists of a semicolon. For example:

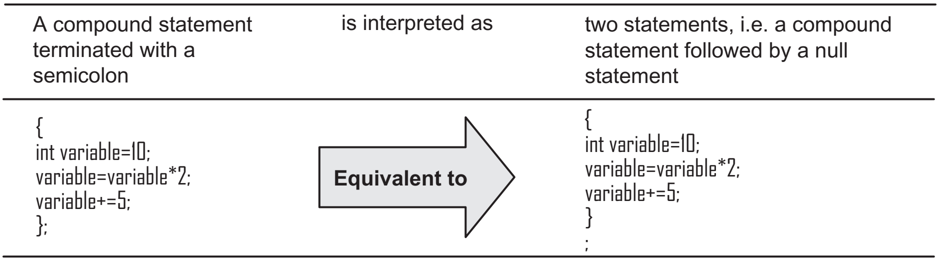
```
;          //← is a null statement
```

A null statement is the simplest form of program statement and **performs no operation**. It is just used as a place-holder, i.e. it is used when the syntax of a language construct requires a statement to be present, but the logic of a program does not require it. A null statement is equivalent to an empty compound statement, i.e. {}. A compound statement need not be terminated with a semicolon. However, if it is terminated with a semicolon, it is interpreted as a compound statement followed by a null statement. Figure 5.2 illustrates the interpretation of a compound statement, which is terminated with a semicolon.

Computations in C language are performed with the help of expression statements. An expression terminated with a semicolon forms an **expression statement**. For example:

```
a=2+3;     //← is an expression statement
```

Expression statements like `printf("Hello Readers");` in which the function call operator (i.e. `()`) is involved are called **function call statements** or **function invocations**.



A compound statement terminated with a semicolon | is interpreted as | two statements, i.e. a compound statement followed by a null statement

```
{
int variable=10;
variable=variable*2;
variable+=5;
};
```

```
{
int variable=10;
variable=variable*2;
variable+=5;
}
;
```

Figure 5.2 | Interpretation of a compound statement terminated with a semicolon

5.3.3.3 Labeled Statements

Labeled statements are rarely used in isolation. They have practical significance only when they are used in conjunction with branching statements. In the following sub-sections, the syntax of labeled statements is described. Their practical application will be discussed along with the branching statements.[††] Labeled statements are of three types:

1. Identifier-labeled statements
2. Case-labeled statements
3. Default-labeled statements

> *i* Practically, an identifier-labeled statement is used in conjunction with a goto[§§] statement. Case-labeled and default-labeled statements are useful only when they are used in conjunction with a switch[¶¶] statement.

5.3.3.3.1 Identifier-labeled Statements

The general form of an **identifier-labeled statement** is:

```
identifier: statement
```

The important points about identifier-labeled statements are listed below:

1. The identifier used in an identifier-labeled statement is called a **label name**. For example, in the following identifier-labeled statement, lab is the label name:

   ```
   lab: printf ("Labeled statement");
   ```

2. Unlike other identifiers, i.e. variable names, function names, etc., label names are not explicitly declared by using declaration statements. They are not explicitly declared because:
   a. There is no type associated with them.
   b. No operation is allowed on them. Unlike other identifiers, they cannot be used as an operand in an expression.
3. Label names are implicitly (i.e. automatically) declared by their syntactic appearance, i.e. an identifier followed by a colon and a statement is implicitly treated as a label name.
4. The statement after the label name in an identifier-labeled statement can be any statement, even some another labeled statement. For example, the following statement is an identifier-labeled statement whose constituent statement is another identifier-labeled statement.

[††] Refer Section 5.4 for a description on branching statements.

[§§] Refer Section 5.4.3 for a description on goto statement.

[¶¶] Refer Section 5.4.6 for a description on switch statement.

```
label1:          //←An identifier-labeled statement whose
    label2:      //←Constituent statement is another identifier-labeled statement
          printf("Identifier labeled statement's statement is another identifier labeled statement");
```

5. Label name should be unique within a function.
6. Label names do not alter the flow of control.[†††]
7. Identifier-labeled statements have practical significance only when they are used in conjunction with a goto statement.

The piece of code in Program 5-4 illustrates that label names do not impede the flow of control.

Prog 5-4.c

```
//Identifier labeled statements
#include<stdio.h>
main()
{
    label1:
        label2:
            printf("Identifier labeled statement");
}
```

Output window

```
Identifier labeled statement
```

Remarks:
- Label names do not alter the flow of control
- label1 followed by label2, followed by the printf statement is one statement. Thus, the mentioned code has only one simple statement

Program 5-4 | A program to illustrate that label names do not alter the flow of control

5.3.3.3.2 Case-labeled Statements

The general form of a **case labeled statement** is:

case constant-expression: statement

The important points about case labeled statements are as follows:

1. A case-labeled statement consists of the keyword case followed by a constant expression (i.e. **case label**), followed by a colon and then a statement. An example of a valid case-labeled statement is as follows:

   case 2: printf("case labeled statement");

2. The case label should be a compile time constant expression of integral type. For example, the following case-labeled statements are valid:

   a. case 2+3: printf("Valid");   //←2+3 is compile time constant expression of int type
   b. case a: printf("Valid");   //←where a is qualified constant of integral type
   c. case 'A': printf("Valid");   //←'A' is a character constant

   The following case-labeled statements are not valid:

   a. case j: printf("Invalid");   //←j is variable and not a constant
   b. case 2.5: printf("Invalid"); //←2.5 is an expression of float type and not of integral type

3. Case-labeled statements can appear only inside the body of a switch[‡‡‡] statement.

[†††] Refer Section 5.3.3.4 for a description on flow of control and flow control statements.

[‡‡‡] Refer Section 5.4.6 for a description on switch statement.

4. The constituting statement of a case-labeled statement can be any statement, even some other case-labeled statement with a different case label. For example, a case-labeled statement whose constituent statement is another case-labeled statement having a different case label is as follows:

```
case 1:              //←Case-labeled statement whose
        case 2:      //←Constituent statement is another case-labeled statement
               printf("Case labeled statement's statement is another case labeled statement");
```

5.3.3.3.3 Default-labeled Statements

The general form of a **default labeled statement** is:

default: statement

The important points about default labeled statements are as follows:

1. A default-labeled statement consists of the keyword default followed by a colon and a statement.
2. A default-labeled statement can appear only inside the body of a switch statement.
3. The constituting statement of a default-labeled statement can be any statement except the default-labeled statement. If a default-labeled statement is the constituting statement of another default-labeled statement, it leads to 'Too many default cases' compilation error. For example, the following default-labeled statement is not valid:

```
default:        //←Default-labeled statement cannot have another default-labeled statement
        default:
               printf("This is not valid");
```

5.3.3.4 Flow Control Statements

By default, statements in a C program are executed in a sequential order. The order in which the program statements are executed is known as **'flow of program control'** or just **'flow of control'**. By default, the program control flows sequentially from top to bottom. All the programs that we have developed till now have default flow of control. Many practical situations like decision making, repetitive execution of a certain task, etc. require deviation or alteration from the default flow of program control. The default flow of control can be altered by using **flow control statements**. Flow control statements are of two types:

1. Branching statements
   a. Selection statements
   b. Jump statements
2. Iteration statements

5.4 Branching Statements

Branching statements are used to transfer the program control from one point to another. They are categorized as:

1. Conditional branching: In **conditional branching**, also known as **selection**, program control is transferred from one point to another based upon the outcome of a certain condition. The following **selection statements** are available in C: if statement, if-else statement and switch statement.
2. Unconditional branching: In **unconditional branching**, also known as **jumping**, program control is transferred from one point to another without checking any condition. The following **jump statements** are available in C: goto statement, break statement, continue statement and return statement.

5.4.1 Selection Statements

Based upon the outcome of a particular condition, **selection statements** transfer control from one point to another. Selection statements select a statement to be executed among a set of various statements. The selection statements available in C are as follows:

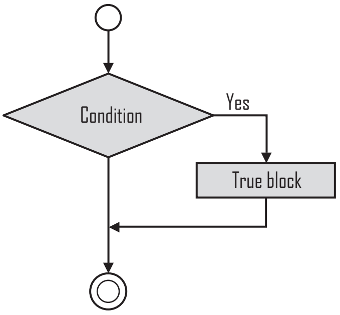
1. if statement
2. if-else statement
3. switch statement

5.4.2 if Statement

The general form of **if statement** is:

```
if(expression)      //←if header
statement           //←if body
```

Syntax of if statement	Flow diagram

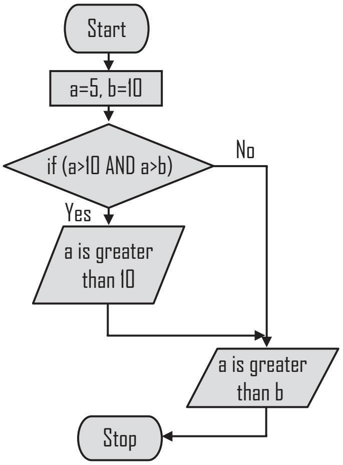
if(condition or expression)
{
Statements;   //code block for true expression
}
 	
Explanation:
 	
if the expression is true then the statement or block of statements gets executed otherwise these statements are skipped.
	

The important points about an if statement are as follows:

1. An if statement consists of an if header and an if body.
2. An if header consists of an if clause followed by an **if controlling expression** enclosed within parentheses.
3. An if statement is executed as follows:
   a. The if controlling expression is evaluated.
   b. If the if controlling expression evaluates to true, the statement constituting if body is executed.
   c. If the if controlling expression evaluates to false, if body is skipped and the execution continues from the statement following the if statement.

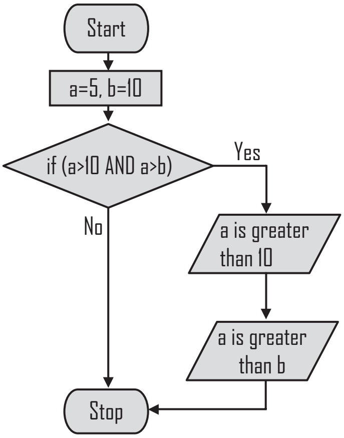
4. The syntax of an if statement permits only a single statement to be associated with if header. Practical applications often require that the execution of two or more statements should depend upon the outcome of a particular condition. In such cases, the dependent statements should be clubbed together to form a compound statement. This concept is clarified with the help of the code snippet listed in Program 5-5 and its corresponding flow chart.

Line	Prog 5-5.c	Flow chart depicting the flow of control in program	Output window

1
2
3
4
5
6
7
8
9
	//if statement
	#include<stdio.h>
	main()
	{
	int a=5, b=10;
	if(a>10 && a>b)
	    printf("a is greater than 10");
	    printf("a is greater than b");
	}
		 Start
		a=5, b=10
		if (a>10 AND a>b)
		No
		Yes
		a is greater than 10
		a is greater than b
		Stop
			a is greater than b
			Reasons:
			• Only one statement can be associated with if header
			• Irrespective of the indentation made in the program, printf statement in line 8 is not associated with the if header and is not dependent upon the result of evaluation of if controlling expression
			• Statement in line 8 is statement next to if statement and will always be executed irrespective of the outcome of if controlling expression
			What to do?
			• Club statements in lines 7 and 8 into one compound statement as shown in Program 5-6

Program 5-5 | A program to illustrate the execution of if statement

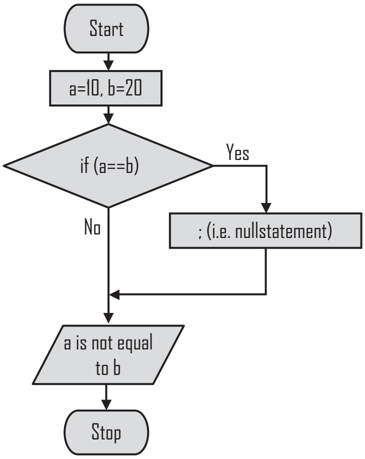
Line	Prog 5-6.c	Flow chart depicting the flow of control in program	Output window

1
2
3
4
5
6
7
8
9
10
11
	//if statement
	#include<stdio.h>
	main()
	{
	int a=5, b=10;
	if(a>10 && a>b)
	{
	    printf("a is greater than 10");
	    printf("a is greater than b");
	}
	}
		 Start
		a=5, b=10
		if (a>10 AND a>b)
		Yes
		No
		a is greater than 10
		a is greater than b
		Stop
			No output
			Reasons:
			• Lines 7–10 constitute a compound statement
			• The execution of both the statements in lines 8 and 9 is dependent upon the result of evaluation of if controlling expression
			• Since the if controlling expression evaluates to false, statements in lines 8 and 9 do not get executed

Program 5-6 | A program to illustrate the execution of if statement

5. No semicolon should be placed at the end of the if header. However, if a semicolon is placed at the end of the if header, there will be no compilation error (although this may lead to logical error). This is one of the potential logical errors most amateur programmers do. The logical error due to the semicolon placed at the end of the if header is depicted in the code listed in Program 5-7.

Line	Prog 5-7.c	Flow chart depicting the flow of control in program	Output window

1
2
3
4
5
6
7
8
	//if statement
	#include<stdio.h>
	main()
	{
	int a=10, b=20;
	if(a==b);
	printf("a is not equal to b");
	}
		 Start
		a=10, b=20
		if (a==b)
		Yes
		No
		; (i.e. nullstatement)
		a is not equal to b
		Stop
			a is not equal to b
			Expected output:
			No output
			Reason for deviation:
			Presence of semicolon at the end of the if header
			How is the listed code interpreted?
			• It is interpreted as:
			if(a==b)
			;
			printf("a is not equal to b");
			• if body is a null statement
			• printf statement is next to the if statement and its execution does not depend upon the outcome of if controlling expression

Program 5-7 | A program to illustrate the effect of the semicolon placed at the end of the if header

5.4.3 if-else Statement

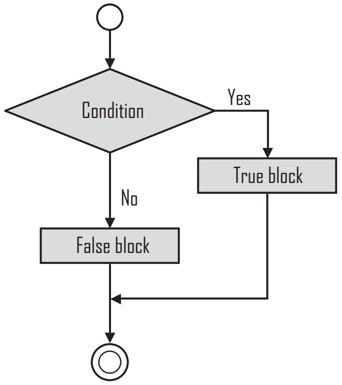
Most of the problems require one set of actions to be performed if a particular condition is true, and another set of actions to be performed if the condition is false. To implement such a decision, C language provides an **if-else statement**. The general form of the if-else statement is:

```
if(expression)          //←if-else header
statement1              //←if body
else                    //←else clause
statement2              //←else body
```

Syntax of if-else statement	Flow diagram

if(condition or expression)
{
True Statements   //code block
}
else
{	

(*Contd...*)

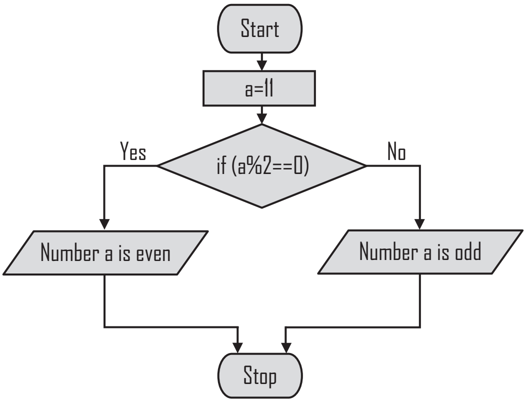
False statements //code block } Explanation: if the expression is true then the true statement or block of statements gets executed and else body is skipped. if the expression is false then the false statement or block of statements gets executed and if - true body is skipped.	

The important points about an if-else statement are as follows:

1. An if-else statement consists of an if-else header, if body, else clause and else body.
2. An if-else header consists of an if clause followed by an **if-else controlling expression** enclosed within parentheses.
3. An if-else statement is executed as follows:
   a. The if-else controlling expression is evaluated.
   b. If the if-else controlling expression evaluates to true, the statement constituting the if body is executed and the else body is skipped.
   c. If the if-else controlling expression evaluates to false, the if body is skipped and the else body is executed.
   d. After the execution of the if body or the else body, the execution continues from the statement following the if-else statement.

The code snippet in Program 5-8 illustrates the use of the if-else statement.

Line	Prog 5-8.c	Flow chart depicting the flow of control in program	Output window

1
2
3
4
5
6
7
8
9
10
11
	//if-else statement
	//Find whether no. is even or odd
	#include<stdio.h>
	main()
	{
	int a=11;
	if(a%2==0)
	    printf("Number a is even");
	else
	    printf("Number a is odd");
	}
		 Start
		a=11
		if (a%2==0)
		Yes
		No
		Number a is even
		Number a is odd
		Stop
			Number a is odd
			Remarks:
			• The if-else controlling expression a%2==0 evaluates to false
			• The if body is skipped and the else body gets executed

Program 5-8 | A program to illustrate the use of the if-else statement

4. The syntax of if-else statement permits only a single statement to be associated with if clause and else clause. However, this single statement can be a compound statement

constituting a number of simple statements. Consider the piece of code in Program 5-9.

Line	Prog 5-9.c	Output window

1
2
3
4
5
6
7
8
9
10
11
	//if-else statement
	#include<stdio.h>
	main()
	{
	int a=11;
	if(a>10)
	    printf("The value of a is %d",a);
	    printf("Value a is greater than 10");
	else
	    printf("Value a is less than 10");
	}
		Compilation error "Misplaced else in function main"
		Reasons:
		• Only a single statement can be associated with if clause and else clause
		• The mentioned code is interpreted as:
		if(a>10) **//←if statement**
		    printf("The value of a is %d",a);
		printf("Value a is greater than 10"); **//←statement next to if statement**
		else **//←else clause without any matching if clause**
		    printf("Value a is less than 10");
		• else clause cannot exist without a matching if clause
		What to do?
		• Club statements in lines 7 and 8 into one compound statement and re-execute the code

Program 5-9 | A program to illustrate the use of the if-else statement

5. Care must be taken that no semicolon is placed at the end of the if-else header or after the else clause.

5.4.4 Nested if Statement

If the body of the if statement is another if statement or contains another if statement (as shown below), then we say that if's are nested and the statement is known as a **nested if statement**. The general form of a nested if statement is:

```
if(expression)
    if statement
```

(a)

Body of an **if** statement is another **if** statement

or

```
if(expression)                    //←nested if statement
{          statement
           -------------
           if statement
           -------------
           statement
}
```

(b)

Body of an **if** statement contains another **if** statement

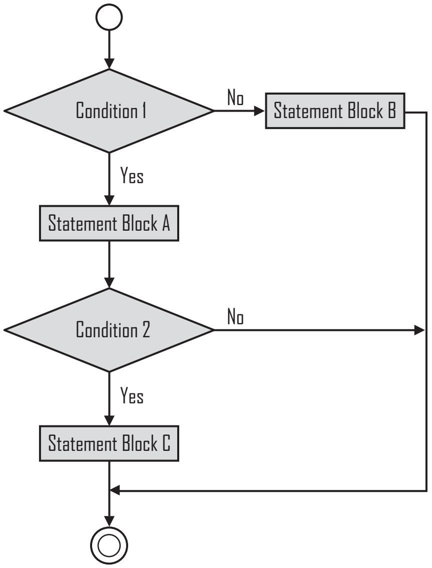
This nesting can be done up to any level as shown below:

```
if(expression1)
          if(expression2)
```

```
                    if(expression-n)
                            statement
```

The above structure seems to form a ladder and is known as the **if ladder**.

Syntax of nested if statement	Flow diagram

if(Condition 1 or expression1)
{
Statement A
   if(Condition 2 or expression2)
   {
    Statement C
    }
else
{
     Statement B
}
 	
Explanation:
if the expression is true then the true statement or block of statements gets executed and executes another if else inside it.
 	
if the expression is false then the false statement or block of statements gets executed and if - true body is skipped.
	 Condition 1
	No
	Statement Block B
	Yes
	Statement Block A
	Condition 2
	No
	Yes
	Statement Block C

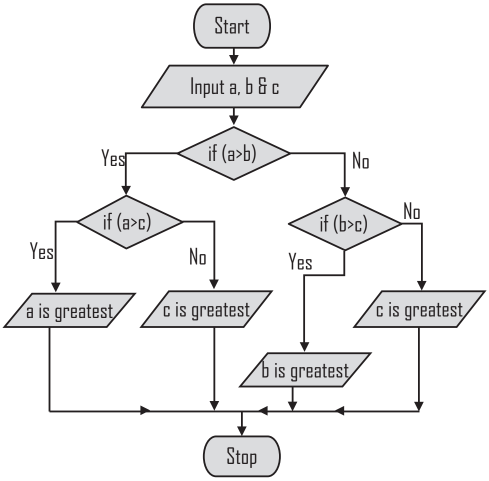
i The number of levels up to which nesting can be done depends upon the translation limits of the compiler. The translation limits constrain the implementation of language translators and libraries.

5.4.5 Nested if-else Statement

In a **nested if-else statement**, the if body or else body of an if-else statement is, or contains, another if statement or if-else statement. Program 5-10 illustrates the use of a nested if-else statement to find the greatest of three numbers.

The careless use of a nested if-else statement introduces a source of potential ambiguity referred to as the **dangling else ambiguity**. When **a statement** contains more number of if clauses than else clauses, then there exists a potential ambiguity regarding with which if clause does the else clause properly matches up. This ambiguity is known as **dangling else problem**. The code listed in the column 1 of Table 5.1 suffers from a dangling else problem. The other columns in the table show the two possible interpretations of the code listed in column 1.

Line	Prog 5-10.c	Flow chart depicting the flow of control in program	Output window

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
	//Nested if-else statement
	#include<stdio.h>
	main()
	{
	int a, b, c;
	printf("Enter three numbers\t");
	scanf("%d %d %d",&a,&b,&c);
	if(a>b)
	{ //←if body starts
	    if(a>c)
	        printf("%d is greatest", a);
	    else
	        printf("%d is greatest",c);
	}//←if body ends
	else
	{//←else body starts
	    if(b>c)
	        printf("%d is greatest",b);
	    else
	        printf("%d is greatest",c);
	}//←else body ends
	}
		 Start
		Input a, b & c
		if (a>b)
		Yes
		No
		if (a>c)
		Yes
		No
		if (b>c)
		Yes
		No
		a is greatest
		c is greatest
		c is greatest
		b is greatest
		Stop
			Enter three numbers 1 4 2
			4 is greatest
			Remarks:
			• The program illustrates the use of nested if-else statement
			• Both if body and else body of if-else statement consists of if-else statement

Program 5-10 | A program that uses a nested if-else statement to find the greatest of three numbers

Table 5.1 | The code in column 1 suffers from dangling else ambiguity. Columns 2 and 3 depict the two possible interpretations of the code listed in column 1

Line	Code suffering from dangling else problem (Column 1)	Interpretation-I (Column 2)	Interpretation-II (Column 3)

1
2
3
4
5
6
7
8
9
10
11
	//Dangling else problem
	#include<stdio.h>
	main()
	{
	int a=10, b=20;
	if(a==100)
	if(b==20)
	printf("Match-I");
	else
	printf("Match-II");
	}
		//Interpretation-I
		#include<stdio.h>
		main()
		{
		int a=10, b=20;
		if(a==100)
		    if(b==20)
		        printf("Match-I");
		else
		    printf("Match-II");
		}
			//Interpretation-II
			#include<stdio.h>
			main()
			{
			int a=10, b=20;
			if(a==100)
			    if(b==20)
			        printf("Match-I");
			    else
			        printf("Match-II");
			}

	Output	**If interpreted in this way, the output would be:**	**If interpreted in this way, the output would be:**
	No output	Match-II	No output

The dangling else problem is solved in two ways:

1. **Implicitly by compiler:** The dangling else ambiguity is implicitly resolved by the compiler by matching the else clause with the last occurring unmatched if, i.e. interpreted in a way as shown in column 3 of Table 5.1. The outputs in columns 1 and 3 are the same. This indicates that the code in column 1 is interpreted in the same way as shown in column 3 of Table 5.1.
2. **Explicitly by user:** The dangling else ambiguity can be explicitly removed by the user by using braces. This is shown in Table 5.2.

Table 5.2 | Dangling else ambiguity removed explicitly by the user

Line	Code suffering from dangling else problem (Column 1)	Dangling else ambiguity removed from the code listed in column 1 by using braces (Column 2)	Dangling else ambiguity removed from the code listed in column 1 by using braces (Column 3)

1
2
3
4
5
6
7
8
9
10
11
	//Dangling else problem
	#include<stdio.h>
	main()
	{
	int a=10, b=20;
	if(a==100)
	if(b==20)
	printf("Match-I");
	else
	printf("Match-II");
	}
		//Dangling else problem
		#include<stdio.h>
		main()
		{
		int a=10, b=20;
		if(a==100)
		{
		if(b==20)
		printf("Match-I");
		}
		else
		printf("Match-II");
		}
			//Dangling else problem
			#include<stdio.h>
			main()
			{
			int a=10, b=20;
			if(a==100)
			{
			if(b==20)
			printf("Match-I");
			else
			printf("Match-II");
			}
			}

	Output	**Output**	**Output**
	No output	Match-II	No output

5.4.6 switch Statement

A **switch statement** is used to control complex branching operations. When there are many conditions, it becomes too difficult and complicated to use if and if-else constructs. In such cases, the switch statement provides an easy and organized way to select among multiple options, depending upon the outcome of a particular condition. The general form of a switch statement is:

```
switch(expression)        //←switch header
statement                 //←switch body
```

Syntax of switch statement

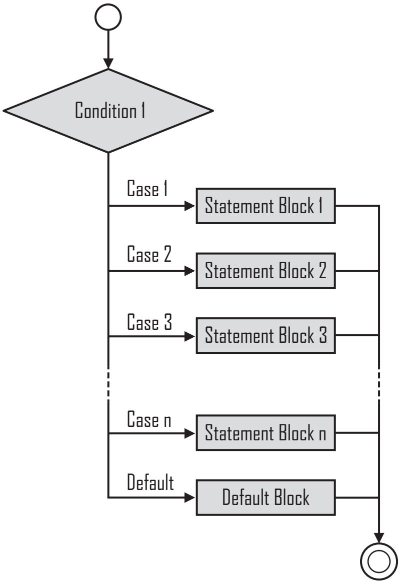
```
switch (expression)
{
case label1:
          statements;        // code for block 1
          break;
case label2:
         statements;        // code for block 2
         break;
case label3:
         statements;        // code for block 3
         break;
          .
          .
          .
case label n:
         statements;        // code for block n
         break;

default:
         statements;     // code for default block
}
```

Explanation:
If a condition is met in switch case then execution continues on into the next case clause

if it is not explicitly specified that the execution should exit the switch statement. This is achieved by using *break* keyword.

If none of the listed conditions is met then default condition executed.

Flow diagram



The important points about a switch statement are as follows:

1. A switch statement consists of a switch header and a switch body.
2. A switch header consists of the keyword switch followed by a switch selection expression enclosed within parentheses.
3. The switch selection expression must be of integral type (i.e. integer type or character type).
4. The switch body consists of a statement. The statement constituting a switch body can be a null statement, an expression statement, a labeled statement, a flow control statement, a compound statement, etc.
5. Generally, a switch body consists of a compound statement, whose constituent statements are case-labeled statements, expression statements, flow control statements and an optional default-labeled statement.
6. Case labels of case-labeled statements constituting the body of a switch statement should be unique, i.e. no two case labels should have or evaluate to the same value.
7. There can be at most one default labeled statement within the switch body.

8. A switch statement is executed as follows:
   a. The switch selection expression is evaluated.
   b. The result of evaluation of switch selection expression is compared with the case labels of the case-labeled statements until there is a match or until all the case-labeled statements have been examined.
      i. If the result of evaluation of switch selection expression is matched with the case label of a case-labeled statement, the execution starts from the matched case-labeled statement, and all the statements after the matched case-labeled statement within the switch body gets executed.
      ii. If no case label of case-labeled statements within the switch body matches the result of evaluation of switch selection expression, the execution starts with the default-labeled statement, if it is present, and all the statements after the default-labeled statement within the switch body gets executed.
      iii. If none of the case labels match the result of evaluation of switch selection expression and there is no default-labeled statement present within the switch body, no statement within the switch body will be executed and the execution continues from the statement following the switch statement.

> *i* It is a common misunderstanding that only the matched case-labeled statement or the default-labeled statement (if none of the case labels match) gets executed. In fact, the execution begins with the matched case labeled statement or the default labeled statement, and all the statements after the matched case labeled statement or the default labeled statement within the switch body get executed.

The code snippets in Programs 5-11 to 5-13 clarify the points discussed above.

Prog 5-11.c

```
//switch statement
#include<stdio.h>
main()
{
int a=1;
switch(a)
{
case 1:
    printf("This is case option 1\n");
    printf("Value of a is %d\n",a);
case 2:
    printf("This is case option 2\n");
default:
    printf("This is default option\n");
}
}
```

Output window

```
This is case option 1
Value of a is 1
This is case option 2
This is default option
```

Remarks:

- A switch body consists of four statements and is interpreted as:

```
{
case 1:                          //←Statement 1: case-labeled statement
    printf("This is case option 1\n");
printf("Value of a is %d\n",a);  //←Statement 2: function call statement
case 2:                          //←Statement 3: case-labeled statement
    printf("This is case option 2\n");
default:                         //←Statement 4: default-labeled statement
    printf("This is default option\n");
}
```

- Since case label 1 matches the result of evaluation of switch selection expression, the execution starts from the statement with the case label 1, and all the statements after it within the switch body gets executed

Program 5-11 | A program to illustrate the working of a switch statement

Prog 5-12.c

```
//switch statement
#include<stdio.h>
main()
{
int a=3;
switch(a)
{
case 1:
    printf(“This is case option 1\n”);
    printf(“Value of a is %d\n”,a);
default:
    printf(“This is default option\n”);
case 2:
    printf(“This is case option 2\n”);
 }
}
```

Output window

```
This is default option
This is case option 2
```

Remarks:

- There is no constraint about the position of a default-labeled statement within the switch body
- Since none of the case labels match the result of evaluation of a switch selection expression, the execution begins with the default-labeled statement
- All the statements after the default-labeled statement within the switch body gets executed

Program 5-12 | A program to illustrate that there is no constraint about the position of a default-labeled statement within the switch body

Prog 5-13.c

```
// switch statement & ranges
#include<stdio.h>
main()
{
char exp=‘E’;
switch(exp)
{
case ‘a’:
case ‘e’:
case ‘i’:
case ‘o’:
case ‘u’:
    printf(“Lower case vowel\n”);
case ‘A’:
case ‘E’:
case ‘I’:
case ‘O’:
case ‘U’:
    printf(“Upper case vowel\n”);
}
}
```

Output window

```
Upper case vowel
```

Remarks:

- A switch body consists of two case-labeled statements. The code is interpreted as:

```
case ‘a’:      //←Statement 1: case-labeled statement
   case ‘e’:     //←Constituent statement of case-labeled statement is
case ‘i’:   //←another case-labeled statement
        case ‘o’:
               case ‘u’:
                   printf(“Lower case vowel\n”);
case ‘A’:    //←Statement 2: case-labeled statement
    case ‘E’:
        case ‘I’:
            case ‘O’:
                case ‘U’:
                    printf(“Upper case vowel\n”);
```

- In this way, the switch statement can be used to switch on ranges
- This is only beneficial when the ranges are small
- C language does not support the following ways for switching on ranges:
    - case ‘A’-‘Z’ //←if used, it will be interpreted as case -25 (i.e. ASCII code of ‘A’- ASCII code of ‘Z’)
    - case ‘A’ to ‘Z’ //←allowed in Visual Basic but not in C language

Program 5-13 | A program to illustrate the use of a switch statement to switch on ranges

5.4.7 Jump Statements

A **jump statement** transfers the control from one point to another without checking any condition, i.e. unconditionally. The following jump statements are present in C language:

1. goto statement
2. break statement
3. continue statement
4. return statement

5.4.8 goto Statement

The **goto statement** is used to branch unconditionally from one point to another within a function. It provides a highly unstructured way of transferring the program control from one point to another within a function. It often makes the program control difficult to understand and modify. Thus, the use of a goto statement is discouraged in powerful structured programming languages like C. The syntactic form of a goto statement is:

```
goto label;
```

The important points about a goto statement are as follows:

1. The goto statement is always used in conjunction with an identifier-labeled statement. Within the body of a function in which the goto statement is present, an identifier-labeled statement with a label name, same as the label name used in the goto statement should be present.
2. The goto statement on execution transfers the program control to an identifier-labeled statement having a label name same as the label name used in the goto statement.
3. The goto statement can be used to make a forward jump as well as a backward jump. If the goto statement is present before its corresponding identifier-labeled statement, the jump made will be known as a **forward jump**. If the goto statement is present after its corresponding identifier-labeled statement, the jump made will be known as a **backward jump**. The forward and backward jumps are shown in Figure 5.3.

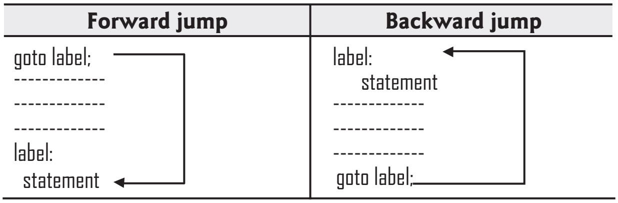


Figure 5.3 | Forward and backward jump

4. There can be two or more goto statements corresponding to an identifier-labeled statement but there cannot be two or more identifier-labeled statements corresponding to a goto statement. The interpretation of this rule is illustrated in Figure 5.4.

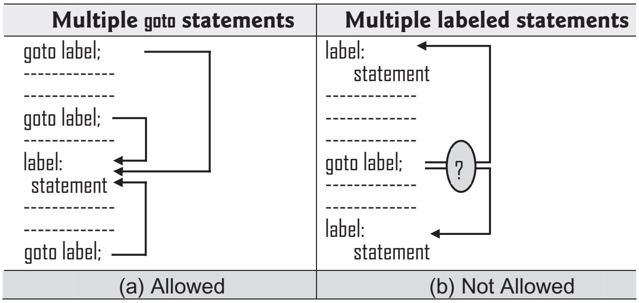


Figure 5.4 | (a) Multiple goto statements corresponding to one identifier-labeled statement are allowed; (b) multiple identifier-labeled statements corresponding to one goto statement are not allowed

5. The goto statement can transfer control anywhere within a function, i.e. it can take control in or out of a nested if statement, nested if-else statement or nested loops. However, a goto statement in no way can take control out of the function in which it is used.

5.4.9 break Statement

The syntactic form of a **break statement** is:

```
break;
```

The important points about a break statement are as follows:

1. A break statement can appear only inside, or as a body of, a switch statement or a loop.[§§§] The code snippet listed in Program 5-14 verifies this fact.

Line	Prog 5-14.c	Output window

1
2
3
4
5
6
7
8
9
10
11
12
	//break statement
	#include<stdio.h>
	main()
	{
	int a=10;
	if(a==10)
	{
	    printf("if controlling expression evaluates to true");
	    break;
	    printf("The value of a is %d",a);
	}
	}
		Compilation error "Misplaced break in function main"
		Reasons:
		• The break statement can appear only inside or as a switch/loop body
		• The break statement present in line 9 is neither a part of a switch body nor a loop body
		What to do?
		• Either remove the break statement from if body or place the if statement inside a loop body or a switch body

Program 5-14 | A program to illustrate that the break statement cannot appear outside the switch body or a loop

2. A break statement terminates the execution of the nearest enclosing switch or the nearest enclosing loop. The execution resumes with the statement present next to the terminated switch statement or the terminated loop. The interpretation of this point is illustrated in the code snippet listed in Program 5-15.

[§§§] Refer Section 5.5 for a description on loops.

Prog 5-15.c

```
//break statement
#include<stdio.h>
main()
{
int a=1;
switch(a)
{
case 1:
     printf("One");
     break;
case 2:
     printf("Two");
     break;
default:
     printf("Default");
}
printf("\nThis statement is next to switch");
}
```

Output window

```
One
This statement is next to switch
```

Remarks:

- The switch body consists of five statements and is interpreted as:

```
{
case 1:              //←Statement 1: case-labeled statement
     printf("One");
break;               //←Statement 2: break statement
case 2:              //←Statement 3: case-labeled statement
     printf("Two");
break;               //←Statement 4: break statement
default:             //←Statement 5: default-labeled statement
     printf("Default");
}
```

- Execution starts from the statement with the case label 1
- Execution of break statement terminates the switch statement and the control immediately transfers to the statement present next to the switch statement, i.e. printf("\nThis statement is next to switch");

Program 5-15 | A program to illustrate the execution of a switch statement

5.4.10 continue Statement

The syntactic form of a **continue statement** is:

```
continue;
```

The important points about a continue statement are as follows:

1. A continue statement can appear only inside, or as the body of, a loop.
2. A continue[¶¶¶] statement terminates the current iteration of the nearest enclosing loop. The semantics of the continue statement will be discussed after iteration statements.

5.4.11 return Statement

The general forms of a **return statement** are:

```
return;    or              //←Form 1
return expression;         //←Form 2
```

The important points about a return statement are as follows:

1. A return statement without an expression (i.e. Form 1) can appear only in a function whose return type is void.
2. A return statement with an expression (i.e. Form 2) should not appear in a function whose return type is void.
3. A return statement terminates the execution of a function and returns the control to the calling function.

[¶¶¶] Refer Section 5.5.7.2 for a description on the semantics of a continue statement.

The syntax and semantics of a return statement will be discussed in Chapter 8.

5.5 Iteration Statements

Iteration is a process of repeating the same set of statements again and again until the specified condition holds true. Humans find iterative tasks boring but computers are very good at performing iterative tasks. Computers execute the same set of statements again and again by putting them in a loop. The C language provides the following three **iteration statements**:

1. for statement
2. while statement
3. do-while statement

In general, loops are classified as:

1. Counter-controlled loops
2. Sentinel-controlled loops

5.5.1 Counter-Controlled Loops

Counter-controlled looping is a form of looping in which the number of iterations to be performed is known in advance. Counter-controlled loops are so named because they use a control variable, known as the **loop counter**, to keep a track of loop iterations. The counter-controlled loop starts with the initial value of the loop counter and terminates when the final value of the loop counter is reached. Since the counter-controlled loops iterate a fixed number of times, which is known in advance, they are also known as **definite repetition loops**. There are three main ingredients of counter-controlled looping:

1. Initialization of the loop counter.
2. An expression (specifically a condition) determining whether the loop body should be executed or not.
3. An expression that manipulates the value of the loop counter so that the condition in step 2 eventually becomes false and the loop terminates.

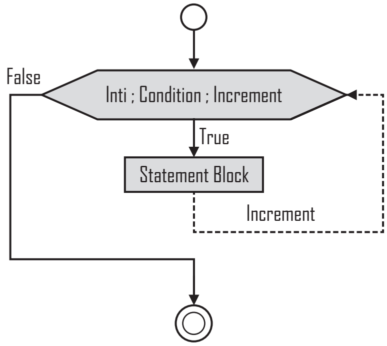
Firstly, I will describe the syntax of looping statements available in C language and how they can be used for counter-controlled looping. In Section 5.5.5, I will describe the use of available iteration statements for sentinel-controlled looping.

5.5.2 for Statement

Out of all the looping constructs available in C, **for statement** is the most popular one. The general form of a for statement is:

```
for(expression1; expression2; expression3)          //←for header
statement                                           //←for body
```

Syntax of for statement	**Flow diagram**

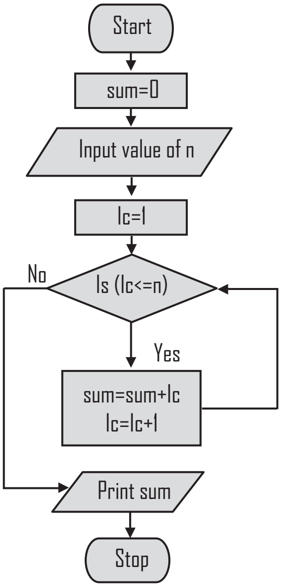
for (init; condition; increment)
{
statement(s);
}
 	
Explanation:
 	
for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
	 False
	Inti ; Condition ; Increment
	True
	Statement Block
	Increment

The important points about a for statement are as follows:

1. The for statement consists of for header and for body.
   Points about for header:
2. The for header consists of the keyword for followed by three expressions separated by semicolons and enclosed within parentheses.
3. All the expressions in the for header are optional and can be skipped. Even if all the expressions are missing, it is mandatory to create three sections by placing two semicolons.
4. Three sections are named as: initialization section, condition section and manipulation section.
   a. **Initialization section:** expression$_1$ constitutes the initialization section. It is used to initialize (i.e. assign a starting value to) the loop counter. If the loop counter has already been initialized, the initialization expression, i.e. expression$_1$ can be skipped. However, a semicolon is necessary and must be placed.
   b. **Condition section:** expression$_2$ forms the condition section. expression$_2$ tests the value of the loop counter. This section determines whether the body of the loop is to be executed or not. In case of infinite loops, the condition section can be skipped.
   c. **Manipulation section:** expression$_3$ is part of the manipulation section. The manipulation expression manipulates the value of the loop counter so that the expression$_2$ present in the condition section eventually evaluates to false and the loop terminates.
5. Care must be taken that the for header is not terminated with a semicolon. If it is terminated with a semicolon, the semicolon is interpreted as a null statement following the for header (i.e. it is treated as for body).
   A point about for body:
6. The syntax of for statement permits only a single statement to be associated with for header. If a number of statements are to be executed repeatedly, the statements should be clubbed together to form a compound statement.
   Execution of for statement:

7. The for statement is executed as follows:
    a. Initialization section is executed only once at the start of the loop.
    b. The expression present in the condition section is evaluated.
        i. If it evaluates to true, the body of the loop is executed.
        ii. If it evaluates to false, the loop terminates and the program control is transferred to the statement present next to the for statement.
    c. After the execution of the body of the loop, the manipulation expression is evaluated.
    d. These three steps represent the first iteration of the for loop. For the next iterations, Steps b and c are repeated until the expression in Step b evaluates to false.

The facts mentioned above are illustrated in Program 5-16.

Line	Prog 5-16.c	Flow chart depicting the flow of control in program	Output window
1	//Use of for statement to find the	 Start sum=0 Input value of n lc=1 Is (lc<=n) No Yes sum=sum+lc lc=lc+1 Print sum Stop	Enter the value of n 10
2	//sum of first n natural numbers		Sum is 55
3	#include<stdio.h>		
4	main()		
5	{		
6	int n, lc, sum=0;		
7	printf("Enter the value of n\t");		
8	scanf("%d",&n);		
9	for(lc=1;lc<=n;lc++)		
10	sum=sum+lc;		
11	printf("Sum is %d",sum);		
12	}		

Program 5-16 | A program to illustrate the use of for statement for finding the sum of first n natural numbers

The codes in Table 5.3 are equivalent to the code specified in Program 5-16.

Table 5.3 | Codes equivalent to the code listed in Program 5-16

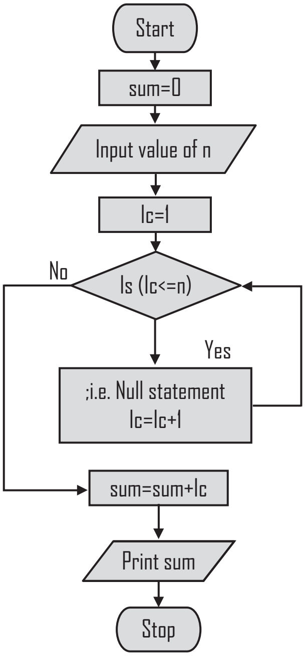
Line	Equivalent Code-I	Equivalent Code-II	Equivalent Code-III
1	//Use of for statement to find the sum	//Use of for statement to find the sum	//Use of for statement to find the
2	// of first n natural numbers	//of first n natural numbers	//sum of first n natural numbers
3	#include<stdio.h>	#include<stdio.h>	#include<stdio.h>
4	main()	main()	main()
5	{	{	{
6	int n, lc=1, sum=0; **//Initialization of lc**	int n, lc, sum=0;	int n, lc=1, sum=0;
7	printf("Enter the value of n\t");	printf("Enter the value of n\t");	printf("Enter the value of n\t");

(Contd...)

8	scanf("%d",&n);	scanf("%d",&n);	scanf("%d",&n);
9	for(;lc<=n;lc++)**//Initialization missing**	for(lc=1;lc<=n;)**//Manipulation missing**	for(;lc<=n;)**//Both missing**
10	sum=sum+lc;	sum=sum+lc++;**//Manipulation of lc**	sum=sum+lc++;
11	printf("Sum is %d",sum);	printf("Sum is %d",sum);	printf("Sum is %d",sum);
12	}	}	}

The code snippet in Program 5-17 illustrates the effect of presence of a semicolon at the end of for header.

Line	Prog 5-17.c	Flow chart depicting the flow of control in program	Output window

1
2
3
4
5
6
7
8
9
10
11
	//Effect of ; at end of for header
	#include<stdio.h>
	main()
	{
	int n, lc, sum=0;
	printf("Enter the value of n\t");
	scanf("%d",&n);
	for(lc=1;lc<=n;lc++);
	sum=sum+lc;
	printf("Sum is %d",sum);
	}
		 Start
		sum=0
		Input value of n
		lc=1
		No
		Is (lc<=n)
		Yes
		;i.e. Null statement
		lc=lc+1
		sum=sum+lc
		Print sum
		Stop
			Enter the value of n 10
			Sum is 11
			Remarks:
			• for header is terminated with a semicolon
			• Semicolon is interpreted as null statement and forms the for body
			• The statement sum=sum+lc; is a statement present next to the for statement and thus gets executed only once
			• The value of lc on the termination of loop will be 11
			• This value of lc is added to sum to produce the mentioned output

Program 5-17 | A program to illustrate the effect of a semicolon at the end of a for header

5.5.3 while Statement

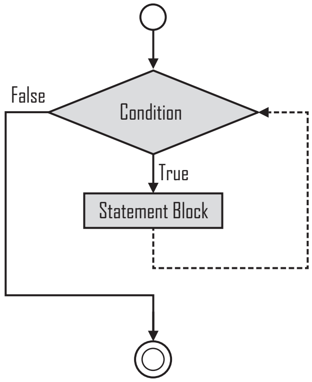
The general form of a **while statement** is:

```
while(expression)          //←while header
statement                  //←while body
```

Syntax of while statement	Flow diagram

while(condition or expression)
{
statement(s);   // code block
}	

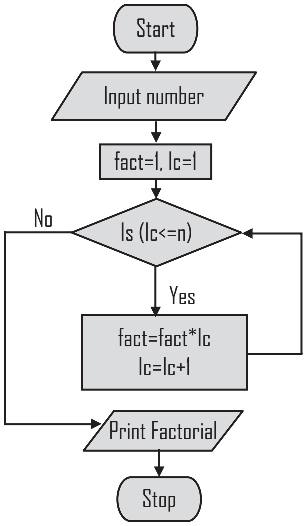
(*Contd...*)

Syntax of while statement	Flow diagram
Explanation: Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.	

The important points about a while statement are as follows:

1. The while statement consists of while header and while body.
2. The while header consists of keyword while followed by while controlling expression enclosed within the parentheses.
3. The controlling expression in while header is mandatory and cannot be skipped.
4. The while header should not be terminated with a semicolon. If it is terminated with a semicolon, the semicolon is interpreted as a null statement following the while header (i.e. it is treated as a while body).
5. The syntax of a while statement permits only a single statement to be associated with while header. If a number of statements are to be executed repeatedly, the statements should be clubbed together to form a compound statement.
6. The while statement is executed as follows:
   a. The while controlling expression is evaluated.
      i. If it evaluates to true, the body of the loop is executed.
      ii. If it evaluates to false, the program control is transferred to the statement present next to the while statement.
   b. After executing the while body, the program control returns back to the while header.
   c. Steps a and b are repeated until the while controlling expression in Step a evaluates to false.
7. While making the use of while statement, always remember to initialize the loop counter before the while controlling expression is evaluated and to manipulate the loop counter inside the body of while statement, i.e. before the while controlling expression is evaluated again.

The facts mentioned above are illustrated in Program 5-18.

Line	Prog 5-18.c	Flow chart depicting the flow of control in program	Output window
1	//Use of while statement to find	 Start Input number fact=1, lc=1 No Is (lc<=n) Yes fact=fact*lc lc=lc+1 Print Factorial Stop	Enter number 5 Factorial is 120 **Remarks:** • Line 9 initializes the value of fact to 1. It is important to initialize fact to 1 else garbage will be the result • Line 10 provides the initialization of loop counter • Line 14 manipulates the loop counter
2	//the factorial of a number		
3	#include<stdio.h>		
4	main()		
5	{		
6	int num, lc, fact;		
7	printf(“Enter number\t”);		
8	scanf(“%d”,&num);		
9	fact=1;		
10	lc=1; //**Initialization of loop counter**		
11	while(lc<=num)		
12	{		
13	fact=fact*lc;		
14	lc=lc+1; //**Manipulation of loop counter**		
15	}		
16	printf(“Factorial is %d”,fact);		
17	}		

Program 5-18 | A program to find the factorial of a number using while loop

The codes in Table 5.4 are equivalent to the code specified in Program 5-18.

Table 5.4 | Codes equivalent to the code listed in Program 5-18

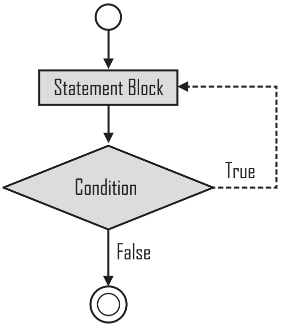
Line	Equivalent Code-I	Equivalent Code-II	Equivalent Code-III
1	//Use of while statement to find	//Use of while statemnt to find	//Use of while statement to find
2	//the factorial of a number	//the factorial of a number	//the factorial of a number
3	#include<stdio.h>	#include<stdio.h>	#include<stdio.h>
4	main()	main()	main()
5	{	{	{
6	int num, lc=1, fact=1;	int num, lc=1, fact=1;	int num, lc=0, fact=1;
7	printf(“Enter number\t”);	printf(“Enter number\t”);	printf(“Enter number\t”);
8	scanf(“%d”,&num);	scanf(“%d”,&num);	scanf(“%d”,&num);
9	while(lc<=num)	while(lc<=num)	while(lc++<num)
10	{	fact=fact*lc++;	fact=fact*lc;
11	fact=fact*lc;	printf(“Factorial is %d”,fact);	printf(“Factorial is %d”,fact);
12	lc=lc+1;	}	}
13	}		
14	printf(“Factorial is %d”,fact);		
15	}		

5.5.4 do-while Statement

The general form of **do-while statement** is:

```
do                          //←do-while header
statement                   //←do-while body
while(expression);          //←while clause
```

Syntax of do-while statement	Flow diagram

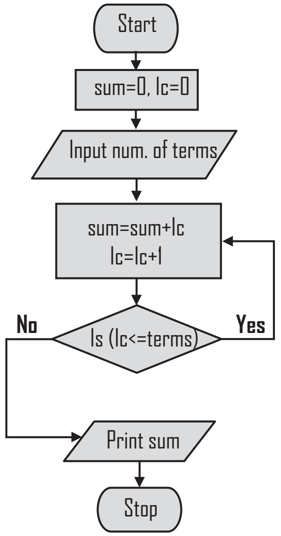
do
{
statement(s); // code block
} while(condition or expression);
Explanation:
The conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.
If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again.
This process repeats until the given condition becomes false.
	

The important points about a do-while statement are as follows:

1. The do-while statement consists of a do clause, followed by a statement that constitutes do-while body, followed by the while clause consisting of while keyword followed by do-while controlling expression enclosed within parentheses. The while clause is terminated with a semicolon.
2. The controlling expression in a do-while statement is mandatory and cannot be skipped.
3. The syntax of a do-while statement permits only a single statement to be present. If a number of statements are to be executed repeatedly, the statements should be clubbed together to form a compound statement.
4. The do-while statement is executed as follows:
   a. The statement, i.e. body of do-while statement, is executed.
   b. After the execution of a do-while body, the do-while controlling expression is evaluated.
      i. If it evaluates to true, the statement, i.e. do-while body is executed again and Step b is repeated.
      ii. If it evaluates to false, the program control is transferred to the statement present next to the do-while statement.
5. While making the use of a do-while statement, always remember to initialize the loop counter before the do-while statement and to manipulate it inside the body of the do-while statement so that the do-while controlling expression eventually becomes false.
6. The statement, i.e. body of the do-while loop is executed once, even when the do-while controlling expression is initially false.

The code snippet in Program 5-19 illustrates the use of a do-while statement to find the sum of the series 1 + 2 + 3… *n* terms.

Line	Prog 5-19.c	Flow chart depicting the flow of control in program	Output window

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
	//Use of do-while statement to find
	//the sum of the series 1+2+3.....n terms
	#include<stdio.h>
	main()
	{
	int terms, sum=0, lc=0;
	printf("Enter number of terms\t");
	scanf("%d",&terms);
	do
	{
	sum=sum+lc;
	lc=lc+1;
	}
	while(lc<=terms);
	printf("Sum of series is %d",sum);
	}
		 Start
		sum=0, lc=0
		Input num. of terms
		sum=sum+lc
		lc=lc+1
		No
		Is (lc<=terms)
		Yes
		Print sum
		Stop
			Enter number of terms 5
			Sum of series is 15
			Remarks:
			• Initialize the value of variable sum to 0 else the result will be garbage
			• Look at the position of controlling expression
			• It is present at the end (i.e. exit point) of the loop
			• That is why, do-while is known as **exit-controlled loop**
			• for and while are known as **entry-controlled loops**

Program 5-19 | A program to illustrate the use of a do-while statement

5.5.5 Sentinel-Controlled Loops

In **sentinel-controlled looping**, the number of times the iteration is to be performed is not known beforehand. The execution or termination of the loop depends upon a special value called the **sentinel value**. If the sentinel value is true, the loop body will be executed, otherwise it will not. Since the number of times a loop will iterate is not known in advance, this type of loop is also known as **indefinite repetition loop**.

Consider the problem of finding the maximum and the minimum from a set of numbers. However, the set (i.e. numbers) and the cardinality of set (i.e. how many numbers are there in the set) are not known beforehand; therefore, the user will enter them at the run time. The mentioned problem can be solved by using sentinel-controlled looping as given in Programs 5-20 and 5-21.

Line	Prog 5-20a.c	Prog 5-20b.c	Output window
1	//while statement for sentinel control	//for statement for sentinel control	Enter number 5
2	#include<stdio.h>	#include<stdio.h>	Want to enter more y
3	#include<conio.h>	#include<conio.h>	Enter number 3
4	main()	main()	Want to enter more y
5	{	{	Enter number 8
6	char choice;	char choice;	Want to enter more y
7	int num, max, min;	int num, max, min;	Enter number -2
8	printf("Enter number\t");	printf("Enter number\t");	Want to enter more n
9	scanf("%d",&num);	scanf("%d",&num);	Maximum is 8

(Contd...)

```
max=min=num;
printf("Want to enter more\t");
choice=getche();
while(choice=='y'||choice=='Y')
{
printf("\nEnter number\t");
scanf("%d",&num);
if(num>max)
    max=num;
else
    if(num<min)
        min=num;
printf("Want to enter more\t");
choice=getche();
}
printf("\nMaximum is %d",max);
printf("\nMinimum is %d",min);
}
```

```
max=min=num;
printf("Want to enter more\t");
choice=getche();
for( ;choice=='y'||choice=='Y'; )
{
printf("\nEnter number\t");
scanf("%d",&num);
if(num>max)
    max=num;
else
    if(num<min)
        min=num;
printf("Want to enter more\t");
choice=getche();
}
printf("\nMaximum is %d",max);
printf("\nMinimum is %d",min);
}
```

```
Minimum is -2
```

Remarks:

- The loop terminates when the user does not enter the choice 'Y' or 'y'
- choice is the **sentinel value**
- The number of iterations after which the user will say 'no' is not known in advance
- The header file conio.h is to be included for using the function getche
- The function getche is used to get a character from the user. It also outputs, i.e. echoes the entered character onto the screen. The character e in getche stands for echo
- The variant of getche function that is used to get a character from the user without echoing it on the screen is getch function

Program 5-20 | A program to illustrate the use of while statement and for statement for sentinel-controlled looping

Line | **Prog 5-21.c** | **Output window**

```
//do-while statement for sentinel controlled looping
#include<stdio.h>
#include<conio.h>
main()
{
char choice;
int num, iteration=1, max, min;
do
{
printf("Enter number\t");
scanf("%d",&num);
if(iteration++==1)
    max=min=num;
else
    if(num>max)
        max=num;
    else
        if(num<min)
            min=num;
```

```
Enter number     5
Want to enter more     y
Enter number     3
Want to enter more     y
Enter number     8
Want to enter more     y
Enter number     -2
Want to enter more     n

Maximum is 8
Minimum is -2
```

Remarks:

- The loop terminates when the user does not enter the choice 'Y' or 'y'
- choice is the sentinel value
- The number of iterations after which the user will say 'no' is not known in advance

(*Contd...*)

Prog 5-21.c

```
printf("Want to enter more\t");
choice=getche();
printf("\n");
}
while(choice=='y'||choice=='Y');
printf("\nMaximum is %d",max);
printf("\nMinimum is %d",min);
}
```

Output window

Program 5-21 | A program to illustrate the use of a do-while statement for sentinel-controlled looping

5.5.6 Nested Loops

If the body of a loop is, or contains another iteration statement, then we say that the **loops are nested**. An example of a nested for loop is given in Program 5-22.

Prog 5-22.c

```
//Nested for loop
#include<stdio.h>
main()
{
int olc,ilc;
for(olc=1;olc<=4;olc++)
{
    for(ilc=1;ilc<=4;ilc++)
        printf("*");
    printf("\n");
}
}
```

Output window

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

Remarks:

- olc is the outer loop counter and ilc is the inner loop counter
- The inner loop is responsible for printing 4 stars in a row
- The outer loop is responsible for printing 4 such rows

Program 5-22 | A program to illustrate the use of a nested for loop

5.5.7 Semantics of break and continue Statements

After the discussion of iteration statements, it is time to discuss the use of break and continue statements. The break statement helps in terminating a loop, while the continue statement helps in terminating the current iteration of a loop.

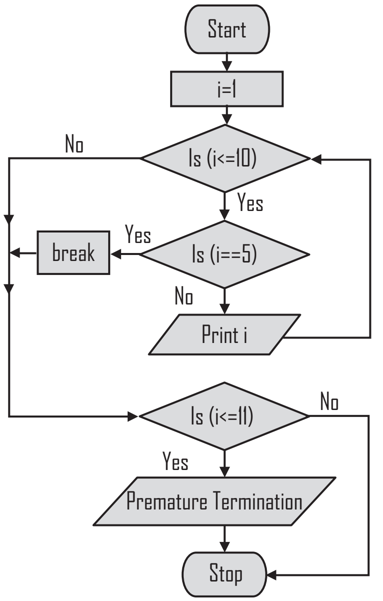
5.5.7.1 Semantics of break Statement

The important points about the usage of a break statement along with loops are as follows:

1. When the break statement present inside a loop is executed, it terminates the loop and the program control is transferred to the statement present next to the loop.
2. When the break statement present inside a nested loop is executed, it only terminates the execution of the nearest enclosing loop. The execution resumes with the statement present next to the terminated loop.
3. There is no constraint about the number of break statements that can be present inside a loop.

The meaning of the above-mentioned points is illustrated in Program 5-23.

Line	Prog 5-23.c	Flow chart depicting the flow of control in program	Output window

1
2
3
4
5
6
7
8
9
10
11
12
13
14
	//Use of break statement
	#include<stdio.h>
	main()
	{
	int i;
	for(i=1;i<=10;i++)
	{
	if(i==5)
	    break;
	printf(“%d ”,i);
	}
	if(i<11)
	    printf(“\nPremature Termination”);
	}
		 Start
		i=1
		No
		Is (i<=10)
		Yes
		Yes
		break
		Is (i==5)
		No
		Print i
		Is (i<=11)
		No
		Yes
		Premature Termination
		Stop
			1 2 3 4
			Premature Termination
			Remark:
			• break statement is used to prematurely terminate the loop

Program 5-23 \| A program to illustrate the use of break statement

Program 5-24 illustrates a break statement, which terminates the nearest enclosing loop.

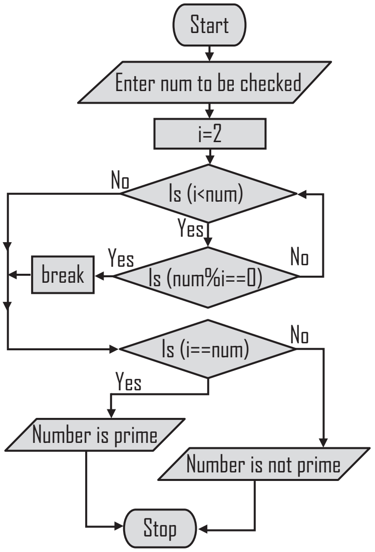
Line	Prog 5-24.c	Values of olc and ilc	Output window

1
2
3
4
5
6
7
8
9
10
11
12
13
	//Use of break statement in nested loops
	#include<stdio.h>
	main()
	{
	int olc, ilc;
	for(olc=1;olc<=3;olc++)
	    for(ilc=1;ilc<=4;ilc++)
	    {
	    if(ilc==3)
	        break;
	    printf(“%d %d\n”,olc,ilc);
	    }
	}
		olc=1
		    ilc=1 //←prints 1 1
		    ilc=2 //←prints 1 2
		    ilc=3 //←break executes & terminates the inner loop
		olc=2
		    ilc=1 //←prints 2 1
		    ilc=2 //←prints 2 2
		    ilc=3 //←break executes
		olc=3
		    ilc=1 //←prints 3 1
		    ilc=2 //←prints 3 2
		    ilc=3 //←break executes
			1 1
			1 2
			2 1
			2 2
			3 1
			3 2
			Remarks:
			• break statement terminates only the inner loop
			• The control still remains inside the outer loop

Program 5-24 \| A program to illustrate that break statement terminates the nearest enclosing loop

The use of a break statement in checking whether a number is prime or not is illustrated in Program 5-25.

Line	Prog 5-25.c	Flow chart depicting the flow of control	Output window

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
	//Use of break statement to check
	//whether a number is prime or not
	#include<stdio.h>
	main()
	{
	int num, i;
	printf("Enter the number\t");
	scanf("%d",&num);
	for(i=2;i<num;i++)
	    if(num%i==0)
	        break;
	if(i==num)
	    printf("Number is prime");
	else
	    printf("Number is not prime");
	}
		 Start
		Enter num to be checked
		i=2
		No
		Is (i<num)
		Yes
		Yes
		No
		break
		Is (num%i==0)
		No
		Is (i==num)
		Yes
		Number is prime
		Number is not prime
		Stop
			Enter the number 9
			Number is not prime
			Remarks:
			• Whether a number is prime or not can be determined by checking whether the number is divisible by any value from 2 to num-1
			• When it is found that the number num is divisible by some value of i, there is no need to check the divisibility of num for rest of the values of i

Program 5-25 \| A program to check whether a number is prime or not

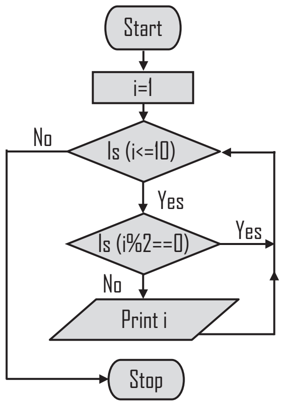
5.5.7.2 Semantics of continue Statement

The important points about a continue statement are as follows:

1. A continue statement terminates the current iteration of the loop.
2. When a continue statement present inside a nested loop is executed, it only terminates the current iteration of the nearest enclosing loop.
3. On the execution of a continue statement, the program control is immediately transferred to the header of the loop.
4. There is no constraint about the number of continue statements that can be present inside a loop.

The semantics of a continue statement is illustrated in Program 5-26.

Line	Prog 5-26.c	Flow chart depicting the flow of control	Output window

1
2
3
4
5
6
7
8
9
10
11
12
	//Use of continue statement
	#include<stdio.h>
	main()
	{
	int i;
	for(i=1;i<=10;i++)
	{
	if(i%2==0)
	    continue;
	printf("%d ",i);
	}
	}
		 Start
		i=1
		No
		Is (i<=10)
		Yes
		Is (i%2==0)
		Yes
		No
		Print i
		Stop
			1 3 5 7 9
			Remark:
			• For even values of i, the printf statement will not be executed as the continue statement transfers the control to the header of the loop

Program 5-26 \| A program to illustrate the use of a continue statement

5.6 Summary

1. Statement is the smallest logical entity that can independently exist in a C program.
2. No entity smaller than a statement, i.e. expressions, variables, constants, etc. can exist in a C program unless and until they are converted into statements.
3. A single statement is known as a simple statement.
4. A group of single statements can be clubbed together into one statement by enclosing them within braces. A clubbed statement is known as a block or a compound statement.
5. Non-executable statements are meant for the compiler. No machine code is generated for non-executable statements.
6. Only executable statements play a role during the execution of a program. Only for these statements, the machine code is generated.
7. A null statement performs no operation and consists of just a semicolon.
8. An expression statement performs the computation and is formed by terminating an expression with a semicolon.
9. By default, the flow of program control is sequential and it flows from top to bottom.
10. Flow of control needs to be altered to implement decision making and iteration.
11. To alter the default flow of control, flow control statements are used.
12. To implement decision making, selection statements are used.
13. Selection statements are: if statement, if-else statement, and switch statement.
14. Selection statements can be nested.
15. Careless use of nested if-else statement may lead to dangling else problem.
16. Dangling else problem can be implicitly as well as explicitly solved.
17. A switch statement is a better alternative to a nested if-else statement and is used in the complex decision making.
18. Looping can be performed by using iteration statements.
19. Three iteration statements available in C are: for statement, while statement and do-while statement.
20. A break statement is used to terminate the nearest enclosing loop.
21. A continue statement is used to terminate the current iteration of the nearest enclosing loop.

Exercise Questions

Conceptual Questions and Answers

1. *What is the smallest logical unit that can independently exist in a C program?*

   Statement is the smallest logical unit that has an independent existence in a C program. No entity smaller than a statement (i.e. expressions, variables and constants, etc.) can exist unless and until they are converted into statements. Consider the following program segment:

```
main()
{
    int a=10,b=20,c;
    c=a+b                          //← Error: Statement missing ; in function main()
    printf("The value of c is %d",c);
}
```

On compilation, the above-mentioned piece of code gives 'Statement missing ; error'. This error is due to the fact that `c=a+b` is an expression and not a statement. Expressions cannot independently exist in a C program. To make them exist, they must be converted into statements by terminating them with a semicolon. The following is the rectified code:

```
main()
{
    int a=10,b=20,c;
    c=a+b;                         //←Expression terminated with a semicolon forming a statement
    printf("The value of c is %d",c);
}
```

2. *What is meant by a simple statement and a compound statement?*

   A simple statement consists of a single statement. For example, `c=a+b;` is a simple statement. A compound statement consists of a sequence of simple statements enclosed within braces. The following is an example of a compound statement:

```
{
    c=a+b;
    a*=2;
    b+3;
}
```

3. *What are executable statements and non-executable statements?*

   Executable statements are the statements that call for a processing action by the computer, such as performing arithmetic, reading data, making decision and so on. Non-executable statements are the statements that provide the information about the nature of data (e.g. declaration statement). Non-executable statements can be placed outside the bodies of functions (i.e. in global scope), but executable statements can only be placed within the body of some function (i.e. local scope).

4. *Write a simple C statement to accomplish the following tasks:*

   a. *Assign sum of* `x` *and* `y` *to* `z` *and increment the value of* `x` *by 1 after the calculation.*
   b. *Decrement the variable* `x` *by 1 then subtract it from the variable* `total`.

   a. `z=x++ + y;`
   Note: Writing `z=x++ + y` is not valid as it is not a statement. It is an expression.
   b. `total-=--x;`
   or
   `total=total- --x;`
   Carefully note the position of white-space character. Writing `total=total---x;` or `total=total-- -x;` is not the same as writing `total=total- --x;`. The difference between them is shown in the table given below:

Column I	Initial values		After execution of statement in Column I	
	total	x	total	x
total-=--x;	15	5	11	4
total=total- --x;	15	5	11	4
total=total-- -x;	15	5	9	5
total=total---x;	15	5	9	5

5. *What is the difference between initialization and assignment?*

   First time assignment at the time of definition is called initialization. Assigning a value to an identifier after initialization will be treated as an assignment. The clear understanding of difference between terms initialization and assignment becomes important when we talk about qualified constants. Consider the following piece of code:

```
main()
{
    const int a=20;    //← Initialization of a qualified constant is valid.
    a=30;              //← Compilation error: Value cannot be assigned to a qualified constant.
}
```

   The above-mentioned code highlights the fact that:
   'We cannot assign a value to a qualified constant but we can initialize it'.

6. *What is null statement and where is it used?*
   A null statement is used when the syntax of a language construct requires a statement to be present but the logic of the program does not require it. Its use is illustrated in the next answer.

7. *How can you print "*Hello World*" without using a semicolon in a C program?*

   The following code segment prints "Hello World" without using a semicolon.

```
main()
{
    if(printf("Hello World"))
    {}       //←Null statement. Syntax of if statement requires a statement to be present but
}          //     the logic of the program does not require it. Hence, null statement is placed.
```

8. *What is dangling* else *problem? How is it solved by a compiler and how can it be avoided*?

9. *Why does the following piece of code on compilation gives an error?*

```
main()
{
int a=1;
if(a==1)
    printf("This is if body\n");
    printf("This statement does not belong to if body");
else
    printf("This is else body");
}
```

The given piece of code on compilation gives 'Misplaced else error.' The source of error can be found by looking at the syntax of an if-else statement. The general form of an if-else statement is:

```
if(expression)        //← if header
statement1            //← if body
else                  //← else clause
statement2            //← else body
```

It should be noted that only one statement can be associated with if clause and else clause. If more than one statement needs to be associated with if clause or else clause, then a block comprising those simple statements must be created. This block of statements, although comprising more than one simple statement, will be treated as a unit, as one statement and can be associated with if clause or else clause. The given piece of code is interpreted as:

```
main()
{
    int a=1;
    if(a==1)
        printf("This is if body\n");                  //←Only this statement is associated with if clause
    printf("This statement does not belong to if body");          //←This statement is not in if body
    else           //←else clause is left without any matching if clause and this leads to error
        printf("This is else body");
}
```

To remove this error, club both the simple statements into a compound statement. The rectified code is as follows:

```
main()
{
    int a=1;
    if(a==1)
    {                         //←Compound statement: It will be treated as a unit
        printf("This is if body\n");
        printf("This statement does not belong to if body");
    }
    else                      //←Now the else clause is properly matched with if clause
        printf("This is else body");
}
```

10. *Can the selection expression of a* switch *statement be a string?*

No, the selection expression of a switch statement cannot be a string. The switch selection expression and case labels must be of integral type. Hence, the switch statement can be used to switch only on integral data types (i.e. character and integer). Consider the following program segment:

```
main()
{
    switch("Hello")
    {
    case "Hello":
        printf("Hello");
    case "Hi":
        printf("Hi");
    }
}
```

In the above-mentioned piece of code, switch selection expression and case labels (shown in bold) are strings. This is not allowed and thus, the code on compilation gives an error.

11. *Can a* switch *statement have more than one* default *label?*

    No, a switch statement cannot have more than one default label. In a switch statement, all the case labels must be unique and at most one default label can be present. The presence of more than one default label or duplicate case labels leads to ambiguity, which results in a compilation error.

12. *Why does the following piece of code on compilation gives an error?*

```
main()
{
int i=65;
switch(i)
{
case 65:
    printf("This statement should get executed\n");
    break;
case 'A':
    printf("A has ASCII code of 65, this statement should get executed\n");
    break;
default:
    printf("Duplicate case labels lead to error\n");
}
}
```

The mentioned piece of code on compilation gives 'Duplicate case in function main' error. This is due to the fact that integers and characters are not treated separately in C language. Characters are stored internally in terms of their ASCII values. Character 'A' has ASCII value 65. So, writing case 'A': is equivalent to writing case 65:. However, case label 65 is already present. Duplicate case labels are not allowed. Hence, this leads to 'Duplicate case in function main' error.

13. *Can we use a* continue *statement within the body of a* switch *statement like we can use a* break *statement within it?*

    No, a continue statement can appear only in or as a loop body. A switch statement is a branching statement and not a looping statement. Hence, the continue statement cannot appear inside the body of a switch statement.

14. *Is it mandatory to have* case *labeled or* default *labeled statements within a* switch *body? If the* switch *body does not contain any* case *or* default *labeled statements, will there be a compilation error?*

    The general form of a switch statement is:

```
switch(expression)        //←switch header
statement                 //←switch body
```

A switch body consists of a statement. This statement can be a null statement, an expression statement, a labeled statement, a flow control statement, a compound statement, etc. There is no constraint that only labeled statements can form the switch body. Hence, it is not mandatory to have case labeled or default labeled statements within the switch body. The following usages of switch statement (without any case labeled or default labeled statements) are valid:

a.
```
switch(expr);                 //←switch body is a null statement
```

b.
```
switch(expr)                  //←switch body consisting of two function call statements
{
    printf("Two expression statements");
    printf("This is valid");
}
```

c.
```
switch(expr)                  //←switch body has a labeled statement, but the labeled
{                             //   statement is an identifier labeled statement and not a case labeled
lab:                          //   or a default labeled statement
printf("This is also valid");
goto lab;
}
```

15. *Can* case *labeled or* default *labeled statement exist outside the* switch *body?*

No, case labeled statements and default labeled statements can appear only inside the switch body. Placing case labeled statements or default labeled statements outside the switch body leads to 'Case/ Default outside of switch' compilation error.

16. *Why does the following piece of code gives an error on compilation?*

```
main()
{
int exp=2;
switch(exp)
{
case 1:
    int j=2;
    printf("The value of j in case 1 is %d\n",j);
case 2:
    printf("The value of j in case 2 is %d\n",j);
}
}
```

This compilation error is due to the fact that the placement of the definition statement associated with a case or default label is illegal unless it is placed within a statement block. The placement of the definition statement within a statement block is mandatory because if the definition is not enclosed within a statement block, the defined identifier would be visible (i.e. can be used) across the case labels, but is initialized only if the case label within which it is defined is executed. The presence of the statement block ensures that the name is initialized whenever it is visible. In the given piece of code, int j=2; is not placed within a statement block. Hence, there is a compilation error. The compilation error can be removed by placing int j=2; within a statement block.

17. *I have tried to rectify the problem in the code mentioned in the previous question. Does the following piece of code compile successfully?*

```
main()
{
int exp=2;
switch(exp)
{
case 1:
{
    int j=2;
    printf("The value of j in case 1 is %d\n",j);
}
case 2:
    printf("The value of j in case 2 is %d\n",j);
}
}
```

No, the given piece of code does not yet compile successfully. The given piece of code on compilation gives 'Undefined symbol 'j' in function main' error. This error is due to the fact that the identifier j defined within the statement block of case label 1 is visible (i.e. can be used) only inside it. The identifier j is not visible (i.e. does not exist) outside the statement block in which it is defined. Hence, reference to j in the printf statement of case label 2 is not valid and leads to the compilation error.

18. *Why does the following piece of code show just a sequence of zeros in its output?*

```
main()
{
    int number=2;
    while(1)
    {
        printf("%d ",number);
        number*=2;
    }
}
```

The code actually outputs

```
2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 -32768 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...infinite times.
```

Initially number is two. It is represented in memory as:

Sign Bit 16 MSB	Magnitude													
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Multiplying by two makes it four (i.e. equivalent to shifting in left direction by 1 bit).

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

This shifting is continued and after 14 iterations, the number becomes:

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

i.e. -32768. If the shifting is further carried out, the number becomes zero.

Sign Bit 16 MSB	Magnitude														
	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

From this point onwards irrespective of how many times shifting is carried out (i.e. number is multiplied by two), the number remains zero. Hence, from this point onwards, the output will only have zeros.

Now, the speed of processing is so fast that first few outputs will be skipped (cannot be seen in the output as the screen scrolls) and only a sequence of zeros can be seen. If you want to see all the outputs, put some delay mechanism inside the loop. This can be done by using either the function getch(), delay(int)✍ or sleep(int). ✍

✍ The function **delay(int)** suspends the execution of the program for a given time interval. The time interval is an integer value and specifies the time in milliseconds. **sleep(int)** is a function equivalent to the delay function. The delay function is provided in the DOS environment and the sleep function is usually available with the WINDOWS environment.

19. *I want to test whether a character entered by the user lies in the range 'A' to 'C' or 'X' to 'Z'. Can I use a* switch *statement to do this?*

Yes, a switch statement can be used to accomplish it. Use the following piece of code to check whether the character entered lies in the range ‘A' to ‘C' or ‘X' to ‘Z'.

```
main()
{
    char ch;
    printf(“Enter a character\t”);
    scanf(“%c”,&ch);
    switch(ch)
    {
        case ‘A’:
        case ‘B’:
        case ‘C’:
            printf(“The entered character is in range A-C”);
            break;
        case ‘X’:
        case ‘Y’:
        case ‘Z’:
            printf(“The entered character is in range X-Z”);
            break;
        default:
            printf(“The entered character is neither in range A-C nor in range X-Z”);
    }
}
```

However, this method would not be practical if the ranges are bigger. In case of bigger ranges, usage of if-else statements with the involvement of logical operators in the controlling expressions is preferred.

20. *Do labels have scope like variables?*

    Yes, labels do have scope like variables. A label name is a type of identifier that has only function scope. It can be used anywhere in the function in which it appears.

21. *Can we use a* goto *statement to take control from one function to some other function?*

    A goto statement in no way can transfer control from one function to another function. Consider the following piece of code:

```
main()
{
    goto here;
}
other_function()
{
    here:
        printf("The label is in other function");
}
```

The above-mentioned code on compilation gives 'Undefined label here in function main' error. To remove this error, use label here somewhere inside the body of main function.

22. *Can a label be followed by another label or should it be followed by a statement?*

    The general forms of labeled statements are:

    1. identifier: statement
    2. case constant-expression: statement
    3. default: statement

    After identifier label, case label or default label, there should be a statement. This statement can itself be another labeled statement. Hence, label can follow another label. For e.g.

```
lab:            //←label followed by another labeled statement
    try:
        printf("This is valid");
```

Due to this definition of a labeled statement, the following form of a switch statement is valid:

```
switch(expr)
{
    case 1:
    case 2:
        printf("Case 1 and Case 2");
    case 3: case 4: case 5:
        printf("Case 3,4 and 5");
}
```

23. *Can a label name be the same as a function name or a variable name?*

    Yes, label name can be the same as a function name or a variable name. Consider the following piece of code:

```
main()
{
    int i=1;
    main:
        printf("Function name is used as label name\n");
    i++;
    if(i==2)
        goto i;
    goto main;
    i:
        printf("Variable name is used as label name\n");
}
```

In the given code, the function name, i.e. main and the variable name, i.e. i are used as label names. The given code on execution gives an output as:

```
Function name is used as label name
Variable name is used as label name
```

24. *Can a reserved word or a keyword like* while, if, *etc. be used as a label name?*

    Reserved words or keywords cannot form valid identifier names. Since label names are identifiers, reserved words or keywords cannot be used as valid label names.

25. *All the identifiers need to be declared before their use. Label names are also identifiers. So, do we need to declare label names?*

    No, label names need not be declared. Label names are identifiers but no type is associated with them. Hence, there is no need to explicitly declare them. Label names are implicitly declared by their syntactic appearance. An identifier followed by a colon and a statement is implicitly treated as a label name.

26. *Is a* goto *statement capable of taking the control in or out of a nested loop?*

    Yes, a goto statement is capable of taking the control in or out of a nested loop. The goto statement is capable of taking control anywhere within a function in which it is used. Consider the following piece of code:

```
main()
{
    int i,j;
    for(i=1;i<5;i++)
        for(j=1;j<5;j++)
        {
            printf("This statement will be executed only once\n");
            goto label;
        }
    label:
        printf("goto statement has taken the control out of nested loop");
}
```

Upon execution, it gives the output as:

```
This statement will be executed only once
goto statement has taken the control out of nested loop
```

27. *Can a single* break *statement be used to terminate a nested loop?*

    No, a single break statement cannot be used to terminate a nested loop. A break statement can only terminate the execution of the nearest enclosing switch or the nearest enclosing loop. Consider the following piece of code:

```
main()
{
    int i,j;
    for(i=1;i<3;i++)
    {
        for(j=1;j<3;j++)
        {
            break;
            printf("This will not get printed");
        }
        printf("This will be executed twice as it is inside outer loop\n");
    }
}
```

The break statement only terminates the inner for loop. The printf statement in the outer for loop executes normally. The above piece of code on execution outputs:

```
This will be executed twice as it is inside outer loop
This will be executed twice as it is inside outer loop
```

28. *What are entry-controlled and exit-controlled loops?*

    In **entry-controlled loops**, condition is checked before the execution of body of the loop. The for loop and while loop are examples of entry-controlled loops. In **exit-controlled loops**, the condition is checked after the execution of body of the loop. do-while is an example of an exit-controlled loop. In entry-controlled loops, if the condition is initially false, the body of the loop will not be executed. However, in exit-controlled loops, even if the condition is initially false, the body of the loop will be executed once. Consider the following piece of code:

```
main()
{
    int i=2;
    do
        printf("Condition is false, but this will be printed");
    while(i<1);
}
```

    The condition of a do-while loop is initially false; even then "Condition is false, but this will be printed" is the output. This indicates that the body of the exit-controlled loop gets executed once, even if the condition of the loop is initially false.

29. *What are counter-controlled and sentinel-controlled loops?*

    Counter-controlled looping is a form of looping in which the number of times the loop will execute is known in advance. The counter-controlled loop starts with the initial value of the loop counter and terminates when the final value of the loop counter is reached. Since a counter-controlled loop iterates a fixed number of times, it is also known as a definite repetition loop. In sentinel-controlled looping, the number of times the loop will execute is not known beforehand. The execution or termination of the loop depends upon a special value called the sentinel value. If the sentinel value is true, the loop body gets executed else not. Since the number of times the loop will iterate is not known in advance, this type of loop is also known as an indefinite repetition loop.

30. *What are the three main ingredients of counter-controlled looping?*

    Three main ingredients of counter-controlled looping are:

1. Initialization of the loop counter.
2. A condition determining whether the loop body should be executed or not.
3. An expression that manipulates the value of the loop counter so that the condition in Step 2 eventually becomes false and the loop terminates.

31. *For every usage of a* for *loop, we can write an equivalent* while *loop. So, when should one prefer to use a* for *loop and when should a* while *loop be preferred?*

A while loop should be preferred over a for loop when the number of iterations to be performed is not known in advance. The termination of the while loop is based on the occurrence of some particular condition, i.e. a specific sentinel value. The usage of a for loop should be preferred when the number of iterations to be performed is known beforehand. In short, a while loop is preferred for sentinel-controlled looping and a for loop is preferred for counter-controlled looping.

32. *Why does the following piece of code on compilation give a compilation error?*

```
main()
{
int i=2;
while(i<10);
{
    printf(“The value of i is %d”,i);
    if(i==5)
        break;
}
}
```

The given piece of code gives a compilation error due to the fact that the break statement can appear only in or as a switch body or a loop body. Here, the break statement does not appear inside the body of the while loop. The body of the while loop consists of a null statement. To rectify the given code, remove the semicolon present at the end of the while header.

33. *I want to terminate the nearest enclosing loop. Which construct in C provides me this functionality?*

To terminate the nearest enclosing loop, a break statement can be used. This can be seen by executing the following piece of code:

```
main()
{
    int i,j;
    for(i=0;i<2;i++)
    {
        for(j=0;j<5;j++)
        {
            if(i!=0 || j!=0)
                break;
            printf(“This will be printed only once\n”);
        }
    printf(“This will be printed two times\n”);
    }
}
```

34. *I want to terminate the current iteration of the nearest enclosing loop. Which construct in C provides me this functionality?*

To terminate the current iteration of the nearest enclosing loop, a continue statement can be used. This can be seen by executing the following piece of code:

```
main()
{
    int i,j;
    for(i=0;i<2;i++)
    {
        for(j=0;j<5;j++)
        {
            if(i!=0 || j!=0)
                continue;
            printf("This will be printed only once\n");
        }
    printf("This will be printed two times\n");
    }
}
```

35. *The syntactic form of a* for *loop is as follows:*

    $\text{for(expression}_1\text{;expression}_2\text{;expression}_3\text{)}$
    statement

    What is the order in which expression_1, expression_2, expression_3 and statement get evaluated?

    The order in which the expressions are evaluated is:

    1. expression_1 is evaluated before the first evaluation of the controlling expression expression_2. expression_1 is evaluated only once.
    2. expression_2 is the controlling expression and is evaluated every time before the execution of the loop body. If expression_2 evaluates to true, the loop (i.e. statement) body will be executed otherwise the control will come out of the loop.
    3. expression_3 is evaluated after the execution of the loop body.

Code Snippets

Determine the output of the following code snippets. Assume that the inclusion of the required header files has been made and there is no prototyping error due to them.

36.
```
int a=10,b=20,c;
c=a+b;
main()
{
    printf("Value of c is %d",c);
}
```

37.
```
main()
{
    int a=10,b=20,c;
    c=a+2*b
    printf("The value of c is %d",c);
}
```

38.
```
main()
  {
      int a=10,b=20;
      if(a==b)
          printf("a=10,b=20");
```

```
              printf("a and b are not equal");
      }
```

```
39. main()
      {
          int a=10,b=20;
          if(a==b)
          {
              printf("a=10, b=20");
              printf("a and b are not equal");
          }
      }
```

```
40. main()
      {
          int a=10,b=20;
          if(a=b)
              printf("a and b are equal");
          else
              printf("a and b are not equal");
      }
```

```
41. main()
    {
        int a=10,b=20;
        if(a==b);
            printf("a and b are equal");
        else
            printf("a and b are not equal");
    }
```

```
42. main()
    {
        int a=10,b=10;
        if(a==b)
            printf("a and b are equal\n");
        else;
            printf("a and b are not equal\n");
    }
```

```
43. main()
    {
        if(1)
            printf("This will always get executed");
        else
            printf("This will never get executed");
    }
```

```
44. main()
      {
          if(printf("Hello"))
              printf("Students");
      }
```

45.
```
main()
{
int a=10,b=20;
if(a==10)
if(b==10)
printf("Value of a and b is 10");
else
printf("Value of a is 10 and b is something else");
}
```

46.
```
main()
{
    int a=10,b=20;
    if(a==10)
    {
    if(b==10)
        printf("Value of a and b is 10");
    }
    else
        printf("Value of a is 10 and b is something else");
}
```

47.
```
main()
{
    int expr=10;
    switch(expr)
        printf("This is valid but will not get executed");
}
```

48.
```
main()
{
    int expr=10;
    switch(expr);
        printf("Tell whether this will get executed or not");
}
```

49.
```
main()
{
    float expr=2.0;
    switch(expr)
    {
        case 1: printf("One");
        case 2: printf("Two");
        default: printf("Default");
    }
}
```

50.
```
main()
{
    int expr=2,j=1;
    switch(expr)
    {
    case j:
            printf(“This is case 1”);
        case 2:
            printf(“This is case 2”);
        default:
            printf(“This is default case”);
        }
    }
```

```
51. main()
    {
        char ch=’A’;
        switch(ch)
        {
            case ‘A’:
                printf(“Case label is A”);
            case “B”:
                printf(“Case label is B”);
        }
    }
```

```
52. main()
    {
        int expr=1;
        switch(expr)
        {
            case 1: printf(“One\n”);
            case 2: printf(“Two\n”);
            default: printf(“Three\n”);
        }
    }
```

```
53. main()
    {
        int expr=1;
        switch(expr)
        {
            case 1:
                printf(“One\n”);
                break;
            case 2:
                 printf(“Two\n”);
                break;
            default: printf(“Three\n”);
        }
    }
```

```
54. main()
    {
        int expr=3;
        switch(expr)
        {
            default: printf(“Three\n”);
            case 1: printf(“One\n”);
            case 2: printf(“Two\n”);
        }
    }
```

```
55. main()
    {
        int expr=2;
        switch(expr)
        {
            case 1:
                printf(“This is case 1”);
            case 2-1:
                printf(“This is case 2”);
        }
    }
```

```
56. main()
    {
        int i=1,j=3;
        switch(i)
        {
            case 1:
                printf(“This is outer case 1\n”);
                switch(j)
                {
                    case 3:
                        printf(“This is inner case 1\n”);
                        break;
                    default:
                        printf(“This is inner default case”);
                }
            case 2:
                printf(“This is outer case 2”);
        }
    }
```

```
57. main()
    {
        int expr=2;
        switch(expr)
        {
            case 1:
                printf(“This is case 1”);
                break;
            case 2:
                printf(“This is case 2”);
                continue;
            default:
                printf(“Default”);
        }
    }
```

58.

```
    main()
    {
        default:
            printf(“This is default labeled statement”);
        goto default;
    }
```

59.

```
    main()
    {
        int i=1;
        while(i<=5)
        {
            printf(“%d ”,i);
            i=i+1;
        }
        printf(“\nThe value of i after the loop is %d”,i);
    }
```

60.

```
    main()
    {
        int i=1;
        while(i<=5);
        {
            printf(“%d\n ”,i);
            i=i+1;
        }
        printf(“The value of i after the loop is %d”,i);
    }
```

61.

```
    main()
    {
        int i=1;
        while(i<=5)
            printf(“%d ”,i);
        printf(“The value of i after loop is %d”,i);
    }
```

62.

```
    main()
    {
        int i=1;
        for(     )
        {
            printf(“%d”,i);
            if(i=5)
                break;
        }
    }
```

```
63. main()
    {
        int i;
        for(i=1;i<=32767;i++)
            printf(“%d ”,i);
    }
```

```
64. main()
    {
        int i=1;
        for(;;)
        {
            printf(“%d ”,i);
            if(i==5)
                break;
        }
    }
```

```
65. main()
    {
        int i=1;
        for(;;)
        {
            printf(“%d”,i);
            if(i=5)
                break;
        }
    }
```

```
66. main()
    {
        int i=1;
        for(;i<=5;printf(“%d ”,i++));
    }
```

```
67. main()
    {
        int i=1;
        for(;i<=10;i++)
        {
            if(i%2==0)
                continue;
            printf(“%d ”,i);
        }
    }
```

```
68. main()
    {
        int i=1;
        loop:
          printf(“%d ”,i++);
       if(i==5) break;
       goto loop;
    }
```

69.
```
main()
{
       int i=1;
       loop:
          printf(“%d ”,i++);
       if(i==5) goto out;
       goto loop;
       out:
       ;
  }
```

70.
```
main()
{
     int i,j;
     for(i=1;i<3;i++)
          for(j=1;j<4;j++)
          {
               if(j==2) break;
               printf(“%d %d\n”,i,j);
          }
}
```

71.
```
main()
{
     int i,j;
     for(i=1;i<3;i++)
          for(j=1;j<4;j++)
          {
               if(j==2) continue;
               printf(“%d %d\n”,i,j);
          }
}
```

72.
```
main()
{
     int i=3;
     for(;i++=0;)
          printf(“%d”,i);
}
```

73.
```
main()
{
     int a=0, b=20;
     char x=1, y=10;
     if(y,x,b,a)
          printf(“hello”);
}
```

```
74. main()
    {
        int i=0;
        for(;i++;)
            printf(“%d”,i);
    }
```

```
75. main()
    {
        int i=0;
        for(;++i;)
            printf(“%d”,i);
    }
```

```
76. main()
    {
        int i=3,j=3;
        for(;i<6,j<4;i++,j++)
            printf(“%d %d\n”,i,j);
    }
```

```
77. main()
    {
        int i=1;
        while (i<=5)
        {
            printf(“%d”,i);
            if (i>2)
                goto here;
            i++;
        }
    }
    other_function()
    {
        here:
            printf(“The label is in other function”);
    }
```

```
78. main()
    {
        int i=3;
        goto label;
        for(i=0;i<5;i++)
        {
            label:
                printf(“%d ”,i);
        }
    }
```

```
79. main()
    {
        int i=5;
        do
        {
            printf("%d",i);
            i++;
        }while(i<10)
    }
```

80.
```
main()
{
    int i=5;
    do
    {
        printf("%d",i);
        i++;
    }while(i<0);
}
```

Multiple-choice Questions

81. The smallest independent logical unit in a C program is

a. Expression	c. Statement
b. Token	d. None of these

82. In C language, statements are terminated with

a. Period	c. New-line character
b. Semicolon	d. None of these

83. By default, statements in a C program are executed

a. Randomly	c. Sequentially in bottom to top order
b. Sequentially in top to bottom order	d. None of these

84. `int ival;` is actually a

a. Declaration statement	c. Neither a declaration statement nor a definition statement
b. Definition statement	d. Declaration as well as a definition statement

85. Sentinel-controlled loop is also known as

a. Definite repetition loop	c. Indefinite repetition loop
b. Infinite repetition loop	d. None of these

86. Case label inside `switch` body must be

a. An expression	c. A constant integral expression
b. An integral expression	d. An integer constant

87. Which of the following forms of `for` statement is syntactically valid

a. `for(;;);`	c. `for(;)`
b. `for(;;)`	d. `for();`

88. The selection expression of `switch` statement must be of

a. Integer type	c. Integral type
b. Float type	d. String type

89. The C construct that is used to terminate the current iteration of a loop is
    a. break statement
    b. continue statement
    c. return statement
    d. None of these

90. Dangling else is an ambiguity that arises when in a statement the number of else clauses are
    a. Equal to the number of if clauses
    b. Less than the number of if clauses
    c. Greater than the number of if clauses
    d. None of these

91. The C construct that is used to terminate a loop is
    a. break statement
    b. continue statement
    c. return statement
    d. None of these

92. Minimum number of times a do-while loop will be executed is
    a. 0
    b. 1
    c. Cannot be predicted
    d. None of these

93. Which of the following statement is true about continue statement?
    a. It terminates the loop
    b. It terminates the current iteration of the loop
    c. It can be used in or as a switch body
    d. None of these

94. The body of a switch statement must consist of
    a. Case-labeled statements
    b. Default-labeled statements
    c. A statement
    d. Null statement

95. A continue statement can only be used in or as
    a. switch body
    b. Loop body
    c. if body
    d. None of these

96. Labels have
    a. Block scope
    b. Global scope
    c. Function scope
    d. File scope

97. A goto statement cannot take control
    a. Out of nested if-else
    b. Out of a nested loop
    c. Out of a function
    d. None of these

98. Consider the following segment of C code:

```
int j,n;
j=1;
while(j<=n)
    j=j*2;
```

The number of comparison made in the execution of the loop for any n>0 is
    a. ceiling(log$_2$n)+2
    b. n
    c. ceiling(log$_2$n)+1
    d. floor(log$_2$n)+2

99. Consider the following fragment of C code in which i, j and n are integer variables.

```
for(i=n,j=0;i>0;i/=2,j+=1);
```

The value of j after the termination of for loop is
    a. floor(log$_2$n)+1
    b. n/2+1
    c. n
    d. ceiling(log$_2$n)+1

100. Consider the following fragment of C code. How many times will the following loop be executed?

```
x=500;
while(x<=500)
{
    x=x-600;
    if(x<0) break;
}
```

a. 0
b. 100
c. 500
d. 1

Outputs and Explanations to Code Snippets

36. Compilation error

Explanation:

Non-executable statements can be placed outside the body of a function but executable statements can only be placed within the body of a function. `c=a+b;` is an executable statement and cannot be placed outside the body of a `main` function. To remove this error, place the statement `c=a+b;` inside the body of the `main` function.

37. Compilation error (Statement missing ; in function main)

Explanation:

`c=a+2*b` is not a statement. It is an expression. No entity smaller than a statement can independently exist in a C program. Hence, the error. To remove this error, convert the expression `c=a+2*b` into a statement by terminating it with a semicolon.

38. a and b are not equal

Explanation:

`printf("a and b are not equal");` does not belongs to `if` body. It is a statement next to `if` statement and will always be executed irrespective of the result of evaluation of `if` controlling expression.

39. No output

Explanation:

The `if` body is a compound statement consisting of two `printf` statements. Being a compound statement, it will be treated as a unit, i.e. a single statement. Either all of its constituent statements will be executed or none will get executed depending upon the outcome of the `if` controlling expression. Here, the `if` controlling expression evaluates to false. Hence, `if` body (i.e. `printf` statements) will not be executed and thus, there is no output.

40. a and b are equal

Explanation:

The controlling expression of `if-else` statement is `a=b`. An assignment operator has been used instead of equality operator. The value of `b` is assigned to `a` and the value of expression comes out to be 20 (i.e. the assigned value of `b`). 20 is a non-zero value, i.e. true. If the `if-else` controlling expression evaluates to true, `if` body will get executed. Hence, `if` body (i.e. `printf("a and b are equal");`) gets executed and `a and b are equal` is the result.

41. Compilation error (Misplaced else in function main)

Explanation:

This error is due to the presence of a semicolon after the `if-else` controlling expression. The mentioned code will be interpreted in the following way:

```
main()
{
    int a=10,b=20;
    if(a==b)
        ;                              //←if body is a null statement
    printf("a and b are equal");       //←This statement is next to if statement
    else                               //←else clause is without any if clause
        printf("a and b are not equal");
}
```

To rectify this code, either remove the semicolon or make the null statement and printf("a and b are equal"); statement a single statement by enclosing them within braces.

42. a and b are equal
    a and b are not equal

    Explanation:

    a and b are not equal is a part of the output due to the presence of a semicolon after the else clause. Null statement forms the else body. printf("a and b are not equal"); statement is a statement next to the if-else statement and will always get executed irrespective of the result of the evaluation of if-else controlling expression.

43. This will always get executed

    Explanation:

    The controlling expression of if-else statement is 1. 1 is a non-zero value and is considered as true. Every time you run this program, This will always get executed is the output as if-else controlling expression always evaluates to true.

44. HelloStudents

    Explanation:

    The controlling expression of if statement is evaluated first. Controlling expression of if statement is printf("Hello"). Function calls are valid expressions, so writing if(printf("Hello")) will not lead to any compilation error. The expression gets evaluated and Hello is printed on the screen. The printf function also returns an integer value. The value returned by the printf function is the number of characters it prints. The number of characters in Hello is 5; hence, printf function returns 5. 5 is a non-zero value and is treated as true. As the controlling expression of if statement evaluates to true, if body gets executed and Students is printed on the screen. Hence, the output that gets printed is: HelloStudents.

45. Value of a is 10 and b is something else

    Explanation:

    The code suffers from dangling else ambiguity. The ambiguity is implicitly resolved by the compiler and the code is interpreted in the following way:

```
main()
{
    int a=10,b=20;
    if(a==10)
        if(b==10)
            printf("Value of a and b is 10");
        else
            printf("Value of a is 10 and b is something else");
}
```

The given code has an if statement whose body consists of an if-else statement. The controlling expression of if statement (i.e. a==10) evaluates to true, so its body (i.e. if-else statement) gets executed. The controlling expression of if-else statement (i.e. b==10) evaluates to false, and hence the else body, i.e. printf("Value of a is 10 and b is something else"); gets executed.

46. No output

    Explanation:

    This code does not suffer from dangling else ambiguity. There is an if-else statement whose if body consists of another if statement and else body consists of a printf statement. The controlling expression of an if-else statement (i.e. a==10) evaluates to true, hence its if body will be executed and else body will be skipped. The if statement present inside the if body of if-else statement starts execution and its controlling expression (i.e. b==10) evaluates to false. Hence, its body will not be executed and thus, nothing gets printed.

47. No output

    Explanation:

    The switch statement is executed according to the rule mentioned below:
    The switch selection expression is evaluated and the result of evaluation of the switch selection expression is compared against the value associated with each case label until either a match is successful or all labels have been examined. If the result of evaluation of the switch selection expression matches the value of a case label, the execution begins from the statement with that case label. The execution continues across case/default boundaries till the end of the switch statement. If there is no match, the execution begins from the statement with the default label if it is present; otherwise the execution of the program continues with the statement following the switch statement.
    According to the above-mentioned rule, execution can start only with the matched case labeled statement or the default labeled statement, if it is present. Since the printf statement is neither a matched case labeled statement nor a default labeled statement, it will not be executed. Hence, there will be no output.

48. Tell whether this will get executed or not

    Explanation:

    Null statement present after the switch controlling expression forms the switch body. The printf statement does not belong to switch body and is a statement present next to the switch statement. This statement will always be executed irrespective of the value of switch selection expression.

49. Compilation error

    Explanation:

    switch selection expression and case labels must be of integral type. Since in the given code switch selection expression is of float type, there will be a compilation error.

50. Compilation error

    Explanation:

    Case label must be a compile time constant integral expression. Since in the given code, variable j is used as case label, there is a violation of syntactic rule and this leads to the compilation error.

51. Compilation error

    Explanation:

    Case label must be of integral type, i.e. either integer type or character type. Usage of string as case label (i.e. case "B") is a violation of syntactic rule and leads to the compilation error.

52. One
    Two
    Three

    Explanation:

    A common misunderstanding is that only the statements associated with the matched case label are executed. Rather, execution begins there and continues across case/default boundaries until the end of switch statement is encountered.

53. One

    Explanation:

    The case label 1 gets matched with the value of switch selection expression. Execution begins from the statement with the case label 1. printf("One\n"); gets executed and One is printed on the screen. The execution of statements would have been carried out till the end of switch statement but the break statement is encountered after the printf statement. This break statement terminates the switch statement. Hence, the rest of the case labeled, default labeled and other statements do not get executed. Thus, One is the output.

54. Three
    One
    Two

    Explanation:

    There is no constraint about the position of default labeled statement within the switch body. It can be placed before the case labeled statements, in-between the case labeled statements or after the case labeled statements. Generally, it is placed after the case labeled statements but it can be placed anywhere within the switch body. In the given piece of code, default labeled statement is placed before the case labeled statements. The result of evaluation of switch selection expression is matched with the case labels. Since none of the case labels (i.e. 1 and 2) get matched with the evaluated value of the switch selection expression (i.e. 3), the execution starts from the statement with the default label and is carried out across the case boundaries till the end of the switch statement. Hence, the printf statements associated with case labels 1 and 2 also gets executed.

55. Compilation error

    Explanation:

    The case labels should be unique. Although the case labels in the given piece of code seems to be unique but they are actually the same. The constant expression 2-1 gets evaluated to 1. Since case label 1 is already present, there is 'Duplicate case in function main' error.

56. This is outer case 1
    This is inner case 1
    This is outer case 2

    Explanation:

    The body of the switch statement consists of three statements:

    1. case labeled statement-1
       case 1:

printf("This is outer case 1\n");

2. switch(j) { ...}
3. case labeled statement-2

   case 2:

   printf("This is outer case 2\n");

The execution of the statements starts from the statement with the matched case label. Since case label 1 gets matched with the value of the switch selection expression (i.e. value of i), the execution starts with printf("This is outer case 1\n");. The execution from this point is carried out till the end of the switch statement. After the execution of the printf statement, statement 2, i.e. the inner switch statement□ starts execution. The body of the inner switch also consists of three statements:

1. Case-labeled statement-1

   case 3:

   printf("This is inner case 1\n");
2. break;
3. Default-labeled statement

   default:

   printf("This is inner default case\n");

Since the value of selection expression of the inner switch (i.e. value of j) matches the case label 3, execution starts with printf("This is inner case 1\n");. Execution from this point would have been carried out till the end of the inner switch statement but after the execution of printf statement break statement is encountered. This break statement terminates the execution of the nearest enclosing switch (i.e. inner switch statement). Hence, the default labeled statement is not executed and the control is immediately transferred to the case labeled statement-2 of the outer switch statement. The statement printf("This is outer case 2\n"); gets executed.

This illustrates that there can be a switch statement within the body of another switch statement. Hence, switch statements can be nested.

57. Compilation error: "Misplaced continue in function main()"

Explanation:

A continue statement shall appear only in or as a loop body. It cannot appear in or as a switch body. In the given piece of code, continue is placed inside the switch body. This is a violation of the syntactic rule and leads to the compilation error 'Misplaced continue in function main.'

58. Compilation error

Explanation:

Remember the following syntactic rules:

1. case labeled and default labeled statements can appear only inside the switch statement.
2. case label and default label cannot be used with a goto statement. Only identifier labels can be used with the goto statement.

Since there is violation of both the above-mentioned rules, there are compilation errors:

1. 'Default outside of switch in function main' (Due to violation of rule 1)
2. 'Goto statement missing label in function main' (Due to violation of rule 2)

59. 1 2 3 4 5
    The value of i after the loop is 6

    Explanation:

    The controlling expression (i<=5) is evaluated first and comes out to be true. The body of the loop is executed. 1 gets printed and value of i becomes 2. The controlling expression is evaluated again with the value of i being 2 (i.e. 2<=5). It comes out to be true and 2 gets printed. In this way 3 4 5 gets printed. The value of i becomes 6. The controlling expression (i.e. 6<=5) becomes false and the loop terminates. The value of i when the loop terminates is 6 and gets printed by the next printf statement.

60. No output

    Caution:

    Infinite loop

    Explanation:

    The presence of a semicolon at the end of the while header makes this program to stick into an infinite loop. The controlling expression of a while statement is true and the body of the while statement gets executed. The body of the while statement is a null statement. Null statement produces no output. There is no expression in the body of the while statement that manipulates the value of the loop counter so that the controlling expression eventually evaluates to false. Due to the absence of a manipulating expression, the controlling expression of the while statement always evaluates to true and keeps on executing the null statement. Hence, there will be no output and the program will not terminate as it is trapped inside an infinite loop.

61. 1 1 1 1 1 ...infinite times

    Caution:

    Infinite loop

    Explanation:

    An expression that manipulates the value of the loop counter is missing. The controlling expression of the while statement always evaluates to true. Thus, an infinite loop.

62. Compilation error

    Explanation:

    The general form of for statement is:
    for(expression$_1$;expression$_2$;expression$_3$)
    statement
    All the expressions in for header are optional and can be skipped. Even if all the expressions are missing, it is mandatory to create three sections by placing two semicolons. In the given code, the for header does not have the required sections. Thus, it is syntactically incorrect and leads to a compilation error.

63. 1 2 3 ...32767 -32768 -32767...32767 -32768 -32767...infinite times

    Caution:

    Infinite loop

    Explanation:

    The loop counter i is initialized to 1. The condition i<=32767 (i.e. 1<=32767) evaluates to true. Hence, the loop body gets executed and 1 is printed. The expression i++ gets evaluated and the value of i becomes 2. Condition i<=32767 (i.e. 2<=32767) evaluates to true. The loop body gets executed and 2 is printed. This process is continued till the value 32767 gets printed. Now, when i++ is evaluated, the value of i does not becomes 32768 as 32768 exceeds the range of the integer data type. Instead it becomes –32768 due to the wrap around effect. Thus, the condition i<=32767 (i.e. -32768<=32767) still evaluates to true. Hence, the condition never becomes false and the loop will not terminate.

64. 1 1 1 1...infinite times

Caution:

Infinite loop

Explanation:

for(;;) is syntactically valid and semantically (i.e. logically) it is an infinite loop. Inside the body of for loop, a break statement is present and it seems to be an exit path from the loop. The break statement will only be executed if the value of i becomes 5. Since the body of the for loop contains no expression to manipulate the value of i, the value of i will never become 5 and thus the break statement will never be executed. Hence, 1 will be printed infinite number of times.

65. 1

Explanation:

The initial value of i is 1. No condition is present inside the header of the for loop. Hence, without checking any condition, the body of the loop starts execution. printf("%d",i) gets executed and the value 1 gets printed. The statement present next to the printf statement is an if statement. The controlling expression of if statement is evaluated. The if controlling expression (i.e. i=5) has an assignment operator instead of an equality operator. The value 5 is assigned to i and the if controlling expression evaluates to true. Thus, the body of if statement, i.e. break statement gets executed. The break statement terminates the for loop. Hence, 1 is the output.

66. 1 2 3 4 5

Explanation:

In the given piece of code, condition i<=5 (i.e. 1<=5) evaluates to true. Thus, the body of the for loop, i.e. a null statement gets executed. After the execution of the body, the manipulation section (i.e. printf("%d ",i++)) gets executed. It prints the current value of i (i.e. 1) and then increments the value of i to 2. Again, the condition is checked and the above process is repeated. In this way 2 3 4 5 also gets printed.

67. 1 3 5 7 9

Explanation:

For even values of i, the if controlling expression i%2==0 evaluates to true. The body of the if statement (i.e. continue statement) gets executed. The continue statement on execution, immediately transfers the control to the header of the loop and the rest of the statements in the body of the loop will not be executed for the current iteration. Thus, for the even values of i, printf statement will not be executed.

68. Compilation error

Explanation:

break statement shall appear only in or as a switch body or a loop body. Logically, we have created a loop by using goto statement but since no looping construct (i.e. for, while or do-while) is used, a break statement cannot be placed there. Hence, the compilation error 'Misplaced break in function main' occurs.

69.
```
1 2 3 4
```

Explanation:

`goto loop;` statement is used to create a logical loop and `goto out;` statement is used to take the control out of this logical loop. The `printf` statement present inside the logical loop prints the value of `i`. The value of `i` is manipulated by the expression `i++`. When the value of `i` becomes `5`, `goto out;` takes the control out of the logical loop and the logical loop terminates.

70.
```
1 1
2 1
```

Explanation:

Value of i	Condition of outer `for` loop	Value of j	Condition of inner `for` loop	Controlling expression of if statement (j==2)	Whether `break` is executed	Whether `printf` statement is executed	The values that get printed
1	True	1	True	False	No	Yes	1 1
		2	True	True	Yes	No	
2	True	1	True	False	No	Yes	2 1
		2	True	True	Yes	No	
3	False	Outer `for` loop is terminated					

71.
```
1 1
1 3
2 1
2 3
```

Explanation:

Value of i	Condition of outer `for` loop	Value of j	Condition of inner `for` loop	Controlling expression of if statement (j==2)	Whether `continue` is executed	Whether `printf` statement is executed	The values that get printed
1	True	1	True	False	No	Yes	1 1
		2	True	True	Yes	No	
		3	True	False	No	Yes	1 3
		4	False	Inner `for` loop is terminated			
2	True	1	True	False	No	Yes	2 1
		2	True	True	Yes	No	
		3	True	False	No	Yes	2 3
		4	False	Inner `for` loop is terminated			
3	False	Outer `for` loop is terminated					

72. Compilation error

    Explanation:

    ++ operator has higher priority than an assignment operator and will get evaluated first. The expression i++ evaluates to an r-value and cannot be placed on the left side of the assignment operator. Thus, i++=0 leads to the compilation error 'L-value required in function main'.

73. No output

    Explanation:

    The if controlling expression is y,x,b,a. In if controlling expression, the sub-expressions y, x, b and a are separated by comma operators. The comma-separated expressions (i.e. y, x, b and a) are evaluated from left to right and the result of evaluation of full expression is the result of evaluation of the right-most sub-expression (i.e. a). Since a is 0, the value of the entire expression y,x,b,a turns out to be 0. 0 is considered as false and hence the if body will not be executed.

74. No output

    Explanation:

    i is initialized with 0. The condition of the for loop (i.e. i++) has post-increment operator. This means, firstly the value of i (i.e. 0) is used for the evaluation of expression and then the value of i will be incremented. Thus, the controlling expression of the loop evaluates to 0, i.e. false. Hence, the body of the loop will not be executed and there will be no output.

75. 1 2 ...32767 -32768 -32767...-1

    Explanation:

    The condition of the for loop has a pre-increment operator. The value of i is incremented first and then used for the evaluation of expression. The value of i first becomes 1 and then is used for the evaluation of the for controlling expression. Since the controlling expression evaluates to true, the body of the loop will be executed and 1 gets printed. The condition is evaluated again, i becomes 2 and gets printed. This process is repeated till 32767 gets printed. Now, when ++i is evaluated, i becomes -32768 instead of 32768 (due to the range wrapping to the other side). This is a non-zero value and will be treated as true and it gets printed. The condition is evaluated again, i becomes -32767 and gets printed. This process is repeated till -1 is printed. After printing of -1, ++i gets evaluated and i becomes 0. 0 is treated as false; hence, the condition of the loop becomes false and the loop terminates.

76. 3 3

    Explanation:

    The condition of the for loop is an expression i<6,j<4. This expression has two sub-expressions separated by a comma operator. The sub-expressions will be evaluated from left to right but the outcome of the full expression, i.e. i<6,j<4 depends upon the outcome of the right-most sub-expression, i.e. j<4. Hence, till the sub-expression j<4 evaluates to true, the body of the loop will be executed. The initial value of j is 3. The sub-expression j<4 (i.e. 3<4) evaluates to true and the body of the loop will be executed. The value that gets printed is 3 3. After the execution of the body of the loop, the expression i++,j++ gets evaluated. Both i and j become 4. Now the condition j<4 (i.e. 4<4), evaluates to false and the loop gets terminated.

77. Compilation error

    Explanation:

    A label name is a type of identifier that has only function scope. In function main, goto here; statement is present but there is no label named here. The label here present inside the body of other_function is not visible inside the function main, as label names have only function scope. Hence, the compilation error 'Undefined label here in function main' occurs.

78. 3 4

Explanation:

The goto statement is capable of taking the program control in or out of a loop. In the given piece of code, the goto statement is used to transfer the program control inside the for loop. Since the goto statement transfers the control inside the for loop, the initialization expression in the for header will not be executed. Hence, the value of i remains 3 instead of being initialized to 0. After this the for loop works normally and 3 4 gets printed.

79. Compilation error

Explanation:

The general form of the do-while statement is:

```
do
statement
while(expression);
```

The semicolon after the while controlling expression is a must, else there will be a compilation error 'Statement ; missing'.

80. 5

Explanation:

do-while is an exit-controlled loop. The body of the loop will be executed once, even if the controlling condition is initially false. In the given piece of code, the controlling expression is initially false, even then the body of the do-while loop is executed once, and 5 gets printed.

Answers to Multiple-choice Questions

81. c 82. b 83. b 84. b 85. c 86. c 87. a 88. c 89. b 90. b 91. a 92. b 93. b 94. c 95. b 96. c 97. c 98. d 99. a 100. d

Programming Exercises

Program 1 | Check whether a given number is even or odd without using modulus operator

Whether a number is even or odd can be determined by checking its **L**east **S**ignificant **B**it (LSB). If the first bit of a number is:

LSB

- **0**, the number is even, e.g. 6, i.e. 0000 0000 0000 011**0**
- **1**, the number is odd, e.g. 13, i.e. 0000 0000 0000 110**1**

PE 5-1.c

```
//Even or odd without using modulus operator
#include<stdio.h>
main()
{
int num;
printf("Enter the number\t");
scanf("%d",&num);
if((num&1)==0)
    printf("Number %d is even",num);
else
    printf("Number %d is odd",num);
}
```

Output window

```
Enter the number    12
Number 12 is even
```

Program 2 | Check whether a given year is leap or not

A year is a **leap year,** if:

- It is divisible by 4 but not by 100, or
- It is divisible by 400.

Line | PE 5-2.c

```
//Leap year
#include<stdio.h>
main()
{
int year;
printf(“Enter the year\t”);
scanf(“%d”,&year);
if(((year%4==0) && (year%100!=0)) || (year%400==0))
    printf(“%d is a leap year”, year);
else
    printf(“%d is not a leap year”, year);
}
```

Output window

```
Enter the year    2004
2004 is a leap year
```

Program 3 | Calculate the roots of a quadratic equation

The roots of a quadratic equation $ax^2 + bx + c = 0$ can be obtained by using the expression $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$, where $b^2 – 4ac$ is called discriminant.

If $b^2–4ac > 0$, the roots are real and unequal.

If $b^2–4ac = 0$, the roots are real and equal, i.e. $x = \frac{-b}{2a}$.

If $b^2–4ac < 0$, the roots are imaginary, i.e. $x = \frac{-b}{2a} \pm \sqrt{\frac{b^2 - 4ac}{2a}}\,i$.

Line | PE 5-3.c

```
//Roots of a quadratic equation
#include<stdio.h>
#include<math.h>
main()
{
int a, b, c, d;
float r1,r2;
int num;
printf(“Enter the coefficients a, b and c\t”);
scanf(“%d %d %d”, &a, &b, &c);
d=b*b-4*a*c;
if(d>0)
{
    r1=(-b+sqrt(d))/(2*a);
    r2=(-b-sqrt(d))/(2*a);
    printf(“Roots are real and unequal\n”);
```

Output window

```
Enter the coefficients a, b and c    1 4 3
Roots are real and unequal
Roots are: -1.000000 -3.000000
```

(Contd...)

```
    printf(“Roots are: %f %f”,r1,r2);
}
else if(d==0)
{
    r1= -b/(2*a);
    printf(“Roots are real and equal\n”);
    printf(“Roots are: %f %f”,r1,r1);
}
else
    printf(“No real roots, roots are imaginary”);
}
```

Program 4 | Find the sum of individual digits in a given positive integer number

Line PE 5-4.c

```
//Find sum of digits of a given number
#include<stdio.h>
main()
{
int num,sum=0,digit;
printf(“Enter the number\t”);
scanf(“%d”,&num);
while(num!=0)
{
    digit=num%10;
    sum=sum+digit;
    num=num/10;
}
printf(“Sum of digits is %d”,sum);
}
```

Output window

```
Enter the number    786
Sum of digits is 21
```

Program 5 | Find the reverse of a given number

Line PE 5-5.c

```
//Reverse of a given number
#include<stdio.h>
main()
{
int num,reverse=0, digit;
printf(“Enter the number\t”);
scanf(“%d”,&num);
while(num!=0)
{
    digit=num%10;
    num=num/10;
    reverse=reverse*10+digit;
}
printf(“Reverse is %d”, reverse);
}
```

Output window

```
Enter the number    534
Reverse is 435
```

Program 6 | Check whether a given number is a palindrome or not

A number is a **palindrome** if the reverse of the number is equal to the number itself, e.g. 121, 535, etc.

PE 5-6.c

```c
//Palindrome
#include<stdio.h>
main()
{
int num, temp, digit, reverse=0;
printf(“Enter the number\t”);
scanf(“%d”,&num);
temp=num;
while(temp!=0)
{
    digit=temp%10;
    temp=temp/10;
    reverse=reverse*10+digit;
}
if(num==reverse)
    printf(“%d is a palindrome”, num);
else
    printf(“%d is not a palindrome”, num);
}
```

Output window

```
Enter the number     1234
1234 is not a palindrome
```

Output window (second execution)

```
Enter the number     12321
12321 is a palindrome
```

Program 7 | Check whether a given number is perfect or not

An integer is said to be a **perfect number** if its factors (including 1) sum to the number, e.g. 6 is a perfect number as 6=1+2+3, 28 is a perfect number as 28=1+2+4+7+14.

PE 5-7.c

```c
//Perfect number
#include<stdio.h>
main()
{
int num, sum=0, i;
printf(“Enter the number\t”);
scanf(“%d”,&num);
for(i=1;i<num;i++)
{
    if(num%i==0)
        sum=sum+i;
}
if(num==sum)
    printf(“%d is a perfect number”, num);
else
    printf(“%d is not a perfect number”, num);
}
```

Output window

```
Enter the number     28
28 is a perfect number
```

Output window (second execution)

```
Enter the number 23
23 is not a perfect number
```

Program 8 | Print first n perfect numbers

PE 5-8.c

```c
//First n perfect numbers
#include<stdio.h>
main()
{
int num=1, sum=0, i, count=1, n;
printf("How many numbers you want to print\t");
scanf("%d", &n);
printf("Perfect numbers are:\n");
while(count<=n)
{
    for(i=1;i<num;i++)
    {
        if(num%i==0)
            sum=sum+i;
    }
    if(num==sum)
    {
        printf("%d\t",num);
        count++;
    }
    num++; sum=0;
}
}
```

Output window

```
How many numbers you want to print      3
Perfect numbers are:
6     28     496
```

Program 9 | Check whether a given number is an Armstrong number or not

A number is said to be an **Armstrong number** if the sum of cube of its digits is equal to the number itself, e.g. 153 is an Armstrong number as $153=1^3+5^3+3^3$, i.e. 153 = 1 + 125 + 27.

PE 5-9.c

```c
//Armstrong number
#include<stdio.h>
main()
{
int num, temp, digit, sum=0;
printf("Enter the number\t");
scanf("%d",&num);
temp=num;
while(temp!=0)
{
    digit=temp%10;
    sum=sum+digit*digit*digit;
    temp=temp/10;
}
if(num==sum)
    printf("%d is an Armstrong number", num);
else
    printf("%d is not an Armstrong number",num);
}
```

Output window

```
Enter the number     153
153 is an Armstrong number
```

Output window (second execution)

```
Enter the number 221
221 is not an Armstrong number
```

Program 10 | Fibonacci series

Fibonacci series is a series in which a term is equal to the sum of the previous two terms. The first term of the series is 0 and the second term is 1.

PE 6-10.c

```
//Fibonacci series:      0 1 1 2 3 5 8 13 21 ...
#include<stdio.h>
main()
{
int n, count=2, a=0, b=1, c;
printf(“How many terms do you want to print\t”);
scanf(“%d”,&n);
printf(“Fibonacci series:\n”);
printf(“%d\t%d\t”,a,b);
while(count<n)
{
    c=a+b;
    printf(“%d\t”, c);
    a=b;
    b=c;
    count++;
}
}
```

Output window

```
How many terms do you want to print      5
Fibonacci series:
0     1     1     2     3
```

Program 11 | Find sum of all odd numbers that lie between 1 and n

PE 5-11.c

```
//Sum of odd numbers 1+3+5+7...+n
#include<stdio.h>
main()
{
int n, sum=0, i=1;
printf(“Enter the value of n\t”);
scanf(“%d”,&n);
while(i<=n)
{
    if(i%2==1)
        sum=sum+i;
    i++;
}
printf(“Sum of odd numbers from %d to %d is %d”,1,n,sum);
}
```

Output window

```
Enter the value of n      5
Sum of odd numbers from 1 to 5 is 9
```

Program 12 | Find the sum of series 1+(1+2)+ (1+2+3) +(1+2+3+4)... n terms

PE 5-12a.c

```
//Sum of the given series
#include<stdio.h>
main()
{
int num, i=1, j, sum=0;
printf(“Enter the number of terms\t”);
```

PE 3-12b.c

```
//Sum of the given series
//Output in a better way
#include<stdio.h>
main()
{
int num, i=1, j, sum=0;
```

Output window PE 3-12a.c

```
Enter the number of terms      3
Sum of the series is 10
```

Output window PE 3-12b.c

```
Enter the number of terms      3
(1)+(1+2)+(1+2+3)= 10
```

(Contd...)

```
scanf("%d",&num);
while(i<=num)
{
    j=1;
    while(j<=i)
    {
        sum=sum+j;
        j++;
    }
    i++;
}
printf("Sum of the series is %d", sum);
}
```

```
printf("Enter the number of terms\t");
scanf("%d",&num);
while(i<=num)
{
    j=1;
    printf("(");
    while(j<=i)
    {
        printf("%d",j);
        sum=sum+j;
        j++;
        if(j<=i)
            printf("+");
        else
            printf(")");
    }
    if(i<num)
        printf("+");
    i++;
}
printf("= %d", sum);
}
```

Program 13 | Find the sum of series $1^2 + 2^2 + 3^2 + \dots$ n terms

Line | **PE 5-13.c**

```
//Sum of the given series
#include<stdio.h>
main()
{
int n, i=1, sum=0;
printf("Enter the number of terms\t");
scanf("%d",&n);
while(i<=n)
{
    sum=sum + i*i;
    i++;
}
printf("Sum of series is %d",sum);
}
```

Output window

```
Enter the number of terms    5
Sum of series is 55
```

Program 14 | Find the sum of series 1+1/2+1/3+... n terms

Line | **PE 5-14.c**

```
//Sum of the given series
#include<stdio.h>
main()
{
int n, i=1;
float sum=0;
printf("Enter the number of terms\t");
scanf("%d",&n);
```

Output window

```
Enter the number of terms    3
Sum of series is 1.833333
```

(Contd...)

Line | **PE 5-14.c** | **Output window**

```
while(i<=n)
{
    sum=sum + 1/(float)i;
    i++;
}
printf(“Sum of series is %f”,sum);
}
```

Program 15 | Making use of sine series, evaluate the value of sin(x), where x is in radians

According to sine series: $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots \frac{x^n}{n!}$

Line | **PE 5-15.c** | **Output window**

```
//Evaluate sin(x)
#include<stdio.h>
main()
{
int i=1,n;
float sum, term, x;
printf(“Enter the value of x in radians\t”);
scanf(“%f”,&x);
printf(“Enter the power of end term\t”);
scanf(“%d”,&n);
sum=0;
term=x;
i=1;
while(i<=n)
{
    sum=sum + term;
    term=(term*x*x*-1)/((i+1)*(i+2));
    i=i+2;
}
printf(“Sin of %4.2f is %f”,x, sum);
}
```

```
Enter the value of x in radians     3.14
Enter the power of end term     25
Sin of 3.14 is 0.001593
```

Program 16 | Reverse, add and check for palindrome

Problem statement: Take a number, reverse its digits and add the reverse to the original. If the sum is not a palindrome, repeat the procedure with the sum until the result is a palindrome. Write a program that takes a number and gives the resulting palindrome and the number of additions it took to find it.

Test case:

```
  354        807       1515
+ 453      + 708     + 5151
 ____      _____     ______
  807       1515       6666
```

Result: Palindrome is 6666 and the number of additions to find it is 3.

PE 5-16.c

```c
//Comment: Reverse and Add
#include<stdio.h>
#include<conio.h>
main()
{
int num, temp, reverse=0, add=0, digit;
printf("Enter the number\t");
scanf("%d",&num);
while(1)
{
    temp=num;           //←Save num in temp
    reverse=0;
    while(temp!=0)      //←Find the reverse of temp
    {
        digit=temp%10;
        reverse=reverse*10+digit;
        temp=temp/10;
    }
    if(num==reverse)      //←Is it a palindrome
    {
        printf("\nPalindrome is %d and no. of addition is %d",reverse, add);
        break;
    }
    else                    //←If no, repeat the procedure with sum
    {
        printf("  %d\n",num);
        printf("+ %d\n",reverse);
        num=num+reverse;
        printf("----------\n");
        printf("  %d\n",num);
        printf("-----------\n");
        add++;          //←Keep track of number of additions performed
    }
}
}
```

Output window

```
Enter the number   354
   354
+  453
----------
   807
----------
   807
+  708
----------
   1515
----------
   1515
+  5151
----------
   6666
----------

Palindrome is 6666 and no. of addition is 3
```

Program 17 | Print pyramid of digits as shown below for n number of lines

Pyramid of digits:

```
            1
         2  3  2
      3  4  5  4  3
   4  5  6  7  6  5  4
   ………………………………..
```

Logic to print the pyramid:

			1			
		2	3	2		
	3	4	5	4	3	
4	5	6	7	6	5	4

1. Get the number of rows in the pyramid, let it be n.
2. In each row r (where r is the row number) leave (n-r) spaces blank and then print (2r-1) values. The printing of values starts with the row number. The first $\left\lceil \frac{2r-1}{2} \right\rceil$ values are printed by incrementing the previously printed value. The next $\left\lfloor \frac{2r-1}{2} \right\rfloor$ values are printed by decrementing the previously printed value.

Line | **PE 5-17.c**

```c
//Print pyramid of digits
#include<stdio.h>
main()
{
int n, r=1, val, j;
printf("Enter the number of rows in the pyramid\t");
scanf("%d",&n);
while(r<=n)                         //←Print n rows
{
    val=r;                          //←Printing starts with row number
    for(j=1;j<=n-r;j++)
        printf("\t");               //←Print n-r blank spaces
    for(j=1;j<=2*r-1;j++)
        if(j<=(2*r-1)/2)            //←Printing left half of the row
            printf("%d\t",val++);
        else if(j==(2*r-1)/2+1)     //←Printing middle element of row
                printf("%d\t",val);
            else                    //←Printing right half of the row
                printf("%d\t",--val);
    printf("\n");
    r++;
}
}
```

Output window

```
Enter the number of rows in the pyramid    4
            1
        2   3    2
    3   4   5    4    3
4   5   6   7    6    5    4
```

Program 18 | Print Floyd's triangle

Floyd's triangle:

```
1
2    3
4    5    6
7    8    9    10
………………………………..
```

Logic to print Floyd's triangle:

1. Get the number of rows in the Floyd's triangle, let it be n.
2. In each row r (where r is the row number), print r values. The printing of values starts with 1. Successive values are printed by incrementing the previously printed values.

(Contd...)

PE 5-18.c

```
//Floyd's triangle
#include<stdio.h>
main()
{i
nt n, r=1, val=1, j;
printf("Enter the number of rows in the triangle\t");
scanf("%d",&n);
while(r<=n)          //←Print n rows
{
      for(j=1;j<=r;j++)                 //←Printing a row
          printf("%d\t",val++);     //←Printing values
     printf("\n");                      //←New-line for next row
     r++;
}
}
```

Output window

```
Enter the number of rows in the triangle    4
1
2    3
4    5    6
7    8    9    10
```

Test Yourself

1. Fill in the blanks in each of the following:
   a. The smallest logical entity that can independently exist in a C program is ____________ .
   b. Statements in C language are terminated with a/an ____________ .
   c. A compound statement is also known as ____________ .
   d. The types of labeled statements are ____________, ____________, ____________ .
   e. A case label should be a compile time constant expression of ____________ type.
   f. The form of looping in which the number of iterations to be performed is known in advance is called ____________ .
   g. The execution or termination of a sentinel-controlled loop depends upon a special value known as ____________ .
   h. Sentinel-controlled loop is also known as ____________ .
   i. The statements for which no machine code is generated are called ____________ .
   j. To alter the default flow of control, ____________ statements are used.
   k. ____________ statement is used to terminate the current iteration of the enclosing loop.
   l. An expression terminated with a semicolon is known as ____________ statement.
   m. ____________ is an exit-controlled loop.
   n. The ____________ statement when executed in a switch statement causes immediate exit from it.
   o. Careless use of nested if-else statement may lead to ____________ problem.

2. State whether each of the following is true or false. If false, explain why.
   a. Only non-executable statements can appear outside the body of a function.
   b. Null statement performs no operation.
   c. An empty compound statement is equivalent to a null statement.
   d. An entry-controlled loop is executed at least once.
   e. Identifier-labeled statement is a branching statement and alters the flow of control.
   f. A continue statement can appear inside, or as a body of switch statement or a loop.
   g. Case-labeled statements can appear only inside the body of a switch statement.
   h. A break statement is used to terminate the current iteration of the loop.
   i. A switch selection expression can be of any type.
   j. In an entry-controlled loop, if the body of the loop is executed n times the expression in the condition section is evaluated n+1 times.

3. Write a simple C statement to accomplish each of the following:
   a. Test if the value of the variable count is greater than 10. If so, print “Count is greater than 10”.
   b. Assign the value 10 to the variables a, b and c.
   c. Increment the value of variable var by 10 and then assign it to variable stud.
   d. Test if the least significant bit of the variable num is 1. If so, assign 10 to variable a else assign 20 to it.
   e. Find factorial of a number n and assign it to variable fact.

4. Programming exercise:
   a. Write a C program that prints the integers between 1 and n which are divisible by 7. Get the value of n from the user.
   b. Write a C program that prints the integers from 1 to n omitting those integers which are divisible by 7. Get the value of n from the user.
   c. Write a C program that prints the integers between 1 and n which are divisible by 3, but not divisible by 4.
   d. Write a C program to find the sum of all integers that lie between 1 and n and are divisible by 7.
   e. Write a C program to evaluate 1×2×3×4×…n. Get the value of n from the user.

f. Write a C program to print first `n` Armstrong numbers. Get the value of `n` from the user.

g. Write a C program to print first `n` prime numbers. Get the value of `n` from the user.

h. Write a C program to evaluate the following series (Get the value `x` and `n` from the user):

i. $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots\infty$

ii. $\cosh(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \cdots\infty$

iii. $\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!}\cdots\infty$

iv. $e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!}\ldots + \frac{1}{n!}$

v. $\frac{\pi^2}{6} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2}\ldots\infty$

i. Write a C program to generate the following patterns (get the number of rows in the pattern from the user):

1.
```
1  2  3  4  5
   1  2  3
      1
```

2.
```
      1
   1  2  3
1  2  3  4  5
   1  2  3
      1
```

PART – III

ARRAYS, POINTERS AND STRINGS

This page is intentionally left blank

6

ARRAYS AND POINTERS

Learning Objectives

In this chapter, you will learn about:

- The limitation of basic data types
- Derived data types: array type and pointer type
- Arrays
- Single-dimensional and multi-dimensional arrays
- Declaration and usage of arrays
- Memory representation of arrays
- Different ways of storing multi-dimensional arrays
- Pointers
- Operations allowed on pointers
- Pointer arithmetic
- void pointer and null pointer
- Relationship between arrays and pointers
- Arrays of pointers
- Pointer to a pointer
- Pointer to an array
- Advantages and limitations of arrays
- Searching
- Sorting

6.1 Introduction

So far you have learnt about the basic data types, expressions and statements. In the previous chapter, you have learnt the use of iteration statements to perform repetitive tasks like summing first n natural numbers, etc. Consider a problem to find the average of marks secured by five students in a course. A piece of code written for it is given in Program 6-1.

Prog 6-1.c

```
//Average of marks secured by students
#include<stdio.h>
main()
{
    int marks1=10, marks2=12, marks3=9, marks4=11, marks5=17;
    int sum; float average;
    sum=marks1+marks2+marks3+marks4+marks5;
    average=sum/5.0;
    printf("Average marks secured is %f ",average);
}
```

Output window

```
Average marks secured is 11.800000
```

Program 6-1 | A program to find average marks secured by students

The powerful iteration statements discussed in Chapter 5 have not been used here to sum up the marks secured by the students because the marks are stored in separate variables and it is not possible to access them in a generalized way. Since there are only five students, it is possible to find the average in the above-mentioned manner. Now suppose there are 200 students in a course. For a problem of this scale, it is not feasible to create separate variables for storing the marks and finding the average in the above-mentioned manner. To solve such problems, a method is required that helps in storing and accessing data in a generalized and an efficient manner. The C language provides this method in the form of a derived data type known as **array type** or just **array**.

Consider another real-time problem that requires storing and processing names like "Sam" entered by the user. There is no basic data type available in C that provides this flexibility. A variable of char type can be used to store only one character but cannot be used to store all the three characters of the name "Sam". The derived array type provides a solution to this problem. An array enables the user to store the characters of the entered name in a contiguous set of memory locations, all of which can be accessed by only one name, i.e. the array name.

The array type has a close relationship with another derived data type, known as the **pointer type** or just **pointer**. Their relationship is so intimate that they cannot be studied in isolation. In this chapter, I will describe both arrays and pointers. Finally, we will look at the operations that can be applied on them and how to use them to solve problems.

6.2 Arrays

An **array** is a data structure that is used for the storage of homogeneous data, i.e. data of the same type. Figure 6.1 depicts arrays of four different types.

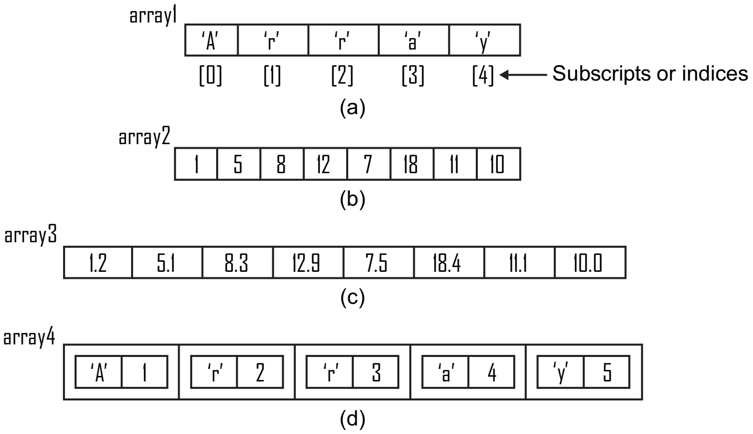


Figure 6.1 | (a) Character array; (b) integer array; (c) float array; (d) array of user-defined type

The important points about arrays are as follows:

1. An **array** is a collection of elements of the same data type. The data type of an element is called **element type**. For example, in Figure 6.1, the element type of array1 is char, array2 is int, array3 is float and array4 is user-defined type.
2. The individual elements of an array are not named. All the elements of an array share a common name, i.e. the **array name**. For example, in Figure 6.1 (a), all the elements of array, i.e. 'A', 'r', 'r', 'a' and 'y' have a common name, i.e. array1.
3. The individual elements of an array are distinguished and are referred to or accessed according to their positions in an array. The position of an element in an array is specified with an integer value known as **index** or **subscript**. Because arrays use indices or subscripts to access their elements, they are also known as **indexed variables** or **subscripted variables**.
4. The array index in C starts with 0, i.e. index of the first element of an array is 0.
5. The memory space required by an array can be computed as (**size of element type**) × (**Number of elements in an array**). For example, in Figure 6.1, array1 takes 1×5, i.e. 5 bytes in the memory, array2 takes 16 bytes (if an integer occupies 2 bytes), array3 takes 32 bytes and array4 takes 15 bytes (if an integer takes 2 bytes) in the memory.
6. Arrays are always stored in contiguous (i.e. continuous) memory locations. For example, in Figure 6.1, if the first element of array1 is stored at memory location 2000, then the successive elements of the array will be stored at the memory locations 2001, 2002, 2003 and 2004. In case of array2, if the first element is stored at memory locations 2000-2001, the next elements will be stored at the memory locations 2002-2003, 2004-2005, and so on.

Data structure is a logical representation of data. It provides systematic mechanisms for storage, retrieval and manipulation of data. Examples of data structures are: **arrays**, stacks, queues, linked lists, trees, etc.

In general, arrays are classified as:

1. Single-dimensional arrays
2. Multi-dimensional arrays

6.3 Single-dimensional Arrays

A **single-dimensional** or **one-dimensional array** consists of a fixed number of elements of the same data type organized as a simple linear sequence. The elements of a single-dimensional array can be accessed by using a single subscript, thus they are also known as **single-subscripted variables**. The other common names of single-dimensional arrays are **linear arrays** and **vectors**. Single-dimensional arrays are shown in Figure 6.2.

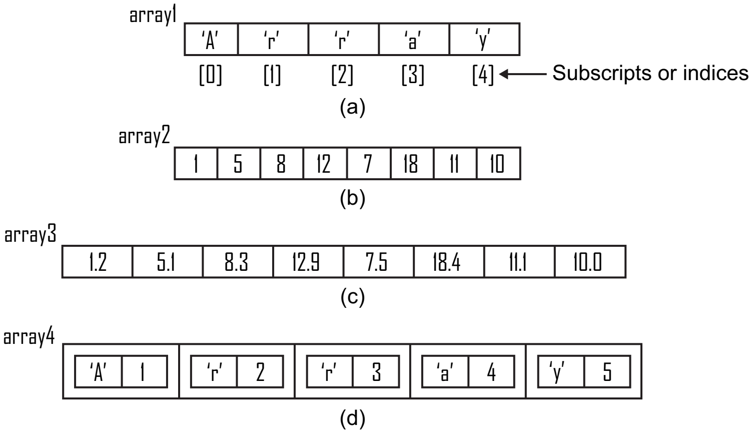


Figure 6.2 | Single-dimensional arrays

There are two aspects of working with arrays:

1. Declaration (i.e. creation) of array
2. Usage (i.e. storing or referring elements) of array

6.3.1 Declaration of a Single-dimensional Array

The general form of a single-dimensional array declaration is:

<storage_class_specifier><type_qualifier><type_mod>**type_specifier identifier[**<size_specifier>**]**<=initialization_list<,...>>**;**

The important points about a single-dimensional array declaration are as follows:

1. The terms enclosed within angular brackets (i.e. <>) are optional and might not be present in a declaration statement. The terms shown in bold are the mandatory parts of a single-dimensional array declaration.

2. A single-dimensional array declaration consists of a type specifier (i.e. **element type**), an identifier (i.e. **name of array**) and a size specifier (i.e. **number of elements** in the array) enclosed within square brackets (i.e. []). The following declarations of single-dimensional arrays are valid:

```
int array1[8];     //←array1 is an array of 8 integers              (Integer array)
float array2[5];   //←array2 is an array of 5 floating point numbers (Floating point array)
char array3[6];    //←array3 is an array of 6 characters             (Character array)
```

3. The size specifier specifies the number of elements in an array. The syntactic rules about the size specifier are as follows:

   a. It should be a compile time constant expression of integral type.

   Reasons:

   i. The memory space to an array is allocated at the compile time. The memory requirement of an array depends upon its element type and the number of elements (i.e. size) in it. Hence, the size of an array must be known at the compile time so that memory can be allocated to it.

   ii. The size of an array cannot be expanded or squeezed at the run-time. Thus, size must be a constant expression so that it cannot be changed at the run-time.

   The following declarations of single-dimensional arrays are valid:

```
int array1[3+5];       //←3+5 is a compile time constant expression of int type
float array2[size];    //←where size is a qualified constant of integral type
char array3[size];     //←where size is a symbolic constant of integral type
```

   The following declarations of single-dimensional arrays are not valid:

```
int array1[j];         //← j is a variable and not a constant
int array2[3.5];       //←It is not possible to create an array of 3.5 locations
```

   b. It should be greater than or equal to one.

   Reason: It is not possible to create an array of size zero, i.e. having no element.

   It is allowed to create an array of size 1, i.e. having only one element. Array of size 1 is like a simple variable and does not provide any significant advantage.

   The following declarations of single-dimensional arrays are not valid:

```
int array1[-1];        //← It is not possible to create an array of -1 locations
char array2[0];        //← It is not possible to create an array of 0 locations
```

   c. The size specifier is mandatory if an array is not explicitly initialized, i.e. if an initialization list is not present.

   Reason: If an initialization list is present, it is possible to determine the size of array from the number of initializers in the initialization list. In that case, the size specification becomes optional.

   The following declaration of a single-dimensional array is not valid:

```
int array1[];          //←Here, it is not possible to determine the size of array
                       //← Hence, the amount of memory to be allocated cannot
                       //   be determined
```

4. **Initializing elements of a single-dimensional array:** Like variables can be initialized, similarly the elements of an array can also be initialized. The syntactic rules about the initialization of array elements are as follows:

   a. The elements of an array can be initialized by using an **initialization list**. An initialization list is a comma-separated list of initializers enclosed within braces.

   b. An **initializer** is an expression that determines the initial value of an element of the array.

   c. If the type of initializers is not the same as the element type of an array, implicit type casting will be done, if the types are compatible. If types are not compatible, there will be a compilation error. The code segment in Program 6-2 illustrates this fact.

Prog 6-2.c

```
//Initializers of compatible but different types
#include<stdio.h>
main()
{
    int arr1[]={2.3, 4.5, 6.9};
    float arr2[]={'A','B','C'};
    printf("Elements of arrays are initialized with\n");
    printf("arr1: %d %d %d\n",arr1[0],arr1[1],arr1[2]);
    printf("arr2: %f %f %f\n",arr2[0],arr2[1],arr2[2]);
}
```

Output window

```
Elements of arrays are initialized with
arr1: 2 4 6
arr2: 65.000000 66.000000 67.000000
```

Remarks:

- The element types of the arrays are different from the types of initializers but the types are compatible
- float initializers are demoted and then elements of arr1 are initialized
- char initializers are promoted before initializing the elements of arr2. ASCII values of characters are used

Program 6-2 | A program to illustrate that the initializer's type can be different from the element type of an array

   d. The number of initializers in the initialization list should be less than or at most equal to the value of size specifier, if it is present.

   The following declarations of single-dimensional arrays are valid:

```
int array1[]={1,2,3,4,5};  //←Initialization list {1,2,3,4,5} present
int array2[]={2+3,a+5}; //←Initializers are 2+3 and a+5, where a is an int variable
char array3[6]={'A','r','r','a','y'};    //←Number of initializers is less than the value of
                                         //   size specifier
```

   The following declaration of a single-dimensional array is not valid:

```
int array1[2]={1,2,3,4,5}; //← Number of initializers cannot be more than the
                           //   value of size specifier
```

   e. If the number of initializers in the initialization list is less than the value of the size specifier, the leading array locations (i.e. occurring first) equal to the number of initializers get initialized with the values of initializers. The rest of the array locations get initialized to 0 (if it is an integer array), 0.0 (if case of floating point array) and '\0' (i.e. null character, if it is an array of character type). The above-mentioned fact is shown in Figure 6.3.

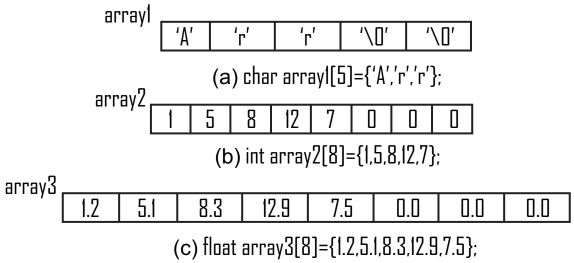

array1
'A'
'r'
'r'
'\0'
'\0'
(a) char array1[5]={'A','r','r'};
array2
1
5
8
12
7
0
0
0
(b) int array2[8]={1,5,8,12,7};
array3
1.2
5.1
8.3
12.9
7.5
0.0
0.0
0.0
(c) float array3[8]={1.2,5.1,8.3,12.9,7.5};

Figure 6.3 | Contents of arrays if the number of initializers is less than their size

6.3.2 Usage of Single-dimensional Array

The elements of a single-dimensional array can be accessed by using a subscript operator (i.e. []) and a subscript. The important points about the usage of single-dimensional arrays are as follows:

1. For accessing the elements of a one-dimensional array, the general form of expression is E1[E2], where E1 and E2 are sub-expressions and [] is the subscript operator. One of the sub-expressions E1 or E2 must be of an array type✍ or a pointer type[†] and the other sub-expression must be of an integral type.
2. The sub-expression of the integral type (i.e. the subscript) must evaluate to a value greater than or equal to 0.
3. The array subscript in C starts with 0, i.e. the subscript of the first element of an array is 0. Thus, if the size of an array is n, the valid subscripts are from 0 to n-1. However, if the array index greater than n-1 is used while accessing an element of the array, there will be no compilation error. This is due to the fact that C language does not provide compile time or run-time **array index out-of-bound check**. However, using an out-of-bound index may lead to run-time error or exceptions. Thus, care must be taken to ensure that the array indices are within bounds, i.e. from 0 to n-1.

✍ An **array type** is one of the derived data types. It is said to be derived from an element type and if the element type is T (where T is a generic term and can be int, float, char or any other type), the array type is called '**array of Ts**'. The construction of an array type from an element type is called '**array type derivation**'. Consider the declaration statement **int array[5];** the array type derived from an element type **int** is **int[5]**.

The code snippet in Program 6-3 illustrates the use of a singe-dimensional array.

[†] Refer Section 6.4 for a description on pointer type.

Prog 6-3.c

```c
//Use of single-dimensional array
#include<stdio.h>
main()
{
    int a[3]={10,20,30};
    printf("Elements of array are:\n");
    printf("%d %d %d",a[0],a[1],a[2]);
}
```

Output window

```
Element of array are:
10 20 30
```

Remarks:

- a is of array type.
- The expression a[0] refers to the first element, a[1] refers to the second element and a[2] refers to the third element of the array

Program 6-3 | A program to illustrate the use of subscript operator

6.3.2.1 Reading, Storing and Accessing Elements of a One-dimensional Array

An iteration statement (i.e. loop) is used for storing and reading the elements of a one-dimensional array. The code snippet in Program 6-4 illustrates a method to read, store and access the elements of a single-dimensional array.

Line | Prog 6-4.c

```c
 1  //Use of single-dimensional array
 2  #include<stdio.h>
 3  main()
 4  {
 5      int marks[200], lc, studs, sum=0;
 6      float average;
 7      printf("Enter the number of students in class\t");
 8      scanf("%d",&studs);
 9      printf("Enter marks of students\n\n");
10      for(lc=0;lc<studs;lc++)
11      {
12          printf("Enter marks of student %d\t",lc+1);
13  //Reading and storing elements in a 1-D array
14          scanf("%d",&marks[lc]);
15      }
16      for(lc=0;lc<studs;lc++)
17  //Accessing elements stored in the 1-D array
18          sum=sum+marks[lc];
19      average=(float)sum/studs;
20      printf("\nAverage marks of the class is %f",average);
21  }
```

Output window

```
Enter the number of students in class     5
Enter marks of students

Enter marks of student 1     10
Enter marks of student 2     12
Enter marks of student 3     9
Enter marks of student 4     11
Enter marks of student 5     17

Average marks of the class is 11.800000
```

Remarks:

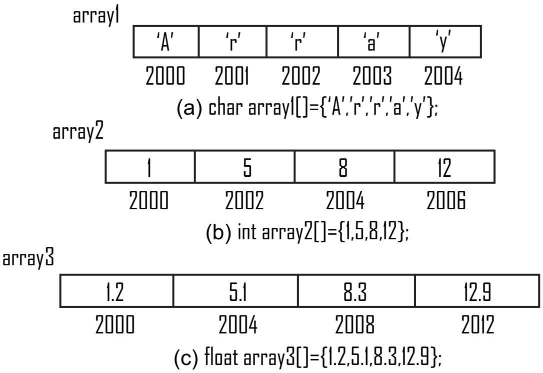
- The marks of 200 students can be stored in an array named marks. The elements of the array can be accessed in general way by writing marks[lc], where lc ∈ {0….199}
- Although at the runtime marks of only 5 students are entered, the size of array is kept 200 to accommodate the worst case (i.e. 200 students)
- 195 locations are not used. Hence, 195*2=390 bytes of memory got wasted
- In line number 19, integer variable sum is explicitly type casted to float

Program 6-4 | A scalable version of Program 6-1

6.3.3 Memory Representation of Single-dimensional Array

The elements of an array are **stored in contiguous** (i.e. continuous) memory locations. This is depicted in Figure 6.4.

i The mentioned addresses refer to the starting addresses of the elements. The first element in Figure 6.4(c) occupies the memory locations 2000–2003.



(a) char array1[]={'A','r','r','a','y'};

(b) int array2[]={1,5,8,12};

(c) float array3[]={1.2,5.1,8.3,12.9};

Figure 6.4 | Elements of the array are stored in contiguous memory locations

6.3.4 Operations on a Single-dimensional Array

6.3.4.1 Subscripting a Single-dimensional Array

The only operation allowed on arrays is **subscripting**. Subscripting is an operation that selects an element from an array. To perform subscripting in C language, a subscript operator (i.e. []) is used. The rules for subscripting have already been discussed in Section 6.3.2.

6.3.4.2 Assigning an Array to Another Array

A variable can be assigned to or initialized with another variable but an array cannot be assigned to or initialized with another array. The following statement is not valid and leads to a compilation error:

array1=array2; //←where array1 and array2 are arrays of the same type and size

Reason: In C language, the name of the array refers to the address of the first element of the array and is a constant object. It does not have a modifiable l-value. Since it does not have a modifiable l-value, it cannot be placed on the left side of the assignment operator.

To assign an array to another array, each element must be assigned individually. The code segment in Program 6-5 illustrates the mentioned fact.

Line	Prog 6-5.c	Output window

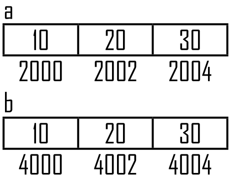
1
2
3
4
5
6
7
8
9
10
	//Assignment of an array to another array
	#include<stdio.h>
	main()
	{
	    int a[3], b[3]={10,20,30};
	    printf("Assigning an array to an array:\n");
	    a=b;
	    printf("Elements of array a are:\n");
	    printf("%d %d %d",a[0],a[1],a[2]);
	}
		Compilation error "L-value required in function main"
		Reasons:
		• The name of the array a refers to the address of the first element of the array and is a constant object
		• It does not refer to a modifiable l-value
		• Hence, it cannot be placed on the left side of the assignment operator
		What to do?
		• Making use of a loop, assign individual elements of array b to the elements of array a by writing a[i]=b[i], where i∈{0,1,2}

Program 6-5 | A program to illustrate that an array cannot be assigned to another array in one step

6.3.4.3 Equating an Array with Another Array

When the operands of an equality operator are of the array type, it always evaluates to false.
Reason: In C language, the name of an array refers to the address of the first element of the array and the addresses of first elements of two arrays can never be the same. Hence, when the operands of an equality operator are of array type, it always evaluates to false. Program 6-6 illustrates the mentioned fact.

Line	Prog 6-6.c	Memory contents	Output window

1
2
3
4
5
6
7
8
9
10
	//Equality operator & arrays
	#include<stdio.h>
	main()
	{
	    int a[3]={10,20,30}, b[3]={10,20,30};
	    if(a==b)
	        printf("Arrays are equal");
	    else
	        printf("Arrays are not equal");
	}
		 a
		10
		20
		30
		2000
		2002
		2004
		b
		10
		20
		30
		4000
		4002
		4004
			Arrays are not equal
			Reasons:
			• The name of arrays a and b refers to the addresses of their first elements, i.e. 2000 and 4000, respectively
			• Since the addresses are different, the equality operator evaluates to false, although the contents of the arrays are the same
			What to do?
			• For checking equality, check the equality of all individual elements

Program 6-6 | A program to illustrate the behavior of equality operator on arrays

To check whether the contents of two arrays are the same or not, check the equality of each individual element.

Programs 6-5 and 6-6 illustrate that the name of an array refers to the address of the first element of the array. An expression of an array type (e.g. the name of array) is automatically converted to an expression of pointer type. This automatic conversion makes the simultaneous discussion of arrays and pointers essential.

6.4 Pointers

A **pointer** is a variable that holds the address of a variable or a function. A pointer is a powerful feature that adds enormous power and flexibility to C language. A pointer variable can be declared as:

[storage_class_specifier][type_qualifier][type_modifier]**type_specifier* identifier**[=l-value[,....]]**;**

The important points about pointers are as follows:

1. The terms enclosed within square brackets (i.e. []) are optional and might not be present in a declaration statement. The terms shown in **bold** are the mandatory parts of a pointer variable declaration.
2. A pointer variable declaration consists of a type specifier (i.e. **referenced type**), **punctuator** * and an identifier (i.e. name of pointer variable). The following declarations are valid:

```
int *iptr;          //←iptr is pointer to an integer
float *fptr;        //←fptr is pointer to a float
char *cptr;         //←cptr is pointer to a character
```

```
const int *ptric;         //←ptric is pointer to an integer constant or constant integer
unsigned int *ptrui;      //←ptrui is pointer to an unsigned integer
```

3. Pointer variable declarations are read from the right side. The punctuator * is read as **'pointer to'**. So the declaration statement int *iptr; is read as 'iptr is a pointer to an integer'.✍

> ✍ The concept of pointer declaration is scalable. It is possible to declare a pointer to a variable, which itself is a pointer variable. Such a pointer is known as a **pointer to a pointer**.[‡] The declaration statement **int **pptr;** declares a pointer to a pointer and is read as '**pptr is a pointer to a pointer to an integer**'.

4. A pointer variable can hold the address of a variable or a function. In Figure 6.5(a) iptr is an integer pointer and holds the address of an integer variable a. In Figure 6.5(c) pptr is pointer to pointer to an integer and holds the address of an integer pointer iptr, which in turn holds the address of an integer variable val.

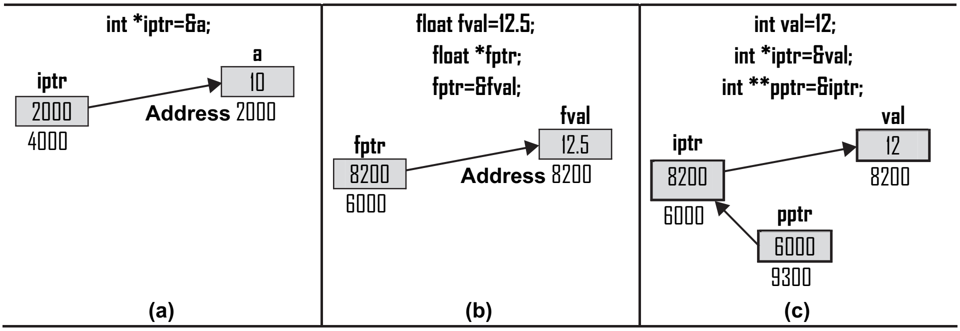


Figure 6.5 | Pointers holding addresses

5. Every pointer variable takes the same amount of memory space irrespective of whether it is a pointer to int, float, char or any other type. This fact is illustrated in the code segment given in Program 6-7.

Prog 6-7.c

```
//Size of pointer variables
#include<stdio.h>
main()
{
    char *cptr;
    int *iptr;
    float *fptr;
    printf("Pointer to character takes %d bytes\n",sizeof(cptr));
    printf("Pointer to integer takes %d bytes\n",sizeof(iptr));
    printf("Pointer to float takes %d bytes\n",sizeof(fptr));
}
```

Output window

```
Pointer to character takes 2 bytes
Pointer to integer takes 2 bytes
Pointer to float takes 2 bytes
```

Remarks:

- The above output is the result of execution using Borland Turbo C 3.0 IDE
- In Borland Turbo C 4.5 or MS-VC++ 6.0 each type of pointer variable takes 4 bytes

Program 6-7 | A program to illustrate that a pointer to any type takes the same amount of memory space

[‡] Refer Section 6.10 for a description on the pointer to a pointer.

6. The value of a pointer variable is printed with %p format specifier. Since the pointer variables hold addresses, which are unsigned integers, %u format specifier can also be used for printing pointer values. However, the use of %p format specifier is recommended over the use of %u format specifier.

6.4.1 Operations on Pointers

The operations allowed on pointers are as follows:

6.4.1.1 Referencing Operation

In **referencing operation**, a pointer variable is made to refer to an object. The reference to an object can be created with the help of a reference operator (i.e. &). The important points about the reference operator are as follows:

1. The reference operator, i.e. & is a unary operator and should appear on the left side of its operand.
2. The operand of the reference operator should be a variable of arithmetic type✍ or pointer type.✍ The operand of the reference operator can also be a function designator, i.e. name of a function.
3. The reference operator is also known as **address-of operator**.

The above-mentioned points are depicted in Figure 6.6.

```
float fval=12.5;        //← fval is a floating point variable initialized with 12.5
   float *fptr;         //← fptr is a pointer to float type
   fptr=&fval;          //← The address-of fval is assigned to fptr. fval is known as
                        //   referenced object and fptr is known as referencing
                        //   object and references fval
```

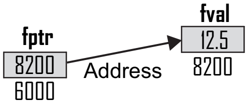


Figure 6.6 | A float pointer referencing a float variable

✍ Integral and floating types are collectively called **arithmetic types**. A **pointer type** describes an object, whose value provides reference to an object of type T. T is a generic term and will be known as **reference type**. It can be int, float, char or any other type. A pointer type derived from the reference type T is called **'pointer to T'**. The construction of a pointer type is called **'pointer-type derivation'**.

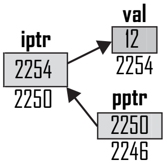
6.4.1.2 Dereferencing a Pointer

The object pointed to or referenced by a pointer can be indirectly accessed by dereferencing the pointer. A dereferencing operation allows a pointer to be followed to the data object to which it points. A pointer can be dereferenced by using a dereference operator (i.e. *). The important points about the dereference operator are as follows:

1. The dereference operator (i.e. *) is a unary operator and should appear on the left side of its operand.
2. The operand of a dereference operator should be of pointer type.
3. The dereference operator is also known as **indirection operator** or **value-at operator**.

The code snippet in Program 6-8 illustrates the use of a dereference operator.

Line	Prog 6-8.c	Memory	Output window

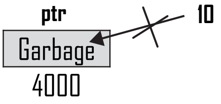
1
2
3
4
5
6
7
8
9
10
11
12
13
	//Dereferencing pointers
	#include<stdio.h>
	main()
	{
	int val=12;
	int *iptr=&val;
	int **pptr=&iptr;
	printf("Value is %d\n",val);
	printf("Value by dereferencing iptr is %d\n",*iptr);
	printf("Value by dereferencing pptr is %d\n",**pptr);
	printf("Value of iptr is %p\n",iptr);
	printf("value of pptr is %p\n",pptr);
	}
		 val
		iptr
		12
		2254
		2254
		2250
		pptr
		2250
		2246
			Value is 12
			Value by dereferencing iptr is 12
			Value by dereferencing pptr is 12
			Value of iptr is 2407:2254
			Value of pptr is 2407:2250
			Remarks:
			• The printed addresses are in the form of **segment address: offset address**
			• The segment address and the offset address are in the **hexadecimal number system**
			• If the memory is assumed to be analogous to a city, the segment address is analogous to a sector number and the offset address is analogous to a house number
			• The addresses that you get in the output may be different from the mentioned addresses as the memory allocation is purely random
			• val=12, iptr=2254 and ptr=2250
			• *iptr=value-at(iptr)=value-at(2254)=12
			• **pptr=value-at(value-at(pptr))=value-at(value-at(2250))=value-at(2254)=12

Program 6-8 | A program to illustrate the dereferencing operation

6.4.1.3 Assigning to a Pointer

1. A pointer can be assigned or initialized with the address of an object. A pointer variable cannot hold a non-address value and thus can only be assigned or initialized with l-values. Program 6-9 illustrates this fact.

Line	Prog 6-9.c	Memory contents	Output window

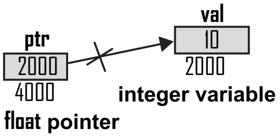
1
2
3
4
5
6
7
8
9
	// Invalid assignment to pointer variable
	#include<stdio.h>
	main()
	{
	    int val=10;
	    int *ptr=val;
	    printf("Value of variable is %d\n",val);
	    printf("Pointer holds %p\n", ptr);
	}
			Compilation error "Cannot convert int to int*"
			Reasons:
			• Pointer variables can only hold addresses
			• A pointer variable ptr cannot hold an integer value val
			What to do?
			• Initialize ptr with the address of variable val by writing &val and re-execute the code

Program 6-9 | A program to illustrate that a pointer variable cannot hold a non-address value

> There is an exception to this rule. The constant zero can be assigned to a pointer. For example, int *iptr=0; is valid. Assignment or initialization with zero makes the pointer a special pointer known as the **null pointer**.[§]

2. A pointer to a type cannot be initialized or assigned the address of an object of another type. Program 6-10 illustrates this fact.

Line	Prog 6-10.c	Memory contents	Output window

1
2
3
4
5
6
7
8
9
	// Invalid assignment to pointer variable
	#include<stdio.h>
	main()
	{
	    int val=10;
	    float *ptr=&val;
	    printf("Value of variable is %d\n",val);
	    printf("Pointer holds %p\n", ptr);
	}
		 val
		ptr
		10
		2000
		2000
		4000
		integer variable
		float pointer
		A float pointer cannot point to an integer variable
			Compilation error "Cannot convert int* to float*"
			Reasons:
			• A pointer variable can only be assigned address of an object of the same type
			• A pointer variable ptr (of type float*) cannot hold the address of an integer variable (i.e. int*)
			What can be done?
			• Explicitly type cast int* to float* by using type cast operator. Write float* ptr=(float*)&val; and then re-execute the code
			Remark:
			• Explicit type casting of pointers may give unexpected results and is not recommended

Program 6-10 | A program to illustrate that a pointer to a type cannot be assigned address of an object of another type

[§] Refer Section 6.6 for a description on null pointer.

3. A pointer can be assigned or initialized with another pointer of the same type. However, it is not possible to assign a pointer of one type to a pointer of another type without explicit type casting.

> There is an exception to Rules 2 and 3. A pointer to any type of object can be assigned to a pointer of type void*[¶] but vice-versa is not true. A **void pointer** cannot be assigned to a pointer to a type without explicit type casting.

6.4.2 Arithmetic Operations (Pointer Arithmetic)

Arithmetic operations can be applied to pointers in a restricted form. When arithmetic operators are applied on pointers, the outcome of the operation is governed by **pointer arithmetic**. The pointer arithmetic rules are mentioned below.

6.4.2.1 Addition Operation

1. An expression of integer type can be added to an expression of pointer type. The result of such operation would have the same type as that of pointer type operand. If ptr is a pointer to an object, then 'adding 1 to pointer' (i.e. ptr+1) points to the next object. Similarly, ptr+i would point to the ith object beyond the one the ptr currently points to. This is shown in Table 6.1.

Table 6.1 | Addition operation on pointers

S. No	Operator	Type of operand 1	Type of operand 2	Resultant type	Example	Initial value	Final value	How to determine?
1.	Addition operator (+)	Pointer to type T	int	Pointer to type T				Result = initial value of pointer + integer operand*sizeof (the reference type T)
	Example1:	**float***	**int**	**float***	**ptr=ptr+1**	**ptr=2000**	**2004**	**2000+1*(4)=2004** **as** **sizeof(float)=4**
	Example2:	**int***	**int**	**int***	**ptr=ptr+5**	**ptr=2000**	**2010**	**2000+5*(2)=2010, if** **sizeof(int)=2**
2.	Addition operator (+)	Pointer	Pointer	Not allowed				

2. Addition of two pointers is not allowed.
3. The addition of a pointer and an integer is commutative, i.e. ptr+1 is same as 1+ptr.

6.4.2.2 Increment Operation

The increment operator can be applied to an operand of pointer type. Table 6.2 depicts the application of an increment operator to an operand of a pointer type.

[¶] Refer Section 6.5 for a description on void pointer.

Table 6.2 | Increment operation on a pointer

S. No	Operator	Type of operand	Resultant type	Example	Initial values	Final values	How to determine?
1.	Increment operator (++)	Pointer to type T	Pointer to type T				**Post-increment:** Result=initial value of pointer **Pre-increment:** Result = initial value of pointer + sizeof (the reference type T) **In both the cases:** Value of pointer=Value of pointer + sizeof (the reference type T)
Example1:	**Post-increment**	float*	float*	ftr=ptr++	ftr=? ptr=2000	ftr=2000 ptr=2004	
Example2:	**Pre-increment**	float*	float*	ftr=++ptr	ftr=? ptr=2000	ftr=2004 ptr=2004	

6.4.2.3 Subtraction Operation

1. A pointer and an integer can be subtracted. The operation along with examples is shown in Table 6.3.

Table 6.3 | Subtraction operation on pointers

S. No	Operator	Type of operand 1	Type of operand 2	Resultant type	Example	Initial value(s)	Final value	How to determine?
1.	Subtraction operator (-)	Pointer to type T	int	Pointer to type T				Result = initial value of pointer - integer operand *sizeof (the reference type T)
	Example1:	float*	int	float*	ptr=ptr-1	ptr=2000	1996	2000-1*(4)=1996 as sizeof(float)=4
	Example2:	int*	int	int*	ptr=ptr-5	ptr=2000	1990	2000-5*(2)=1990, if sizeof(int)=2
2.	Subtraction operator (-)	Pointer to type T	Pointer to type T	int				Result=(operand1-operand2)/ sizeof (the reference type T)
	Example3:	float*	float*	int	a=p2-p1	p1=2000 p2=2008	2	(2008-2000)/ sizeof(float)= (2008-2000)/4=2

2. Subtraction of integer and pointer is not commutative, i.e. ptr-1 is not the same as 1-ptr. The operation 1-ptr is illegal.
3. Two pointers can also be subtracted. Pointer subtraction is meaningful only if both the pointers point to the elements of the same array. The result of the operation is the difference in subscripts of two array elements. The mentioned rule is described in Table 6.3 and is depicted in Figure 6.7.

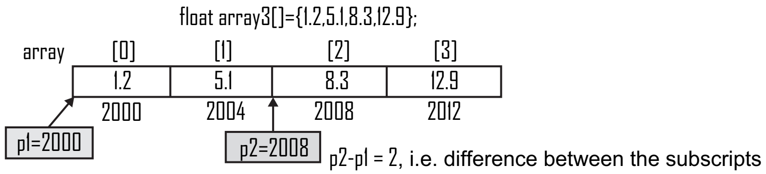

float array3[]={1.2,5.1,8.3,12.9};
array
[0]
[1]
[2]
[3]
1.2
5.1
8.3
12.9
2000
2004
2008
2012
p1=2000
p2=2008
p2-p1 = 2, i.e. difference between the subscripts

Figure 6.7 | Pointer subtracted from a pointer

6.4.2.4 Decrement Operation

The decrement operator can be applied to an operand of pointer type. Table 6.4 depicts the application of a decrement operator to an operand of pointer type.

Table 6.4 | Decrement operation on a pointer

S.No	Operator	Type of operand	Resultant type	Example	Initial values	Final values	How to determine?
1.	Decrement operator (--)	Pointer to type T	Pointer to type T				**Post- decrement:** Result=initial value of pointer **Pre-decrement:** Result = initial value of pointer - sizeof (the reference type T) **In both the cases:** Value of pointer=Value of pointer - sizeof (the reference type T)
Example1:	**Post-decrement**	**float***	**float***	**ftr=ptr--**	**ftr=?** **ptr=2000**	**ftr=2000** **ptr=1996**	
Example2:	**Pre-decrement**	**float***	**float***	**ftr=--ptr**	**ftr=?** **ptr=2000**	**ftr=1996** **ptr=1996**	

6.4.3 Relational (Comparison) Operations

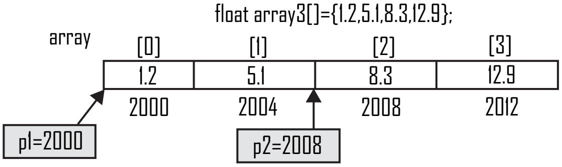
A pointer can be compared with a pointer of the same type or with zero. A comparison of pointers is meaningful only when they point to the elements of the same array. Table 6.5 depicts the comparison of pointers.

Table 6.5 | Relational operations on pointers

S.No	Operator	Type of operand 1	Type of operand 2	Resultant type	Example	Initial values	Final value	How to determine?
1.	Comparison operators (==, !=, <, <=, >, >=)	Pointer to type T	Pointer to type T	int (0 i.e. false or 1 i.e. true)				
	Example1:	**float***	**float***	**int**	**r=p1!=p2**	**p1=2000** **p2=2008**	**1**	
	Example2:	**float***	**float***	**int**	**r=p1<p2**	**p1=2000** **p2=2008**	**1**	
	Example3:	**float***	**float***	**int**	**r=p2>=p1**	**p1=2000** **p2=2008**	**1**	

(Contd...)

S.No	Operator	Type of operand 1	Type of operand 2	Resultant type	Example	Initial values	Final value	How to determine?
	Example4:	float*	float*	int	r=p2==p1	p1=2000 p2=2008	0	



6.4.4 Illegal Pointer Operations

The following operations on pointers are not allowed:

1. Addition of two pointers is not allowed.
2. Only integers can be added to pointers. It is not valid to add a float or a double value to a pointer.
3. Multiplication and division operators cannot be applied on pointers.
4. Bitwise operators cannot be applied on pointers.
5. A pointer of one type cannot be assigned to a pointer of another type (except void*) without explicit type casting.
6. A pointer variable cannot be assigned a non-address value (except zero).

The pointer arithmetic discussed above is not applicable to void pointers. However, what actually are void pointers?

6.5 void pointer

void is one of the basic data types available in C language. void means nothing or not known. It is not possible to create an object of type void. For example, the following declaration statement is not valid and leads to 'Size of var unknown or zero' compilation error.

```
void var;
```

Although an object of type void cannot be created, it is possible to create a pointer to void. Such a pointer is known as a **void pointer** and has type void*. **void pointer** is a generic pointer and can point to any type of object. Figure 6.8 depicts the mentioned fact.

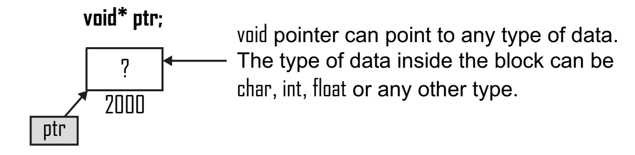

void* ptr;
?
2000
ptr
void pointer can point to any type of data. The type of data inside the block can be char, int, float or any other type.

Figure 6.8 | void pointer

6.5.1 Operations on void Pointer

The following operations on void pointer are allowed:

1. A pointer to any type of object can be assigned to a void pointer. This is a standard conversion and the compiler will do it implicitly without any explicit type casting. This is shown in Program 6-11.

Line	Prog 6-11.c	Output window
1	//Assigning a pointer to a void pointer	int* is implicitly converted to void* **Remarks:** • iptr is of int* type • vptr is of void* type • int* implicitly gets converted to void* in line number 7
2	#include<stdio.h>	
3	main()	
4	{	
5	int a=10;	
6	int *iptr=&a;	
7	void *vptr=iptr;	
8	printf("int* is implicitly converted to void*");	
9	}	

Program 6-11 | A program to illustrate that pointer to any type implicitly gets converted to void*

2. void pointers can be compared for equality and inequality.

The following operations on void pointers are not allowed:

1. A void pointer cannot be dereferenced.
2. Pointer arithmetic is not allowed on void pointers.

Reason: A void pointer cannot be dereferenced and pointer arithmetic is not applicable on it because the compiler does not know what kind of object the void pointer is really pointing to. Hence, the precise number of bytes to which the pointer refers to is not known. The compiler must know the number of bytes to which a pointer refers to in order to apply dereference operation and pointer arithmetic.

Before the application of dereference operator or arithmetic operator on a void pointer, it must be explicitly type casted to a pointer to a specific type.

6.6 Null Pointer

A **null pointer** is a special pointer that does not point anywhere. It does not hold the address of any object or function. It has numeric value 0. The following declaration statement declares nptr as a null pointer:

```
int *nptr=0;
```

The macro or symbolic constant NULL defined in the header files stdio.h, stddef.h, stdlib.h, alloc.h and mem.h can also be used for the creation of a null pointer. The following declaration statement is equivalent to the declaration statement mentioned above:

```
int *nptr=NULL;
```

The important points about null pointers are as follows:

1. When a null pointer is compared with a pointer to any object or a function, the result of comparison is always false.
2. Two null pointers always compare equal.
3. Dereferencing a null pointer leads to a runtime error.

6.7 Relationship Between Arrays and Pointers

In C language, **arrays and pointers** are so closely related that they cannot be studied in isolation. They are often used interchangeably. The following relationships exist between arrays and pointers:

1. The name of an array refers to the address of the first element of the array, i.e. an expression of array type decomposes to pointer type. Program 6-12 illustrates this fact.

Prog 6-12.c

```
//Arrays and pointers relationship-I
#include<stdio.h>
main()
{
    int arr[3]={10,15,20};
    printf("First element of array is at %p\n",arr);
    printf("Second element of array is at %p\n",arr+1);
    printf("Third element of array is at %p\n",arr+2);
}
```

Output window

```
First element of array is at 2407:2242
Second element of array is at 2407:2244
Third element of array is at 2407:2246
```

Remarks:

- The name of the array (i.e. arr) refers to the address of the first element of the array and is a constant object
- The expression arr+1 decomposes to pointer type
- Thus, in expression arr+1, the arithmetic involved is pointer arithmetic
- Note that ++arr cannot be written instead of arr+1 as arr is a constant object

Program 6-12 | A program to depict the relationship between arrays and pointers

The name of an array refers to the address of the first element of the array but there are two exceptions to this rule:

a. When an array name is operand of sizeof operator it does not decompose to the address of its first element. Program 6-13 illustrates this fact.

Prog 6-13.c

```
//sizeof operator and arrays
#include<stdio.h>
main()
{
    int array[5]={10,15,20,25,30};
    printf("The result of sizeof operator is %d\n",sizeof(array));
}
```

Output window

```
The result of sizeof operator is 10
```

Remarks:

- The result of the sizeof operator is the size of the complete array (i.e. 5 elements * 2 bytes each = 10 bytes)
- This example clearly indicates that the name of the array is not decomposed into pointer type
- If it would have been decomposed into pointer type, the result would have been 2 as integer pointer takes 2 bytes in the memory (in case of Borland Turbo C 3.0)

Program 6-13 | A program to illustrate the application of the sizeof operator on arrays

b. When an array name is an operand of reference or address-of operator it does not decompose to the address of its first element.

2. In C language, any operation that involves array subscripting is done by using pointers. The expression of form E1[E2] is automatically converted into an equivalent expression of form *(E1+E2). Program 6-14 illustrates this fact.

Line	Prog 6-14.c	Output window

1
2
3
4
5
6
7
8
9
	//Arrays and pointers relationship-II
	#include<stdio.h>
	main()
	{
	    int array[3]={10,15,20};
	    printf("Elements are %d %d %d\n",array[0], array[1],array[2]);
	    printf("Elements are %d %d %d\n",*(array+0),*(array+1),*(array+2));
	    printf("Elements are %d %d %d\n",0[array],1[array],2[array]);
	}
		Elements are 10 15 20
		Elements are 10 15 20
		Elements are 10 15 20
		Remarks:
		• E1[E2] is the usual way of subscripting (used in line number 6)
		• E1[E2] gets converted to *(E1+E2). The transformed way of subscripting is used in line number 7
		• 0[array] used in line number 8 is also valid because 0[array] will automatically be converted to *(0+array), which is equivalent to *(array+0), + being a commutative operation
		• *(array+0) is equivalent to array[0]. Hence, 0[array] is equivalent to array[0]

Program 6-14 | A program to depict the relationship between arrays and pointers

6.8 Scaling up the Concept

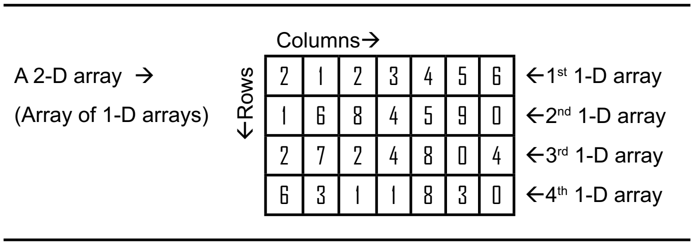
With all this knowledge at hand, it is the time to scale up the concept and look at array of arrays (i.e. multi-dimensional arrays), array of pointers, pointer to a pointer and pointers to arrays.

6.8.1 Array of Arrays (Multi-dimensional Arrays)

A **2-D array** is an array of 1-D (i.e. single dimensional) arrays and can be visualized as a plane that has rows and columns. Each row is a single-dimensional array. A **3-D array** is an array of 2-D arrays and can be visualized as a cube that has planes. Each plane is a 2-D array. This concept can be scaled up to any level and in general, an **n-D array** is an array of (n-1)-D arrays. Arrays having dimensions higher than three are generally not needed unless and until highly data-extensive applications are to be developed. Therefore, I will restrict the discussion only to three-dimensional arrays.

6.8.2 Two-dimensional Arrays

A **two-dimensional array** has its elements arranged in a rectangular grid of rows and columns. The elements of a two-dimensional array can be accessed by using a row subscript (i.e. row number) and a column subscript (i.e. column number). Both the row subscript and the column subscript are required to select an element of a two-dimensional array. A two-dimensional array is popularly known as a **matrix**. Figure 6.9 depicts a two-dimensional array as an array of 1-D arrays.



2	1	2	3	4	5	6
1	6	8	4	5	9	0
2	7	2	4	8	0	4
6	3	1	1	8	3	0

Figure 6.9 | A two-dimensional array

6.8.2.1 Declaration of a Two-dimensional Array

The general form of a two-dimensional array declaration is:

<sclass_specifier><type_qualifier><type_modifier>**type identifier[**<row_specifier>**][column_specifier]**<=initialization_list<,...>>**;**

The important points about a two-dimensional array declaration are as follows:

1. The terms enclosed within angular brackets (i.e. <>) are optional and might not be present in a declaration statement. The terms shown in bold are the mandatory parts of a two-dimensional array declaration.
2. A two-dimensional array declaration consists of a type specifier (i.e. element type), an identifier (i.e. name of the array), a row size specifier (i.e. number of rows in an array) and a column size specifier (i.e. number of columns in each row). The size specifiers are enclosed within square brackets. The following declarations of two-dimensional arrays are valid:

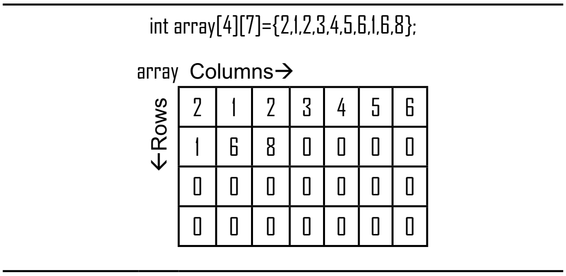
```
int array1[2][3];          //←array1 is an integer array of 2 rows and 3 columns
float array2[5][1];        //←array2 is a float array of 5 rows and 1 column
char array3[3][3];         //←array3 is a character array of 3 rows and 3 columns
```

3. The row size specifier and column size specifier should be a compile time constant expression greater than zero.
4. The specification of a row size and column size is mandatory if an initialization list is not present. If the initialization list is present, the row size specifier can be skipped but it is mandatory to mention the column size specifier.
5. **Initializing elements of two-dimensional arrays:** Like one-dimensional arrays, the elements of two-dimensional arrays can also be initialized by providing an initialization list.

   The syntactic rules about the initialization of elements of a two-dimensional array are as follows:

   a. The number of initializers in the initialization list should be less than or at most equal to the number of elements (i.e. row size × column size) in the array.
   b. The array locations are initialized row-wise. If the number of initializers in the initialization list is less than the number of elements in the array, the array locations that do not get initialized will automatically be initialized to 0 (if it is an integer

array), 0.0 (in case of a floating point array) and '\0' (i.e. null character if it is an array of character type). The mentioned fact is shown in Figure 6.10.

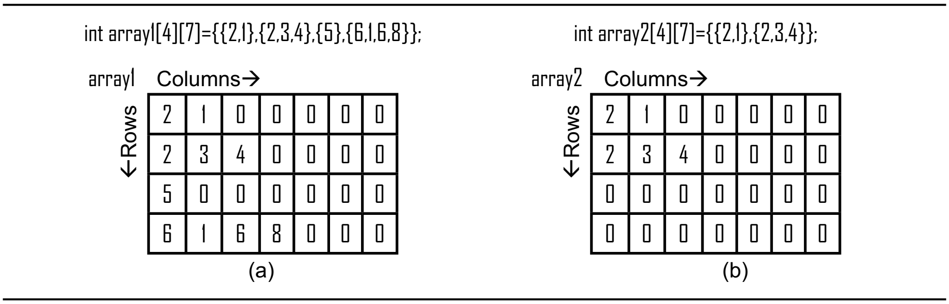


```
int array[4][7]={2,1,2,3,4,5,6,1,6,8};
```

2	1	2	3	4	5	6
1	6	8	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Figure 6.10 | Initialization of a two-dimensional array

c. The initializers in the initialization list can be braced to initialize elements of the individual rows. If the number of initializers within the inner braces is less than the row size, trailing locations of the corresponding row get initialized to 0, 0.0 or '\0', depending upon the element type of the array. The mentioned fact is shown in Figure 6.11.



```
int array1[4][7]={{2,1},{2,3,4},{5},{6,1,6,8}};
```

2	1	0	0	0	0	0
2	3	4	0	0	0	0
5	0	0	0	0	0	0
6	1	6	8	0	0	0

(a)

```
int array2[4][7]={{2,1},{2,3,4}};
```

2	1	0	0	0	0	0
2	3	4	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

(b)

Figure 6.11 | Initialization of individual rows of a two-dimensional array

6.8.2.2 Usage of a Two-dimensional Array

The elements of a two-dimensional array can be accessed by using row and column subscripts. The important points about the usage of a two-dimensional array are as follows:

1. An element of a two-dimensional array can be accessed by writing E1[E2][E3], where E1, E2 and E3 are sub-expressions. One of the sub-expressions E1 or E2 must be of an array type or a pointer type, and the other sub-expressions must be of integral type. Program 6-15 illustrates the use of a subscript operator to access the elements of a two-dimensional array.

Line	Prog 6-15.c	Memory contents	Output window

1
2
3
4
5
6
7
8
9
	//Two-dimensional arrays
	#include<stdio.h>
	main()
	{
	int a[2][3]={2,1,3,2,3,4};
	printf(“Elements of array are:\n”);
	printf(“%d %d %d\n”,a[0][0], a[0][1], a[0][2]);
	printf(“%d %d %d\n”,1[a][0], 1[a][1], 1[a][2]);
	}
		a [0] [1] [2]
		[0] 2 1 3
		[1] 2 3 4
			Elements of array are:
			2 1 3
			2 3 4
			Remarks:
			• The general form of an expression for accessing an element of a 2-D array is E1[E2][E3]
			• In line number 7, E1 is of array type and E2 is of int type
			• In line number 8, E1 is of int type and E2 is of array type
			• Both types of usage are valid
			• The sub-expression E3 must be of integral type and cannot be of array type

Program 6-15 | A program to illustrate the usage of a two-dimensional array

2. The expression E1[E2][E3] is implicitly converted into an equivalent expression of form *(*(E1+E2)+E3). Program 6-16 illustrates this fact.

Line	Prog 6-16.c	Output window

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
	// Subscript operator and equivalent conversion to pointer form
	#include<stdio.h>
	main()
	{
	int a[2][3]={2,1,3,2,3,4};
	printf(“Use of subscript operator:\n”);
	printf(“%d %d %d\n”,a[0][0], a[0][1], a[0][2]);
	printf(“%d %d %d\n”,a[1][0], a[1][1], a[1][2]);
	printf(“Use of pointer expressions:\n”);
	printf(“%d %d %d\n”,*(*(a+0)+0), *(*(a+0)+1), *(*(a+0)+2));
	printf(“%d %d %d\n”, *(*(a+1)+0), *(*(a+1)+1), *(*(a+1)+2));
	printf(“Use of mixed form of expressions:\n”);
	printf(“%d %d %d\n”,*(a[0]+0), *(a[0]+1), *(a[0]+2));
	printf(“%d %d %d\n”, *(a[1]+0), *(a[1]+1), *(a[1]+2));
	}
		Use of subscript operator:
		2 1 3
		2 3 4
		Use of pointer expressions:
		2 1 3
		2 3 4
		Use of mixed form of expressions:
		2 1 3
		2 3 4
		Remark:
		• The expression *(a+i) is equivalent to a[i]. Hence, the expression *(*(a+i)+j) is equivalent to *(a[i]+j), which is further equivalent to a[i][j]

Program 6-16 | A program to illustrate the conversion of a subscript operator into an equivalent pointer form

3. In an expression that involves an array, if the number of subscripts used with the array name is less than the dimensions of the array, the expression refers to an address instead of a value. Program 6-17 illustrates this fact.

Line | **Prog 6-17.c**

```
1   //Number of subscripts and values
2   #include<stdio.h>
3   main()
4   {
5       int a[2][2]={2,1,3,4};
6       printf("No subscript used:\n");
7       printf("%p\n",a);
8       printf("One subscript used:\n");
9       printf("%p %p\n",a[0], a[1]);
10      printf("Two subscripts used:\n");
11      printf("%d %d\n",a[0][0], a[0][1]);
12      printf("%d %d\n",a[1][0], a[1][1]);
13  }
```

Memory contents

a

Indices	[0]	[1]
[0]	2	1
	2234	2236
[1]	3	4
	2238	2240

Output window

```
No subscript used:
234F:2234
One subscript used:
234F:2234 234F:2238
Two subscripts used:
2 1
3 4
```

Remarks:

- When no subscript is used, the expression a refers to the starting address of the first element (i.e. first row) of the array
- When one subscript is used, the expressions a[0] and a[1] refer to the starting address of the first row and the second row, respectively
- When two subscripts are used, the expressions in line numbers 11 and 12 refer to the value of the corresponding array element

Program 6-17 | A program to illustrate the outcome of an expression that uses lesser subscripts than dimensions

6.8.2.2.1 Reading, storing and accessing elements of a 2-D array

The elements can be read and stored in a 2-D array by making use of nested loops. Program 6-18 illustrates the method to read, store and access the elements of a two-dimensional array.

Line | **Prog 6-18.c**

```
// Reading, storing and accessing elements of a two-dimensional array
#include<stdio.h>
main()
{
    int a[10][10], olc, ilc, rows, cols;
    printf("Enter the number of rows(<10):\t");
    scanf("%d",&rows);
    printf("Enter the number of cols(<10)\t");
    scanf("%d",&cols);
    printf("Enter the elements:\n");
    for(olc=0;olc<rows;olc++)
        for(ilc=0;ilc<cols;ilc++)
            scanf("%d",&a[olc][ilc]);    //←Reading and storing elements
    printf("The entered elements were:\n");
    for(olc=0;olc<rows;olc++)
    {
        for(ilc=0;ilc<cols;ilc++)
            printf("%d ",a[olc][ilc]);    //←Accessing elements
        printf("\n");
    }
}
```

Output window

```
Enter the number of rows(<10):    2
Enter the number of cols(<10):    2
Enter the elements:
2
3
3
4
The entered elements were:
2 3
3 4
```

Remarks:

- olc is the outer loop counter
- ilc is the inner loop counter
- To read and store elements in a 2-D array, a nested loop consisting of two loops is required
- The outer loop is for getting the rows, and the inner loop is for getting the elements of a row (i.e. columns)

Program 6-18 | A program to illustrate the method of reading, storing and accessing elements of a two-dimensional array

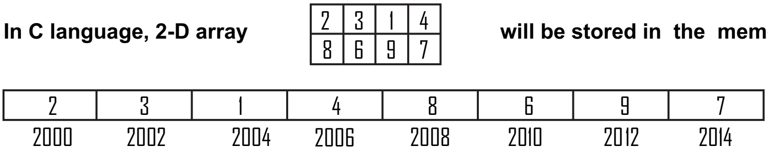
6.8.2.3 Memory Representation of a Two-dimensional Array

A 2-D array can be visualized as a plane, which has rows and columns. Although multi-dimensional arrays are visualized in this way, they are actually stored in the memory, which is linear (i.e. one dimensional). Hence, a multi-dimensional array is to be stored in one dimension. There are two ways of doing this:

1. Row major order of storage
2. Column major order of storage

6.8.2.3.1 Row Major Order of Storage

In **row major order of storage**, the elements of an array are stored row-wise. **In C language, multi-dimensional arrays are stored in the memory by using row major order of storage**. Figure 6.12 shows the row major order of storage.



In C language, 2-D array

2	3	1	4
8	6	9	7

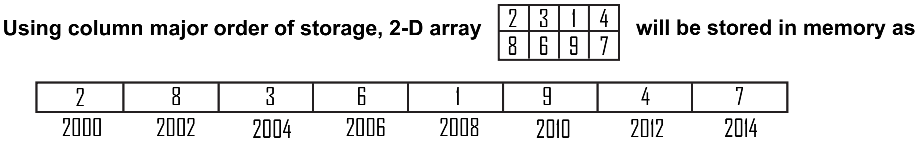
will be stored in the memory as

2	3	1	4	8	6	9	7
2000	2002	2004	2006	2008	2010	2012	2014

Figure 6.12 | Row major order of array storage

6.8.2.3.2 Column Major Order of Storage

In **column major order of storage**, the elements of an array are stored column-wise. Column major order of array storage is used in the languages like FORTRAN, MATLAB, etc. Figure 6.13 shows the column major order of storage.



Using column major order of storage, 2-D array

2	3	1	4
8	6	9	7

will be stored in memory as

2	8	3	6	1	9	4	7
2000	2002	2004	2006	2008	2010	2012	2014

Figure 6.13 | Column major order of array storage

6.8.3 Three-dimensional Arrays

A **three-dimensional array** can be visualized as a cube that has a number of planes. Each plane is a two-dimensional array. Thus, a three-dimensional array is made up of two-dimensional arrays. Figure 6.14 depicts a three-dimensional array as an array of 2-D arrays.

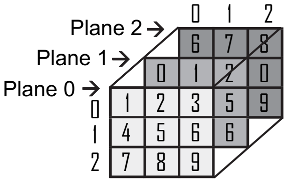


Figure 6.14 | A three-dimensional array

6.8.3.1 Declaration of a Three-dimensional Array

The general form of a three-dimensional array declaration is:

<scspec*><type_qual><type_mod>**type identifier[**<plane_specifier>**][row_specifier][column_specifier]**<=init_list<,...>>;

*- scspec means storage class specifier

The important points about a three-dimensional array declaration are as follows:

1. The terms enclosed within angular brackets (i.e. <>) are optional and might not be present in a declaration statement. The terms shown in **bold** are the mandatory parts of a three-dimensional array declaration.
2. A three-dimensional array declaration consists of a type specifier (i.e. element type), an identifier (i.e. name of array), a plane size specifier, a row size specifier and a column size specifier. The size specifiers are enclosed within the square brackets (i.e. []).
3. The plane size specifier, row size specifier and column size specifier should be a compile time constant expression greater than zero.
4. The specification of all size specifiers is mandatory if the elements of an array are not explicitly initialized. If an initialization list is present, the plane size specifier can be skipped but it is mandatory to mention the row size specifier and column size specifier. The general rule is **'While declaring n-D arrays, even if initialization list is present, it is mandatory to specify (n-1) fastest varying specifiers'**. In case of two-dimensional arrays, the column size specifier varies faster as compared to the row size specifier. In case of three-dimensional arrays, column size specifier and row size specifier vary faster than a plane size specifier.
5. **Initializing elements of three-dimensional arrays:** The elements of a three-dimensional array can be initialized in the same way as the elements of a two-dimensional array are initialized, i.e. by providing an initialization list.

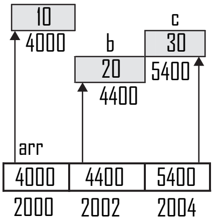
6.9 Array of Pointers

An **array of pointers** is a collection of addresses. The addresses in an array of pointers could be the addresses of isolated variables or the addresses of array elements or any other addresses. The only constraint is that all the pointers in an array must be of the same type. Program 6-19 illustrates the use of array of pointers.

Prog 6-19.c

```
// Array of pointers
#include<stdio.h>
main()
{
    int a=10,b=20, c=30;
    int* arr[3]={&a, &b, &c};
    printf("The values of variables are:\n");
    printf("%d %d %d\n",a,b,c);
    printf("%d %d %d\n",*arr[0],*arr[1],*arr[2]);
}
```

Memory contents



Output window

```
The values of variables are:
10 20 30
10 20 30
```

Remarks:

- arr is an array of integer pointers and holds the addresses of variables a, b and c
- All the variables are of the same type

Program 6-19 | A program to illustrate the use of array of pointers

6.10 Pointer to a Pointer

A pointer that holds the address of another pointer variable is known as a **pointer to a pointer**. Such a pointer is said to exhibit multiple levels of indirection. There can be many levels of indirection in a single declaration statement. Consider the code snippet in Program 6-20.

Prog 6-20.c

```
// Pointer to a pointer
#include<stdio.h>
main()
{
    int i=10;
    int *p1=&i;        //←Pointer to int
    int **p2=&p1;   //←Pointer to pointer to int
    int ***p3=&p2; //←Pointer to pointer to pointer to int
    int ****p4=&p3;//←.................concept scales up
    int *****p5=&p4;
    int ******p6=&p5;
    int *******p7=&p6;
    int ********p8=&p7;
    int *********p9=&p8;
    int **********p10=&p9;
    int ***********p11=&p10;
    int ************p12=&p11;
    printf("The values of variables are:\n");
    printf("%d\t",*p1);
    printf("%d\t",**p2);
    printf("%d\t",***p3);
    printf("%d\t",****p4);
    printf("%d\t",*****p5);
    printf("%d\t",******p6);
    printf("%d\t",*******p7);
    printf("%d\t",********p8);
    printf("%d\t",*********p9);
    printf("%d\t",**********p10);
    printf("%d\t",***********p11);
    printf("%d\t",************p12);
}
```

Output window

```
The values of variables are:
10 10 10 10 10 10 10 10 10 10 10 10
```

Remarks:

- The ANSI C standard says that all compilers must handle at least 12 levels of indirection
- Some compilers may support more levels of indirection
- Two levels of indirection are common
- Level of indirection higher than two becomes difficult to understand and visualize
- In an expression, if the number of indirection operators used to dereference a pointer is less than the number of punctuators (*) used to declare the pointer, then the pointer will not be completely dereferenced and the expression refers to an address
- The number of indirection operators required to completely dereference a pointer is equal to the number of punctuators (*) used while declaring it
- For example, in the mentioned code the expression *p2 refers to an address, i.e. address of p1. In the expression **p2, p2 is completely dereferenced and refers to the value of i, i.e. 10

Program 6-20 | A program to illustrate the use of multi-level pointers

6.11 Pointer to an Array

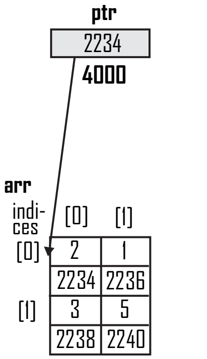
It is possible to create a pointer that points to a complete array instead of pointing to the individual elements of an array or isolated variables. Such a pointer is known as a **pointer to an array**. The following declaration statements declare such pointers:

```
int (*p1)[5];          //←p1 is a pointer to an array of 5 integers
int (*p2)[2][2];       //←p2 is a pointer to an integer array of 2 rows and 2 columns
int (*p3)[2][3][4];    //←p3 is a pointer to an integer array having 2 planes. Each plane
                       //←has 3 rows and 4 columns
```

> While declaring pointer to an array, parentheses, i.e. ( ) are used because [] binds more tightly than *. If parentheses are not used, the declaration **int *p1[5];** declares p1 as an array of 5 integer pointers. In the said declaration, p1 becomes an array instead of becoming a pointer because [] binds p1 more tightly than *. To make p1 a pointer to an array of 5 integers, write it as **int(*p1)[5]**. In this declaration, parentheses are used to bind p1 with *.

Program 6-21 illustrates the use of a pointer to an array.

Line	Prog 6-21.c	Memory contents	Output window

1
2
3
4
5
6
7
8
9
10
11
12
	// Pointer to an array
	#include<stdio.h>
	main()
	{
	    int arr[2][2]={{2,1},{3,5}};
	    int (*ptr)[2]=arr;
	    printf("Address of row 1 is %p\n",arr[0]);
	    printf("Address of row 2 is %p\n",ptr+1);
	    printf("1st element of row 1 is %d\n", arr[0][0]);
	    printf("1st element of row 2 is %d\n",ptr[1][0]);
	
	}
		 ptr
		2234
		4000
		arr
		indi-ces
		[0]
		[1]
		[0]
		2
		1
		2234
		2236
		[1]
		3
		5
		2238
		2240
			Address of row 1 is 234F:2234
			Address of row 2 is 234F:2238
			1st element of row 1 is 2
			1st element of row 2 is 3
			Remarks:
			• arr refers to the address of the first element of the array
			• Elements of a 2-D array are 1-D arrays
			• Thus, arr refers to the address of first 1-D array of two integers (i.e. first row)
			• The type of arr is int(*)[2]
			• Type of ptr is int(*)[2]
			• ptr is initialized with the starting address of row 1
			• ptr+1 will point to the next row
			• As types of arr and ptr are same and both refer to the same address, the expression ptr[1][0] is equivalent to the expression arr[1][0]

Program 6-21 | A program that illustrates the creation and usage of a pointer to an array

6.12 Advantages and Limitations of Arrays

The direct indexing supported by arrays is their biggest advantage. **Direct indexing** means the time required to access any element in an array of any dimension is almost the same irrespective of its location in the array.

The limitations of arrays are as follows:

1. The memory to an array is allocated at the compile time.
2. Arrays are static in nature. The size of an array cannot be expanded or cannot be squeezed at the run time.
3. The size of an array has to be kept big enough to accommodate the worst cases. Therefore, memory usage in case of arrays is inefficient.

6.13 Searching

There are many situations where we want to find out whether a particular item is present in a list or not. For instance, in a given voter list of a colony a person may search his name to ascertain whether he is a valid voter or not. For similar reasons, passengers look for their names in the railway reservation lists.

Note: System programs extensively search symbols, literals, mnemonics, '*compiler and assembler*' directives, etc.

In fact, search is an operation in which a given list is searched for a particular value. The location of the searched element is informed. *Search* can be precisely defined an activity of looking for a value or item in a list.

A list can be searched *sequentially* wherein the search for the data item starts from the beginning and continues till the end of the list. This simple method of search is also called **linear search.** It may be noted that for a list of size 1,000, the worst case is 1,000 comparisons.

Let us consider a situation wherein we are interested in searching a list of numbers called 'numList' for an element having value equal to the contents of a variable called val. It is desired that the location of the element, if found, be displayed.

The list of numbers can be comfortably stored in an array called numList of type int. To find the element, the list would be travelled in such a manner that each visited element would be compared to the variable 'val'. If the match is found, then the location of the corresponding position would be stored in a variable called Pos. As the number of elements in the list is known, For-loop would be used to travel the array.

```
Algorithm searchList()
{
    step
        1. read numList
        2. read Val
        3. Pos = −1 'initialize Pos to a non-existing position'
        4. for (i = 0; i < N; i++)
            {
                4.1 if (val == numList[i])
                        Pos = i;
            }
        5. if (Pos !5 −1)
                print Pos.
}
```

The 'C' implementation of the above algorithm is given below:

```
/* This program searches a given value called Val in a list called numList */
#include<stdio.h>
void main()
{
        int numList[20];
        int N;              /* Size of the list*/
        int Pos, val, i;
        printf ("\n Enter the size of the List");
        scanf ("%d", &N);
        printf ("\n Enter the elements one by one");
        for (i = 0; i < N; i++)
        {
            scanf ("%d", &numList[i]);
        }
        printf ("\n Enter the value to be searched");
        scanf ("%d", &val);
                    /* Search the element and its position in the list*/
        Pos = –1;
        for (i = 0; i < N; i++)
        {
            if (val == numList[i])
                {
                    Pos = i;
                    break;
                }
                    /* The element is found – come out of loop*/
        }
        if (Pos != –1)
                printf ("\n The element found at %d location", Pos);
        else
                printf ("\n Search Failed");
}
```

Example 1: Write a program that finds the second largest element in a given list of N numbers.

Solution: The most basic solution for this problem is that if there are two elements in the list then the smaller of the two will be the second largest. In this case, we would set the second largest number to '–9999', a value not possible in the list.

In given problem, the list will be searched linearly for the required second largest number. Two variables called firstLarge and secLarge would be employed to store the first and second largest numbers. The following algorithm will be used:

```
Algorithm SecLarge()
{
        step
            1. read num;
            2. firstLarge = num;
            3. secLarge = -9999;
            4. for (i = 2; i < = N; i++)
                {
                    4.1 read num;
                    4.2 if (firstLarge < num)
                        {secLarge = firstLarge;
                        firstLarge = num;
                }
                            else
                            if (secLarge < num)
                                secLarge = num;
                }
            4. prompt "Second Large =";
            5. write secLarge;
}
```

The equivalent program is given below:

```
/* This program finds the second largest in a given list of numbers */
#include <stdio.h>
int Num, firstLarge, secLarge;
int N, i;
void main()
{
   printf ("\n Enter the size of the list");
   scanf ("%d", & N);
   printf ("\n Enter the list one by one");
   scanf ("%d", &Num);
   firstLarge = Num;
   secLarge = −9999;
   for (i = 2; i < = N; i++)
   {
         scanf ("%d", &Num);
         if (firstLarge < Num)
         {secLarge = firstLarge;
         firstLarge = Num;
         }
         else
            if (secLarge < Num)
            secLarge = Num;
   }
         printf ("\n Second Large = %d", secLarge);
}
```

The above discussed search through a list, stored in an array, has the following characteristics:

- The search is linear.
- The search starts from the first element and continues in a sequential fashion from element to element till the desired entry is found.
- In the worst case, a total number of N steps need to be taken for a list of size N.

Thus, the linear search is slow and to some extent inefficient. In special circumstances, faster searches can be applied.

For instance, binary search is a faster method as compared to linear search. It mimics the process of searching a name in a directory wherein one opens a page in the middle of the directory and examines the page for the required name. If it is found, the search stops; otherwise, the search is applied either to first half of the directory or to the second half.

6.13.1 Binary Search

If a list is already sorted, then the search for an entry (say Val) in the list can be made faster by using '*divide and conquer*' technique. The list is divided into two halves separated by the middle element as shown in Figure 6.15.

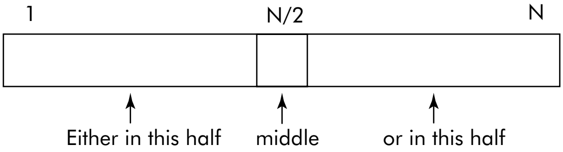


Figure 6.15 | Binary search

The binary search follows the following steps:

Step

- The middle element is tested for the required entry. If found, then its position is reported else the following test is made.
- If Val < middle, search the left half of the list, else search the right half of the list.
- Repeat step 1 and 2 on the selected half until the entry is found otherwise report failure.

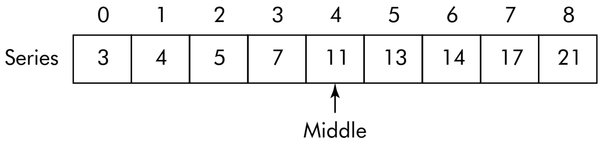
This search is called binary because in each iteration, the given list is divided into two (i.e., binary) parts. Therefore, in next iteration the search becomes limited to half the size of the list to be searched. For instance, in first iteration the size of the list is N which reduces to almost N/2 in the second iteration and so on.

Let us consider a sorted list stored in an array called 'Series' given in Figure 6.16.

	0	1	2	3	4	5	6	7	8
Series	3	4	5	7	11	13	14	17	21

Figure 6.16 | The 'Series' containing nine elements

Suppose we desire to search the *Series* for a value (say 14) and its position in it. Binary search begins by looking at the middle value in the Series. The middle index of the array is approximated by averaging the first and last indices and truncating the result, i.e., (0 + 8)/2 = 4. Now, the content of the fourth location in Series happens to be '11' as shown in Figure 6.17. Since the value we are looking for (i.e., 14) is greater than 11, the middle value, it may be present in the right half (Series [5] to Series [8]).



	0	1	2	3	4	5	6	7	8
Series	3	4	5	7	11	13	14	17	21

Figure 6.17 | The middle value in the Series (1st step)

Now the middle of the right half is approximated i.e., (5 + 8)/2 = 6. We find that the desired element exists at the middle of the right half, i.e., Series [6] = 14 as shown in Figure 6.18.

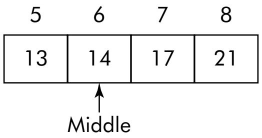


Figure 6.18 | The middle value in the Series (2nd step)

It may be noted that the desired element has been found only in two steps. Thus, it is a much faster method as compared to the linear search. For instance, a list of 1,000 sorted elements would require 10 comparisons to search the entire list. An algorithm for this method is given below.

In this algorithm, we would employ a Boolean variable called flag. The flag will indicate the presence or absence of the element being searched.

```
Algorithm binSearch ()
  {
     Step
        1. First = 0;
        2. Last = N − 1;
        3. Pos =-1;
        4. Flag = false;
        5. While (First < = Last and Flag == false)
        {
              5.1 Middle = (first + last) div 2
                    if (Series [middle] == Val)
                       {Pos= middle;
                       flag = true;
                       break from the loop;
        }
                    else
                       if (Series[middle] < Val) First = middle + 1;
                       else Last = middle − 1;
        }
        6. if (flag == true)
              prompt "The value found at";
              write pos;
           else prompt "The value not found";
  }
```

It may be noted that 'div' operator has been used to indicate that it is an integer division. The integer division will truncate the results to the nearest integer. If the desired value is found, then the flag is set to true and the *while loop* terminates; otherwise, a stage arrives when first becomes greater than the last, indicating the failure of the search. Thus, the variables First and Last keep track of the lower and the upper bounds of the array, respectively.

Example 2: Write a program that uses binary search to search a given value called Val in a list of N numbers called Series.

Solution: The Algorithm binSearch() discussed above is used to write the required program.

```
/* This program uses binary search to find a given value called val in a list of N numbers */
#include <stdio.h>
#define true 1
#define false 0
void main()
{
    int First;
    int Last;
    int Middle;
    int Series[20]; /*The list of N sorted numbers*/
    int Val;
    int flag; /*The value to be searched */
    int N, Pos, i;

    printf ("\n Enter the size of the list");
    scanf ("%d", & N);

    printf ("\n Enter the sorted list one by one");

    for (i = 0; i< N; i++)
    {
        scanf ("%d", &Series[i]);
    }
    printf ("\n Enter the number to be searched");
    scanf ("%d", & Val);
                                    /* BIN SEARCH begins */
    Pos = −1;                       /* Non-existing position */
    flag = false;                   /* Assume search failure */

    First = 0;
    Last = N − 1;

        while ((First <= Last) && (flag == false))
        {
            Middle = (First + Last)/2;
            if (Series [Middle] == Val)
            {Pos = Middle;
                flag = true;
                break;
        }
        else
            if (Series[Middle] < Val)
            First = Middle + 1;
        else
            Last = Middle − 1;
        }
        if (flag == true)
            printf ("\n The value found at %d", Pos);
        else
            printf ("\n The value not found");
    }
```

Binary search through a list, stored in an array, has the following characteristics:

- The list must be sorted, i.e., ordered.
- It is faster as compared to the linear search.
- A list with large number of elements would increase the total execution time, the reason being that the list must be ordered which requires extra effort.

6.14 Sorting

It is an operation in which all the elements of a list are arranged in a predetermined order. The elements can be arranged in a sequence from smallest to largest such that every element is less than or equal to its next neighbour in the list. Such an arrangement is called *ascending order*. Assuming an array called List containing N elements, the ascending order can be defined by the following relation:

```
List[i] <= List [i + 1], 0 < i < N – 1
```

Similarly in descending order, the elements are arranged in a sequence from largest to smallest such that every element is greater than or equal to its next neighbour in the list. The descending order can be defined by the following relation:

```
List[i] >= List [i + 1] , 0 < i < N – 1
```

It has been estimated that in a data processing environment, 25 per cent of the time is consumed in sorting of data. Many sorting algorithms have been developed. Some of the most popular sorting algorithms that can be applied to arrays are *in-place* sort algorithms. An *in-place* algorithm is generally a *comparison-based* algorithm that stores the sorted elements of the list in the same array as occupied by the original one. A detailed discussion on sorting algorithms is given in subsequent sections.

6.14.1 Selection Sort

It is a very simple and natural way of sorting a list. It finds the smallest element in the list and exchanges it with the element present at the head of the list as shown in Figure 6.19.

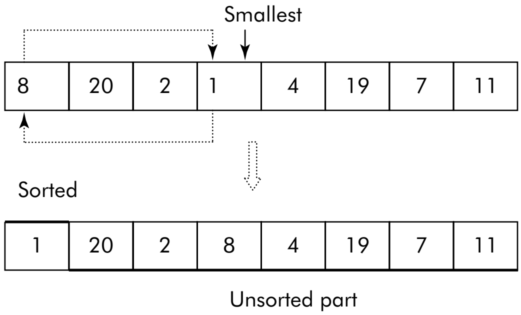


Figure 6.19 | Selection sort (first pass)

It may be noted from Figure 6.19 that initially, whole of the list was unsorted. After the exchange of smallest with the element on the head of the list, the list is divided into two parts: sorted and unsorted.

Now the smallest is searched in the unsorted part of the list, i.e., '2' and exchanged with the element at the head of unsorted part, i.e., '20' as shown in Figure 6.20.

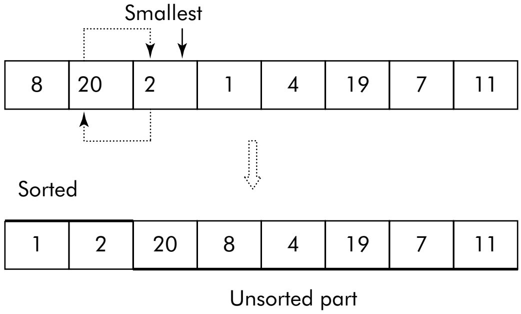


Figure 6.20 | Selection sort (second pass)

This process of selection and exchange (i.e., a pass) continues in this fashion until all the elements in the list are sorted (see Figure 6.21). Thus, in selection sort, two steps are important—*selection* and *exchange.*

From Figures 6.20 and 6.21, it may be observed that it is a case of nested loops. The outer loop is required for passes over the list and the inner loop for searching smallest element within the unsorted part of the list. In fact, for N number of elements, N − 1 passes are made.

An algorithm for selection sort is given below. In this algorithm, the elements of a list stored in an array called LIST[N] are sorted in ascending order. Two variables called Small and Pos are used to locate the smallest element in the unsorted part of the list. Temp is the variable used to interchange the selected element with the first element of the unsorted part of the list.

```
Algorithm SelSort()
 {
    Step
       1. For I = 1 to N – 1 /* Outer Loop */
       {
          1.1 small = List [I];
          1.2 Pos = I;
          1.3 For J = I + 1 to N /* Inner Loop */
          {
             1.3.1 if (List [J] < small)
             {
                small = List[J];
                Pos = J; /* Note the position of the smallest*/
             }
          }
          1.4 Temp = List [I]; /*Exchange smallest with the Head */
          1.5 List [I] = List [Pos];
```

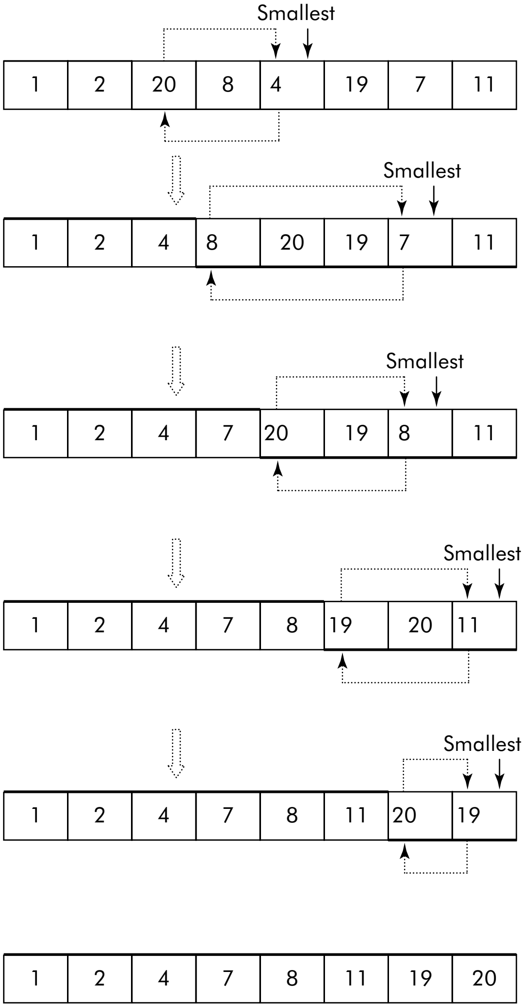


Figure 6.21 | Selection sort

```
            1.6 List [Pos] = Temp;
        }
        2. Print the sorted list
    }
```

Example 3: Given is a list of N randomly ordered numbers. Write program that sorts the list in ascending order by using selection sort.

Solution: The required program is given below:

In this program, the elements of a list are stored in an array called List. The elements are sorted using above given Algorithm selSort(). Two variables small and pos have been used to locate the smallest element in the unsorted part of the list. Temp is a variable used to interchange the selected element with the first element of the unsorted part of the list. With each step, the unsorted part becomes smaller. The process is repeated till all the elements are sorted.

```
/* This program sorts a list by using selection sort */
#include <stdio.h>
main()
{
   int list [10];
   int small, pos, N, i, j, temp;
   printf ("\n Enter the size of the list:");
   scanf ("%d", & N);

   printf ("\n Enter the list: ");
   for (i = 0; i < N; i++)
   {
      printf ("\n Enter Number:");
      scanf ("%d", &list[i]);
   }
                                       /* Sort the list */

   for (i = 0; i < N – 1; i++)
   {
      small = list[i];

      pos = i;
                                       /* Find the smallest of the unsorted list */
      for (j = i+1; j < N; j++)
      {
         if (small > list [j])
         {
            small = list [j];
            pos = j;
         }
      }
                                       /* Exchange the small with the
                                       first element of unsorted list */
      temp = list [i];
      list [i] = list [pos];
      list [pos] = temp;
   }
   printf ("\n The sorted list ...");
   for (i = 0; i < N; i++)
   printf ("%d ", list[i]);
}
```

6.14.2 Bubble Sort

It is also a very simple sorting algorithm. It proceeds by looking at the list from left to right. Each adjacent pair of elements is compared. Whenever a pair is found not to be in order, the elements are exchanged. Therefore after the first pass, the largest element bubbles up to the right end of the list. A trace of first pass on a list of numbers is shown in Figure 6.22.

It may be noted that after the pass is over, the largest element in the list (i.e., 20) has bubbled up to the end of the list and six exchanges were made. Now the same process can be repeated for the list for second pass as shown in Figure 6.23.

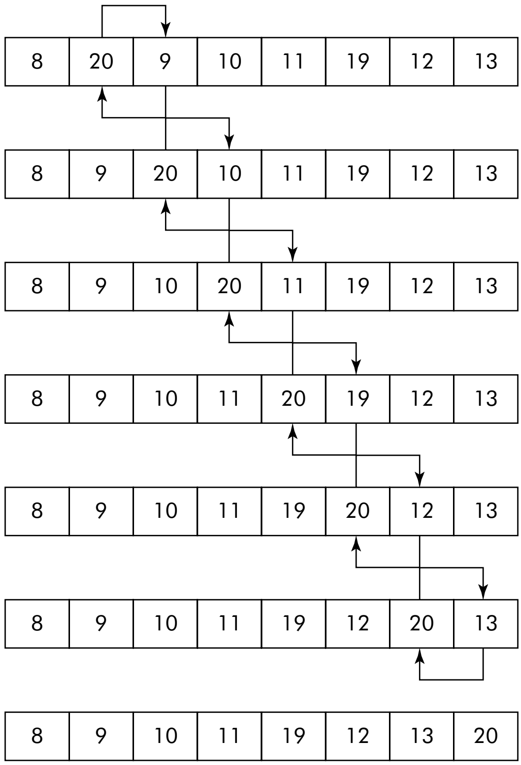


Figure 6.22 | First pass of bubble sort on a list of numbers

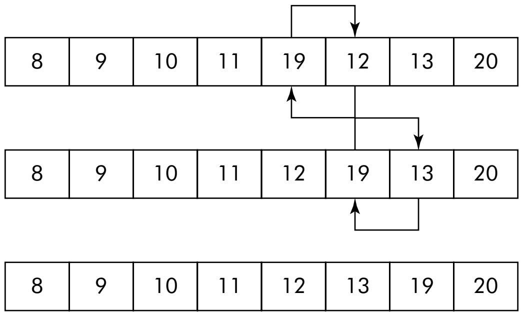


Figure 6.23 | Second pass of the bubble sort

We observe that after the second pass is over, the list has become sorted and only two exchanges were made. Now a point worth noting is as to how and when it will be decided that the list has become sorted. The simple criteria would be to check whether or not any exchange(s) has been made during the current pass. If 'yes', then the list is not yet sorted otherwise if it is 'no', then it can be decided that the list has just become sorted.

An algorithm for bubble sort is given below. In this algorithm, the elements of a list, stored in an array called List[N], are sorted in an ascending order. The algorithm uses two loops—the outer while loop and inner For loop. The inner For loop makes a pass on the list. If during the pass, any exchange(s) is made then it is recorded in a variable called flag, i.e., flag is set to false. The outer while loop keeps track of the flag. As soon as the flag informs that no exchange(s) took place during the current pass indicating that the list is now sorted, the algorithm stops.

```
Algorithm bubbleSort()
{
    Step
    1. Flag = false;
    2. While (Flag == false)
    {
        2.1 Flag = true;
        2.2 For j = 0 to N − 2
        {
            2.2.1 if (List [J] > List [J + 1])
            {
                temp = List[J];
                List[J] = List [J + 1];
                List [J + 1] = temp;
                Flag=false;
            }
        }
    }
    3. Print the sorted list
    4. Stop
}
```

It may be noted that the algorithm stops as soon as the list becomes sorted. Thus, 'bubble sort' is a very useful algorithm when the list is almost sorted i.e., only a very small percentage of elements are out of order.

Example 4: Given is a list of N randomly ordered numbers. Write a program that sorts the list in ascending order by using bubble sort.

Solution: The required program is given below:

In this program, the elements of a list are stored in an array called List. The elements are sorted using above given algorithm bubbleSort().

```
/ *This program sorts a given list of numbers in ascending order, using bubble sort */
#include <stdio.h>
#define N 20
```

```
#define true 1
#define false 0
void main()
{
    int List[N];
    int flag;
    int size;
    int i, j, temp;
    int count; /* counts the number of passes*/
    printf (“\n Enter the size of the list (< 20)”);
    scanf (“%d”, &size);
    printf (“\n Enter the list one by one”);
    for (i = 0; i< size; i++)
    {
        scanf (“%d”, &List[i]);
    }
                        /* Sort the list by bubble sort */
    flag = false;
    count = 0;
    while (flag == false)
    {
        flag = true; /* Assume no exchange takes place*/
        count++;
        for (j =0; j< size−1; j++)
        {
            if (List[j] > List[j+1])
            {           /* Exchange the contents */
                temp = List[j];
                List[j] = List[j + 1];
                List[j + 1] = temp;
                flag = false;    /* Record the exchange operation*/
            }
        }
    }
                        /* Print the sorted list*/
            printf (“\n The sorted list is ....”);
            for (i = 0; i< size; i++)
                printf (“%d ”, List[i]);
            printf (“\n The number of passes made = %d”, count);
}
```

It may be noted that the above program has used a variable called count that counts the number of passes made while sorting the list. The test runs conducted on the program have established that the program is very efficient in case of almost sorted list of elements. In fact, it takes only one scan to establish that the supplied list is already sorted.

Note: Bubble sort is also called a **sinking sort** meaning that the elements sink down in the list to their proper position.

6.14.3 Insertion Sort

This algorithm mimics the process of arranging a pack of playing cards. In the pack of cards, the first two cards are put in correct relative order. The third card is inserted at correct place relative to the first two. The fourth card is inserted at the correct place relative to the first three, and so on.

Given a list of numbers, it divides the list into two part—sorted part and unsorted part. The first element becomes the sorted part and the rest of the list becomes the unsorted part as shown in Figure 6.24. It picks up one element from the front of the unsorted part and inserts it at its proper position in the sorted part of the list. This insertion action is repeated till the unsorted part is exhausted.

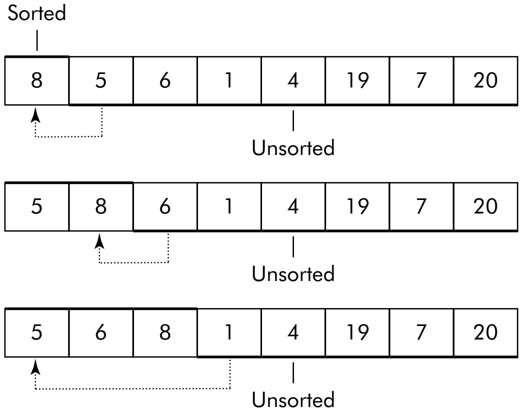


Figure 6.24 | Insertion sort

It may be noted that the insertion operation requires following steps:
Step

1. Scan the sorted part to find the place where the element, from unsorted part, can be inserted. While scanning, shift the elements towards right to create space.
2. Insert the element, from unsorted part, into the created space.

The algorithm for the insertion sort is given below. In this algorithm, the elements of a list stored in an array called List[N] are sorted in an ascending order. The algorithm uses two loops—the outer For loop and inner while loop. The inner while loop shifts the elements of the sorted part by one step to right so that proper place for incoming element is created. The outer For loop inserts the element from unsorted part into the created place and moves to next element of the unsorted part.

```
Algorithm insertSort()
{
   Step
      1. For I = 2 to N /* The first element becomes the sorted part */
         {
         1.1 Temp = List [I]; /* Save the element from unsorted part into temp */
         1.2 J = I – 1;
         1.3 While (Temp < = List [J] AND J > = 0)
```

```
                {
                    List[J + 1] = List[J];     /* Shift elements towards right */
                    J = J - 1;
                }
            1.4 List [J + 1] = Temp;
            }
        2. Print the list
        3. Stop
}
```

Example 5: Given is a list of N randomly ordered numbers. Write a program that sorts the list in ascending order by using insertion sort.

Solution: The required program is given below:

```
/*This program sorts a given list of numbers in ascending order using insertion sort */
#include <stdio.h>
#define N 20
void main()
{
    int List[N];
    int size;
    int i, j, temp;
    printf ("\n Enter the size of the list (< 20)");
    scanf ("%d", &size);
    printf ("\n Enter the list one by one");
    for (i = 0; i < size; i++)
    {
        scanf ("%d", &List[i]);
    }
                        /* Sort the list by Insertion sort */
    for (i=1; i<size; i++)
    {
        temp = List[i]; /* Pick and save the first element of the unsorted part*/
        j= i - 1;
        while ((temp < List[j])&& (j>=0)) /* Scan for proper place */
        {
            List[j + 1] = List[j];
            j = j - 1;
        }
            List[j+1] = temp; /* Insert the element at the proper place */
    }
    /* Print the sorted list*/
    printf ("\n The sorted list is ....");
    for (i = 0; i< size; i++)
    {
        printf ("%d ", List[i]);
    }
}
```

6.14.4 Merge Sort

This method uses following two concepts:

- If a list is empty or it contains only one element, then the list is already sorted. A list that contains only one element is also called **singleton**.
- It uses the old proven technique of 'divide and conquer' to recursively divide the list into sub-lists until it is left with either empty or singleton lists.

In fact, this algorithm divides a given list into two almost equal sub-lists. Each sub-list, thus obtained, is recursively divided into further two sub-lists and so on till singletons or empty lists are left as shown in Figure 6.25.

Since the singletons and empty lists are inherently sorted, the only step left is to merge the singletons into sub-lists containing two elements each (see Figure 6.26) which are further merged into sub-lists containing four elements each and so on. This merging operation is recursively carried out till a final merged list is obtained as shown in Figure 6.26.

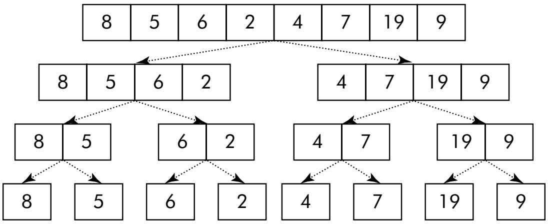


Figure 6.25 | First step of merge sort (divide)

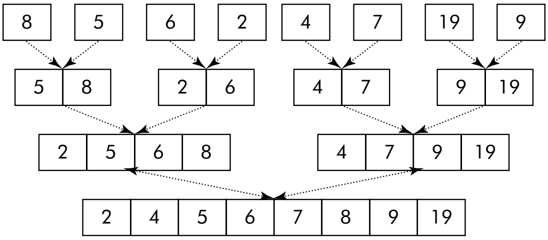


Figure 6.26 | Second step of merge sort (merge)

Note: The merge operation is a time consuming and slow operation. The working of merge operation is discussed in the next section.

Merging of lists It is an operation in which two ordered lists are merged into a single ordered list. The merging of two lists PAR1 and PAR2 can be done by examining the elements at the head of the two lists and selecting the smaller of the two. The smaller element is then stored into a third list called mergeList. For example, consider the lists PAR1 and PAR2 given in Figure 6.27. Let Ptr1, Ptr2, and Ptr3 variables point to the first locations of lists PAR1, PAR2, and PAR3, respectively. The comparison of PAR1[Ptr1] and PAR2[Ptr2] shows that the element of PAR1 (i.e., '2') is smaller. Thus, this element will be placed in the mergeList as per the following operation:

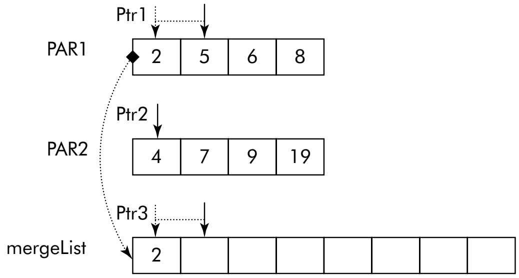


Figure 6.27 | Merging of lists (first step)

```
mergeList[Ptr3] = PAR1[Ptr1];
Ptr1++;
Ptr3++;
```

Since an element from the list PAR1 has been taken to mergeList, the variable Ptr1 is accordingly incremented to point to the next location in the list. The variable Ptr3 is also incremented to point to next vacant location in mergeList.

This process of comparing, storing and shifting is repeated till both the lists are merged and stored in mergeList as shown in Figure 6.28.

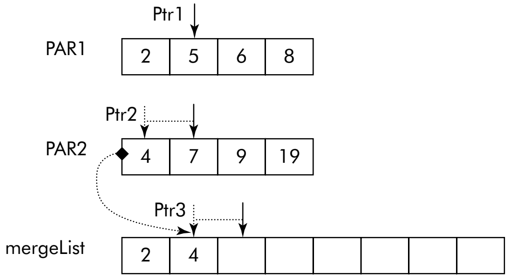


Figure 6.28 | Merging of lists (second step)

It may be noted here that during this merging process, a situation may arise when we run out of elements in one of the lists. We must, therefore, stop the merging process and copy rest of the elements from unfinished list into the final list.

The algorithm for merging of lists is given below. In this algorithm, the two sub-lists are part of the same array List[N]. The first sub-list is stored in locations List[lb] to List[mid] and the second sub-list is stored in locations List [mid+1] to List [ub] where lb and ub mean lower and upper bounds of the array, respectively.

```
Algorithm merge (List, lb, mid, ub)
 {
        Step
           1. ptr1 = lb; /* index of first list */
           2. ptr2 = mid; /* index of second list */
           3. ptr3 = lb; /* index of merged list */
```

```
            4. while ((ptr1 <mid) && ptr2 < = ub) /* merge the lists */
               {
               4.1 if (List[ptr1] < = List [ptr2])
                  {mergeList [ptr3] = List[ptr1]; /* element from first list is taken */
                  ptr1++; /* move to next element in the list*/
                  ptr3++;
               }
               4.2 else
                     {mergeList [ptr3] = List[ptr2]; /* element from second list is taken*/
                  ptr2++; /* move to next element in the list*/
                  ptr3++;
               }
               }
            5. while (ptr1 < mid) /* copy remaining first list */
               {
               5.1 mergeList [ptr3] = List[ptr1];
               5.2 ptr1++;
               5.3 ptr3++;
               }
            6. while (ptr2 <= ub) /* copy remaining second list */
               {
               6.1 mergeList [ptr3] = List[ptr2];
               6.2 ptr2++;
               6.3 ptr3++;
               }
            7. for (i = lb; i<ptr3; i++) /* copy merged list back into original list */
               7.1 List[i] = mergeList[i];
            8. Stop
  }
```

It may be noted that an extra temporary array called mergeList is required to store the intermediate merged sub-lists. The contents of the mergeList are finally copied back into the original list.

The algorithm for the merge sort is given below. In this algorithm, the elements of a list stored in an array called List[N] are sorted in an ascending order. The algorithm has two parts—mergeSort and merge. The merge algorithm, given above, merges two given sorted lists into a third list, which is also sorted. The mergeSort algorithm takes a list and stores into an array called List[N]. It uses two variables lb and ub to keep track of lower and upper bounds of list or sub-lists as the case may be. It recursively divides the list into almost equal parts till singletons or empty lists are left. The sub-lists are recursively merged through merge algorithm to produce final sorted list.

```
Algorithm mergeSort (List, lb, ub)
   {
      Step
         1. if (lb < ub)
         {
            1.1 mid = (lb + ub)/2; /* divide the list into two sub-lists */
            1.2 mergeSort (List, lb, mid); /* sort the left sub-list */
            1.3 mergeSort (List, mid +1, ub); /* sort the right sub-list */
            1.4 merge(List, lb,mid+1,ub); /* merge the lists */
```

```
        }
        2. Stop
    }
```

Example 6: Given is a list of N randomly ordered numbers. Write a program that sorts the list in ascending order by using merge sort.

Solution: The required program uses both the algorithms—mergeSort() and merge().

```
/* This program sorts a given list of numbers in ascending order using merge sort */
#include <stdio.h>
#include <conio.h>
#define N 20
void mergeSort (int List[], int lb, int ub);
void merge (int List[], int lb, int mid, int ub);
void main()
{
    int List[N];
    int i, size;
    int mid;
    printf ("\n Enter the size of the list (< 20)");
    scanf ("%d", &size);
    printf ("\n Enter the list one by one");
    for (i=0; i< size; i++)
    {
        scanf ("%d", &List[i]);
    }
                        /* Sort the list by merge sort */
    mergeSort (List,0,size – 1);
    printf ("\n The sorted list is ....");
    for (i = 0; i< size; i++)
    {
        printf ("%d ", List[i]);
    }
}
void mergeSort (int List[], int lb, int ub)
{
    int mid;
    if (lb < ub)
    {
        mid = (lb + ub)/2;
        mergeSort (List, lb, mid);
        mergeSort (List, mid + 1, ub);
        merge(List, lb, mid + 1,ub);
    }
}
void merge (int List[], int lb, int mid, int ub)
{
    int mergeList[20];
    int ptr1, ptr2, ptr3;
```

```
        int i;
    ptr1=lb;
    ptr2=mid;
    ptr3=lb;
    while ((ptr1 <mid) && ptr2 < = ub)
    {
        if (List[ptr1] < = List [ptr2])
        {mergeList [ptr3] = List[ptr1];
        ptr1++;
        ptr3++;
        }
        else
        {mergeList [ptr3] = List[ptr2];
        ptr2++;
        ptr3++;
        }
    }
    while (ptr1 < mid)
    {mergeList [ptr3] = List[ptr1];
    ptr1++;
    ptr3++;
    }

    while (ptr2 <= ub)
    {mergeList [ptr3] = List[ptr2];
    ptr2++;
    ptr3++;
    }
    for (i = lb; i<ptr3; i++)
    {
        List[i] = mergeList[i];
    }
}
```

6.14.5 Quick Sort

This method also uses the technique of 'divide and conquer'. On the basis of a selected element (pivot) from of the list, it partitions the rest of the list into two parts—a sub-list that contains elements less than the pivot and other sub-list containing elements greater than the pivot. The pivot is inserted between the two sub-lists. The algorithm is recursively applied to the sub-lists until the size of each sub-list becomes 1, indicating that the whole list has become sorted.

Consider the list given in Figure 6.29. Let the first element (i.e., 8) be the pivot. Now the rest of the list can be divided into two parts—a sub-list that contains elements less than '8' and the other sub-list that contains elements greater than '8' as shown in Figure 6.29.

Now this process can be recursively applied on the two sub-lists to completely sort the whole list. For instance, '7' becomes the pivot for left sub-list and '19' becomes pivot for the right sub-list.

Note: Two sub-lists can be safely joined when every element in the first sub-list is smaller than every element in the second sub-list. **Since 'join' is a faster operation as compared to a 'merge' operation, this sort is rightly named as a 'quick sort**`.

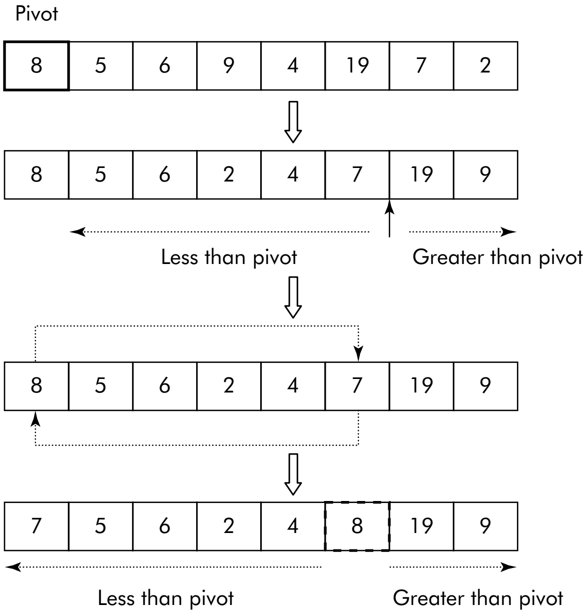


Figure 6.29 | Quick sort

The algorithm for the quick sort is given below:

In this algorithm, the elements of a list, stored in an array called List[N], are sorted in an ascending order. The algorithm has two parts—*quickSort* and *partition*. The partition algorithm divides the list into two sub-lists around a pivot. The quickSort algorithm takes a list and stores it into an array called List[N]. It uses two variables lb and ub to keep track of lower and upper bounds of list or sub-lists as the case may be. It employs partition algorithm to sort the sub-lists.

```
Algorithm quickSort()
 {
    Step
       1. Lb 5 0; /*set lower bound */
       2. ub = N – 1; /* set upper bound */
       3. pivot = List [lb];
       4. lb++;
       5. partition (pivot, List, lb, ub);
 }

Algorithm partition (pivot, List, lb, ub)
 {
    Step
       1. i = lb;
       2. j = ub;
       3. while (i<=j)
          {
                         /* travel the list from lb till an element greater than the pivot is found */
          3.1 while (List[i] <= pivot) i++;
```

```
                         /* travel the list from ub till an element smaller than the pivot is found */
        3.2 while (List[j] > pivot) j--;
        3.3 if (i < = j)    /* exchange the elements */
            {
            temp = List[i];
            List[i] = List[j];
            List[j] = temp;
            }
        }
      4. temp = List[j]; /* place the pivot at mid of the sub-lists */
      5. List[j] = List[lb – 1];
      6. List[lb – 1] = temp;
      7. if (j > lb) quicksort (List, lb, j – 1); /* sort left sub-list */
      8. if (j < ub) quicksort (List, j + 1,ub); /* sort the right sub-list */
}
```

Example 7: Given is a list of N randomly ordered numbers. Write a program that sorts the list in ascending order by using quick sort.

Solution: The required program uses both the algorithms—quickSort() and partition(). In this program, a variable called *Key* has been used that acts as a pivot.

```
/* This program sorts a given list of numbers in ascending order, using quick sort */
#include <stdio.h>
#include <conio.h>
#define N 20
void partition (int Key, int List[], int lb, int ub);
void quicksort (int List[], int lb, int ub);
void main()
{
   int List[N];
   int i, size, Pos, temp;
   int lb, ub;

   printf ("\n Enter the size of the list (< 20)");
   scanf ("%d", &size);

   printf ("\n Enter the list one by one");
   for (i = 0; i < size; i++)
   {
      scanf ("%d", &List[i]);
   }
                         /* Sort the list by quick sort */
   quicksort (List, 0, size – 1);
                         /* Print the sorted list*/
   printf ("\n The sorted list is ....");
   for (i = 0; i< size; i++)
   {
      printf ("%d ", List[i]);
```

```
    }
}
void quicksort (int List[], int lb, int ub)
{
    int Key; /* The Pivot */
    Key =List [lb]; lb++;
    partition (Key, List, lb, ub);
}
void partition (int Key, int List[], int lb, int ub)
{
    int i, j, temp;
    i = lb;
    j = ub;
        while (i<=j)
        {
        while (List[i] <= Key) i++;
        while (List[j] > Key) j--;
        printf(“\ni=%d j=%d”, i, j);
        getch();
        if (i <=j)
        {
        temp = List[i];
        List[i] = List[j];
        List[j] = temp;
        }
    }
    temp = List[j];
    List[j] = List[lb - 1];
    List[lb - 1] = temp;
    if (j > lb) quicksort (List, lb, j - 1);
    if (j < ub) quicksort (List, j + 1, ub);
}
```

6.14.6 Shell Sort

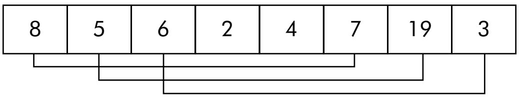
It is an improvement on insertion sort. Given a list of List[N], it is divided into *k* sets of N/k items each. *k* can assume values from a set {1, 3, 5, 19, 41…}. Thus, the *i*th set will have the following elements:

```
Set i = List[i] List[i + k] List[i + 2k] ...
```

Consider the list given in Figure 6.30.

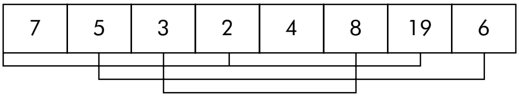
For *k* = 5, the list is divided into the following sets:

Set 0 = 8, 7
Set 1 = 5, 19
Set 2 = 6, 3
Set 3 = 2
Set 4 = 4



8	5	6	2	4	7	19	3

Figure 6.30 | The sets of a list for k = 5



7	5	3	2	4	8	19	6

Figure 6.31 | The list after first pass

Now on each set, the insertion sort is applied resulting in the arrangement shown in Figure 6.31.

For $k = 3$, the list is divided into the following sets:

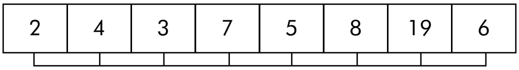
Set 0 = 7, 2, 19
Set 1 = 5, 4, 6
Set 2 = 3, 8

Now on each set, the insertion sort is applied resulting into the arrangement shown in Figure 6.32.

For $k = 1$, the list is represented by the following set:

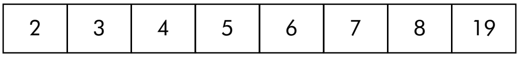
Set 0 = 2, 4, 3, 7, 5, 8, 19, 6

Application of insertion sort on the above set results in the arrangement shown in Figure 6.33.



2	4	3	7	5	8	19	6

Figure 6.32 | The list after second pass



2	3	4	5	6	7	8	19

Figure 6.33 | The list after third pass

It may be noted that the list is now sorted after the third pass. Since the variable k takes the diminishing values—5, 3, and 1; the shell sort is also called ***diminishing step sort***.

The algorithm for the shell sort is given below. In this algorithm, a list of elements, stored in an array called `List[N]`, are sorted in an ascending order. The algorithm divides the list into sets as per the description given above. The diminishing step values are stored in a list called `dimStep`. A variable '*s*' is used that moves the insertion operation to the next set. The variable k takes the step size from `dimStep` and moves the index i within the set from one element to another.

```
Algorithm shellSort()
{
        Step
           1. initialize the set dimStep to values 1,3,5,....
                                              /* sort the list by shell sort */
           2. for (step =0; step <3; step++)
           {
               2.1 k = dimStep[step]; /* set k to diminishing step */
               2.2 s = 0; /* start from the set of the list */
               2.3 for (i = s + k; i <size; i + = k)
                   {
                   temp = List[i]; /* save the element from the set */
                   j = i – k;
                                              /* find the place for insertion */
                   while ((temp <List[j]) && (j > = 0))
                   {
                       List[j + k] = List[j];
                       j = j – k;
                   }
                       List[j + k] = temp; /* insert the saved element at its place */
                       s++; /* go to next set */
                   }
           }
           3. print the sorted list
           4. Stop
}
```

Example 8: Given is a list of N randomly ordered numbers. Write a program that sorts the list in ascending order by using shell sort.

Solution: The required program uses the algorithm shellSort().

```
/* This program sorts a given list of numbers in ascending order using Shell sort */
#include <stdio.h>
#define N 20
void main()
{
    int List[N];
    int size;
    int i, j, k, p;
    int temp, s, step;
    int dimStep[] = {5,3,1}; /* the diminishing steps */
    printf (“\n Enter the size of the list (< 20)”);
    scanf (“%d”, &size);
    printf (“\n Enter the list one by one”);
    for (i=0; i< size; i++)
    {
        scanf (“%d”, &List[i]);
    }
                            /* sort the list by shell sort */
    for (step =0; step <3; step++)
    {
        k = dimStep[step]; /* set k to diminishing step */
        s=0; /* start from the set of the list */
        for (i = s + k; i <size; i += k)
        {
            temp = List[i]; /* save the element from the set */
            j=i – k;
            while ((temp <List[j]) && (j >=0)) /* find the place for insertion */
            {
                List[j + k] = List[j];
                j = j – k;
            }
            List[j + k] = temp; /* insert the saved element at its place */
            s++; /* go to next set */
        }
    }                                           /* Print the sorted list*/
    printf ("\n The sorted list is ....");
    for (i = 0; i< size; i++)
    {
        printf ("%d ", List[i]);
    }
}
```

The analysis of shell sort is beyond the scope of the book.

6.14.7 Radix Sort

It is a non-comparison-based algorithm suitable for integer values to be sorted. It sorts the elements digit by digit. It starts sorting the elements according to the least significant digit of the elements. This partially sorted list is then sorted on the basis of second least significant bit and so on.

Note: This algorithm is suitable to be implemented through linked lists. Therefore, discussion on radix sort is currently being postponed and it would be dealt with later in the book.

6.15 Summary

1. An array is used to store homogeneous data, i.e. data of the same type.
2. All the elements of an array have the same name, i.e. the array name. They are distinguished on the basis of their locations in the array. Locations are specified by using an integer value known as an index or a subscript.
3. Arrays are also known as indexed variables or subscripted variables.
4. Array index in C starts with 0.
5. C does not provide array index out-of-bound check.
6. Arrays are stored in contiguous (i.e. continuous) memory locations.

7. Arrays are classified as single-dimensional arrays and multi-dimensional arrays.
8. Subscript operator is used to access the elements of an array.
9. The array name refers to the address of the first element of the array and is a constant object.
10. An array cannot be assigned to or initialized with another array.
11. If an array is equated with another array, it always evaluates to false.
12. A pointer is a variable that holds the address of a variable or a function.
13. Restricted arithmetic can be applied on pointers. Arithmetic on pointers is governed by pointer arithmetic.
14. Addition of two pointers, addition of a float or a double value to a pointer, application of multiplication and division operators on pointers are not allowed.
15. void pointer is a generic pointer and can point to any type of object.
16. Dereferencing a void pointer and applying pointer arithmetic to it is not allowed.
17. Null pointer is a special pointer that does not point anywhere.
18. Dereferencing a null pointer leads to run time error.
19. An n-D array is an array of (n-1)-D arrays.
20. The expression of form E1[E2] is implicitly converted to an expression of the form *(E1+E2).
21. In C language, multi-dimensional arrays are stored in the memory by using row major order of storage.

Exercise Questions

Conceptual Questions and Answers

1. *What is a pointer? Where is it used?*

   A pointer is an object that holds the address of another object. A pointer is used to indirectly manipulate the value of an object to which it points.

2. *I know about basic data types in C language but what is pointer type?*

   Apart from basic data types, the C language allows to derive types from the basic data types. These types are called **derived data types**. A pointer type is one of the derived data types.

3. *What will the output of the following piece of code be?*

```
main()
{
    int *a;
    float *b;
    char *c;
    printf("%d %d %d", sizeof(a),sizeof(b),sizeof(c));
}
```

   The output of the given piece of code is dependent on the execution environment and the compiler used. If the code is executed using Borland TC 3.0 compiler for DOS, it outputs 2 2 2. If the same code is executed using Borland TC 4.5 compiler for Windows or Microsoft VC++ 6.0 compiler, it outputs 4 4 4.

   An important point to be noted here is that all pointers take 2 or 4 bytes in the memory (depending upon the execution environment and the compiler used), irrespective of whether they are pointers to int, float, char or some other data type. The difference between pointers of different data types is neither in the representation of the pointer nor in their values. The difference, rather, is in the type of the object being addressed.

4. *Why does the following piece of code on execution using Borland TC 3.0 compiler for DOS outputs* 2 1 *instead of* 2 2 *and if executed using Borland TC4.5 compiler for Windows or Microsoft VC++ 6.0 compiler outputs* 4 1 *instead of* 4 4*?*

```
main()
{
    char *a,b;
    printf("%d %d",sizeof(a),sizeof(b));
}
```

The code actually gives a correct output. The syntactic rule concerned with the declaration of a pointer states that '**A pointer is declared by prefixing an identifier with punctuator *. In a comma separated declaration list, the punctuator * must precede each identifier intended to serve as a pointer**'.
Thus, in the declaration statement char *a,b;, a is declared as 'pointer to an object of type char' and b is declared as 'data object of type char' and not as a pointer.

5. *The bitwise AND operator (&) and multiplication operator (*) are binary operators. In the following piece of code, these operators are used with only one operand. Even then the code compiles successfully. How is it possible?*

```
main()
{
    int a=10, b=20;
    int *ptr;
    ptr=&a;
    printf("The object to which ptr points has value %d",*ptr);
    ptr=&b;
    printf("The object to which ptr points now has value %d",*ptr);
}
```

The & symbol can be used as a bitwise AND operator and as a reference operator. Similarly, the symbol * can be used as a multiplication operator and as a dereference operator. The particular instance of a symbol corresponds to which operator depends upon the context in which it is used. The context can be determined by looking at:

1. Number of operands
2. Type of operands

The following are the possible combinations:

Symbol	Number of operands	Type of operands	Meaning of arithmetic, scalar and pointer type	Operator
&	Two	Arithmetic type	Integer, float and character	Bitwise AND operator
	One	Scalar type	Arithmetic type and pointer type	Reference operator
*	Two	Arithmetic type	Integer, float and character	Multiplication operator
	One	Pointer type	Pointer to a data type	Dereference operator

The symbol & when used as a reference operator should appear as a prefix unary operator and should be applied on the operands of scalar type that have l-values. The symbol * when used as a dereference operator should appear as a prefix unary operator and should be applied on the operands of the pointer type. In the mentioned piece of code: & symbol refers to the reference operator and * symbol refers to the dereference operator, which are unary operators. Hence the code compiles successfully.

6. *Why does the following piece of code not compile successfully?*

```
main()
{
    int *ptr=10;
    printf("The value pointed to by pointer is %d",*ptr);
}
```

The mentioned piece of code does not compile successfully because of illegal initialization statement whereby a pointer variable ptr is tried to be initialized with an integer value 10. The compiler gives 'Cannot convert int to int*' error because the types int and int* are incompatible and the compiler will not carry out int to int*conversion implicitly. This error can however be removed by making use of explicit type casting and writing the statement as int ptr=(int*)10;. In this statement, the programmer has forcefully converted int to int* and will himself or herself be responsible for the results. This type of explicit type conversion is not recommended.

7. *Can* const *qualifier be used with pointer types like it can be used with basic data types?*

Yes, const qualifier can be used with pointer types. It is important to understand the use of const qualifier when it is mixed with pointer type. const qualifier can be mixed with pointer type in the following ways:

S.No	Use of const qualifier with pointer type (Column 2)	Meaning of statements in Column 2	What is constant?
1.	const int *ptr	ptr is a pointer to an integer constant	Integer object pointed to by ptr
2.	int const *ptr	ptr is a pointer to a constant integer (same as declaration at S.No. 1)	Integer object pointed to by ptr
3.	int *const ptr	ptr is a constant pointer to an integer	ptr is constant
4.	const int *const ptr	ptr is a constant pointer to an integer constant	Both ptr and the integer object pointed to by ptr are constant
5.	int const *const ptr	ptr is a constant pointer to a constant integer (same as declaration S.No. 4)	Both ptr and the integer object pointed to by ptr are constant

8. *What is pointer arithmetic?*

Refer Section 6.4.2.

9. *In the expression* *pointer++, *which entity gets incremented: pointer or the value to which the pointer points?*

Dereference operator * and the increment operator ++ are unary operators and unary operators are right-to-left associative. The expression *pointer++ will be interpreted as *(pointer++). Thus, in this expression, the value of the pointer instead of the value pointed by the pointer gets incremented.

10. *I want to print the memory address to which a pointer points. Which format specifier should I use to print it?*

The format specifier used for printing pointers (addresses) is %p. The printed format depends upon which memory model is used. It will either be XXXX:YYYY (segment:offset) or YYYY (offset only).

Consider the following piece of code:

```
main()
{
    int a=10;
    int *ptr=&a;
    printf("The value of pointer is %p",ptr);
}
```

If the code is executed using Borland TC 3.0 compiler for DOS and small memory model, it prints FFF4 (offset address only). If worked with huge memory model, it prints 900E:00FE (segment and offset address).

11. *What is array type and how is it declared?*

    Refer Sections 6.2 and 6.3.

12. *I want to store an integer value, a* float *value and a character value in an array. Is it possible?*

    No, arrays can only be used for storage of homogeneous data (i.e. data of the same type). Arrays cannot be used for storage of heterogeneous data (i.e. data of different types). For storage of heterogeneous data, structures and unions are used.

13. *How is the declaration* int * a[10] *different from* int (*a) [10]*?*

    While reading C declarations remember that [] binds more tightly than *. In the declaration statement int *a[10]; the identifier name a is bound to [] instead of * and it is read as 'a is an array of 10 integer pointers'. In the declaration statement int (*a)[10];, () is used to bind a to *. Hence, the declaration is read as 'a is a pointer to an array of 10 integers'.

14. *How is an expression involving a subscript operator internally represented?*

    The general form of an expression involving a subscript operator is E1[E2], where both E1 and E2 are sub-expressions. One of the sub-expressions E1 or E2 must be of array type or pointer type and the other expression must be of integer type. Every expression of the form E1[E2] automatically gets converted to an equivalent expression of the form *(E1+E2). Hence, the expression E1[E2] is internally represented as *(E1+E2).

    Consider the following piece of code:

```
main()
{
    int array[4]={4,5,6,7};
    int *pointer=array;
    printf("%d %d %d\n", array[0],pointer[1],*(array+2), *(pointer+3));
}
```

E1 is of array type

E1 is of pointer type

Transformed form of subscript operator

The mentioned code on execution outputs:

```
4 5 6 7
```

15. *Are the expressions* arr *and* &arr *same, if* arr *is an array of type* T*?*

    No, **the expressions arr and &arr are not the same**. The expression &arr yields 'a pointer to an array of type T' and the expression arr yields 'a pointer to type T'. The expression arr refers to the address of first element of the array and the expression &arr refers to the base address of the entire array. To understand the difference between arr and &arr, consider the following piece of code:

```
main()
{
    int arr[5]={1,2,3,4,5};
    printf("The base address of array is %p or %p\n",arr,&arr);
    printf("After incrementing by one they point to %p and %p",arr+1,&arr+1);
}
```

The code on execution outputs:

```
The base address of array is 1B6F:223A or 1B6F:223A
After incrementing by one they point to 1B6F:223C and 1B6F:2244
```

Increment of one in arr increments it by 2-bytes as it is of type int* while increment of one in &arr increments it by 10-bytes as its type is int(*)[5] (i.e. pointer to an array of 5 integers).

16. *Does the C language provide array index out-of-bound check?*

    No, the C language does not provide compile-time or run-time array index out-of-bound check. If an array is declared as T array[size], the maximum valid index is size-1, as array index in C language starts from 0. Nothing stops a programmer from stepping across an array boundary and accessing the array with an index greater than size-1. The program having array index out-of-bound will compile and execute but will access to the memory location that does not belong to the array. This illegal memory access may be fatal and may even crash the program.

17. *Why does the following piece of code on execution give a garbage value?*

```
main()
{
    int array[3]={1,2,3};
    printf("The last element of array is %d",array[3]);
}
```

The mentioned piece of code gives a garbage value because the array index is out-of-bound. The maximum valid array index is 2. Since the array is indexed with 3, reference has been made to the memory location that does not belong to the array (i.e. garbage field). Hence, the code on execution gives a garbage value. Note that in some cases the program may even crash, i.e. terminate.

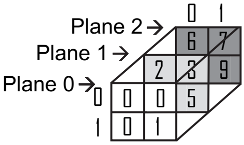
18. *How will you visualize a multi-dimensional array?*

    Refer Section 6.8.1.

19. *How are multi-dimensional arrays stored in C?*

    Refer Section 6.8.1.

    Suppose a three-dimensional array is declared as int a[3][2][2]={0,0,0,1,2,3,4,5,6,7,8,9};. It can be visualized as:



The shown 3-D array will actually be stored in the physical memory as:

a

[0][0][0]	[0][0][1]	[0][1][0]	[0][1][1]	[1][0][0]	[1][0][1]	[1][1][0]	[1][1][1]	[2][0][0]	[2][0][1]	[2][1][0]	[2][1][1]
0	0	0	1	2	3	4	5	6	7	8	9
2000-01	2002-03	2004-05	2006-07	2008-09	2010-11	2012-13	2014-15	2016-17	2018-19	2020-21	2022-23

20. *What would be the result of a* sizeof *operator, when it is applied on an array type?*

    When the sizeof operator is applied on an operand of array type, the result is the total number of bytes occupied by the array.

21. *What is null pointer? Is null pointer same as uninitialized pointer?*

    Refer Section 6.6.

    No, null pointer is not the same as uninitialized pointer. A null pointer does not point to any object or function, while an uninitialized pointer might point anywhere. The declaration statements int *ptr=0; and int *ptr=NULL; create a null pointer, named ptr, and the declaration statement int *ptr; creates an uninitialized pointer.

22. *What is a* void *pointer?*

    Refer Section 6.5.

23. *Why is pointer arithmetic not applicable on* void *pointers?*

    Pointer arithmetic is not applicable on void pointers because the compiler does not know what kind of object the void pointer is really pointing to. Before applying an arithmetic operator on void*, explicitly type cast void* to a pointer to a specific type.

24. *Given the declaration statement,* int array[10],i=2; *what are the types of expressions* array, &array, *array, array[i]*?*

    The types of expressions:

array is int*	(i.e. pointer to the first element of array)
&array is int(*)[10]	(i.e. pointer to the entire array)
*array is int	(i.e. value of first element of the array)
array[i] is int	(i.e. value of (i+1)th element of the array)

25. *Given the declaration statement,* int array[10][10],i=2,j=2; *what are the types of expressions* array, &array, *array, array[i], **array, array[i][j]*?*

    The types of expressions:

array is int(*)[10]	(i.e. pointer to the first row of array)
&array is int(*)[10][10]	(i.e. pointer to the entire array)
array is int	(i.e. pointer to first element in the first row of array)
array[i] is int*	(i.e. pointer to first element in (i+1)th row of the array)
**array is int	(i.e. value of first element in first row of the array)
array[i][j] is int	(i.e. value of element in (i+1)th row and (j+1)th column of the array)

Code Snippets

Determine the output of the following code snippets. Assume that the inclusion of the required header files has been made and there is no prototyping error due to them.

26.
```
main()
{
    char*p1,p2;
    printf("%d %d",sizeof(p1),sizeof(p2));
}
```

27.
```
main()
{
    printf("%d %d %d",sizeof(char*),sizeof(int*),sizeof(float*));
}
```

```
28 main()
   {
       char far *p1, near *p2, huge *p3;
       printf(“%d %d %d”,sizeof(p1),sizeof(p2),sizeof(p3));
   }
```

```
29. main()
    {
        int a=10;
        int *ptr=&a;
        printf(“%d %d”,++*ptr,*ptr++);
    }
```

```
30. main()
    {
        int a=10;
        int *ptr=&a;
        printf(“%d %d”,*ptr++,++*ptr);
    }
```

```
31. main()
    {
        int a=10;
        const int *ptr=&a;
        *ptr=50;
        printf(“The changed value of pointed object is %d”,*ptr);
    }
```

```
32. main()
    {
        int a=10,b=20;
        int *const ptr=&a;
        *ptr=20;
        printf(“The changed value of pointed object a is %d”,*ptr);
        ptr=&b;
        *ptr=10;
        printf(“The changed value of pointed object b is %d”,*ptr);
    }
```

```
33. main()
    {
        int a=10,b=20;
        const int *const ptr=&a;
        *ptr=20;
        printf(“The changed value of pointed object a is %d”,*ptr);
        ptr=&b;
        *ptr=10;
        printf(“The changed value of pointed object b is %d”,*ptr);
    }
```

```
34. main()
    {
         int *ptr=10;
         printf(“The value of pointer is %p”,ptr);
     }
35.  main()
     {
         int *ptr=0;
         printf(“The value of pointer is %p”,ptr);
     }
36.  main()
     {
         int *ptr1=0;
         int *ptr2=NULL;
         if(ptr1==ptr2)
             printf(“ptr1 becomes a NULL pointer”);
         else
             printf(“ptr1 does not become a NULL pointer”);
     }
37.  main()
     {
         int arr[ ];
         arr[0]=arr[1]=arr[2]=5;
         printf(“%d %d %d”,arr[0],arr[1],arr[2]);
     }
38.  main()
     {
         int size=3;
         int arr[size];
         arr[0]=arr[1]=arr[2]=5;
         printf(“%d %d %d”,arr[0],arr[1],arr[2]);
     }
39.  main()
     {
         int a[]={1,2,3};
         printf(“%d %d %d”,a[0],a[1],a[2]);
     }
40.  main()
     {
         int a[2]={1,2,3};
         printf(“%d %d %d”,a[0],a[1],a[2]);
     }
41.  main()
     {
         int arr[6]={1,2,3,4};
         int i;
         for(i=0;i<6;i++)
         printf(“%d ”,arr[i]);
     }
```

42.
```
main()
{
    int arr[3]={1,2,3};
    printf(“%d %d %d”,arr[1],arr[2],arr[3]);
}
```

43.
```
main()
{
    int arr[]={1,2,3};
    arr[0,1,2]=10;
    printf(“%d %d %d”,arr[0],arr[1],arr[2]);
}
```

44.
```
main()
{
    int arr[]={1,2,3,4,5},i;
    arr[1+2]=10;
    for(i=0;i<5;i++)
        printf(“%d ”,arr[i]);
}
```

45.
```
main()
{
    int arr[]={1,2,3,4,5},i;
    arr[2.5+1.5]=10;
    for(i=0;i<5;i++)
        printf(“%d ”,arr[i]);
}
```

46.
```
main()
{
    int array[]={1,2,3,4};
    printf(“The number of elements in array are %d”,sizeof(array)/sizeof(array[0]));
}
```

47.
```
main()
{
    int a=10,b;
    int arr[]={1,2,3}, brr[3];
    printf(“Assigning the content of a to b\n”);
    b=a;
    printf(“Assigning the contents of one array to another\n”);
    brr=arr;
    printf(“Contents of brr are %d %d %d”,brr[0],brr[1],brr[2]);
}
```

48.
```
main()
{
    int arr[]={1,2,3},brr[]={1,2,3};
    if(arr==brr)
        printf(“Contents of array arr and brr are same\n”);
    else
        printf(“Contents of array arr and brr are not same”);
}
```

```
49. main()
    {
        int a[]={1,2,3,4,5};
        int *ptr=a;
        printf(“%d %d\n%p %p”,*a,*ptr,a,ptr);
    }
50. main()
    {
        int arr[]={1,2,3,4,5};
        printf(“%p %p\n”,arr,&arr);
        printf(“%p %p”,++arr,++&arr);
    }
51. main()
    {
        int arr[]={1,2,3,4,5};
        printf(“%p %p\n”,arr,&arr);
        printf(“%p %p”,arr+1,&arr+1);
    }
52. main()
    {
        int a[]={1,2,3,4,5};
        printf(“%d %d %d %d %d”,*a,*(a+0),*(0+a),a[0],0[a]);
    }
53. main()
    {
        int *ptr;
        int arr[]={1,2,3,4};
        ptr=arr;
        printf(“%d %d”,arr[2],ptr[2]);
    }
54. main()
    {
        int arr[ ]={2.8,3.4,4,6.7,5};
        int j,*ptr=arr;
        for(j=0;j<5;j++)
        {
            printf(“ %d “,*ptr);
            ++ptr;
        }
    }
55. main( )
    {
        int j=20;
        int arr[ ] = {10,j,30,40,50},i,*ptr;
        ptr = arr;
        for(i=0; i<5; i++)
        {
            printf(“%d ” ,*ptr);
            ptr++;
        }
    }
56. main()
    {
        int arr[2][3]={1,2,3,4};
        printf(“%d %d %d %d %d %d”,arr[0][0],arr[0][1],arr[0][2],arr[1][0],arr[1][1],arr[1][2]);
    }
57. main()
    {
        int arr[2][3]={{1,2},{3,4}};
        printf(“%d %d %d %d %d %d”,arr[0][0],arr[0][1],arr[0][2],arr[1][0],arr[1][1],arr[1][2]);
    }
58. main()
    {
        int arr[][]={1,2,3,4};
        printf(“%d %d %d %d %d %d”,arr[0][0],arr[0][1],arr[0][2],arr[1][0],arr[1][1],arr[1][2]);
    }
59. main()
    {
        int arr[2][][]={1,2,3,4,5,6,7,8};
        int i, j, k;
        for(i=0;i<2;i++)
            for(j=0;j<2;j++)
                for(k=0;k<2;k++)
                    printf(“%d”,arr[i][j][k]);
    }
60. main()
    {
        int arr[][3]={1,2,3,4};
        printf(“%d %d %d %d %d %d”,arr[0][0],arr[0][1],arr[0][2],arr[1][0],arr[1][1],arr[1][2]);
    }
61. main()
    {
        int arr[2][2]={1,2,3,4};
        printf(“%p %p\n%p %p”,&arr[0][0],&arr[0][1],&arr[1][0],&arr[1][1]);
    }
62. main()
    {
        int arr[2][3]={1,2,3,4,5,6};
        printf(“%d %d %d”,arr[1][2],1[arr][2],*(*(arr+1)+2));
    }
63. main()
    {
        int arr[2][3]={1,2,3,4,5,6};
        printf(“%d %d %d”,arr[1][2],1[arr][2],1[2][arr]);
    }
```

64.
```
main( )
{
    int a[ ] = {0,1,2,3,4};
    int *p[ ] = {a,a+1,a+2,a+3,a+4};
    int **ptr = p;
    printf(“%d %p %p %p %p %p\n”,**ptr,&ptr,*ptr,*p,p,a);
}
```

65.
```
main()
{
    int a[2][2][2]={1,2,3,4,5,6,7,8};
    printf(“%p %p %p\n”,a,a[0],a[0][0]);
    printf(“%p %p %p\n”,a,a[1],a[1][1]);
    printf(“%d %d”,a[0][0][0],a[1][1][1]);
}
```

66.
```
main()
{
    void a,b;
    void *ptr;
    ptr=&a;
    printf(“ptr points to a\n”);
    ptr=&b;
    printf(“ptr now points to b”);
}
```

67.
```
main()
{
    int a=10;
    int* i_ptr=&a;
    void* v_ptr=i_ptr;
    *i_ptr++;
    *v_ptr++;
    printf(“The value of objects pointed to by pointers are %d %d”,*i_ptr,*v_ptr);
}
```

68.
```
main()
{
    int arr[]={1,2,3,4,5};
    int *ptr=arr;
    ptr=ptr+1;
    printf(“The value pointed by ptr is %d”,*ptr);
}
```

69.
```
main()
{
    int arr[]={1,2,3,4,5};
    int *ptr1=arr;
    int *ptr2=arr+3;
    printf(“The result of ptr2-ptr1 is %d”,ptr2-ptr1);
}
```

70.
```
main()
{
    int array[]={1,2,3,4,5};
    int *ptr1=array;
    int *ptr2;
    ptr2=ptr1*2;
    printf("The value of ptr2 is %p",ptr2);
  }
```

Multiple-choice Questions

71. Arrays are used to store the elements of
    - a. The same type
    - b. Different types
    - c. Multiple types
    - d. None of these

72. Array index in C language starts from
    - a. 1
    - b. 0
    - c. Any integer value
    - d. None of these

73. The size specifier in the array declaration must be
    - a. An expression
    - b. A constant expression
    - c. A constant expression of integral type
    - d. A constant expression of integral type having a value greater than zero

74. In C language, elements of two-dimensional arrays are stored in
    - a. Random order
    - b. Column major order
    - c. Row major order
    - d. None of these

75. The elements of an array are stored in
    - a. Contiguous memory locations
    - b. Discontinuous memory locations
    - c. Randomly allocated memory locations
    - d. None of these

76. If one of the operands of subscript operator is of array type, the other operand of the subscript operator can be
    - a. An expression
    - b. An expression of integral type
    - c. An integral constant only
    - d. None of these

77. If `arr` is an array of integers, which of the following expression(s) is equivalent to the expression `arr[0]`?
    - a. `*arr`
    - b. `*(arr+0)`
    - c. `0[arr]`
    - d. All of these

78. Given the declaration statement `int arr[5];`, the type of expression `arr` is
    - a. `int*`
    - b. `int(*)[5]`
    - c. `int*[5]`
    - d. None of these

79. Given the declaration statement `int arr[5];`, the type of expression `&arr` is
    - a. `int*`
    - b. `int(*)[5]`
    - c. `int*[5]`
    - d. None of these

80. Given the declaration statement `int arr[5][7];`, the linear offset from the beginning of the array to any given element `arr[2][3]` can be computed as

a. 2*7+3
b. 2+3*5
c. 2*5+3*7
d. None of these

81. Given the declaration statement int array[3][2][2]={1,2,3,4,5,6,7,8,9,10,11,12};, what is the value of array[2][1][0]?

a. 3
b. 5
c. 7
d. 11

82. Given the statement int a[8]={0,1,2,3};, the definition of a explicitly initializes its first four elements. Which one of the following describes how the compiler treats the remaining four elements?

a. The remaining four elements are initialized to zero
b. It is illegal to initialize only a portion of the array
c. C standard defines the particular behavior as implementation dependent
d. None of these

83. In the C language, pointer is

a. Address of a variable
b. An indication of the variable to accessed next
c. A variable for storing address
d. None of these

84. Which of the following is a derived type?

a. Pointer type
b. Array type
c. Function type
d. All of these

85. Which of the following is a correct way to declare two integer pointers a and b?

a. int* a,b;
b. int *a,*b;
c. int* a,int* b;
d. None of these

86. A null pointer points to

a. No object
b. Null value
c. Null character stored at the end of string
d. None of these

87. Pointer arithmetic cannot be performed on

a. void pointers
b. Uninitialized pointers
c. Dangling pointers
d. None of these

88. Which of the following conversions is carried out implicitly by the compiler?

a. Conversion of void pointer to any other pointer type on assignment
b. Conversion of integer constant zero into null pointer of desired type on assignment
c. Conversion of pointer of one type to the pointer of another type on assignment
d. None of these

89. Given the declaration statement int* a[2][3][4];, which of the following definitions and initialization of p is valid?

a. int* (*p)[3][4]=a;
b. int ****p=a;
c. int* (*p)[2][3][4]=a;
d. None of these

90. Given the declaration statement int const* ptr;, which of the following objects is constant?

a. ptr
b. The object pointed to by ptr
c. Both ptr and the object pointed to by ptr
d. The given declaration is not valid

91. Given the declaration statement const int* ptr;, which of the following objects is constant?
    a. ptr
    b. The object pointed to by ptr
    c. Both ptr and the object pointed to by ptr
    d. The given declaration is not valid

92. Given the declaration statement int* const ptr;, which of the following objects is constant?
    a. ptr
    b. The object pointed to by ptr
    c. Both ptr and the object pointed to by ptr
    d. The given declaration is not valid

93. Given the declaration statement int const* const ptr;, which of the following objects is constant?
    a. ptr
    b. The object pointed to by ptr
    c. Both ptr and the object pointed to by ptr
    d. The given declaration is not valid

94. In the expression ++*ptr, the value of which entity gets incremented?
    a. ptr
    b The object pointed to by ptr
    c. Both ptr and the object pointed to by ptr
    d. The given expression is not valid

95. In the expression *ptr++, the value of which entity gets incremented?
    a. ptr
    b. The object pointed to by ptr
    c. Both ptr and the object pointed to by ptr
    d. The given expression is not valid

Outputs and Explanations to Code Snippets

26. 4 1 (If executed using Borland TC 4.5 for Windows or Microsoft VC++ 6.0)
    2 1 (If executed using Borland 3.0 for DOS)

    Explanation:

    Refer Answer numbers 3 and 4.

27. 4 4 4 (If executed using Borland TC 4.5 for Windows or Microsoft VC++ 6.0)
    2 2 2 (If executed using Borland 3.0 for DOS)

    Explanation:

    Refer Answer number 3.

    sizeof operator yields the size of its operand in bytes. The operand can be an expression or parenthesized name of a type. In the given code, the operands of sizeof operators are parenthesized name of the derived types (i.e. char*, int* and float*).

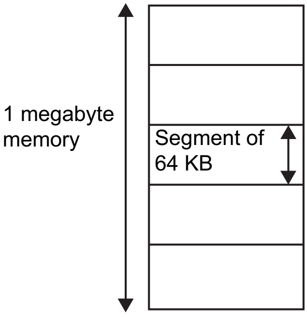
28. 4 2 4 (If executed using Borland 3.0 for DOS)

    Explanation:

    In DOS, the total amount of memory accessible is 1 MB, i.e. 1 megabyte. The entire block of memory is divided into various segments that are 64 K, i.e. 64 kilobytes in size. There are various segments like Code Segment (CS), Data Segment (DS), Extra Segment (ES), etc.

    The type of pointer to be used for accessing the memory location depends upon whether the memory location to be accessed lies in the same segment or different segments. If the memory location to be accessed lies in the same segment, the access is called **intra-segment access** and if it lies in a different segment then it is called **inter-segment access.**

    If intra-segment access is to be made, pointer of 16-bits is sufficient to refer to all the memory locations (as 2^{16}= 64 K). The 16-bit (2 bytes) pointer that is used for intra-segment access is known as a **near pointer.** However, if inter-segment access is to be made, the pointer of 16-bits falls short of its memory addressing capability. Hence, a



bigger pointer of 32-bits is used to make inter-segment access. The 32-bit (4 bytes) pointers that are used for inter-segment access are known as **far pointers** and **huge pointers.**

The far pointer contains a 16-bit segment address and a 16-bit offset address (i.e. address within a segment) while the near pointer has only a 16-bit offset address. huge pointers are essentially far pointers but in a normalized form.

The concept of far, near and huge pointers is available in DOS, which has less memory accessible. It is not a part of the C standard and is an extension to the language provided by some of the compilers (e.g. Borland Turbo C 3.0). Refer to the compiler documentation before using these non-standardized qualifiers as they might not be supported by all the compilers (e.g. Borland Turbo C 4.5 and MS-VC++ 6.0 compilers do not support these non-standardized extensions).

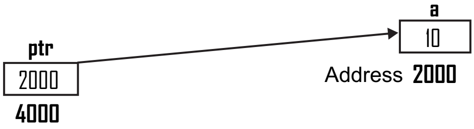
29. Garbage Value 10

Caution:

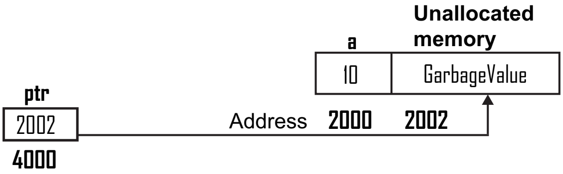
Program may even abnormally terminate.

Explanation:

Suppose variable a gets allocated at the memory address 2000 and variable ptr gets allocated at the memory address 4000. The variable ptr is initialized with the address of variable a. This can be illustrated as:



The arguments of printf functions are evaluated from right to left. So, the expression *ptr++ will be evaluated first and will be interpreted as *(ptr++). Due to post-increment, firstly the value of ptr is used for the evaluation of expression and then the value of ptr will be incremented. The expression evaluates to 10 and the value of ptr becomes 2002. This can be illustrated as:

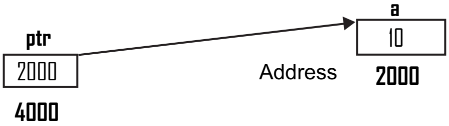


After evaluation of expression *ptr++, the expression ++*ptr will be evaluated. The expression will be interpreted as ++(*ptr). ptr being pointing to an unallocated memory location, i.e. 2002, the behavior of the operation ++(*ptr) is undefined. It will give a garbage value and in extreme cases, the program may even terminate abnormally.

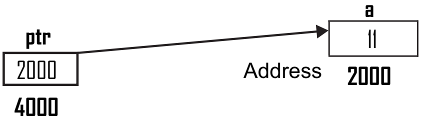
30. 11 11

Explanation:

Suppose variable a gets allocated at the memory address 2000 and variable ptr gets allocated at the memory address 4000. The variable ptr is initialized with the address of variable a. This can be illustrated as:



The expression ++*ptr will be evaluated first and will be treated as ++(*ptr). This expression makes the value pointed to by the pointer ptr to increment by 1. This can be shown as:

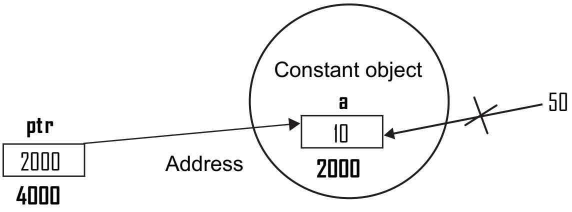


After the evaluation of expression ++*ptr, *ptr++ starts evaluation. The expression *ptr++ will be treated as *(ptr++). Being post-incremented, the value of ptr used for the evaluation of expression will be 2000. The expression evaluates to 11 and the value of ptr becomes 2002.

31. Compilation Error (Cannot modify a constant object)

Explanation:

In the declaration statement const int *ptr=&a;, ptr is declared as 'pointer to a constant integer'. The object to which pointer ptr points is constant and cannot be modified.

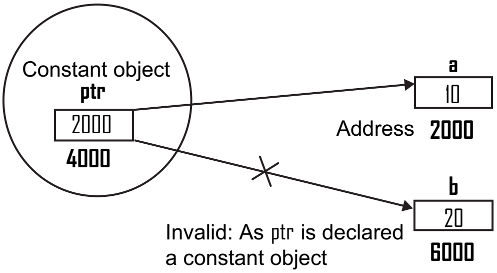


Hence, writing *ptr=50; is not valid and leads to a compilation error.

32. Compilation Error (Cannot modify a constant object)

Explanation:

The declaration statement int *const ptr=&a; declares ptr as 'constant pointer to integer'. Pointer is constant and must point to the same object throughout. It cannot be made to point to a different object throughout the execution of the program.

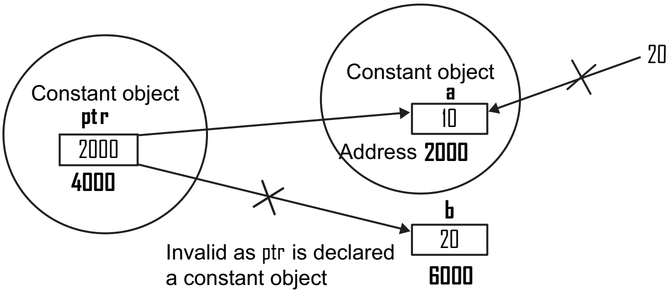


Hence, writing ptr=&b; is invalid and leads to a compilation error.

33. Compilation Error (Cannot modify a constant object)

Explanation:

The declaration statement const int* const ptr=&a; declares ptr as 'constant pointer to a constant integer'. Both the pointer and the object to which the pointer points are constant.



Hence, both the statements *ptr=20; and ptr=&b; are invalid and lead to a compilation error.

34. Compilation error (Cannot convert int to int*)

Explanation:

An integer value 10 is assigned to a pointer variable of type int*. This is not a standard conversion and the compiler will not be able to carry it out implicitly and gives a compilation error 'Cannot convert int to int*'. The error can be removed by explicitly type casting int to int* by writing int*ptr=(int*)10;. However, this conversion is not recommended.

35. The value of pointer is 0000:0000

Explanation:

The conversion of integer value zero to pointer type is standard conversion and is carried out implicitly by the compiler. When a variable or expression of pointer type is initialized, assigned or compared with 0, the constant 0 is implicitly converted into correctly typed null pointer. Hence, there will be no compilation error as in Question number 34.

36. ptr1 becomes a NULL pointer

Explanation:

The integer constant zero is implicitly converted to null pointer of the correct type. NULL is a predefined macro that specifies null pointer value. Hence, in the given code both ptr1 and ptr2 become null pointers. As two null pointers always compare equal, the if condition evaluates to true and 'ptr1 becomes a NULL pointer' gets printed.

37. Compilation Error (Size of 'arr' is unknown)

Explanation:

Memory to an array is allocated at the compile time. To allocate the memory, the compiler should be able to determine the number of bytes to be allocated. To determine it, the compiler needs to know:

1. The element type of array
2. The size of array

The size information can be provided by giving:

1. Size specifier (it should be a compile time constant expression), and/or
2. Initialization list (number of initializers in the initialization list determines the size of array)

Since in the declaration statement `int arr[];` both the size specifier and the initialization list are not given, the compiler will not be able to determine the number of bytes to be allocated to `arr`. This leads to a compilation error.

38. `Compilation error (Constant expression required)`

    Explanation:

    Refer Sections 6.2 and 6.3.

    Although, `size` is initialized with a literal constant, `size` itself is a non-constant object (i.e. it is a variable). Access to its value can only be accomplished at the run time, so it is illegal to use it as an array size specifier. To remove this error, make `size` a qualified constant by using `const` qualifier and write it as `const int size=3;` instead of `int size=3;`.

39. `1 2 3`

    Explanation:

    The compiler uses the initializers in the initialization list to determine the size of the array and to initialize the array locations.

40. `Compilation error (Too many initializers)`

    Explanation:

    The number of initializers in the initialization list cannot be more than the value of the size specifier. They can be less than or at most equal to the value of the size specifier.

41. `1 2 3 4 0 0`

    Explanation:

    If the number of initializers in the initialization list is less than the value of the size specifier, the leading array locations equal to the number of initializers get initialized with the values of initializers. The rest of the array locations get initialized to `0` (if it is an integer array), `0.0` (in case of `float` array) and '`\0`', i.e. null character (if array is of character type).

42. `2 3 Garbage Value`

    Explanation:

    In C language, an array index starts from `0` and the maximum value of the valid index is `size-1`. However, if the index value greater than the maximum valid value is used, there will be no compilation error as C language does not provide an array index out-of-bound check. If array is accessed with an out-of-bound index, the result will be a garbage value. In the extreme case, the program may terminate.

43. `1 2 10`

    Explanation:

    To access an array location, an expression of array type and an expression of integral type are used with a subscript operator. In the statement `arr[0,1,2]=10;`, `arr` is an expression of array type and `0,1,2` is an expression of integer type. The expression `0,1,2` evaluates to `2` as the result of the evaluation of a comma operator is the result of evaluation of the right-most sub-expression. Hence, `10` is assigned to `arr[2]`.

44. 1 2 3 10 5

Explanation:

Refer Answer number 43.

45. Compilation error (illegal use of floating point)

Explanation:

An expression of float type cannot be used with a subscript operator. Hence, writing a[2.5+1.5] is not valid and leads to a compilation error.

46. The number of elements in array are 4

Explanation:

When the sizeof operator is applied on the operand of an array type, the result is the total number of bytes allocated to the array. So, sizeof(array) results in 8. However, sizeof(array[0]) gives the size of one element of the array, i.e. 2. Hence, sizeof(array)/sizeof(array[0]) results in 4.

47. Compilation Error (L-value required error)

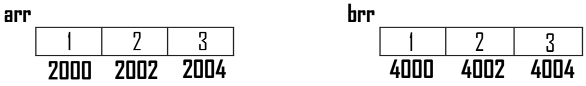
Explanation:

The name of an array refers to the address of the first element of an array and does not have a modifiable l-value. Since it does not have a modifiable l-value, it cannot be placed on the left side of the assignment operator. Hence, writing brr=arr is not valid and leads to a compilation error.

48. Contents of array arr and brr are not same

Explanation:

Suppose, the array arr gets allocated at the memory location 2000 and brr gets allocated at the memory location 4000. This is shown in the figure below:



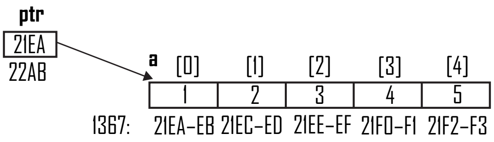
Since, the name of the array refers to the address of the first element of the array, arr refers to 2000 and brr refers to 4000. The addresses are not equal (in fact they can never be) and hence, the printf statement of the else body gets executed to produce the mentioned result.

49. 1 1
1367:21EA 1367:21EA

Explanation:

Refer Answer numbers 15 and 20.

The name of type 'array of type T' is implicitly converted to pointer of type 'pointer to T' (with two exceptions). The pointer refers to the address of the first element of the array. Hence, the initialization statement int *ptr=a; initializes ptr with the address of the first element of the array.



Therefore, *a and *ptr give the value of the first element of the array, a and ptr give the address of the first element of the array (i.e. base address of the array).

The mentioned addresses are in hexadecimal number system.

50. Compilation error (L-value required error)

    Explanation:

    Refer Answer number 15.

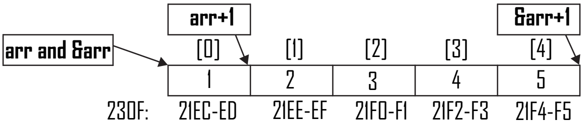
    The name arr refers to the address of the first element of the array and &arr refers to the base address of the entire array. Both arr and &arr are constant objects and do not have a modifiable l-value. The increment operator can operate only on operands that have a modifiable l-value. Hence, the expressions ++arr and ++&arr are erroneous.

51. 230F:21EC 230F:21EC
    230F:21EE 230F:21F6

    Explanation:

    Refer Answer number 15.

    Both the expressions arr and &arr refer to the starting address of the array arr, say 230F:21EC. The expression arr+1 evaluates to 230F:21EE as the type of arr is int* and the sizeof(int*) is 2. The expression &arr+1 evaluates to 230F:21F6 as the type of arr is int(*)[5] and the sizeof(int(*)[5]) is 10.



52. 1 1 1 1 1

    Explanation:

    All the expressions *a, *(a+0), *(0+a), a[0] and 0[a] are equivalent and refer to the first element of array, i.e. 1.

53. 3 3

    Explanation:

    Both the expressions arr[2] and ptr[2] are equivalent to *(arr+2) and *(ptr+2), respectively. Since arr has been assigned to ptr, *(ptr+2) is equivalent to *(arr+2), i.e. 3.

54. 2 3 4 6 5

    Explanation:

    Since initializers in the initialization list of an integer array are of float type, the initializers will be demoted before initializing array locations. As ptr is initialized with arr, it points to the first element of the array arr. During every iteration of the loop, the value of element pointed to by ptr is printed, and ptr is made to point to the next element of the array arr. In this way, the entire array gets printed.

55. 10 20 30 40 50

    Explanation:

    Initializers in the initialization list can be pre-defined variables. Hence, writing int arr[]={10,j,30,40,50} is valid as j is a predefined variable having a value of 20.

56. 1 2 3 4 0 0

Explanation:

As the number of initializers in the initialization list is less than the value of the size specifier, the leading array locations equal to the number of initializers get initialized with the values of initializers. The rest of the array locations get initialized to 0. Since multi-dimensional arrays in C are stored in row major order, elements of the array are initialized row by row. Thus, the contents of initialized array will be:

2-D array **arr** →

Rows ↓ \ Cols →	0	1	2
0	1	2	3
1	4	0	0

57. 1 2 0 3 4 0

Explanation:

The initializers in the initialization list are bracketed to initialize individual rows. Since the number of initializers within the inner brackets is less than the row size, the last element of each row gets initialized to 0. The contents of the initialized array are as follows:

2-D array **arr** →

Row ↓ \ Cols →	0	1	2
0	1	2	0
1	3	4	0

58. Compilation error (Size of type is unknown or zero)

Explanation:

While declaring 2-D arrays, even if the initialization list is present, both the row size specifier and the column size specifier cannot be skipped. In the declaration statement int arr[][]={1,2,3,4}; there are four initializers, so the number of elements in the array will be at least four. There are three different ways to create an array of four elements:

1. int arr[1][4]={1,2,3,4}, i.e. array having one row and four columns, or
2. int arr[4][1]={1,2,3,4}, i.e. array having four rows and one column, or
3. int arr[2][2]={1,2,3,4}, i.e. array having two rows and two columns

So, the compiler will not be able to determine the number of rows and columns in an array.

Since arrays are stored in row major order, if the number of columns in a row of an array is specified, the compiler will be able to determine the number of rows and can create the array. Look at the following declarations and the arrays that get created:

1. int arr[][4]={1,2,3,4}

arr	0	1	2	3
0	1	2	3	4

2. int arr[][2]={1,2,3,4}

arr	0	1
0	1	2
1	3	4

3. int arr[][1]={1,2,3,4}

arr	0
0	1
1	2
2	3
3	4

4. int arr[][3]={1,2,3,4} (Number of elements in the array will be greater than 4)

arr	0	1	2
0	1	2	3
1	4	0	0

5. int arr[][5]={1,2,3,4} (Number of elements in the array will be greater than 4)

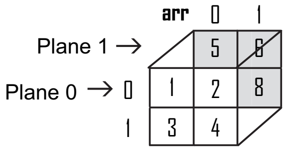
arr	0	1	2	3	4
0	1	2	3	4	0

59. Compilation error (Size of type is unknown or zero)

Explanation:

"While declaring n-D array, even if initialization list is present, it is mandatory to specify (n-1) fastest varying size specifiers so that compiler can uniquely determine the dimensions and create the array".

In case of 3-D arrays, even if the initialization list is present, it is mandatory to mention both the column size specifier and the row size specifier, as they vary faster as compared to plane size specifier as shown in the figure below. The plane size specifier can be skipped, if the initialization list is present, e.g. the declaration int arr[][2][2]={1,2,3,4,5,6,7,8}; is valid and the compiler will create an array that has two planes, each having two rows and two columns, as shown in the figure below:



Plane 0				Plane 1			
Row 0		Row 1		Row 0		Row 1	
Col 0	Col 1	Col 0	Col 1	Col 0	Col 1	Col 0	Col 1
a[0][0][0]	a[0][0][1]	a[0][1][0]	a[0][1][1]	a[1][0][0]	a[1][0][1]	a[1][1][0]	a[1][1][1]
1	2	3	4	5	6	7	8
21F8	21FA	21FC	21FE	2200	2202	2204	2206

In the declaration statement present in the given question, the plane size specifier is mentioned but the column size and the row size specifier are not mentioned. Hence, the compiler cannot uniquely determine the dimensions of the array. This leads to a compilation error.

60. 1 2 3 4 0 0

Explanation:

Refer Answer numbers 58 and 59.

61. 2367:21EA 2367:21EC
    2367:21EE 2367:21F0

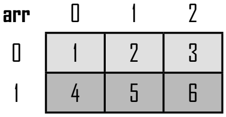
    Explanation:

    The printf statement prints the addresses of array elements. The printed addresses show that the elements of an array are stored in the memory using row major order of storage.

62. 6 6 6

    Explanation:

    The declaration statement int arr[2][3]={1,2,3,4,5,6}; creates an array as shown in the figure below:



arr	0	1	2
0	1	2	3
1	4	5	6

    The expression of form E1[E2][E3] (where one of the sub-expressions E1 or E2 is of array type or pointer type and the other sub-expressions are of integral type) gets converted to expression of form *(*(E1+E2)+E3). Hence, all the expressions arr[1][2], 1[arr][2], *(*(arr+1)+2) are equivalent and refer to the element in row 1 and column 2, i.e. 6.

63. Compilation error (Invalid indirection in function main)

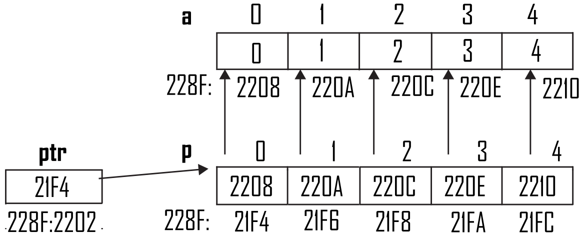
    Explanation:

    The expression 1[2][arr] gets converted to expression of form *(*(1+2)+arr). Application of dereference operator * on the expression of integer type, i.e. (1+2) is not valid and leads to a compilation error.

64. 0 228F:2202 228F:2208 228F:2208 228F:21F4 228F:2208

    Explanation:

    Suppose that the defined arrays and the pointer variable have been allocated memory as shown in the figure below. p and a being names of the arrays refer to the address of the first element of the array. Hence, the expressions p and a result in 228F:21F4 and 228F:2208, respectively. Both the expressions *p and *ptr refer to the value at memory address 228F:21F4 and result in 228F:2208. The expression &ptr refers to the address of variable ptr, i.e. 228F:2202. The expression **ptr, refers to value at memory address 228F:2208 and results in 0.



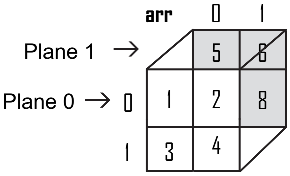
    The printf statement prints the values of the evaluated expressions to produce the mentioned result.

i As memory allocation is purely random the values of printed addresses may vary, if the code is executed on different machines or at different times.

65.
```
242F:21F8 242F:21F8 242F:21F8
242F:21F8 242F:2200 242F:2204
1 8
```

Explanation:

In an expression, if the number of subscripts used with an array name is less than the dimensions of the array, then the expression refers to an address. Suppose that array a gets allocated at the memory location 21F8 and is stored in the memory as shown in figure below:



Plane 0				Plane 1			
Row 0		Row 1		Row 0		Row 1	
Col 0	Col 1	Col 0	Col 1	Col 0	Col 1	Col 0	Col 1
a[0][0][0]	a[0][0][1]	a[0][1][0]	a[0][1][1]	a[1][0][0]	a[1][0][1]	a[1][1][0]	a[1][1][1]
1	2	3	4	5	6	7	8
21F8	21FA	21FC	21FE	2200	2202	2204	2206

The expression:

1. a refers to the starting address of the first element of the array (plane 0), i.e. 242F:21F8.
2. a[0] refers to the starting address of plane 0, i.e. 242F:21F8.
3. a[0][0] refers to the address of plane 0 and row 0, i.e. 242F:21F8.
4. a[1] refers to the address of plane 1, i.e. 242F:2200.
5. a[1][1] refers to the address of plane 1 and row 1, i.e. 242F:2204.
6. a[0][0][0] refers to the value at plane 0, row 0 and column 0, i.e. 1.
7. a[1][1][1] refers to the value at plane 1, row 1 and column 1, i.e. 8.

66. Compilation error (Size of a and b is unknown in function main)

Explanation:

Declaring an object of type void is not allowed. Hence, the declaration statement void a,b; is erroneous.

67. Compilation error (Size of type is unknown or zero)

Explanation:

Pointer arithmetic is not allowed on void pointers. Hence, the statement *v_ptr++; is erroneous.

68. The value pointed by ptr is 2

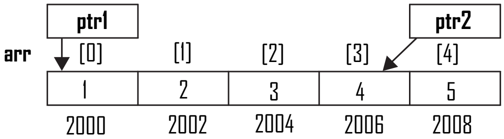
Explanation:

The pointer ptr is initialized with the address of the first element of the array arr. After incrementing it by 1, it points to the next element of the array, i.e. 2. The printf statement prints the value of the element pointed to by the pointer ptr.

69. The result of ptr2-ptr1 is 3

    Explanation:

    Suppose, the array arr gets allocated at the memory location 2000. ptr1 is initialized with the address of the first element of the array, i.e. 2000 and ptr2 is initialized with the address of arr[3], i.e. 2006. This is depicted in the figure below:



The expression ptr2-ptr1 will be computed as (ptr2-ptr1)/sizeof(int), i.e. (2006-2000)/2=3.

70. Compilation error (illegal use of pointers)

    Explanation:

    Application of multiplication operator on pointers is not allowed.

Answers to Multiple-choice Questions

71. a 72. b 73. d 74. c 75. a 76. b 77. d 78. a 79. b 80. a 81. d 82. a 83. c 84. d
85. b 86. a 87. a 88. b 89. a 90. b 91. b 92. a 93. c 94. b 95. a

Programming Exercises

Program 1 | Maximum-Minimum: Find the maximum and minimum element in a set of n elements

Algorithm:

```
Step 1: Start
Step 2: Assign the first array element to two different variables (i.e. max and min) that will hold the maximum
        and minimum value
Step 3: Loop through the remaining elements, starting from the second element. When a value larger than the
        present maximum value is found, it becomes the new maximum. Similarly, when a value smaller than the
        present minimum value is found, it becomes the new minimum
Step 4: After the termination of the loop, print the maximum and minimum values
Step 5: Stop
```

PE 6-1.c

```
//Maximum and minimum
#include<stdio.h>
main()
{
int elements[20], num, i, max, min;
printf("Enter the number of elements in the set (max. 20)\t");
scanf("%d",&num);
printf("Enter the elements:\n");
for(i=0;i<num;i++)
    scanf("%d",&elements[i]);
max=min=elements[0]; //←Let max and min is the first item
for(i=1;i<num;i++)
    if(elements[i]>max) //←if element[i]>max, then set max=element[i]
        max=elements[i];
```

Output window

```
Enter the number of elements in the set (max. 20)   5
Enter the elements:
12
-3
45
67
8
Maximum element in the set is 67
Minimum element in the set is -3
```

(Contd...)

PE 6-1.c

```c
    else if(elements[i]<min)     //←else if element[i]<min, then
            min=elements[i];     //← set min=element[i]
printf("Maximum element in the set is %d\n",max);
printf("Minimum element in the set is %d\n",min);
}
```

Output window

Program 2 | Find arithmetic mean, variance and standard deviation of n elements

Arithmetic mean is given as: $\overline{x} = \dfrac{\sum_{i=1}^{n} x_i}{n}$

Variance is given as: $\sigma_x = \dfrac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n}$

Standard deviation is given as: $\sqrt{\dfrac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n}}$

PE 6-2.c

```c
//Arithmetic mean, variance and standard deviation
#include<stdio.h>
#include<math.h>
main()
{
float elements[20], sum=0.0, mean, var, sd;
int num, i;
printf("Enter the number of elements (max. 20)\t");
scanf("%d",&num);
printf("Enter the elements:\n");
for(i=0;i<num;i++)
    scanf("%f",&elements[i]);
for(i=0;i<num;i++)
    sum=sum+elements[i];
mean=sum/num;
sum=0.0;
for(i=0;i<num;i++)
    sum=sum+(elements[i]-mean)*(elements[i]-mean);
var=sum/num;
sd=sqrt(var);
printf("Arithmetic mean is %f\n",mean);
printf("Variance is %f\n",var);
printf("Standard deviation is %f\n",sd);
}
```

Output window

```
Enter the number of elements(max. 20)    6
Enter the elements:
2.1
2.9
2.3
2.4
1.8
2.5
Arithmetic mean is 2.333333
Variance is 0.115556
Standard deviation is 0.339935
```

Program 3 | Linear Search: Given a list of n elements and a key. Find whether the given key exists in the list or not. If it exists, print its position in the list

Algorithm:

Step 1: Start
Step 2: Read the elements present in the list and store them in an array
Step 3: Read the key to be searched in the list

(*Contd...*)

Step 4: Loop to compare every element in the array with the key. When an equal value is found, print the location where the match has been found. If the loop finishes without finding a match, the search fails and print the message that key is not present in the list

Step 5: Stop

PE 6-3.c

```
//Linear Search
#include<stdio.h>
main()
{
int elements[20], num, i, key, found=0;
printf("Enter the number of elements (max. 20)\t");
scanf("%d",&num);
printf("Enter the elements:\n");
for(i=0;i<num;i++)
  scanf("%d",&elements[i]);    //←Read elements in the list
printf("Enter the key that you want to search\t");
scanf("%d",&key);           //←Read the key to be searched
for(i=0;i<num;i++)          //←Loop
  if(elements[i]==key)      //←Comparison of element & key
   {                        //←Key found
       printf("%d exists at location no. %d\n",key, i+1);
       found=1;
   }
if(found==0)                //←Key not found in the list
     printf("%d does not exist in the list",key);
}
```

Output window

```
Enter the number of elements(max. 20)     6
Enter the elements:
12
10
5
-3
14
2
Enter the key that you want to search     -3
-3 exists at location no. 4
```

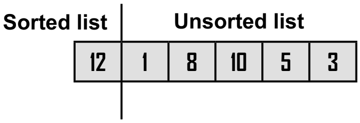
Program 4 | Insertion Sort: Given list of n elements. Arrange them in an ascending order

Principle:

Insertion Sort works on the principle of sorting by insertion. Any given unsorted list can be divided into two lists such that one is sorted and the other is unsorted. For example, the given unsorted list

12	1	8	10	5	3

can be divided into two parts such that one list is sorted and other list is unsorted. The divided list is shown as:

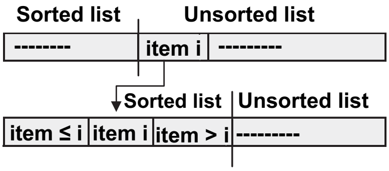


Initially the sorted list consists of zero or one element, as the list containing zero or one element is always sorted and the unsorted list consists of the rest of the elements.

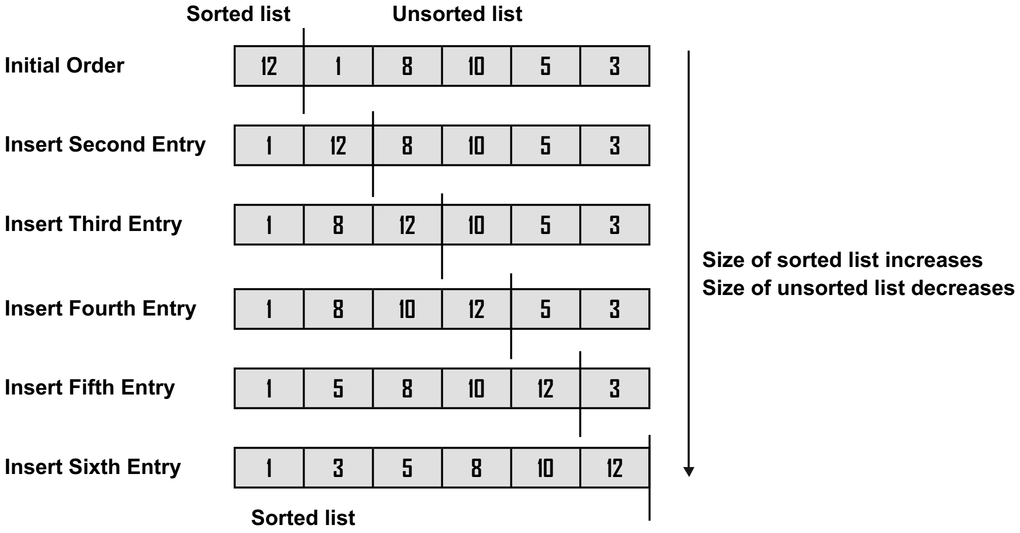
Insertion Sort sorts by removing one element from the unsorted list at a time and inserting it at a proper position in the sorted listed. To make room for the insertion, some of the elements in the sorted list need to be moved. Each iteration of Insertion Sort reduces the size of the unsorted list by one and increases the size of the sorted list by one. Ultimately, the unsorted list will vanish and the entire list will be sorted.

The general procedure of the Insertion Sort is shown in the figure below:

(*Contd...*)



Insertion Sort sorts the given list as shown below:


Sorted list
Unsorted list
Size of sorted list increases
Size of unsorted list decreases
Sorted list

Initial Order 12 | 1 8 10 5 3

Insert Second Entry 1 12 | 8 10 5 3

Insert Third Entry 1 8 12 | 10 5 3

Insert Fourth Entry 1 8 10 12 | 5 3

Insert Fifth Entry 1 5 8 10 12 | 3

Insert Sixth Entry 1 3 5 8 10 12 |

PE 6-4.c

```
//Insertion Sort
#include<stdio.h>
main()
{
int list[20], num, current, i, j;
printf("Enter the number of elements (max. 20)\t");
scanf("%d",&num);     //←Read the number of elements in the list
printf("Enter the elements:\n");
for(i=0;i<num;i++)
     scanf("%d",&list[i]);     //←Read the elements of the list
//←First element is sorted and the rest of the list is unsorted
for(i=1;i<num;i++)
     if(list[i]<list[i-1])          //←Remove element from the unsorted
     {                              //list and place it at proper position
          current=list[i];     //in the sorted list
          for(j=i-1;j>=0;j--)
          {
               list[j+1]=list[j];
                    if(j==0||list[j-1]<=current)
                         break;
          }
```

Output window

```
Enter the number of elements(max. 20)    6
Enter the elements:
12
10
5
-3
14
2
After sorting, elements are:
-3
2
5
10
12
14
```

(*Contd...*)

PE 6-4.c	Output window

` list[j]=current;`
` }`
`printf("After sorting, elements are:\n");`
`for(i=0;i<num;i++)      //←Print sorted list`
`  printf("%d\n",list[i]);`
`}`	

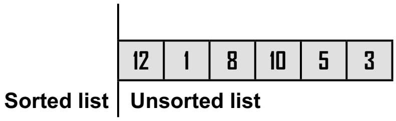
Program 5 | Selection Sort: Given a list of n elements. Arrange them in an ascending order

Insertion Sort has one major disadvantage. Insertion of an element removed from the unsorted list into the sorted list requires the elements in the sorted list to be moved to create space for the new element. Consider the insertion of sixth entry in the previous program. Insertion of 3 into the sorted list requires the movement of 5, 8, 10 and 12. These excessive movements become very expensive especially if the elements are very large such as records of employee's personal file or student transcripts. It would be far more efficient if an entry being moved could immediately be placed in its final position. Selection Sort accomplishes this goal and works on the following principle:

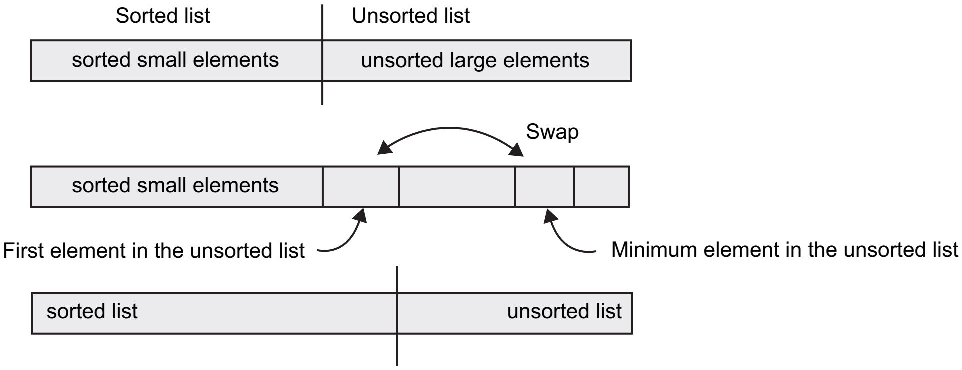
Principle:
Selection Sort works on the principle of sorting by selection. The given unsorted list is initially divided into two lists—the sorted list containing no element and the unsorted list containing all the elements. For example, the given unsorted list

12	1	8	10	5	3

can be divided into two parts as:

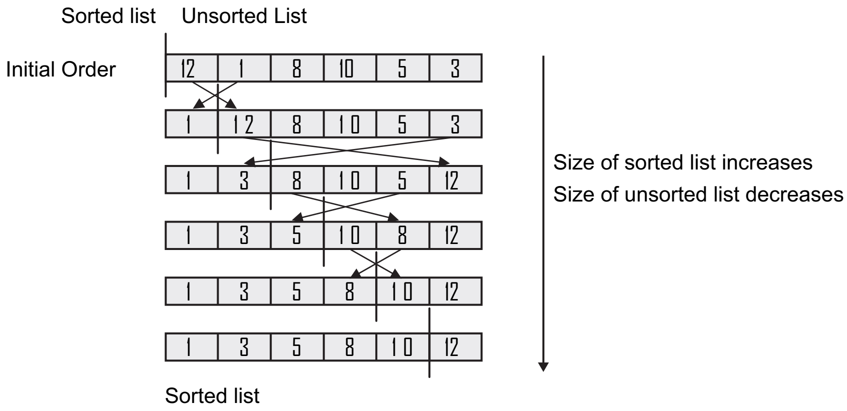


Selection Sort selects the minimum element from the unsorted list and exchanges it with the first element in the unsorted list. The selected element has moved to its final position; hence, the size of the sorted list is increased by one and the size of the unsorted list is decreased by one. This process of selecting the minimum element from the unsorted list, exchanging it with the first element in the unsorted list and then increasing the size of the sorted list and decreasing the size of the unsorted list by one is repeatedly followed till the entire list becomes sorted. The general procedure of Selection Sort is shown in the figure below:



(*Contd...*)

Selection Sort sorts the given list as shown below:



PE 6-5.c

```
//Selection Sort
#include<stdio.h>
main()
{
int list[20], num,min, temp, i, j;
printf("Enter the number of elements (max. 20)\t");
scanf("%d",&num);     //←Read number of elements in the list
printf("Enter the elements:\n");
for(i=0;i<num;i++)
     scanf("%d",&list[i]);     //←Read the elements
for(i=0;i<num-1;i++)          //←Initially entire list is unsorted
{
     min=i;
     for(j=i+1;j<num;j++)     //←Select minimum element in the list
          if(list[j]<list[min])
               min=j;
     {  //←Place selected element at 1st position in the unsorted list
     temp=list[min];
     list[min]=list[i];
     list[i]=temp;
     }
}
printf("After sorting, elements are:\n");
for(i=0;i<num;i++)          //←Print sorted list
     printf("%d\n",list[i]);
}
```

Output window

```
Enter the number of elements(max. 20)     6
Enter the elements:
12
10
5
-3
14
2
After sorting, elements are:
-3
2
5
10
12
14
```

Program 6 | Bubble Sort: Given a list of n elements. Arrange them in an ascending order

Principle:

Bubble Sort works on the following observation:

'Bubbles (or lighter elements) rise up in water and heavier elements sink'

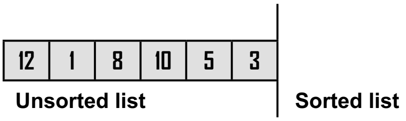
The given unsorted list is initially divided into two lists—the sorted list containing no element and the unsorted list containing all the elements. For example, the given unsorted list

(Contd...)

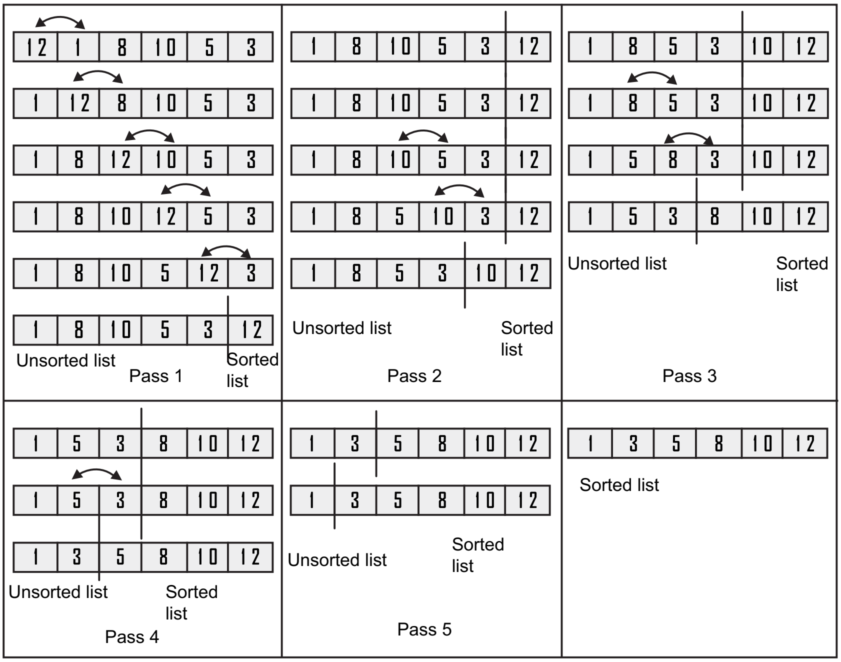
Program 6 | Bubble Sort: Given a list of n elements. Arrange them in an ascending order

12	1	8	10	5	3

can be divided into two parts as:



Bubble Sort scans the unsorted list from left to right and swaps elements when a pair of adjacent elements is found to be out of order. After one complete iteration (also known as a **pass**), the heaviest element (i.e. the largest element) is at the right end of the unsorted list, but the earlier elements may still be out of order. The size of unsorted list is decreased by one and the size of the sorted list is increased by one. This process is repeated till the unsorted list vanishes and the entire list becomes sorted. Bubble Sort sorts the given list as shown below:



PE 6-6.c

```
//Bubble Sort
#include<stdio.h>
main()
{
int list[20], num,min, temp, i, j;
printf(“Enter the number of elements (max. 20)\t”);
scanf(“%d”,&num);          //←Read number of elements in the list
printf(“Enter the elements:\n”);
for(i=0;i<num;i++)
     scanf(“%d”,&list[i]);   //←Read the elements
for(i=0;i<num-1;i++)         //←Passes
```

Output window

```
Enter the number of elements(max. 20)     6
Enter the elements:
12
10
5
-3
14
2
After sorting, elements are:
-3
2
```

(*Contd...*)

PE 6-6.c

```
    for(j=0;j<num-1-i;j++)
        if(list[j]>list[j+1])     //←If elements are out of order, swap them
        {
            temp=list[j];
            list[j]=list[j+1];
            list[j+1]=temp;
        }
printf(“After sorting, elements are:\n”);
for(i=0;i<num;i++)              //←Print sorted list
    printf(“%d\n”,list[i]);
}
```

Output window

```
5
10
12
14
```

Program 7 | Given two sorted one-dimensional arrays A and B of size m and n, respectively. Merge them into a single-sorted array C that contains every element from arrays A and B in ascending order

PE 6-7.c

```
//Merge two Sorted arrays into one
#include<stdio.h>
main()
{
int A[20], B[20], C[40] ;
int i, j, l, h, m, n;
printf(“Enter the number of elements in A (max. 20)\t”);
scanf(“%d”,&m);            //←Read number of elements in A
printf(“Enter the elements in sorted order:\n”);
for(i=0;i<m;i++)
    scanf(“%d”,&A[i]);     //←Read the elements
printf(“Enter the number of elements in B (max. 20)\t”);
scanf(“%d”,&n);            //←Read number of elements in B
printf(“Enter the elements in sorted order:\n”);
for(i=0;i<n;i++)
    scanf(“%d”,&B[i]);     //←Read the elements
i=0; j=0; h=0;
while(i<m || j<n)
{
    if(A[i]<=B[j])
    {
        C[h]=A[i];
        i++;
    }
    else
    {
        C[h]=B[j];
        j++;
    }
    h++;
}
if(i==m)
    for(l=j;l<n;l++)
        C[h++]=B[l];
else if(j==n)
```

Output window

```
Enter the number of elements in A (max. 20)     5
Enter the elements in sorted order:
1 3 5 7 9
Enter the number of elements in B (max. 20)     4
Enter the elements in sorted order:
2 4 6 8
After merging, elements are:
1 2 3 4 5 6 7 8 9
```

(Contd...)

PE 6-7.c

```
    for(l=i;l<m;l++)
        C[h++]=A[l];
printf("After merging, elements are:\n");
for(i=0;i<m+n;i++)          //←Print merged array C
    printf("%d ",C[i]);
}
```

Output window

Program 8 | Matrix addition: Add two matrices of order m × n

PE 6-8.c

```
//Matrix Addition
#include<stdio.h>
main()
{
int mat1[10][10], mat2[10][10], resultant[10][10];
int m, n, row, col;
printf("Enter the order of matrices (max. 10 by 10)\t");
scanf("%d %d",&m, &n);
printf("Enter the elements of matrix-1:\n");
for(row=0;row<m;row++)
{
    for(col=0;col<n;col++)
        scanf("%d",&mat1[row][col]);
}
printf("Enter the elements of matrix-2:\n");
for(row=0;row<m;row++)
{
    for(col=0;col<n;col++)
        scanf("%d",&mat2[row][col]);
}
for(row=0;row<m;row++)
    for(col=0;col<n;col++)
        resultant[row][col]=mat1[row][col]+mat2[row][col];
printf("The result of matrix addition is:\n");
for(row=0;row<m;row++)
{
    for(col=0;col<n;col++)
        printf("%d ",resultant[row][col]);
    printf("\n");
}
}
```

Output window

```
Enter the order of matrices (max. 10 by 10)     3 3
Enter the elements of matrix-1:
1 2 3
4 5 6
7 8 9
Enter the elements of matrix-2:
2 3 4
1 2 3
1 1 0
The result of matrix addition is :
3 5 7
5 7 9
8 9 9
```

Program 9 | Matrix multiplication: Multiply two matrices

Given two matrices A and B

$$A = \begin{bmatrix} \multicolumn{2}{c}{\overrightarrow{A_{11} \quad A_{12}}} \\ A_{21} & A_{22} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} B_{11} & \big\downarrow & B_{12} & B_{13} \\ B_{21} & & B_{22} & B_{12} \end{bmatrix}$$

(*Contd...*)

The result of the matrix multiplication is given as:

$$C_{2\times3} = A_{2\times2}\,B_{2\times3} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} & A_{11}B_{13} + A_{12}B_{23} \\ A_{21}B_{31} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} & A_{21}B_{13} + A_{22}B_{23} \end{bmatrix}$$

PE 6-9.c

```c
//Matrix Multiplication
#include<stdio.h>
#include<stdlib.h>
main()
{
int mat1[10][10], mat2[10][10], resultant[10][10]={0};
int  m1, n1, m2, n2, i, j, k;
printf("Enter the order of matrix-1 (max. 10 by 10)\t");
scanf("%d %d",&m1, &n1);
printf("Enter the elements of matrix-1:\n");
for(i=0;i<m1;i++)
{
    for(j=0;j<n1;j++)
        scanf("%d",&mat1[i][j]);
}
printf("Enter the order of matrix-2 (max. 10 by 10)\t");
scanf("%d %d",&m2, &n2);
printf("Enter the elements of matrix-2:\n");
for(i=0;i<m2;i++)
{
    for(j=0;j<n2;j++)
        scanf("%d",&mat2[i][j]);
}
if(n1!=m2)
{
    printf("Matrices are not compatible for multiplication\n");
    exit(1);
}
else
{
    for(i=0;i<m1;i++)
        for(j=0;j<n2;j++)
            for(k=0;k<n1;k++)
                resultant[i][j]=resultant[i][j]+mat1[i][k]*mat2[k][j];
}
printf("The result of matrix multiplication is:\n");
for(i=0;i<m1;i++)
{
    for(j=0;j<n2;j++)
        printf("%d ",resultant[i][j]);
    printf("\n");
}
}
```

Output window

```
Enter the order of matrix-1 (max. 10 by 10)     2 3
Enter the elements of matrix-1:
1 2 3
4 5 6
Enter the order of matrix-2 (max. 10 by 10)     3 3
Enter the elements of matrix-2:
2 3 4
1  2 3
1  1  0
The result of matrix multiplication is :
7 10 10
19 28 31
```

Program 10 | Find the sum of principal diagonal elements of a square matrix

The set of elements extending from the upper-left-most corner to the lower-right-most corner in a square matrix are known as **principal diagonal elements**. An element A_{ij} of a square matrix is principle diagonal element if and only if i=j.

PE 6-10.c

```
//Sum of principle diagonal elements
#include<stdio.h>
main()
{
int matrix[10][10];
int order, sum=0, i, j;
printf("Enter the order of the square matrix(max. 10)\t");
scanf("%d",&order);
printf("Enter the elements of matrix:\n");
for(i=0;i<order;i++)
{
    for(j=0;j<order;j++)
        scanf("%d",&matrix[i][j]);
}
for(i=0;i<order;i++)
    sum=sum+matrix[i][i];
printf("Sum of elements of principal diagonal is %d",sum);
}
```

Output window

```
Enter the order of the square matrix (max. 10)    3
Enter the elements of matrix:
1 2 3
4 5 6
7 8 9
Sum of elements of principal diagonal is 15
```

Program 11 | Matrix transpose: Find the transpose of a given matrix

The **transpose** of the matrix A is another matrix A^T, which can be found by any one of the following actions:

1. Writing the rows of A as the columns of A^T
2. Writing the columns of A as the rows of A^T
3. Reflect A about its main diagonal to obtain A^T(only possible in case of square matrix).

The transpose of an m×n matrix A with elements A_{ij} is an n×m matrix $A^T=A_{ji}$, $1 \le i \le n$ and $1 \le j \le m$.

PE 6-11.c

```
//Matrix Transpose
#include<stdio.h>
main()
{
int matrix[10][10], matrix_transpose[10][10];
int  m, n, i, j;
printf("Enter the order of the matrix(max. 10 by 10)\t");
scanf("%d %d",&m, &n);
printf("Enter the elements of the matrix:\n");
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
        scanf("%d",&matrix[i][j]);
}
for(i=0;i<n;i++)
```

Output window

```
Enter the order of matrix (max. 10 by 10)    2 4
Enter the elements of matrix:
1 2 3 4
5 6 7 8
Transpose of the matrix is:
1 5
2 6
3 7
4 8
```

(Contd...)

PE 6-11.c

```
    for(j=0;j<m;j++)
        matrix_transpose[i][j]=matrix[j][i];
printf("Transpose of the matrix is:\n");
for(i=0;i<n;i++)
{
    for(j=0;j<m;j++)
        printf("%d ",matrix_transpose[i][j]);
    printf("\n");
}
}
```

Output window

Program 12 | Check whether a given square matrix is symmetric or not

A square matrix A is **symmetric** if $A=A^T$ (i.e. the matrix is equal to its transpose).

PE 6-12.c

```
//Symmetric matrix
#include<stdio.h>
main()
{
int matrix[10][10], matrix_transpose[10][10], unequal=0;
int i,j,order;
printf("Enter the order of the square matrix(max. 10 by 10)\t");
scanf("%d",&order);
printf("Enter the elements of the matrix:\n");
for(i=0;i<order;i++)
{
    for(j=0;j<order;j++)
        scanf("%d",&matrix[i][j]);
}
for(i=0;i<order;i++)
    for(j=0;j<order;j++)
        matrix_transpose[i][j]=matrix[j][i];
for(i=0;i<order;i++)
{
    for(j=0;j<order;j++)
        if(matrix[i][j]!=matrix_transpose[i][j])
        {
            unequal=1;
            break;
        }
    if(unequal==1)
        break;
}
if(unequal==0)
    printf("The matrix is symmetric\n");
else
    printf("The matrix is not symmetric\n");
}
```

Output window

```
Enter the order of the square matrix (max. 10 by 10)     3
Enter the elements of matrix:
1 2 3
2 1 4
3 4 1
The matrix is symmetric
```

Program 13 | Upper triangular matrix: Extract the upper triangular matrix from a square matrix

A square matrix in which all the elements below the main (i.e. principal) diagonal are zero is known as **upper triangular matrix** and a square matrix in which all the elements above the main diagonal are zero is known as **lower triangular matrix**.

Upper triangular matrix can be extracted from a square matrix by extracting the elements of principle diagonal and the elements that lie above it.

PE 6-13.c

```
//Extraction of Upper Triangular matrix
#include<stdio.h>
main()
{
int matrix[10][10], ut_matrix[10][10], unequal=0;
int i,j,order;
printf(“Enter the order of the square matrix(max. 10 by 10)\t”);
scanf(“%d”,&order);
printf(“Enter the elements of the matrix:\n”);
for(i=0;i<order;i++)
{
    for(j=0;j<order;j++)
        scanf(“%d”,&matrix[i][j]);
}
for(i=0;i<order;i++)
    for(j=0;j<order;j++)
        if(i<=j)
            ut_matrix[i][j]=matrix[i][j];
        else
            ut_matrix[i][j]=0;
printf(“Upper Triangular matrix is:\n”);
for(i=0;i<order;i++)
{
    for(j=0;j<order;j++)
        printf(“%d “,ut_matrix[i][j]);
    printf(“\n”);
}
}
```

Output window

```
Enter the order of the square matrix (max. 10 by 10)     3
Enter the elements of the matrix:
1 2 3
2 1 4
3 4 1
Upper Triangular matrix is:
1 2 3
0 1 4
0 0 1
```

Program 14 | Strictly upper triangular matrix: Check whether a given matrix is strictly upper triangular or not

An upper triangular matrix is **strictly upper triangular** if the elements of the principal diagonal are zero.

PE 6-14.c

```
//Strictly Upper Triangular matrix
#include<stdio.h>
main()
{
```

Output window

```
Enter the order of the square matrix (max. 10 by 10)     3
Enter the elements of matrix:
0 2 3
0 0 4
```

(*Contd...*)

PE 6-14.c

```
int matrix[10][10], notzero=0;
int i,j,order;
printf("Enter the order of the square matrix(max. 10 by 10)\t");
scanf("%d",&order);
printf("Enter the elements of the matrix:\n");
for(i=0;i<order;i++)
{
    for(j=0;j<order;j++)
        scanf("%d",&matrix[i][j]);
}
for(i=0;i<order;i++)
{
    for(j=0;j<order;j++)
        if(i>=j)
            if(matrix[i][j]!=0)
            {
                notzero=1;
                break;
            }
    if(notzero==1)
        break;
}
if(notzero==1)
    printf("The given matrix is not strictly upper triangular\n");
else
    printf("The given matrix is strictly upper triangular\n");
}
```

Output window

```
0 0 0
The given matrix is strictly upper triangular
```

Output window (second execution)

```
Enter the order of the square matrix (max. 10 by 10)     3
Enter the elements of matrix:
6 2 3
0 0 4
2 0 0
The given matrix is not strictly upper triangular
```

Program 15 | Matrix Inverse: Find the inverse of a 3 × 3 matrix

The **inverse** of a square matrix A, is a matrix A^{-1} such that $AA^{-1}=I$, where I is the identity matrix. The matrix A has an inverse if and only if the determinant of A (written as |A|) is not equal to zero. A matrix whose inverse exists is known as **invertible matrix**

Finding the inverse of a matrix using Gauss–Jordan elimination method:

Step 1: Start

Step 2: Read the elements of the matrix A whose inverse is to be found

Step 3: Check whether its determinant is zero or not. If it is zero, print that inverse does not exist and stop, else proceed to Step 4

Step 4: Form the augmented matrix B. It is formed by augmenting the matrix A with an identity matrix of the same dimensions. If the matrix A is of order m × n, the augmented matrix B = [A|I] is of order m × 2n. It consists of two parts: the first part corresponds to A and the second part corresponds to I

Step 5: Apply elementary row operations on the augmented matrix B so that its first part reduces to identity matrix. By performing these row operations, the inverse of the matrix A appears in the second part

Step 6: The matrix augmentation can be undone to retrieve the inverse of the matrix

Step 7: Print the inverse of the matrix

Step 8: Stop

(*Contd...*)

Example:

$$A = \begin{bmatrix} 2 & 3 & 1 \\ 1 & 1 & 2 \\ 2 & 3 & 4 \end{bmatrix} \qquad I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & 3 & 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 0 & 1 & 0 \\ 2 & 3 & 4 & 0 & 0 & 1 \end{bmatrix}$$

Step 1: R_1->R_1/B[0][0]

$$B = \begin{bmatrix} 1 & 3/2 & 1/2 & 1/2 & 0 & 0 \\ 1 & 1 & 2 & 0 & 1 & 0 \\ 2 & 3 & 4 & 0 & 0 & 1 \end{bmatrix}$$

Step 2: R_2->R_2-B[1][0]*R_1

$$B = \begin{bmatrix} 1 & 3/2 & 1/2 & 1/2 & 0 & 0 \\ 0 & -1/2 & 3/2 & -1/2 & 1 & 0 \\ 2 & 3 & 4 & 0 & 0 & 1 \end{bmatrix}$$

Step 3: R_3->R_3-B[2][0]*R_1

$$B = \begin{bmatrix} 1 & 3/2 & 1/2 & 1/2 & 0 & 0 \\ 0 & -1/2 & 3/2 & -1/2 & 1 & 0 \\ 0 & 0 & 3 & -1 & 0 & 1 \end{bmatrix}$$

Step 4: R_2->R_2/B[1][1]

$$B = \begin{bmatrix} 1 & 3/2 & 1/2 & 1/2 & 0 & 0 \\ 0 & 1 & -3 & 1 & -2 & 0 \\ 0 & 0 & 3 & -1 & 0 & 1 \end{bmatrix}$$

Step 5: R_1->R_1-B[0][1]*R_2

$$B = \begin{bmatrix} 1 & 0 & 5 & -1 & 3 & 0 \\ 0 & 1 & -3 & 1 & -2 & 0 \\ 0 & 0 & 3 & -1 & 0 & 1 \end{bmatrix}$$

Step 6: R_3->R_3-B[2][0]*R_2

$$B = \begin{bmatrix} 1 & 0 & 5 & -1 & 3 & 0 \\ 0 & 1 & -3 & 1 & -2 & 0 \\ 0 & 0 & 3 & -1 & 0 & 1 \end{bmatrix}$$

Step 7: R_3->R_3/B[2][2]

$$B = \begin{bmatrix} 1 & 0 & 5 & -1 & 3 & 0 \\ 0 & 1 & -3 & 1 & -2 & 0 \\ 0 & 0 & 1 & -1/3 & 0 & 1/3 \end{bmatrix}$$

Step 8: R_1->R_1-B[0][2]*R_3

$$B = \begin{bmatrix} 1 & 0 & 0 & 2/3 & 3 & -5/3 \\ 0 & 1 & -3 & 1 & -2 & 0 \\ 0 & 0 & 1 & -1/3 & 0 & 1/3 \end{bmatrix}$$

(Contd...)

Step 9: R_2->R_2-B[1][2]*R_3

$$B = \begin{bmatrix} 1 & 0 & 0 & 2/3 & 3 & -5/3 \\ 0 & 1 & 0 & 0 & -2 & 1 \\ 0 & 0 & 1 & -1/3 & 0 & 1/3 \end{bmatrix}$$

Step 10: Undo the matrix augmentation and print inverse

$$A^{-1} = \begin{bmatrix} 2/3 & 3 & -5/3 \\ 0 & -2 & 1 \\ -1/3 & 0 & 1/3 \end{bmatrix}$$

PE 6-15.c

```
//Inverse of a matrix
#include<stdio.h>
main()
{
float matrix[3][3], aug_matrix[3][6];
float identity[3][3]={1,0,0,0,1,0,0,0,1};     //←3×3 identity matrix
float c,r, sub, det;
int i,j,order=3,k, row, col;
printf("Enter the elements of 3 by 3 matrix:\n");
for(i=0;i<order;i++)
{
    for(j=0;j<order;j++)
        scanf("%f",&matrix[i][j]);     //←Read the elements of the matrix
}
//←Calculate the determinant of the matrix
det=matrix[0][0]*(matrix[1][1]*matrix[2][2]-matrix[2][1]*matrix[1][2]) -
matrix[0][1]*(matrix[1][0]*matrix[2][2]-matrix[2][0]*matrix[1][2]) +
matrix[0][2]*(matrix[1][0]*matrix[2][1]-matrix[2][0]*matrix[1][1]);
if(det!=0)     //←if determinant is not zero inverse can be found
{
    for(i=0;i<order;i++)     //←augmenting the matrix with identity matrix
        for(j=0;j<order;j++)
        {
            aug_matrix[i][j]=matrix[i][j];
            aug_matrix[i][j+3]=identity[i][j];
        }
//←Elementary row operations
    for(i=0;i<order;i++)
        for(j=0;j<order;j++)
        {
            if(i==j)
            {
//←Implementing Steps 1, 4 and 7 described in the example above
                 c=aug_matrix[i][i];
                 for(k=0;k<6;k++)
                     aug_matrix[i][k]=aug_matrix[i][k]/c;
//←Implementing Steps 2,3,5,6,8 and 9 described in the example above
                for(row=0;row<order;row++)
```

Output window

```
Enter the elements of 3 by 3 matrix:
2 3 1
1 1 2
2 3 4
Inverse of the matrix is:
 0.67      3.00     -1.67
 0.00     -2.00      1.00
-0.33      0.00      0.33
```

(*Contd...*)

PE 6-15.c

```
                {
                    sub=aug_matrix[row][j];
                    for(col=0;col<6;col++)
                    if(row!=i)
                    {
                        aug_matrix[row][col]-=sub*aug_matrix[i][col];
                    }
                }
            }
        }
    printf(“Inverse of the matrix is:\n”);
    for(i=0;i<order;i++)     //←Printing the inverse of the matrix
    {
        for(j=0;j<order;j++)
            printf(“%5.2f ”,aug_matrix[i][j+3]);
        printf(“\n”);
    }
}
else     //←if determinant is zero, print that inverse does not exist
    printf(“Inverse does not exist”);
}
```

Output window

Test Yourself

1. Fill in the blanks in each of the following:
   a. An array is used for the storage of ____________ data.
   b. The array index in C language starts with ____________.
   c. The elements of an array are always stored in ____________ memory locations.
   d. The size specifier in an array declaration must be a compile time expression of ____________ type.
   e. The elements of an array can be accessed by using ____________ operator.
   f. The object pointed to by a pointer can be indirectly accessed by using ____________ operator.
   g. The expression equivalent to the expression arr[5][4], where arr is an integer array is ____________.
   h. The biggest advantage of arrays is their ____________ capabilities.
   i. In an expression, if the number of subscripts used with the array is less than the dimensions of the array, the expression always refers to a/an ____________.
   j. The comparison of two null pointers always results in ____________.

2. State whether each of the following is true or false. If false, explain why.

   a. In an array declaration, the number of initializers in the initialization list should be less than or at most equal to the value of size specifier.
   b. The index of an array must be a positive integer greater than zero.
   c. A pointer variable can be initialized with a constant value zero.
   d. A pointer to any type of object can be assigned to a pointer of type void* without explicit type casting.
   e. A void pointer can be assigned to a pointer variable without explicit type casting.
   f. The name of the array refers to the base address of the complete array.
   g. The size of an array cannot be changed at the run time.
   h. If the size specifier is not mentioned in an array declaration, the size of the array is automatically initialized to a single element.
   i. Multi-dimensional arrays in C are stored in the memory using column major order of storage.
   j. The declaration statement int* a[10]; declares a as a pointer to an integer array of 10 elements.

3. Programming exercises:

   a. Write a C program to find the sum of all the elements of an array.
   b. An array consists of integers. Write a C program to count the number of elements less than, greater than and equal to zero.
   c. Write a C program to check whether a given matrix is skew-symmetric or not.
   d. Write a C program to extract lower-triangular matrix from a square matrix.
   e. Write a C program that returns the position of the largest element in an array.
   f. In a class there are twenty students and each student undergoes five courses. Write a C program to find out the average marks secured by each student and the overall average of the class.

7

STRINGS AND CHARACTER ARRAYS

Learning Objectives

In this chapter, you will learn about:

- Strings
- How strings are represented in C language
- The usage of character arrays to store strings
- Null character and its importance in string representation
- Various string operations like copy, compare, concatenate, etc.
- String library functions
- How to store and work with a list of strings
- Command line arguments

7.1 Introduction

The character string is one of the most useful and important data types. You have used the character strings all the way in the previous chapters, but there is still much to learn about them. The C string library provides a wide range of functions for strings like reading, writing, copying, comparing, combining, searching, etc. This chapter will add these capabilities to your programming skills.

7.2 Strings

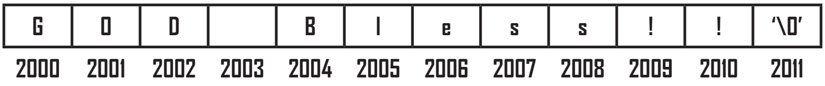
A **character string literal constant** or just a **string literal** is a sequence of zero or more characters enclosed within double quotes. For example, "GOD Bless!!" is a string literal constant. Knowingly or unknowingly, you have used strings in abundance with the printf function in previous chapters.

The important points about the string literal constants are as follows:

1. String literals are enclosed within double quotes, whereas character literals are enclosed within single quotes, e.g. "A" is a string literal constant while 'A' is a character literal constant.
2. The used double quotes are not part of the string literal and are used only to delimit it.
3. Every string literal constant is automatically terminated by the **null character**,✍ i.e. '\0'.

✍ The character constant with an ASCII value of zero is known as a **null character** and is written as '\0'.

4. Like other literal constants, string literal constants are also stored in the memory. The characters enclosed within double quotes and the terminating null character are stored in the contiguous memory locations in a similar manner as arrays are stored in the memory. Thus, a string literal constant "GOD Bless!!" will be stored in the memory as shown in Figure 7.1.



G	O	D		B	l	e	s	s	!	!	'\0'
2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011

Figure 7.1 | Storage of string literal constant "GOD Bless!!"

5. Unlike other literal constants, the amount of the memory space required for storing a string literal constant is not fixed and depends upon the number of characters present in a string literal.
6. The number of bytes required to store a string literal constant is one more than the number of characters present in it. The additional byte is required for storing the terminating null character. For example, the memory required to store the string literal "xyz" is 4 bytes. The code snippet in Program 7-1 illustrates this fact.

Prog 7-1.c	Output window

//Memory requirement of string literal
#include<stdio.h>
main()
{
 printf(“Memory requirement of \"xyz\" is %d bytes”,sizeof(“xyz”));
}
	Memory requirement of “xyz” is 4 bytes
	Remarks:
	• Escape sequence \" is used to print double quotes
	• The additional byte is required to store the terminating null character

Program 7-1 | A program to illustrate that the memory space required by a string literal constant is one more than the number of characters in it

7. The **length of a string** is defined as the number of characters present in it. The terminating null character is not counted while determining the length of a string. For example, the length of the string literal “xyz” is 3. The code snippet in Program 7-2 verifies this fact.

Prog 7-2.c	Output window

//Length of string literal
#include<stdio.h>
//string.h header file is to be included for using string library functions
#include<string.h>
main()
{
printf(“Length of string literal \"xyz\" is %d characters”,strlen(“xyz”));
}
	Length of string literal “xyz” is 3 characters
	Remarks:
	• The terminating null character is not counted while determining the length of a string
	• strlen is a string library function that determines the length of a string
	• The prototype of the strlen function is present in the header file string.h

Program 7-2 | A program to find the length of a string

8. A string literal constant of zero length is known as an **empty string**. The empty string is written as "", i.e. no character enclosed within double quotes. Although an empty string is of zero length, it still takes 1 byte in the memory for the storage of a null character.
9. In C language, **string type** is not separately available, and **character pointers** are used to represent strings. Thus, the type of string literal (e.g. “xyz”) is **const char***. The constant pointer refers to the address of the first element of the string. The strings represented and interpreted in this way are known as **C-style character strings**. The code snippet in Program 7-3 illustrates that a string literal decomposes into a pointer (const char*) pointing to the first character of the string.

Prog 7-3.c	Output window

//C-style character strings are represented by const char*
#include<stdio.h>
main()
{
  printf(“The first character of string literal \"xyz\" is %c\n”,*“xyz”);
  printf(“The second character of string literal \"xyz\" is %c”,*(“xyz”+1));
}
	The first character of string literal “xyz” is x
	The second character of string literal “xyz” is y
	Remarks:
	• The type of string literals is const char*
	• “xyz” refers to the address of the first element of the string, i.e. the address of x
	• Hence, dereferencing “xyz” outputs x

Program 7-3 | A program to illustrate that the string literal constant refers to the address of its first element

10. Since a string literal constant refers to a constant character pointer and does not have a modifiable l-value, only the operations that can be applied on constant pointers can be applied on C-style character strings. The application of any other operator on string literals that cannot be applied on constant pointers leads to 'L-value required' compilation error. The code snippet in Program 7-4 illustrates this fact.

Line	Prog 7-4.c	Output window

1
2
3
4
5
6
	//String literal refers to constant character pointer
	#include<stdio.h>
	main()
	{
	    printf("The first character of string literal \"xyz\" is %c",*"xyz"++);
	}
		Compilation error "L-value required"
		Remarks:
		• The expression *"xyz"++ will be interpreted as *("xyz"++)
		• The application of the post-increment operator on "xyz" leads to the compilation error as "xyz" does not have a modifiable l-value

Program 7-4 | A program to illustrate that a string literal constant refers to a constant pointer and does not have a modifiable l-value

11. Since C-style character string is of const char* type, it can be assigned to or initialized to a character pointer variable. The following statements are valid:

```
char *string="Strings!!!";
string="Trings!!!";
```

12. Adjacent string literal constants are concatenated. This concatenation is carried out during the preprocessing phase. The code snippet in Program 7-5 illustrates this fact.

Line	Prog 7-5.c	Output window

1
2
3
4
5
6
	//Adjacent string literal constants get concatenated
	#include<stdio.h>
	main()
	{
	    printf("GOD Bless " "us" "!!!");
	}
		GOD Bless us!!!
		Remark:
		• Adjacent string literal constants in line number 5 are concatenated and then printed

Program 7-5 | A program to illustrate that the adjacent string literal constants get concatenated

7.3 Character Arrays

An integer variable can store the value of an integer constant. For example, statement int a=10; creates a variable a to store an integer constant 10. Similarly, float variables can store floating point constants, and character variables can be used to store character constants. Now, the question that arises here is: 'Can we create a variable that can be used to store a string literal constant?'. The answer to this question is YES! We can create variables of type char[] (i.e. **character arrays**) to store string constants.

The general form of a **string variable** or **a character array** declaration is:

<s_class_specf><type_qualifier><type_modifier>**char identifier[**<size_specifier>**]**<=initialization_list OR string literal>;

The important points about string variable declarations are as follows:

1. The terms enclosed within angular brackets (i.e. <>) are optional and might not be present in a declaration statement. The terms shown in **bold** are the mandatory parts of a string variable declaration.
2. Since a string variable is a character array, all the syntactic rules discussed in Chapter 6 for declaring arrays are applicable for declaring string variables as well.
3. The size specification is optional if a string variable is explicitly initialized.
4. The string variable or character array can be initialized in two different ways:
    a. **By using string literal constant:** In the declaration statement char str[6]="Hello"; the character array or string variable str is initialized with a string literal constant "Hello". It will be stored in the memory as shown in Figure 7.2.
    b. **By using initialization list:** The alternate way to initialize a character array is by using a list of character initializers. The declaration statement char str[6]= {'H','e','l','l','o','\0'}; initializes the locations of the character array str with character initializers. The character array str will be stored in the memory in the same way as shown in Figure 7.2.

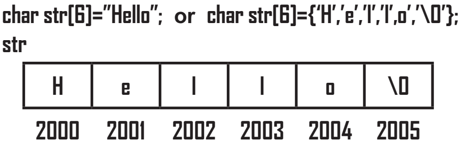

char str[6]="Hello"; or char str[6]={'H','e','l','l','o','\0'};
str
H
e
l
l
o
\0
2000
2001
2002
2003
2004
2005

Figure 7.2 | Two different ways to initialize a string variable or a character array

When a character array is initialized with a list of character initializers, the terminating null character is to be explicitly placed but when it is initialized with a string literal constant, the terminating null character is automatically placed (if the size of the character array is one more than the length of the string literal constant).

7.4 Importance of Terminating Null Character

The terminating null character in strings is very important. Every string operation checks the presence of the null character to determine the end of a string. Consider the piece of code snippet in Program 7-6 that illustrates the importance of terminating a null character in strings.

Line | **Prog 7-6.c**

```
 1  //Importance of terminating null character
 2  #include<stdio.h>
 3  #include<string.h>
 4  main()
 5  {
 6      char str[5]={'H','e','l','l','o'};
 7      printf("The string is\t");
 8      puts(str);
 9      printf("Its length is %d",strlen(str));
10   }
```

Memory contents

str

H	e	l	l	o	G	G	G	G	G	\0

4000 4010

Output window

```
The string is     Hello0¥¤§¶
Its length is 10
```

Remarks:

- The printf function and the puts function print the characters starting from the memory location pointed to by its argument till the terminating null character is encountered
- In line number 6, the character array str is initialized with a list of characters and the null character is not explicitly placed at its end
- If a character array is not terminated with a null character, the output of the strlen function would be indeterminate and depends upon where the null character is present in the memory
- Thus, the puts function in line number 8 while printing str gives garbage (any arbitrary value) as it starts printing from the memory location pointed to by its argument (i.e. 4000) and keeps on printing till a terminating null character is encountered
- The number of garbage characters in the output depends upon where the first null character is encountered in the memory
- Executing the same code at different times or on different machines may give different outputs (i.e. Hello followed by different and/or different number of garbage characters)
- The strlen function determines the length of the string by counting the number of characters in the string starting from the memory location pointed to by its argument till the null character is encountered. The terminating null character is not counted while determining the length of a string

Line | **Prog 7-6.c** | **Output window**

- Thus, it is very important to explicitly place the null character at the end when a character array is initialized with the character initializers or when its content are manipulated
- The null character is automatically placed at the end of a character array when it is initialized with a string literal constant or when scanf and gets functions are used to read a string from the user

Program 7-6 | A program to illustrate the importance of the terminating null character in the strings

7.5 String Library Functions

The C string library provides a large number of functions that can be used for string manipulations. The commonly used C string library functions are given in Table 7.1.

Table 7.1 | C string library functions

S. No	Function name	Prototype	Role
1.	strlen	int strlen(const char* s);	Calculates the length of a string s
2.	strcpy	char* strcpy(char* dest, const char* src);	Copies the source string str to the destination string dest
3.	strcat	char* strcat(char *dest, const char*src);	Appends a copy of the string src to the end of the string dest
4.	strcmp	int strcmp(const char*s1, const char* s2);	Compares two strings
5.	strcmpi	int strcmpi(const char*s1, const char* s2);	Compares two strings without case sensitivity
6.	strrev	char* strrev(char* s);	Reverses the content of a string s
7.	strlwr	char* strlwr(char* s);	Converts the string to lowercase
8.	strupr	char* strupr(char* s);	Converts the string to uppercase
9.	strset	char* strset(char* s, int ch);	Set all characters in a string s to the character ch
10.	strchr	char* strchr(const char* s, int c);	Scans a string for the first occurrence of a given character
11.	strrchr	char* strrchr(const char* s, int c);	Finds the last occurrence of a character c in the string s
12.	strstr	char* strstr(const char* s1, const char* s2);	Finds the first occurrence of a substring (i.e. s2) in another string (i.e. s1)
13.	strncpy	char* strncpy(char* dest, const char* src, int n);	Copies at the most n characters of the string src to the string dest
14.	strncat	char* strncat(char* dest, const char* src, int n);	Appends at the most n characters of the string src to the string dest
15.	strncmp	int strncmp(const char* s1, const char* s2, int n);	Compares at the most n characters of two strings s1 and s2
16.	strncmpi	int strncmpi(const char* s1, const char* s2, int n);	Compares at the most n characters of two strings s1 and s2 without case sensitivity
17.	strnset	char* strnset(char* s, int ch, int n);	Sets the first n characters of the string s to the character ch

The following sub-sections illustrate the use of the above-mentioned string library functions along with the development of user-defined functions with the same functionality.

7.5.1 strlen Function

Role: The strlen function is used to find the length of a string.

Input: The input to the strlen function can be a string literal constant or a character array holding a string or a character pointer pointing to a string.

Output: The strlen function returns the length of the string. The terminating null character is not counted while determining the length of the string.

Usage: The code snippets in Program 7-7 illustrate the use of the strlen function and the development of the strlen functionality.

Prog 7-7a.c Using library function	Prog 7-7b.c Using user-defined function	Output window

//Finding length of a string
#include<stdio.h>
#include<string.h>
main()
{
char *ptr="Dear";
char name[50]="Reader";
printf(“The length of strings:\n”);
printf(“Hello is %d\n”,strlen(“Hello”));
printf(“Dear is %d\n”,strlen(ptr));
printf(“Reader is %d\n”,strlen(name));
}
	//Finding length of a string
	#include<stdio.h>
	int mystrlen(char* s);
	main()
	{
	char *ptr="Dear";
	char name[50]="Reader";
	printf(“The length of strings:\n”);
	printf(“Hello is %d\n”,mystrlen(“Hello”));
	printf(“Dear is %d\n”,mystrlen(ptr));
	printf(“Reader is %d\n”,mystrlen(name));
	}
	int mystrlen(char *s)
	{
	int i=0;
	while(*(s+i)!='\0')
	i++;
	return i;
	}
		The length of strings:
		Hello is 5
		Dear is 4
		Reader is 6
		Remarks:
		• The strlen function returns the number of characters that precede the terminating null character
		• If a terminating null character is not present at the end of a string, the strlen function gives an arbitrary result

Program 7-7 | A program to find the length of a string (a) using a library function and (b) using a user-defined function

7.5.2 strcpy Function

Role: The strcpy function copies the source string to the destination string.

Inputs: A source string and a destination string. The source string can be a string literal or a character array or a character pointer pointing to a string. The destination should be a character array or a character pointer to the memory location in which the source string is to be copied.

Output: The strcpy function copies the source string to the destination and returns a pointer to the destination string.

Usage: The code snippets in Program 7-8 illustrate the use of the strcpy function and the development of the strcpy functionality.

Line	Prog 7-8a.c Using library function	Prog 7-8b.c Using user-defined function	Output window
1	//Copying one string to another	//Copying one string to another	Source string is
2	#include<stdio.h>	#include<stdio.h>	Hello
3	#include<string.h>	char* mystrcpy(char* dest, const char* src);	Destination string is
4	main()	main()	Hello
5	{	{	**Remark:**
6	char src[50]="Hello";	char src[50]="Hello";	• If the number of characters in the source string is more than the number of characters that the destination can hold, a memory exception may arise
7	char dest[50];	char dest[50];	
8	puts(“Source string is”);	puts(“Source string is”);	
9	puts(src);	puts(src);	
10	strcpy(dest,src);	mystrcpy(dest,src);	
11	puts(“Destination string is”);	puts(“Destination string is”);	
12	puts(dest);	puts(dest);	
13	}	}	
14		char* mystrcpy(char* dest, const char* src)	
15		{	
16		int i=0;	
17		while(src[i]!='\0')	
18		{	
19		dest[i]=src[i];	
20		i++;	
21		}	
22		//Null character should be explicitly placed at	
23		//the end of the string.	
24		dest[i]='\0';	
25		return dest;	
26		}	

Program 7-8 | A program to copy a string (a) using a library function and (b) using a user-defined function

> The destination character array or the destination memory block to which the character pointer points should be big enough to hold the source string. If they are not big enough, a run time exception may occur. Refer Question number 12 and its answer for more details.

7.5.3 strcat Function

Role: The strcat function concatenates one string with another. It appends a source string to the destination string.

Inputs: The source string to be appended and the destination string to which the source string is to be appended. The first argument of the function strcat can be a character array or a character pointer but should not be a string literal constant.

Output: The strcat function appends a source string to the destination string and returns a pointer to the destination string.

Usage: The code snippets in Program 7-9 illustrate the use of the strcat function and the development of the strcat functionality.

Line	Prog 7-9a.c Using library function	Prog 7-9b.c Using user-defined function	Output window
1	//Concatenating a string with another	//Concatenating a string with another	The strings are::
2	#include<stdio.h>	#include<stdio.h>	Hello
3	#include<string.h>	char* mystrcat(char* dest, const char* src);	Readers!!
4	main()	main()	After concatenation::
5	{	{	HelloReaders!!
6	char dest[50]="Hello";	char dest[50]="Hello";	**Remarks:**
7	char src[50]="Readers!!";	char src[50]="Readers!!";	• The length of the destination string after concatenation = the length of the destination string before concatenation plus the length of the source string
8	puts("The strings are::");	puts("The strings are::");	
9	puts(dest);	puts(dest);	
10	puts(src);	puts(src);	
11	strcat(dest,src);	mystrcat(dest,src);	
12	puts("After concatenation::");	puts("After concatenation::");	
13	puts(dest);	puts(dest);	
14	}	}	• The destination should be big enough to hold the destination string plus the source string
15		char* mystrcat(char* dest, const char* src)	
16		{	
17		int i=0, j=0;	
18		while(dest[i]!='\0')	• If it is not big enough, the characters of the resulting string would be placed in unreserved memory and may lead to memory violation. Hence memory exception may occur
19		i++;	
20		while(src[j]!='\0')	
21		{	
22		dest[i]=src[j];	
23		i++;j++;	
24		}	
25		dest[i]='\0';	
26		return dest;	
27		}	

Program 7-9 | A program to concatenate a string with another (a) using a library function and (b) using a user-defined function

7.5.4 strcmp Function

Role: The strcmp function compares two strings.

Inputs: Two strings str1 and str2 that are to be compared in the form of string literal constants or character arrays or character pointers to the memory locations in which str1 and str2 are stored.

Output: The strcmp function performs the comparison of str1 and str2 character by character, starting with the first character in each string and continuing with the subsequent characters until the corresponding characters differ or until the end of the strings is reached. It returns the ASCII difference of the first dissimilar corresponding characters or zero if none of the corresponding characters in both the strings are different.

Usage: The code snippets in Program 7-10 illustrate the use of the strcmp function and the development of the strcmp functionality.

Prog 7-10a.c
Using library function

```c
//Comparing two strings
#include<stdio.h>
#include<string.h>
main()
{
    char str1[20],str2[20];
    int res;
    puts("Enter string 1:");
    gets(str1);
    puts("Enter string 2:");
    gets(str2);
    res=strcmp(str1,str2);
    if(res==0)
        puts("Strings are equal");
    else
        puts("Strings are not equal");
}
```

Prog 7-10b.c
Using user-defined function

```c
//Comparing two strings
#include<stdio.h>
int mystrcmp(const char* s1, const char* s2);
main()
{
    char str1[20],str2[20];
    int res;
    puts("Enter string 1:");
    gets(str1);
    puts("Enter string 2:");
    gets(str2);
    res=mystrcmp(str1,str2);
    if(res==0)
        puts("Strings are equal");
    else
        puts("Strings are not equal");
}
int mystrcmp(const char* s1, const char* s2)
{
int i=0;
while(s1[i]!='\0' || s2[i]!='\0')
{
    if(s1[i]!=s2[i])
        return(s1[i]-s2[i]);
    i++;
}
return 0;
}
```

Output window

```
Enter string 1:
Hello
Enter string 2:
Hi
Strings are not equal
```

Output window
(second execution)

```
Enter string 1:
Hello
Enter string 2:
Hello
Strings are equal
```

Output window
(third execution)

```
Enter string 1:
hello
Enter string 2:
HELLO
Strings are not equal
```

Remarks:

- **strcmp(str1,str2)** returns a value:
- **0** if str1 and str2 are equal, or
- **>0** if str1 is greater than str2, i.e. str1 comes after str2 in lexicographic order (i.e. dictionary order), or
- **<0** if str1 is lesser than str2 i.e. str1 comes before str2, in lexicographic order

Program 7-10 | A program to compare two strings (a) using a library function and (b) using a user-defined function

7.5.5 strcmpi Function

Role: The strcmpi function compares two strings without case sensitivity. The suffix character 'i' in strcmpi stands for ignore case.

Inputs: Two strings str1 and str2 that are to be compared, in the form of string literal constants or character arrays or character pointers to the memory locations in which str1 and str2 are stored.

Output: The strcmpi function performs a comparison of strings str1 and str2 without case sensitivity. It returns the ASCII difference of the first different corresponding characters or zero if none of the corresponding characters in both the strings are different.

Usage: The code snippets in Program 7-11 illustrate the use of the strcmpi function and the development of the strcmpi functionality.

Prog 7-11a.c
Using library function

```c
//Comparing two strings without
//case sensitivity
#include<stdio.h>
#include<string.h>
main()
{
    char str1[20],str2[20];
    int res;
    puts(“Enter string 1:”);
    gets(str1);
    puts(“Enter string 2:”);
    gets(str2);
    res=strcmpi(str1,str2);
    if(res==0)
        puts(“Strings are equal”);
    else
        puts(“Strings are not equal”);
}
```

Prog 7-11b.c
Using user-defined function

```c
//Comparing two strings without
//case sensitivity
#include<stdio.h>
int mystrcmpi(const char* s1, const char* s2);
main()
{
    char str1[20],str2[20];
    int res;
    puts(“Enter string 1:”);
    gets(str1);
    puts(“Enter string 2:”);
    gets(str2);
    res=mystrcmpi(str1,str2);
    if(res==0)
        puts(“Strings are equal”);
    else
        puts(“Strings are not equal”);
}
int mystrcmpi(const char* s1, const char* s2)
{
int i=0;
while(s1[i]!=‘\0’ || s2[i]!=‘\0’)
{
if((s1[i]==s2[i])||(s1[i]-s2[i])==32||(s1[i]-s2[i])==-32)
    i++;
else
    return(s1[i]-s2[i]);
}
return 0;
}
```

Output window

```
Enter string 1:
HELLO
Enter string 2:
hello
Strings are equal
```

Output window (second execution)

```
Enter string 1:
Hello
Enter string 2:
Hi
Strings are not equal
```

Output window (third execution)

```
Enter string 1:
hello
Enter string 2:
HELLO
Strings are equal
```

Remarks:

- The difference between the ASCII values of lowercase letters and their uppercase counterparts is 32
- For example, ‘a’ has an ASCII value of 97 while ‘A’ has an ASCII value of 65

Program 7-11 | A program to compare two strings without case sensitivity (a) using a library function and (b) using a user-defined function

7.5.6 strrev Function

Role: The strrev function reverses all the characters of a string except the terminating null character.

Input: A string in the form of a character array or a character pointer or a string literal constant.

Output: The strrev function reverses the string and returns a pointer to the reversed string.

Usage: The code snippets in Program 7-12 illustrate the use of the strrev function and the development of the strrev functionality.

Line	Prog 7-12a.c Using library function	Prog 7-12b.c Using user-defined function	Output window
1	//Reversing the contents of a string	//Reversing the contents of a string	Enter a string:
2	#include<stdio.h>	#include<stdio.h>	Hello
3	#include<string.h>	char* mystrrev(char* s);	After reversal, the string is:
4	main()	main()	olleH
5	{	{	**Output window**
6	char str[20];	char str[20];	**(second execution)**
7	puts(“Enter a string:”);	puts(“Enter a string:”);	Enter a string:
8	gets(str);	gets(str);	Hello Readers
9	strrev(str);	mystrrev(str);	After reversal, the string is:
10	puts(“After reversal, the string is:”);	puts(“After reversal, the string is:”);	sredaeR olleH
11	puts(str);	puts(str);	**Remarks:**
12	}	}	• The strrev function can also be applied on the string literals, i.e. strrev(“Hello”)=”olleH”
13		char* mystrrev(char* s)	
14		{	
15		int i=0, j=0;	• strrev(strrev(“String”))=”String”
16		char temp;	• Reversal of reverse of a string is the string itself
17		while(s[i]!='\0')	
18		i++;	
19		i--;	
20		while(i>j)	
21		{	
22		temp=s[i];	
23		s[i]=s[j];	
24		s[j]=temp;	
25		j++;i--;	
26		}	
27		return s;	
28		}	

Program 7-12 | A program that reverses contents of a string (a) using a library function and (b) using a user-defined function

7.5.7 strlwr Function

Role: The strlwr function converts all the letters in a string to lowercase.

Input: A string in the form of a character array or a character pointer or a string literal constant.

Output: It returns a pointer to the converted string.
Usage: The code snippets in Program 7-13 illustrate the use of the strlwr function and the development of the strlwr functionality.

Prog 7-13a.c
Using library function

```
//Converting all the characters of a
//string to lower case
#include<stdio.h>
#include<string.h>
main()
{
    char str[20];
    puts("Enter a string:");
    gets(str);
    strlwr(str);
    puts("Lowercase string is:");
    puts(str);
}
```

Prog 7-13b.c
Using user-defined function

```
//Converting all the characters of a
//string to lower case
#include<stdio.h>
char* mystrlwr(char* s);
main()
{
    char str[20];
    puts("Enter a string:");
    gets(str);
    mystrlwr(str);
    puts("Lowercase string is:");
    puts(str);
}
char* mystrlwr(char* s)
{
int i=0;
while(s[i]!='\0')
{
    if(s[i]>=65 && s[i]<=90)
        s[i]=s[i]+32;
    i++;
}
return s;
}
```

Output window

```
Enter a string:
HELLO
Lowercase string is:
hello
```

Output window (second execution)

```
Enter a string:
HELLO READERS!!
Lowercase string is:
hello readers!!
```

Remarks:

- Digits, special characters and white-space characters within the string remain unchanged

Program 7-13 | A program that converts all the characters of a string to lowercase (a) using a library function and (b) using a user-defined function

7.5.8 strupr Function

Role: The strupr function converts all the letters in a string to uppercase.
Input: A string in the form of a character array or a character pointer or a string literal constant.
Output: It returns a pointer to the converted string.
Usage: The code snippets in Program 7-14 illustrate the use of the strupr function and the development of the strupr functionality.

Line	Prog 7-14a.c Using library function	Prog 7-14b.c Using user-defined function	Output window
1	//Converting all the characters of a	//Converting all the characters of a	Enter a string:
2	//string to uppercase	//string to uppercase	hello
3	#include<stdio.h>	#include<stdio.h>	Uppercase string is:
4	#include<string.h>	char* mystrupr(char* s);	HELLO
5	main()	main()	**Output window**
6	{	{	**(second execution)**
7	char str[20];	char str[20];	Enter a string:
8	puts(“Enter a string:”);	puts(“Enter a string:”);	hello readers!!
9	gets(str);	gets(str);	Uppercase string is:
10	strupr(str);	mystrupr(str);	HELLO READERS!!
11	puts(“Uppercase string is:”);	puts(“Uppercase string is:”);	**Remark:**
12	puts(str);	puts(str);	• Digits, special characters and white-space characters within a string remain unchanged
13	}	}	
14		char* mystrupr(char* s)	
15		{	
16		int i=0;	
17		while(s[i]!=’\0’)	
18		{	
19		if(s[i]>=97 && s[i]<=122)	
20		s[i]=s[i]-32;	
21		i++;	
22		}	
23		return s;	
24		}	

Program 7-14 | A program that converts all the characters of a string to uppercase (a) using a library function and (b) using a user-defined function

7.5.9 strset Function

Role: The strset function sets all characters in a string to a specific character.

Inputs: A string and a character. The string can be in the form of a character array or a character pointer or a string literal constant.

Output: The strset function sets all the characters in the string to the given character and returns a pointer to the string.

Usage: The code snippets in Program 7-15 illustrate the use of the strset function and the development of the strset functionality.

Line	Prog 7-15a.c Using library function	Prog 7-15b.c Using user-defined function	Output window
1	//Setting all the characters of a string to	//Setting all the characters of a string to	Before using strset(), string is:
2	//a specific character	//a specific character	123456789
3	#include<stdio.h>	#include<stdio.h>	After using strset(), string is:
4	#include<string.h>	char* mystrset(char* s, int ch);	ccccccccc
5	main()	main()	

(Contd...)

6	{	{	**Remark:** • All the characters (letters, digits, special characters and white-space characters) within a string are set to a specific character
7	char str[10]="123456789";	char str[10]="123456789";	
8	char ch='c';	char ch='c';	
9	puts(“Before using strset(), string is:”);	puts(“Before using strset(), string is:”);	
10	puts(str);	puts(str);	
11	strset(str,ch);	mystrset(str,ch);	
12	puts(“After using strset(), string is:”);	puts(“After using strset(), string is:”);	
13	puts(str);	puts(str);	
14	}	}	
15		char* mystrset(char* s, int ch)	
16		{	
17		int i=0;	
18		while(s[i]!='\0')	
19		{	
21		s[i]=ch;	
22		i++;	
23		}	
24		return s;	
25		}	

Program 7-15 | A program that sets all the characters of a string to a specific character (a) using a library function and (b) using a user-defined function

7.5.10 strchr Function

Role: The strchr function scans a string for the first occurrence of a given character.

Inputs: A string and a character to be found in the string. The string can be in the form of a character array or a character pointer or a string literal constant.

Output: The strchr function scans the input string in the forward direction, looking for the specific character. If the character is found, it returns a pointer to the first occurrence of the character in the given string. If the character is not found it returns NULL.

Usage: The code snippets in Program 7-16 illustrate the use of the strchr function and the development of the strchr functionality.

Line	Prog 7-16a.c Using library function	Prog 7-16b.c Using user-defined function	Output window
1	//Scans a string for the first occurrence	//Scans a string for the first occurrence	Enter a string:
2	//of a given character	//of a given character	Hello
3	#include<stdio.h>	#include<stdio.h>	Enter a character to be found:
4	#include<string.h>	char* mystrchr(const char* s, int c);	e
5	main()	main()	Located at the index 1
6	{	{	
7	char str[20], ch;	char str[20], ch;	**Output window**
8	char* ptr;	char* ptr;	**(second execution)**
9	puts(“Enter a string:”);	puts(“Enter a string:”);	Enter a string:
10	gets(str);	gets(str);	Hello

(*Contd...*)

Line	Prog 7-16a.c Using library function	Prog 7-16b.c Using user-defined function	Output window
11	puts("Enter a character to be found:");	puts("Enter a character to be found:");	Enter a character to be found:
12	scanf("%c",&ch);	scanf("%c",&ch);	y
13	ptr=strchr(str,ch);	ptr=mystrchr(str,ch);	Character not found
14	if(ptr==NULL)	if(ptr==NULL)	**Remark:**
15	puts("Character not found");	puts("Character not found");	• The terminating null character is also considered to be a part of the string
16	else	else	
17	printf("Located at the index %d",ptr-str);	printf("Located at the index %d",ptr-str);	
18	}	}	
19		char* mystrchr(const char* s, int c)	
20		{	
21		int i=0;	
22		while(s[i]!='\0')	
23		{	
24		if(s[i]==c)	
25		return((char*)s+i);	
26		i++;	
27		}	
28		return NULL;	
29		}	

Program 7-16 | A program that scans a string for the first occurrence of a given character (a) using a library function and (b) using a user-defined function

7.5.11 strrchr Function

Role: The strrchr function locates the last occurrence of a character in a given string.

Inputs: A string and a character to be found in the string. The string can be in the form of a character array or a character pointer or a string literal constant.

Output: The strrchr function scans the input string in the reverse direction, looking for a specific character. If the character is found, it returns a pointer to the first occurrence of the character in the given string. If the character is not found, it returns NULL.

Usage: The code snippets in Program 7-17 illustrate the use of the strrchr function and the development of the strrchr functionality.

Line	Prog 7-17a.c Using library function	Prog 7-17b.c Using user-defined function	Output window
1	//Scans a string in the reverse direction	//Scans a string in the reverse direction	Enter a string:
2	//for the first occurrence of a given	//for the first occurrence of a given	Hello
3	//character	//character	Enter a character to be found:
4	#include<stdio.h>	#include<stdio.h>	o
5	#include<string.h>	char* mystrrchr(const char* s, int c);	Located at the index 4
6	main()	main()	
7	{	{	
8	char str[20], ch;	char str[20], ch;	
9	char* ptr;	char* ptr;	
10	puts("Enter a string:");	puts("Enter a string:");	

(Contd...)

11	gets(str);	gets(str);	**Output window**
12	puts("Enter a character to be found:");	puts("Enter a character to be found:");	**(second execution)**
13	scanf("%c",&ch);	scanf("%c",&ch);	Enter a string:
14	ptr=strrchr(str,ch);	ptr=mystrrchr(str,ch);	Hello
15	if(ptr==NULL)	if(ptr==NULL)	Enter a character to be found:
16	puts("Character not found");	puts("Character not found");	y
17	else	else	Character not found
18	printf("Located at the index %d",ptr-str);	printf("Located at the index %d",ptr-str);	**Output window**
19	}	}	**(third execution)**
20		char* mystrrchr(const char* s, int c)	Enter a string:
21		{	Hello
22		int i=0;	Enter a character to be found:
23		while(s[i]!='\0')	l
24		i++;	Located at the index 3
25		i--;	**Remark:**
26		while(i>=0)	• The terminating null
27		{	character is also con-
28		if(s[i]==c)	sidered to be a part
29		return((char*)s+i);	of the string
30		i--;	
31		}	
32		return NULL;	
33		}	

Program 7-17 | A program that scans a string in the reverse direction for the first occurrence of a given character (a) using a library function and (b) using a user-defined function

7.5.12 strstr Function

Role: The strstr function finds the first occurrence of a string in another string.

Inputs: Two strings str1 and str2. The strings can be in the form of a character array or a character pointer or a string literal constant.

Output: The strstr function finds the first occurrence of the string (i.e. str2) in the string (i.e. str1). If the string str2 is found, it returns a pointer to the position from where the string starts. If the string str2 is not found in the string str1, it returns NULL.

Usage: The code snippets in Program 7-18 illustrate the use of the strstr function and the development of the strstr functionality.

Line	Prog 7-18a.c Using library function	Prog 7-18b.c Using user-defined function	Output window
1	//Finding string within a string	//Finding string within a string	Enter a string:
2	#include<stdio.h>	#include<stdio.h>	Hello Readers!!
3	#include<string.h>	char* mystrstr(const char* s1, const char* s2);	Enter the string to be found:
4	main()	main()	Read
5	{	{	Found at the index 6
6	char* ptr;	char* ptr;	Found in Readers!!
7	char str1[20];	char str1[20];	
8	char str2[20]	char str2[20];	
9	puts("Enter a string:");	puts("Enter a string:");	

(*Contd...*)

```
    gets(str1);
    puts("Enter the string to be found:");
    gets(str2);
    ptr=strstr(str1,str2);
    if(ptr==NULL)
        puts("String not found");
    else
    {
    printf("Found at the index %d\n",ptr-str1);
    printf("Found in %s",ptr);
    }
}
```

```
    gets(str1);
    puts("Enter the string to be found:");
    gets(str2);
    ptr=mystrstr(str1,str2);
    if(ptr==NULL)
        puts("String not found");
    else
    {
    printf("Found at the index %d\n",ptr-str1);
    printf("Found in %s",ptr);
    }
}
char* mystrstr(const char* s1, const char* s2)
{
int i=0,j=0,k;
while(s1[i]!='\0')
{
    k=i;
    while(s2[j]!='\0')
    {
        if(s1[k]!=s2[j])
            break;
        k++;j++;
    }
    if(s2[j]=='\0')
        return (char*)s1+i;
    else
        i++;j=0;
}
return NULL;
}
```

Output window (second execution)

```
Enter a string:
Hello Readers!!
Enter the string to be found:
Student
String not found
```

Program 7-18 | A program that finds a string within a string (a) using a library function and (b) using a user-defined function

7.5.13 strncpy Function

Role: The strncpy function copies at the most n characters of a source string to the destination string.

Inputs: A character array or a character pointer to the memory location where the source string is to be copied (i.e. destination), the source string that is to be copied and an integer value that specifies the number of characters of the source string that is to be copied.

Output: The strncpy function copies at the most n characters of the source string to the destination and returns a pointer to the destination string.

Usage: The code snippets in Program 7-19 illustrate the use of the strncpy function and the development of the strncpy functionality.

Line	Prog 7-19a.c Using library function	Prog 7-19b.c Using user-defined function	Output window
1	//Copying at the most n characters of	//Copying at the most n characters of	Enter source string: Hello Readers!! Enter the value of n: 5 Source string is: Hello Readers!! Destination string is: Hello **Remarks:** • If the source string contains more than n characters, n characters are copied and the null character is not placed at the end. The terminating null character is to be explicitly placed as done in line number 18 • If the source string is shorter than n characters, the terminating null character is copied into the destination string **Try:** • Comment line number 18 • Execute the code with the same input and observe the garbage characters in the output in some of the executions
2	//a source string to the destination	//a source string to the destination	
3	//string	//string	
4	#include<stdio.h>	#include<stdio.h>	
5	#include<string.h>	char* mystrncpy(char* dest, const char* src, int n);	
6	main()	main()	
7	{	{	
8	char src[50];	char src[50];	
9	char dest[50];	char dest[50];	
10	int n;	int n;	
11	puts(“Enter source string:”);	puts(“Enter source string:”);	
12	gets(src);	gets(src);	
13	puts(“Enter the value of n:”);	puts(“Enter the value of n:”);	
14	scanf(“%d”,&n);	scanf(“%d”,&n);	
15	puts(“Source string is:”);	puts(“Source string is:”);	
16	puts(src);	puts(src);	
17	strncpy(dest,src,n);	mystrncpy(dest,src,n);	
18	dest[n]=’\0’;	dest[n]=’\0’;	
19	puts(“Destination string is:”);	puts(“Destination string is:”);	
20	puts(dest);	puts(dest);	
21	}	}	
22		char* mystrncpy(char* dest, const char* src, int n)	
23		{	
24		int i=0;	
25		while(i<n)	
26		{	
27		if(src[i]=='\0')	
28		{	
29		dest[i]='\0';	
30		break;	
31		}	
32		else	
33		{	
34		dest[i]=src[i];	
35		i++;	
36		}	
37		}	
38		return dest;	
39		}	

Program 7-19 | A program that copies at most n characters of a source string to a destination string (a) using a library function and (b) using a user-defined function

7.5.14 strncat Function

Role: The strncat function concatenates a portion of one string with another. It appends at the most n characters of a source string to a destination string.

Inputs: The source string to be appended, the destination string to which the source string is to be appended and the number of characters to be appended. The destination string should be a character array or a character pointer but should not be a string literal constant.

Output: The strncat function appends at the most n characters of the source string to the destination string and returns a pointer to the destination string.

Usage: The code snippets in Program 7-20 illustrate the use of the strncat function and the development of the strncat functionality.

Line	Prog 7-20a.c Using library function	Prog 7-20b.c Using user-defined function	Output window
1	//String Concatenation	//String Concatenation	Enter strings:
2	#include<stdio.h>	#include<stdio.h>	Hello
3	#include<string.h>	char* mystrncat(char* dest, const char* src, int n);	Readers!!
4	main()	main()	Enter the value of n:
5	{	{	7
6	char dest[50], src[50];	char dest[50], src[50];	The strings are:
7	int n;	int n;	Hello
8	puts(“Enter strings:”);	puts(“Enter strings:”);	Readers!!
9	gets(dest);	gets(dest);	After concatenation:
10	gets(src);	gets(src);	HelloReaders
11	puts(“Enter the value of n:”);	puts(“Enter the value of n:”);	**Remarks:**
12	scanf(“%d”,&n);	scanf(“%d”,&n);	• Unlike strncpy, a terminating null character is always appended to the result
13	puts(“The strings are:”);	puts(“The strings are:”);	
14	puts(dest);	puts(dest);	
15	puts(src);	puts(src);	
16	strncat(dest,src,n);	mystrncat(dest,src,n);	• The maximum number of characters in the destination string after the execution of strncat would be the number of characters in the dest (before execution of strncat)+n+1
17	puts(“After concatenation:”);	puts(“After concatenation:”);	
18	puts(dest);	puts(dest);	
19	}	}	
20		char* mystrncat(char* dest, const char* src,int n)	
21		{	
22		int i=0, j=0,k=1;	
23		while(dest[i]!='\0')	
24		i++;	
25		while(src[j]!='\0' && k<=n)	
26		{	
27		dest[i]=src[j];	
28		i++;j++;k++;	
29		}	
30		dest[i]='\0';	
31		return dest;	
32		}	

Program 7-20 | A program that concatenates at the most n characters of a source string with the destination string (a) using a library function and (b) using a user-defined function

7.5.15 strncmp Function

Role: The strncmp function compares a portion of two strings.

Inputs: Two strings str1 and str2 and the value of n, i.e. the number of characters to be compared.

Output: The strncmp function performs the comparison of str1 and str2, starting with the first character in each string and continuing with the subsequent characters until the corresponding characters differ or until the end of strings is reached or n characters have been compared. It returns the ASCII difference of the first dissimilar corresponding characters or zero if none of the corresponding n characters in both the strings are different.

Usage: The code snippets in Program 7-21 illustrate the use of the strncmp function and the development of the strncmp functionality.

Prog 7-21a.c
Using library function

```
//Comparing a portion of two strings
#include<stdio.h>
#include<string.h>
main()
{
    char str1[20],str2[20];
    int res, n;
    puts(“Enter string 1:”);
    gets(str1);
    puts(“Enter string 2:”);
    gets(str2);
    puts(“Enter the value of n:”);
    scanf(“%d”,&n);
    res=strncmp(str1,str2,n);
    if(res==0)
      puts(“String portions are equal”);
    else
     puts(“String portions are not equal”);
}
```

Prog 7-21b.c
Using user-defined function

```
//Comparing a portion of two strings
#include<stdio.h>
int mystrncmp(const char* s1, const char* s2, int n);
main()
{
    char str1[20],str2[20];
    int res,n;
    puts(“Enter string 1:”);
    gets(str1);
    puts(“Enter string 2:”);
    gets(str2);
    puts(“Enter the value of n:”);
    scanf(“%d”,&n);
    res=mystrncmp(str1,str2,n);
    if(res==0)
      puts(“String portions are equal”);
    else
      puts(“String portions are not equal”);
}
int mystrncmp(const char* s1, const char* s2,int n)
{
int i=0;
while((s1[i]!='\0' || s2[i]!='\0') && i<n)
{
    if(s1[i]!=s2[i])
        return(s1[i]-s2[i]);
    i++;
}
return 0;
}
```

Output window

```
Enter string 1:
Hello
Enter string 2:
Hi
Enter the value of n:
1
String portions are equal
```

Output window (second execution)

```
Enter string 1:
Hello
Enter string 2:
Hello
Enter the value of n:
4
String portions are equal
```

Output window (third execution)

```
Enter string 1:
hello
Enter string 2:
HELLO
Enter the value of n:
3
String portions are not equal
```

Program 7-21 | A program that compares a portion of two strings (a) using a library function and (b) using a user-defined function

7.5.16 strncmpi Function

Role: The strncmpi function compares a portion of two strings without case sensitivity.

Inputs: Two strings str1 and str2 and the value of n, i.e. the number of characters to be compared.

Output: The strncmpi function performs the comparison of str1 and str2 without case sensitivity, starting with the first character in each string and continuing with the subsequent characters until the corresponding characters differ or until the end of strings is reached or n characters have been compared. It returns the ASCII difference of the first different corresponding characters or zero if none of the corresponding n characters in both the strings are different.

Usage: The code snippets in Program 7-22 illustrate the use of the strncmpi function and the development of the strncmpi functionality.

Prog 7-22a.c
Using library function

```
//Comparing a portion of strings
//without case sensitivity
#include<stdio.h>
#include<string.h>
main()
{
    char str1[20],str2[20];
    int res, n;
    puts(“Enter string 1:”);
    gets(str1);
    puts(“Enter string 2:”);
    gets(str2);
    puts(“Enter the value of n:”);
    scanf(“%d”,&n);
    res=strncmpi(str1,str2,n);
if(res==0)
    puts(“String portions are equal”);
else
    puts(“String portions are not equal”);
}
```

Prog 7-22b.c
Using user-defined function

```
//Comparing a portion of strings without
//case sensitivity
#include<stdio.h>
int mystrncmpi(const char* s1, const char* s2, int n);
main()
{
    char str1[20],str2[20];
    int res,n;
    puts(“Enter string 1:”);
    gets(str1);
    puts(“Enter string 2:”);
    gets(str2);
    puts(“Enter the value of n:”);
    scanf(“%d”,&n);
    res=mystrncmpi(str1,str2,n);
    if(res==0)
      puts(“String portions are equal”);
    else
      puts(“String portions are not equal”);
}
int mystrncmpi(const char* s1, const char* s2,int n)
{
int i=0;
while((s1[i]!=‘\0’ || s2[i]!=‘\0’) && i<n)
{
if((s1[i]==s2[i])|| (s1[i]-s2[i])==32|| (s1[i]-s2[i])==-32)
    i++;
else
    return(s1[i]-s2[i]);
}
return 0;
}
```

Output window

```
Enter string 1:
Hello
Enter string 2:
Hi
Enter the value of n:
2
String portions are not equal
```

Output window (second execution)

```
Enter string 1:
Hello
Enter string 2:
Hello
Enter the value of n:
5
String portions are equal
```

Output window (third execution)

```
Enter string 1:
hello
Enter string 2:
HELLO
Enter the value of n:
4
String portions are equal
```

Program 7-22 | A program that compares a portion of two strings without case sensitivity (a) using a library function and (b) using a user-defined function

7.5.17 strnset Function

Role: The strnset function sets the first n characters in a string to a specific character.

Inputs: A string, a character and an integer value n.

Output: The strnset function sets the first n characters in a string to the given character and returns a pointer to the string.

Usage: The code snippets in Program 7-23 illustrate the use of the strnset function and the development of the strnset functionality

Prog 7-23a.c
Using library function

```
//Setting the first n characters of a string
//to a specific character
#include<stdio.h>
#include<string.h>
main()
{
    char str[20], ch;
    int n;
    puts(“Enter the string:”);
    gets(str);
    puts(“Enter the character:”);
    scanf(“%c”,&ch);
    puts(“Enter the value of n:”);
    scanf(“%d”,&n);
    puts(“Before using strnset(), string is:”);
    puts(str);
    strnset(str,ch,n);
    puts(“After using strnset(), string is:”);
    puts(str);
}
```

Prog 7-23b.c
Using user-defined function

```
//Setting the first n characters of a
//string to a specific character
#include<stdio.h>
char* mystrnset(char* s, int ch, int n);
main()
{
    char str[20], ch;
    int n;
    puts(“Enter the string:”);
    gets(str);
    puts(“Enter the character:”);
    scanf(“%c”,&ch);
    puts(“Enter the value of n:”);
    scanf(“%d”,&n);
    puts(“Before using strnset(), string is:”);
    puts(str);
    mystrnset(str,ch,n);
    puts(“After using strnset(), string is:”);
    puts(str);
}
char* mystrnset(char* s, int ch, int n)
{
int i=0;
while(s[i]!=‘\0’ && i<n)
{
    s[i]=ch;
    i++;
}
return s;
}
```

Output window

```
Enter the string:
Hello Readers!!
Enter the character:
X
Enter the value of n:
6
Before using strnset(), string is:
Hello Readers!!
After using strnset(), string is:
XXXXXXReaders!!
```

Remark:

- If the length of the string is less than the value of n then the strnset function sets all the characters of the string to the specific character

Program 7-23 | A program that sets the first n characters of a string to a specific character (a) using a library function and (b) using a user-defined function

7.6 List of Strings

In the previous sections, we have seen how to store the strings in character arrays and the functions that can be used to manipulate them. However, real-time applications often require storage and manipulation of a number of strings (i.e. **list of strings**) and not only a single string. A list of strings can be stored in two ways:

1. Using an array of strings
2. Using an array of character pointers

7.6.1 Array of strings

If an application requires the storage of multiple strings, an array of strings can be used to store them. Since a string itself is stored in a one-dimensional character array, the list of strings can be stored by creating an array of one-dimensional character arrays, i.e. two-dimensional character array. Figure 7.3 depicts an array of strings.

A 2-D char array →	R	a	m	a	n	\0		←1st string
(Array of strings)	S	a	m	\0				←2nd string
	V	i	s	h	a	l	\0	←3rd string
	N	e	h	a	\0			←4th string

Figure 7.3 | Array of strings

7.6.1.1 Declaration of Array of strings

The general form of an **array of strings declaration** is:

```
<sclass_specifier><type_qualifier><type_modifier>char identifier[<row_specifier>][column_specifier]<=initialization_list>;
```

The important points about an array of strings declaration are as follows:

1. Array of strings declaration consists of char type specifier, an identifier name, row size specifier and column size specifier. The following declarations are valid:

   ```
   char array1[2][30];    //←array1 can store 2 strings of maximum 30 characters each
   char array2[5][5];     //←array2 can store 5 strings of maximum 5 characters each
   ```

2. All the syntactic rules discussed in Chapter 6 for declaring two-dimensional arrays are applicable for declaring arrays of strings as well.

3. **Initialization of array of strings:** Array of strings can be initialized in two ways:

   a. **Using string literal constants:** Using string literal constants, an array of strings can be initialized as:

   ```
           char str[][20]={
                       "Raman",
                       "Sam",
                       "Vishal",
                       "Neha"
                       };
   ```

   b. **Using a list of character initializers:** Using a list of character initializers, an array of strings can be initialized as:

```
char str[][20]={
        {'R','a','m','a',n','\0'},
        {'S','a','m','\0'},
        {'V','i','s','h','a','l','\0'},
        {'N','e','h','a','\0'}
    };
```

7.6.1.2 Reading List of Strings from the Terminal

A list of strings can be read from the terminal by iteratively calling the gets or scanf function. Program 7-24 reads a list of strings from the terminal and stores them in an array of strings.

Prog 7-24.c

```
//Reading a list of strings from the terminal
#include<stdio.h>
main()
{
    int i=0,j=0, marks[10], max;
    char students[10][20], ch;
    printf("Enter names of students and their marks:\n");
    while(1)
    {
        scanf("%s %d",students[i], &marks[i]);
        printf("Do you want to enter more(Y/N)\t");
        flushall();
        scanf("%c",&ch);
        if(ch=='Y'||ch=='y')
            i=i+1;
        else
            break;
        if(i==10)
        {
            printf("Cannot hold more names\n");
            break;
        }
    }
    max=0;
    for(j=0;j<i;j++)
        if(marks[j]>marks[max])
            max=j;
    printf("\n%s got maximum marks",students[max]);
}
```

Output window

```
Enter names of students and their marks:
Praveen 89
Do you want to enter more(Y/N)    Y
Ashok 80
Do you want to enter more(Y/N)    Y
Manish 90
Do you want to enter more(Y/N)    Y
Ameet  85
Do you want to enter more(Y/N)    N

Manish got maximum marks
```

Remarks:

- List of strings can be read by iteratively using the gets or scanf function
- The role of the flushall function is to flush (i.e. clear) the contents of all the streams
- Refer Question number 15 for a description on streams and the flushall function

Program 7-24 | A program that demonstrates a method to read a list of strings

7.6.2 Array of Character Pointers

An array of strings can also be stored by using an array of character pointers. The starting addresses of strings are stored in an array of character pointers as shown in Figure 7.4.

char* languages[]={“Basic”, “Java”, “Fortran”, “C”, “C++”};

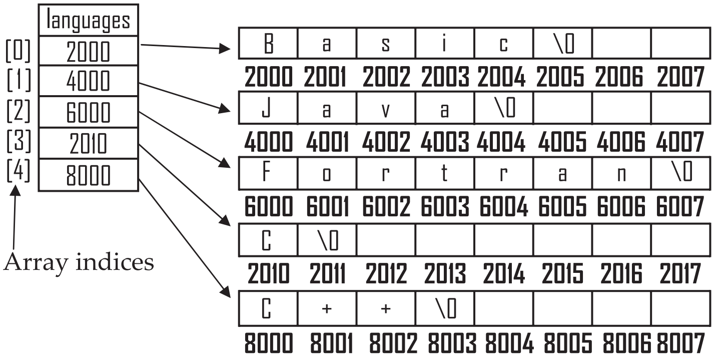


Figure 7.4 | Storing a list of strings using an array of character pointers

7.6.2.1 Use of Array of Character Pointers

Program 7-25 demonstrates the use of an array of character pointers to store a list of strings.

Prog 7-25.c

```
 1  //Use of array of character pointers
 2  #include<stdio.h>
 3  main()
 4  {
 5  int i,a[4];
 6  char* states[]={“Punjab”, “Bihar”, “Rajasthan”, “Gujarat”} ;
 7  char* capitals[]={“Gandhinagar”, “Chandigarh”, “Jaipur”, “Patna”};
 8  printf(“States\t\t\tCapitals\n”);
 9  printf(“------------------------------------------\n”);
10  for(i=0;i<4;i++)
11  printf(“%d. %-10s\t\t%d. %s\n”,i+1,states[i],i+1,capitals[i]);
12  printf(“\nMatch states in Col. 1 with capitals in Col. 2\n”);
13  printf(“(Enter only Sr. Nos.)\n”);
14  printf(“------------------------------------------\n”);
15  for(i=0;i<4;i++)
16  {
17   printf(“Capital of state %d is at\t”,i+1);
18   scanf(“%d”,&a[i]);
19  }
20  printf(“------------------------------------------\n”);
21  for(i=0;i<4;i++)
22  printf(“%-11s is capital of %s\n”,capitals[a[i]-1],states[i]);
23  }
```

Output window

```
States                  Capitals
------------------------------------------
1. Punjab               1. Gandhinagar
2. Bihar                2. Chandigarh
3. Rajasthan            3. Jaipur
4. Gujarat              4. Patna

Match states in Col. 1 with capitals in Col. 2
(Enter only Sr. Nos.)
------------------------------------------
Capital of state 1 is at     2
Capital of state 2 is at     4
Capital of state 3 is at     3
Capital of state 4 is at     1
------------------------------------------
Chandigarh    is capital of Punjab
Patna         is capital of Bihar
Jaipur        is capital of Rajasthan
Gandhinagar   is capital of Gujarat
```

Remark:

- Lists of strings are stored using arrays of character pointers in line number 6 and 7

Program 7-25 | A program that illustrates the use of an array of character pointers

7.7 Command Line Arguments

In Chapter 8, we will see that inputs are given to the functions by means of arguments. main is also a function. Therefore, can we give inputs to the function main also by supplying arguments? The answer to this question is YES! Inputs to the function main are given by making use of special arguments known as **command line arguments**.

If you have used DOS, you must have used copy command. The copy command looks like:

copy source_file.txt dest_file.txt

To the copy program, the name of the source file (i.e. source_file.txt) and the name of the destination file (i.e. dest_file.txt) are given as inputs. These inputs are given at the **command line** or **command prompt** and are known **command line arguments**.

C provides a fairly simple mechanism for retrieving command line arguments entered by the user at the command line. To retrieve the command line arguments, the function main should be defined as:

```
main(int argc, char* argv[])        //←Header of the function main
{
//.......Statements.......
//.......Body.....
//.......Statements........
}
```

In the header of the function main, two parameters are given, namely:

1. argc: The parameter argc stands for **argument count** and is of integer type.
2. argv: The parameter argv stands for **argument vector** and is an array of character pointers.

> The names of parameters are dummy and can be anything like abc, xyz, etc. but generally the names argc and argv are used.

Suppose that on the command prompt, the user has entered:

```
prog opt1 opt2 sfile dfile
```

The important points about the given input are as follows:

1. The command line arguments are separated by blank spaces. In the given input, there are five arguments. The name of the program file (actually executable file) will also be counted while determining the argument count.
2. The parameter argc will receive a value equal to the number of arguments specified on the command prompt. In the given example, argc will have the value 5.
3. The first argument is the name of the program file (actually executable file). The file prog.exe should be present in the current working directory.
4. The contents of the parameter argv will be:

```
argv[0]="prog"
argv[1]="opt1"
argv[2]="opt2"
argv[3]="sfile"
argv[4]="dfile"
```

The contents of the array argv are shown in Figure 7.5.

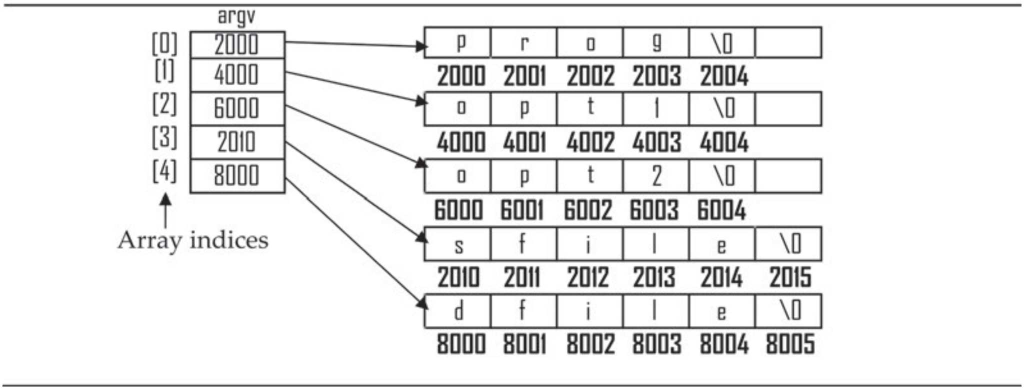


Figure 7.5 | Contents of the array argv

Program 7-26 illustrates the use of command line arguments.

Prog 7-26 mycopy.c

```
//Command line arguments
#include<stdio.h>
main(int argc, char* argv[])
{
    int i=0;
    printf("The number of arguments are %d\n", argc);
    printf("Arguments are:\n");
    for(i=0;i<argc;i++)
        printf("%s\n",argv[i]);
}
```

Command prompt

```
c:\tc\bin>mycopy source.txt dest.txt
The number of arguments are 3
Arguments are:
c:\tc\bin>mycopy.exe
source.txt
dest.txt
```

Program 7-26 | A program that illustrates the use of command line arguments

To execute Program 7-26, follow these steps:

1. Save the program with .c extension. Suppose the name given to the program file is mycopy.c.
2. Compile the program and check for compilation errors.
3. If there are no errors, build an executable file by invoking Make or Build all option in the Compile Menu of Turbo C 3.0 or by invoking Make all or Build all option in the Project menu, if using Turbo C 4.5. By default, the name of the executable file would be the same as the name of the program file. However, if a different name is given to the executable file, note it.
4. Observe the name and path of the directory in which the executable file is created.
5. Invoke the command prompt. Change the directory and make the current working directory the same as the directory in which the executable file was created.

6. Execute the program by writing the name of the executable file followed by blank separated arguments, e.g. mycopy source.txt dest.txt
7. If using Turbo C 3.0, the other way to execute Program 7-26 is by providing arguments from the IDE. Invoke Arguments... option is available in the Run Menu. Provide the arguments and execute the program. Note that if using this option, all the arguments except the name of the program file are to be provided. The name of the program is used by default and should not be specified.

Practically, the command line arguments are used in the applications that involve file handling.

7.8 Summary

1. A string literal is a sequence of zero or more characters enclosed within double quotes.
2. A string literal is automatically terminated by a null character.
3. A null character has an ASCII value of 0 and is written as '\0'.
4. Due to this additional null character, a string constant takes 1 byte more than the number of characters present in the string.
5. String literals are stored in character arrays.
6. In C language, string type is not separately available and character pointers are used to represent a string.
7. The type of string literal constants is const char*. The constant pointer refers to the address of the first character of the string.
8. Strings can be read from the keyboard by using scanf and gets functions.
9. The scanf function is used for reading single-word strings while the gets function can be used for reading multi-word strings.
10. Strings are printed on the screen by using printf and puts functions.
11. The printf function does not place a new line character after printing the string but the puts function places a new line character after printing the string.
12. The C string library provides a rich set of functionality to manipulate strings in the form of library functions like strcpy, strcmp, strcat, strrev, etc.
13. Real-time applications often require storage and processing of a number of strings at a time. A list of strings can be stored by using an array of strings or by using an array of character pointers.
14. The main function can also take string inputs from the command line. The arguments given to the function main from the command line are known as command line arguments.

Exercise Questions

Conceptual Questions and Answers

1. *What is a null character?*

   A character constant with an ASCII value of zero is known as the null character and is written as '\0'.

2. *What is a character string literal constant? How is it written and stored in the memory?*

   Refer Section 7.2 for a description on character string literal constants.

3. *What can the maximum number of characters in a character literal constant be?*

   The character constant can be one (e.g. 'A') or two (e.g. '\n') characters long. Hence, the maximum number of characters in a character literal constant can be two.

4. *What would be the size of the following arrays:*

```
char str1[]= "Hello";
char str2[]={'H','e','l','l','o'};
```

   The character array str1 is initialized with a character string literal constant "Hello". Since a character string literal constant is terminated by a null character '\0', the contents stored in the character array str1 will be (say array is allocated at 2000):

str1	H	e	l	l	o	\0
Memory addresses →	**2000**	**2001**	**2002**	**2003**	**2004**	**2005**

   The character array str2 is initialized with the five initializers in the initialization list. Hence, the contents of str2 will be (say array is allocated at 4000):

str2	H	e	l	l	o
Memory addresses →	**4000**	**4001**	**4002**	**4003**	**4004**

   Therefore, the size of array str1 is 6 and that of str2 is 5.

5. *What are the different ways to print character arrays?*

   The following code illustrates four different ways to print character arrays:

```
main()
{
    char character_array[]="Example";
    int i;
    printf(character_array);                  //← Way 1
    printf("\n%s\n",character_array);         //← Way 2
    puts(character_array);                    //← Way 3
    for(i=0;character_array[i]!='\0';i++)     //← Way 4
        printf("%c",character_array[i]);
}
```

   In way 1, the character array is printed without using any format specifier. The first argument of the printf function must be of type const char* and the array name character_array is implicitly converted to pointer type char*. Since the types const char* and char* are compatible, the compiler implicitly converts char* to const char*. Therefore, this usage is perfectly valid. This type of usage however has a limitation that only one character array can be printed at a time.

In way 2, the character array is printed by using a %s format specifier. This type of usage has an advantage that many character arrays can be printed by a single call to the printf function by using multiple %s specifiers. For example:

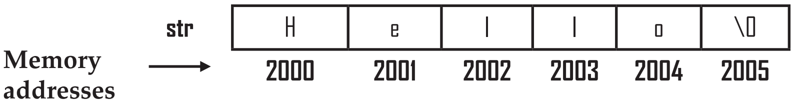
```
main()
{
    char character_array1[]="Hello";
    char character_array2[]="Readers";
    printf("%s %s",character_array1,character_array2);
}
```

In way 3, the puts function is used to print the character array. The difference between the puts and printf function is that the puts function places a new line character at the end, while the printf function does not do so.

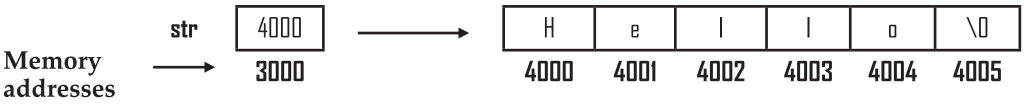
In way 4, a for loop is used to print all the characters of the array character_array one by one.

6. *Is the declaration* char str[6]="Hello" *same as* char *str="Hello"*?*

No, the declaration char str[6]="Hello"; is not the same as char *str="Hello";. The first declaration statement declares str to be a character array of size six and initializes the elements of array str with the characters of the string literal constant "Hello". However, the second declaration statement declares str to be a pointer to the character type and initializes it with the base address of string "Hello". The difference between the two declarations is shown in the figure below:



(a) char str[6]="Hello";



(b) char *str="Hello";

Another difference is that the first declaration statement allocates six bytes of the memory space to str, while the second declaration allocates two bytes to str (since it is a pointer).

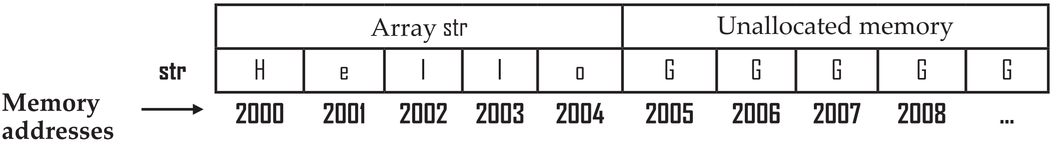
i It is very important to note that arrays are not pointers, although they are very closely related and sometimes have similar usage. For example, it is valid to write puts(str) and printf(str), where str is either declared by (a) or (b) as shown in the figure above.

7. *The following piece of code on execution gives some garbage. Why?*

```
main()
{
    char str[5]="Hello";
    puts(str);
}
```

The puts function outputs a sequence of characters (i.e. a string) on the screen. The output starts from the character pointed to by the pointer argument and is carried out till the null character is encountered.

The declaration char str[5]="Hello"; creates a character array of five locations and initializes the locations with the characters 'H', 'e', 'l', 'l' and 'o'. The array does not have the space to accommodate the null character. The array allocation and the memory contents are shown in the figure below:

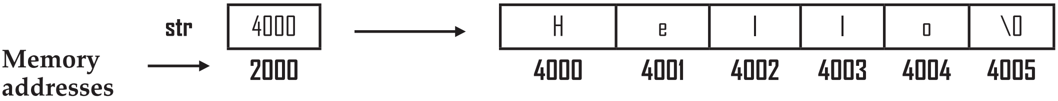


The function call puts(str) prints the characters starting from location 2000 till the null character is encountered. Since the character array str does not have the terminating null character, the output will be **Hello followed by some garbage characters**. The number of garbage characters depends upon where the null character (i.e. 0 value) is encountered in the memory. Execution of the same code at different times or on different machines may give a different number of garbage characters.

8. *Will the following piece of code also give some garbage as the previous code does?*

```
main()
 {
       char *str="Hello";
       puts(str);
 }
```

No, the mentioned piece of code outputs Hello and does not give any garbage character. The declaration char* str="Hello"; creates str as a 'pointer to character' and initializes it with the base addresses of string literal constant "Hello". This can be depicted as:



The function call puts(str) starts printing the characters from the memory location 4000 till the null character is encountered. Since there is a null character '\0' available at the memory location 4005, the output will be Hello only without any garbage.

9. *Why does the following piece of code not work? Rectify it.*

```
main()
{
    char string[15]="Hello Readers";
    strcat(string,'!');
    puts(string);
}
```

The following piece of code on compilation gives 'Cannot convert char to char*' error. The error is due to the fact that the strcat function expects two arguments of type char* (i.e. both the arguments should be strings). In the function call, strcat(string,'!'); the second argument is a character (i.e. of type char) and is not a string (i.e. of type char*). The conversion from type 'char to char*' is not a standard conversion, hence, the compiler will not carry it out implicitly and flags it as an error. The rectified call to the strcat function is strcat(string,"!");.

10. *What is the difference between* strchr *and* strrchr *functions?*

    Refer Sections 7.5.10 and 7.5.11 for a description on strchr and strrchr functions.

11. *Describe the behavior of the* scanf *function when applied on strings.*

    Refer Section 4.6 for a description on the behavior of the scanf function when applied on strings.

12. *The following piece of code compiles successfully. However, on execution gives an exception. Why? Rectify it.*

```
main()
{
    char *str;
    printf("Enter a string\t");
    gets(str);
    printf("The string entered was\t");
    puts(str);
}
```

The given code on execution gives an exception because before calling the gets function we have not allocated sufficient memory space to store the string entered by the user. There will be no compilation error because the gets function has no way to check whether the memory space pointed to by str is allocated or not.

The following are the rectified pieces of equivalent code:

```
#include<stdio.h>
main()
{
    char str[10];
    printf("Enter a string\t");
    gets(str);
    printf("The entered string was\t"):
    puts(str);
}
```

Rectified code 1

```
#include<stdio.h>
#include<alloc.h>
main()
{
    char *str=(char*)malloc(10);
    printf("Enter a string\t");
    gets(str);
    printf("The entered string was\t");
    puts(str);
}
```

Rectified code 2

In the rectified code 1, str has been declared to be a character array of size 10. Hence, 10 bytes are allocated to str at the compile time. In the rectified code 2, the **malloc** (i.e. memory allocate) function is used to allocate 10 bytes of memory. malloc function returns a void pointer to the allocated memory space. The void* is type casted to char* and is assigned to str, i.e. str is made to point to the allocated memory space.

i Some of the compilers like GNU GCC compiler, Borland Turbo C 3.0, etc. may not generate an exception, if the uninitialized pointer like str is used with the gets function.

13. *What would be the output of the following piece of code?*

```
main()
{
    char str[10]="ab\n\tcd";
    printf("Size of string is %d",strlen(str));
}
```

The given piece of code on execution outputs:

Size of string is 6

Character sequences like \n are interpreted at the compile time. When a backslash and an adjacent character n appear in a character constant or a string literal constant, they are immediately translated into a single new line character, i.e. one token. Similar translations also occur for other character escape sequences like \t, \b, \r, etc.

Hence, the string literal constant "ab\n\tcd" has six characters namely 'a', 'b','\n', i.e. new line character, '\t,' i.e. tab character, 'c' and 'd'.

14. *Consider the following piece of code:*

```
main()
{
    char str[10];
    gets(str);
    printf("Size of string is %d",strlen(str));
}
```

What would the output of the mentioned piece of code be, if the user entered the same string as in the previous question, i.e. "ab\n\tcd"?

On execution of the code, if the user enters the string "ab\n\tcd", the output of the code would be:

Size of string is 8

The output of this code is different from the output of the previous question because of the fact that when strings are taken from the user or read from a file at the run time, no interpretation of character sequences like \n, \t, etc. is performed. '\' and 'n' are treated as separate characters and are not transformed into single characters. The same is true for other escape sequences.

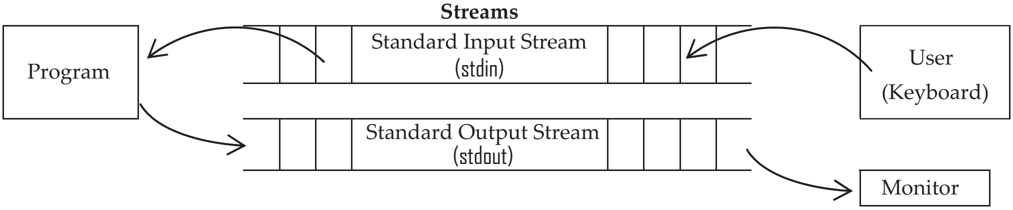
Hence, the string "ab\n\tcd" entered by the user at the run time has eight characters namely 'a', 'b','\', 'n', '\', 't', 'c' and 'd'.

15 *Consider the following piece of code:*

```
main()
{
    char str1[20],str2[20];
    printf("This code demonstrates two different ways to read strings\n");
    printf("Enter string 1\t");
    scanf("%s",str1);
    printf("Enter string 2\t");
    gets(str2);
    printf("\nThe strings entered were\n");
    puts(str2);
    puts(str1);
}
```

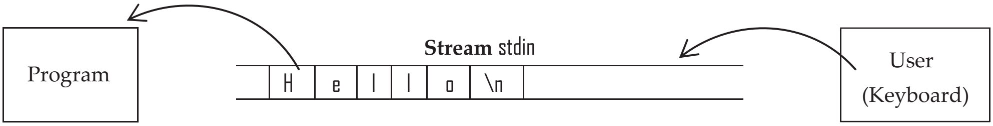
On execution, the code does not use the prompt to enter string 2 and directly starts printing the strings. Why?

The reason behind this behavior can be understood by learning how input and output are done in C. All the input and output in C are done with **streams.** A **stream** can be thought of as a buffer from which a sequence of data elements is made available during input or to which a sequence of data elements is written during output. The figure shown below depicts how input and output are done by means of input and output streams.



All the input functions like scanf, gets, getc etc. read from the **standard input stream** stdin and prompt the user to enter the data only if the stream is empty. If the stream already contains data or some characters, the input function will not prompt the user and silently retrieves the already available characters from the stream.

Suppose on execution of the given code, the user typed Hello and pressed the Enter key. The contents entered into the standard input stream are shown in the figure below.



The scanf function retrieves all the characters from stdin up to but not including the white-space character. Hence, after the execution of the function call scanf("%s",str1);, Hello is removed from stdin and is stored in str1 but the new line character still remains in the stream stdin. The call to the function gets(str2); finds the new line character in the stream. That is why it does not prompt the user to make input. It silently removes that new line character from stdin and stores it in str2.

This problem can be solved by removing the new line character from the stdin stream before giving call to the gets function. This can be done either by calling function flushall(); or fflush(stdin);. The rectified piece of code is as follows:

```
main()
{
    char str1[20],str2[20];
    printf("This code demonstrates two different ways to read strings\n");
    printf("Enter string 1\t");
    scanf("%s",str1);
    printf("Enter string 2\t");
    flushall();    //flushall(); flushes all the streams
                   //or fflush(stdin); flushes only stdin stream.
    gets(str2);
    printf("\nThe strings entered were\n");
    puts(str2);
    puts(str1);
}
```

Code Snippets

Determine the output of the following code snippets. Assume that the inclusion of the required header files has been made and there is no prototyping error due to them.

16.
```
main()
{
    char str1[]= "Strings";
    char str2[]={'S','t','r','i','n','g','s'};
```

```
         puts(str1);
         puts(str2);
    }
17. main()
    {
         char str[]=”Strings”;
         int i;
         for(i=0;str[i];i++)
         printf(“%c”,str[i]);
    }
18. main()
    {
         printf(“%d %d”,sizeof(‘A’),sizeof(“A”));
    }
19. main()
    {
         char str1[]=”Hello”;
         char *str2=”Hello”;
         printf(“%d %d\n”,sizeof(str1),sizeof(str2));
         printf(“%d %d”,sizeof(*str1),sizeof(*str2));
    }
20. main()
    {
         char str[]=”Characters”;
         printf(“%d %d”,strlen(str) ,sizeof(str));
    }
21. main()
    {
         char str1[]=”Hello”;
         char str2[]=”Readers!”;
         printf(“Hello ””Readers!””\n”);
         puts(“Hello ””Readers!”);
         printf(“%s %s”,str1,str2);
    }
22. main()
    {
         char str1[]=”Hello”;
         char str2[]=”Readers!”,
         puts(str1,str2);
    }
23. main()
    {
         char str1[]=”Hello”;
         char str2[]=”Readers!”;
         puts((str1,str2));
    }
24. main()
    {
         char *str;
```

```
     str="Hello","Readers!";
     puts(str);
}
```

25.
```
main()
{
     char *str;
     str=("Hello","Readers!");
     puts(str);
}
```

26.
```
main()
{
     char str1[]="Hello";
     char str2[]="Readers!";
     printf(str1,str2);
}
```

27.
```
main()
{
     char str[]="HelloReaders!";
     printf(“%s %s %s”,&str[5],&5[str],str+5);
}
```

28.
```
main()
{
     char str[]="Hello Readers!";
     printf(“%c %c %c”,str[6],6[str],*(str+6));
}
```

29.
```
main()
{
     printf(“Hello Readers!”+6);
}
```

30.
```
main()
{
     putchar(“Hello Readers!”[6]);
     putchar(6["Hello Readers!"]);
}
```

31.
```
main()
{
     printf(“The size of string is %d\n”,sizeof(“Hello Readers!”));
     printf(“The string is allocated memory starting at %p”,&"Hello Readers!");
}
```

32.
```
main()
{
     char str1[]="Strings!";
     char str2[]="Strings!";
     if(str1==str2)
          printf(“Strings are same!!”);
     else
          printf(“Strings are different!!”);
}
```

```
33. main()
    {
        char str1[]="Strings!";
        char str2[]="Strings!";
        if(strcmp(str1,str2)==0)
            printf(“Strings are same!!”);
        else
            printf(“Strings are different!!”);
    }
```

```
34. main()
    {
        char str1[]="strings!";
        char str2[]="STRINGS!";
        if(strcmp(str1,str2)==0)
            printf(“Strings are same!!”);
        else
            printf(“Strings are different!!”);
    }
```

```
35. main()
    {
        char str1[]="strings!";
        char str2[]="STRINGS!";
        if(strcmpi(str1,str2)==0)
            printf(“Strings are same!!”);
        else
            printf(“Strings are different!!”);
    }
```

```
36. main()
    {
        if(strcmp(“Strings”,"Strings\0”))
            printf(“Strings are different!!”);
        else
            printf(“Strings are same!!”);
    }
```

```
37. main()
    {
        char str1[]={‘S’,’t’,’r’,’i’,’n’,’g’,’s’};
        char str2[]="Strings";
        if(strcmp(str1,str2))
            printf(“Strings are different!!”);
        else
        printf(“Strings are same!!”);
    }
```

```
38. main()
    {
        char format[]="%d\n";
        format[1]=’c’;
    printf(format,65);
}
```

39.
```
main()
{
    char format[]={37,111,32,37,120,0};
    printf(format,format[0],format[1]);
}
```

40.
```
main()
{
    char str1[]=”Strings”;
    char str2[10];
    str2=str1;
    puts(str1);
    puts(str2);
}
```

41.
```
main()
{
    char src[]=”Strings”;
    char dest[10];
    strcpy(dest,src);
    puts(src);
    puts(dest);
}
```

42.
```
main()
{
    char dest[]=”Visual Basic”;
    char src[]=”C++”;
    puts(strcpy(&dest[7],src));
}
```

43.
```
main()
{
    char dest[]=”Visual Basic”;
    char src[]=”C++”;
    strcpy(&dest[7],src);
    puts(dest);
}
```

44.
```
main()
{
    char dest[]=”Visual Basic”;
    char src[]=”Visual C++”;
    strcpy(&dest[7],&src[7]);
    puts(dest);
}
```

45.
```
main()
{
    if(printf(“\0”))
        printf(“Characters”);
    else
```

```
          printf(“Strings”);
    }
46. main()
    {
        char cities[][11]={“Delhi”,”Chandigarh”,”Noida”};
        int i;
        for(i=0;i<3;i++)
            puts(cities[i]);
    }
47. main()
    {
        char languages[5][20]={“Visual Basic”,”Java”,”Fortran”,”C”,”C++”};
        int i; char *t;
        t=languages[3];
        languages[3]=languages[4];
        languages[4]=t;
        for (i=0;i<=4;i++)
            printf(“%s\n”,languages[i]);
    }
48. main()
    {
        char *languages[]={“Basic”,”Java”,”Fortran”,”C”,”C++”};
        int i; char *t;
        t=languages[3];
        languages[3]=languages[4];
        languages[4]=t;
        for (i=0;i<=4;i++)
            printf(“%s\n”,languages[i]);
    }
49. main()
    {
        char lang[5][20]={“Visual Basic”,”Java”,”Fortran”,”C”,”C++”};
        int i; char *t;
        t=lang[0];
        while(*t++!=32);
            for(i=0;i<5;i++)
            {
                puts(lang[0]);
                strcpy(t,lang[i+1]);
            }
    }
50. main()
    {
        int i,len;
        char *ptr=”String”;
        len=strlen(ptr);
        for(i=0;i<len;i++)
        {
            puts(ptr);
            ptr++;
        }
    }
```

51.

```
string_manipulation(char[][]);
main()
{
    char arr[][10]={“Hello”,”Students”};
    string_manipulation(arr);
    printf(“%s %s”,arr[0],arr[1]);
}
string_manipulation(char arr[][])
{
    strcpy(arr[1],”Readers!!”);
}
```

52.

```
string_manipulation(char(*)[10]);
main()
{
    char arr[][10]={“Hello”,”Students”};
    string_manipulation(arr);
    printf(“%s %s”,arr[0],arr[1]);
}
string_manipulation(char (*arr)[10])
{
    strcpy(arr[1],”Readers!!”);
}
```

53.

```
string_manipulation(char[][10]);
main()
{
    char arr[][10]={“Hello”,”Students”};
    string_manipulation(arr);
    printf(“%s %s”,arr[0],arr[1]);
}
string_manipulation(char arr[][10])
{
    strcpy(arr[1],”Readers!!”);
}
```

54.

```
char[20] print_string()
{
    char str[20]=”Strings!!”;
    return str;
}
main()
{
 puts(print_string());
}
```

55.
```
main(int argc,char*argv[])
{
    int i;
    printf("The argument count is %d\n",argc);
    printf("The content of argument vector i.e. array is\n");
    for(i=0;i<argc;i++)
        printf("%s\n",argv[i]);
}
```

Suppose the name of the program file is ques55.c *and the executable file* ques55.exe *is invoked from the command prompt as follows:*

```
c:\>ques55 Hello Readers!!
```

Multiple-choice Questions

56. The maximum number of characters in a character literal constant can be

a. One
b. Two
c. Three
d. As many as the user likes

57. The size occupied by a string literal constant in the memory is

a. One more than the number of characters in the string
b. Same as the number of characters in the string
c. One less than the number of characters in the string
d. None of these

58. The value returned by the strlen function when a string literal constant is given to it as an argument is

a. One more than the number of characters in the string argument
b. Same as the number of characters in the string argument
c. One less than the number of characters in the string argument
d. None of these

59. String literal constants are terminated by

a. New line character
b. Carriage return character
c. Null character
d. None of these

60. The ASCII code of the null character is

a. 32
b. 27
c. 13
d. 0

61. The output of the statement printf("%d","123456"[1]); is

a. 1
b. 2
c. 50
d. None of these

62. The output of the statement printf("%s","123456"+1); is

a. 123456
b. 123457
c. 23456
d. None of these

63. The correct way to compare two string literal constants "Hello" and "Hi" is

a. "Hello"="Hi"
b. "Hello"=="Hi"
c. strcmp("Hello","Hi")
d. None of these

64. The output of the statement puts(“\0ABCD\0”); is

a. ABCD
b. No output
c. \0ABCD
d. Compilation error

65. The result of evaluation of the expression strcmp(“Hello”,“Hi”); will be

a. 0
b. 4
c. –4
d. None of these

66. The correct statement to copy a string literal constant “Hello” to a character array str is

a. str=“Hello”;
b. strcpy(str,“Hello”);
c. strcpy(“Hello”,str);
d. None of these

67. Adjacent string literal constants

a. Are always concatenated
b. Are treated as two separate tokens
c. Leads to compilation error
d. None of these

68. The invocation of the function call strcat(“Hi”,“Readers!!”); leads to

a. HiReaders!!
b. Compilation error
c. Run-time exception
d. None of these

69. The invocation of the function call puts(“Hi”,“Readers!!”); leads to

a. HiReaders!!
b. Compilation error
c. Run-time exception
d. None of these

70. The invocation of the function call puts(“Hi””Readers!!”); leads to

a. HiReaders!!
b. Compilation error
c. Run-time exception
d. None of these

71. The output of the following program file ques71.c, if executed from the command line as ques71 1 2 3, is

```
main(int argc, char* argv[])
{
    int val;
    val=argv[1]+argv[2]+argv[3];
    printf(“%d”,val);
}
```

a. 6
b. 123
c. Compilation error
d. None of these

72. The output of the following program file ques72.c, if executed from the command line as ques72 1 2 3, is

```
#include<stdlib.h>          //←atoi function converts string to an integer and its
                             //←prototype is in stdlib.h
main(int argc, char* argv[])
{
    int val;
    val=atoi(argv[1])+atoi(argv[2])+atoi(argv[3]);
   printf(“%d”,val);
}
```

a. 6
b. 123
c. Compilation error
d. None of these

73. The output of the following program file ques73.c, if executed from the command line as ques73 1 2, is

```
main(int argc, char* argv[])
{
    char str[10];
    strcpy(str,argv[1]);
    strcpy(str,argv[2]);
    printf("%s",str);
}
```

a. 1
b. 2
c. 12
d. None of these

74. The output of the following program file ques74.c, if executed from the command line as ques74 1 2, is

```
main(int argc, char* argv[])
{
    char str[10];
    strcpy(str,argv[1]);
    strcat(str,argv[2]);
    printf("%s",str);
}
```

a. 1
b. 2
c. 12
d. None of these

75. Which of the following is true about argv?

a. It is an array of character pointers
b. It is a pointer to an array of character pointers
c. It is an array of characters
d. None of these

Outputs and Explanations to Code Snippets

16. Strings
    Strings!¥¤§¶

    Explanation:

    As the character array str1 is initialized with the character string literal constant, str1[7] will be a null character. However, as the character array str2 is initialized with a list of characters, i.e. 'S', 't', 'r', 'i', 'n', 'g' and 's', no terminating null character is placed in it. Hence, the puts function while printing str2 gives garbage as it prints from the memory location pointed to by its argument till the terminating null character is encountered.

17. Strings

    Explanation:

    The for loop is used to print the elements of character array str one by one. The loop terminates when the value of i becomes 7 and str[i] evaluates to 0 (i.e. the ASCII value of null character). for(i=0;str[i];i++) is equivalent to writing for(i=0;str[i]!='\0';i++).

18. 1 2

    Explanation:

    'A' is a character constant and characters take one byte in the memory. "A" is a string literal constant and string literal constants are terminated by a null character, i.e. '\0'. So "A" is actually made up of two characters, i.e. 'A' and '\0'. Hence, sizeof("A") comes out to be 2.

19.
```
6 2
1 1
```
Explanation:

str1 is a character array of 6 locations while str2 is a pointer to character. Hence, size of str1 and str2 would be 6 and 2, respectively (in Borland TC 3.0 for DOS), and 6 and 4 (in Borland TC 4.5 for Windows or Microsoft Visual C++ 6.0). The usage of * with str1 and str2 dereferences them to a character and hence, sizeof(*str1) and sizeof(*str2) would be 1 and 1.

20.
```
10 11
```
Explanation:

The strlen function computes the length of a string given to it as an argument. The strlen function does not count the null character while computing the length of the string. It returns the number of characters that precedes the terminating null character. On the other hand, the sizeof function also counts the memory required by the null character while computing the number of memory bytes occupied by the string.

21.
```
Hello Readers!
Hello Readers!
Hello Readers!
```
Explanation:

Adjacent character string literal constants are concatenated. Hence, writing "Hello ""Readers!""\n" is equivalent to writing "Hello Readers!\n". The printf function outputs this character string literal constant onto the screen. The puts function also does the same with a difference that it places the new line character at the end. Hence, there is no requirement of the new line character in the string given to puts. The last call to the printf function uses %s specifiers to output the contents of the character arrays str1 and str2.

22.
```
Compilation error "Extra parameter in call to puts"
```
Explanation:

The puts function expects only one argument of char* type while in the call puts(str1,str2); two arguments of type char* are provided. Hence, there is an extra parameter in the function call, which is the source of error.

23.
```
Readers!
```
Explanation:

The argument of the puts function is an expression (str1,str2). Now, the instance of the comma symbol separating str1 and str2 in the expression (str1,str2) is treated as a comma operator. The comma operator guarantees left-to-right evaluation and returns the result of the rightmost sub-expression. Therefore, the expression (str1,str2) evaluates to str2. Hence, the string Readers! gets printed.

24.
```
Hello
```
Explanation:

The string literal constant refers to the address of its initial element except in the cases when it is an operand of the sizeof operator or the unary & operator. Hence, "Hello" and "Readers!" refer to the starting addresses of the strings "Hello" and "Readers!". In the expression, str="Hello","Readers!", the assignment operator has a higher priority as compared to the comma operator. Hence, the assignment operator is evaluated first and the starting address of the string literal constant "Hello" is assigned to str. In the next statement, the string pointed to by str is printed by the puts function. Hence, "Hello" is the output.

25. Readers!

Explanation:

The use of parentheses makes the comma operator to be evaluated first. The comma operator returns the result of the rightmost sub-expression. Therefore, in the expression str=("Hello","Readers!"), the starting address of the string "Readers!" is assigned to str. In the next statement, the string pointed to by str is printed by the puts function. Hence "Readers!" is the output.

26. Hello

Explanation:

On compilation, the given code does not produce a compilation error as the code of Question number 22 does. This is due to the fact that printf is a variable argument function. It can take a variable number of arguments while puts can only take one argument. Examples when the printf function takes 1, 2 and 3 arguments are as follows:

```
printf("Hello Readers");     //← Only one argument
printf("%d",2);              //← Two arguments
printf("%d %d",2,3);         //← Three arguments
```

The following important points should also be remembered:

1. The comma symbol appearing in a printf function call is not treated as a comma operator.
2. The printf function expects the first argument to be a string (commonly known as a format string). It actually prints only the first argument while the other arguments available in the printf function replace the format specifiers in the first string, if they are present. If no format specifiers are present in the format string, the arguments following the first argument are ignored.

Hence, the output of printf(str1, str2); is Hello, as the string str1 does not contain a format specifier.

27. Readers! Readers! Readers!

Explanation:

The declaration statement char str[]="HelloReaders!"; allocates str, say at the memory location 2000. The contents of the array are shown in the figure below:

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
str	H	e	l	l	o	R	e	a	d	e	r	s	!	\0
	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013

The printf function prints the sequence of characters from the address given as an argument till null character is encountered. All the expressions &str[5], &5[str] and str+5 evaluate to 2005. Therefore, the printing starts from the character present at the location 2005 and is carried out till a null character is encountered.

28. R R R

Explanation:

The expressions str[6], 6[str] and *(str+6) refer to the seventh character of the array str, i.e. R.

29. Readers!

Explanation:

Suppose the string literal constant "Hello Readers!" is allocated the memory space from the address 2000 to 2014 as depicted in the figure below:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
H	e	l	l	o		R	e	a	d	e	r	s	!	\0
2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014

String literal constant refers to the address of its initial element except in the cases when it is an operand of the sizeof operator or the unary & operator. Hence, the expression "Hello Readers!" evaluates to 2000, and the expression "Hello Readers!"+6 evaluates to 2006. When the expression "Hello Readers!"+6 is given as an argument to the printf function, the printf function starts printing the characters from 2006 till a null character is encountered. Hence, the output comes out to be Readers!.

30. RR

    Explanation:

    The function putchar outputs the character given to it as an argument on the screen. Suppose the string "Hello Readers!" is allocated the same memory location as in Answer number 29.

    In the first call to the function putchar, the argument is an expression "Hello Readers!"[6]. This expression gets converted to the form *("Hello Readers!"+6). The expression *("Hello Readers!"+6) evaluates to *(2000+6), i.e. *(2006), i.e. R (refer to the explanation given in Answer number 29). Similarly, 6["Hello Readers!"] evaluates to R.

31. The size of string is 15
    The string is allocated memory starting at 2A4F:00AD

    Explanation:

    The string literal constant expression does not decompose into the pointer to its initial element when it is an operand of the sizeof operator or the unary & operator.

32. Strings are different!!

    Explanation:

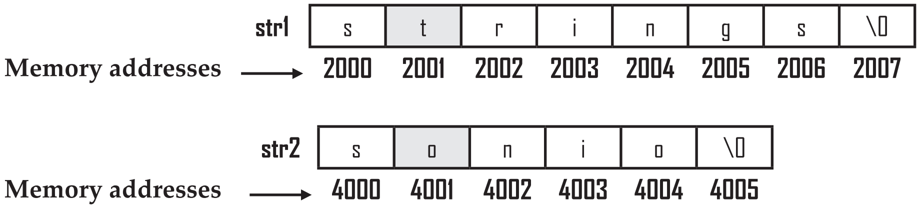
    In the expression str1==str2, str1 and str2 are the names of character arrays and refer to the addresses of their first elements. Since the addresses of the first element of two arrays can never be the same, the expression str1==str2 evaluates to false and Strings are different!! is the output.

33. Strings are same!!

    Explanation:

    strcmp(str1,str2) performs a comparison between str1 and str2, starting with the first character in each string and continuing with the subsequent characters until the corresponding characters differ or until the end of strings is reached. It returns the ASCII difference of the first dissimilar corresponding characters or zero if none of the corresponding characters in both the strings are different.

    For example, when strcmp function is applied on the strings str1 (say strings) and str2 (say sonio) shown in the figure below, it returns 116-111 (i.e. ASCII code of 't' – ASCII code of 'o') = 5.



str1	s	t	r	i	n	g	s	\0
Memory addresses →	2000	2001	2002	2003	2004	2005	2006	2007

str2	s	o	n	i	o	\0
Memory addresses →	4000	4001	4002	4003	4004	4005

strcmp(str1,str2) returns a value equal to:

- 0 if str1 and str2 are equal, or
- >0 if str1 is greater than str2, i.e. str1 comes after str2 in lexicographic order, or
- <0 if str1 is lesser than str2, i.e. str1 comes before str2 in lexicographic order.

In the given question, str1 and str2 being the same, the expression strcmp(str1,str2)==0 evaluates to true as the strcmp function returns 0. Hence, Strings are same!! is the output.

34. Strings are different!!

Explanation:

The strcmp function when used to compare str1 and str2 returns 115-83 (i.e. ASCII code of 's'-ASCII code of 'S') = 32. The returned value is not equal to zero. Hence, the expression strcmp(str1,str2)==0 evaluates to false and Strings are different!! is the output. Note that the strcmp function considers the case sensitivity of the characters while comparing the strings.

35. Strings are same!!

Explanation:

The strcmpi function compares the strings without case sensitivity. The character i in the strcmpi function stands for ignore case.

36. Strings are same!!

Explanation:

Suppose, the character string literal constants "Strings" and "Strings\0" are allocated the memory space at the addresses 2000 and 4000 as shown in the figure below:

	S	t	r	i	n	g	s	\0	
Memory addresses →	**2000**	**2001**	**2002**	**2003**	**2004**	**2005**	**2006**	**2007**	
	S	t	r	i	n	g	s	\0	\0
Memory addresses →	**4000**	**4001**	**4002**	**4003**	**4004**	**4005**	**4006**	**4007**	**4008**

The function call strcmp("Strings","Strings\0") compares characters of both the strings one by one until the corresponding characters differ or until the end of the strings is reached. In the given piece of code, the function call terminates by comparing the null characters located at the locations 2007 and 4007 and returns 0. Hence, Strings are same!! is the output.

37. Strings are different!!

Explanation:

Suppose that the character array str1 gets allocated at the memory address 2000 as shown in the figure below. The first seven elements of the character array str1 are initialized with the characters 'S', 't', 'r', 'i', 'n', 'g' and 's', and the memory locations following 2006 contain garbage values.

	[0]	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**					
str1	S	t	r	i	n	g	s	G	G	G	G	
	2000	**2001**	**2002**	**2003**	**2004**	**2005**	**2006**	**2007**	**2008**	**2009**	**2010**	**...**

G (in the above figure) means garbage value.

The contents of the character array str2 allocated at the memory address 4000 are shown in the figure below:

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]				
str2	S	t	r	i	n	g	s	\0	G	G	G	G
	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	...

The function strcmp(str1,str2) returns the ASCII difference of the first dissimilar corresponding characters or zero if there is no dissimilarity. The first dissimilarity in str1 and str2 is in the characters located at 2007 and 4007, respectively. Hence, the function strcmp returns the difference between garbage value G and 0 (i.e. the ASCII code of the null character). There is high probability that this garbage value is a non-zero value, and hence Strings are different!! is the output. However, by chance, if the garbage value located at 2007 is 0 (which has lesser probability), then the output will be Strings are same!!.

38. A

Explanation:

The contents of the character array format after initialization are shown in the figure below:

	[0]	[1]	[2]	[3]
format	%	d	\n	\0
	2000	2001	2002	2003

After the execution of the assignment statement format[1]='c'; the contents of the character array format become:

	[0]	[1]	[2]	[3]
format	%	c	\n	\0
	2000	2001	2002	2003

Writing printf(format,65); is equivalent to writing printf("%c\n",65);. Printing of integer value 65 is done according to the %c format specifier; hence A is the output (since 65 is the ASCII value of 'A').

39. 45 6f

Explanation:

The character array format is initialized with an initialization list consisting of integer values. If the initialization list of a character array consists of integer values, then the locations of the array are initialized with the characters whose ASCII values are equivalent to the integer values in the initialization list. The characters having an ASCII value of 37 is '%', 111 is 'o', 32 is ' ', (i.e. blank space), 120 is 'x' and 0 is '\0', i.e. null character. The initialized contents of the character array format are shown in the figure below:

	[0]	[1]	[2]	[3]	[4]	[5]
format	%	o		%	x	\0
	2000	2001	2002	2003	2004	2005

Now, printf(format,format[0],format[1]); prints the value of format[0] (i.e. 37) and format[1] (i.e. 111) according to %o and %x format specifiers, respectively. Hence, the output comes out to be 45 and 6f as the octal equivalent of 37 is 45 and the hexadecimal equivalent of 111 is 6f.

40. Compilation error "L-value required"

Explanation:

str2 is the name of the character array and is a constant object. It cannot be placed on the left side of an assignment operator. Hence, writing str2=str1 is not valid and gives 'L-value required' error.

41. Strings
Strings

Explanation:

The strcpy(dest,src); copies the string pointed by src into the memory location pointed to by dest. The copying terminates after the terminating null character of src has been copied to dest. The strcpy function returns the starting address of the memory location where the string has been copied. Thus, after the execution of the function call strcpy(dest,src); both str and dest contain "Strings", and their contents are printed using the puts function.

42 C++

Explanation:

The contents of the character array dest and src are shown in the figure below:

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
dest	V	i	s	u	a	l		B	a	s	i	c	\0
	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012

	[0]	[1]	[2]	[3]
src	C	+	+	\0
	4000	4001	4002	4003

The function call strcpy(&dest[7],src); copies the contents of the source string src to the memory locations starting from 2007, i.e. &dest[7]. The contents of dest after the function call are as follows:

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
dest	V	i	s	u	a	l		C	+	+	\0	c	\0
	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012

The strcpy function returns the address of the memory location where the string has been copied. Hence, strcpy(&dest[7],src); returns 2007. The puts prints the sequence of characters starting from the memory location 2007 till a null character is encountered. Hence, C++ is the output.

43. Visual C++

Explanation:

Refer to the explanation given in Answer number 42.

44.
```
Visual C++
```
Explanation:

Refer to the explanation given in Answer number 42.

45.
```
Strings
```
Explanation:

The printf function returns an integer value equivalent to the number of characters printed. Printing of the null character using the printf function returns zero. Hence, Strings is the output.

46.
```
Delhi
Chandigarh
Noida
```
Explanation:

The content of a two-dimensional character array cities is shown in the figure below:

cities	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
[0]	D	e	l	h	i	\0					
	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010
[1]	C	h	a	n	d	i	g	a	r	h	\0
	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021
[2]	N	o	i	d	a	\0					
	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031	2032

Referring to a two-dimensional array with only one subscript gives the starting address of a row. Hence, the expression cities[0] refers to the starting address of the first row, i.e. 2000, and cities[1] refers to the starting address of the second row, i.e. 2011. The function call puts(cities[i]), prints the strings in the first, second and third rows.

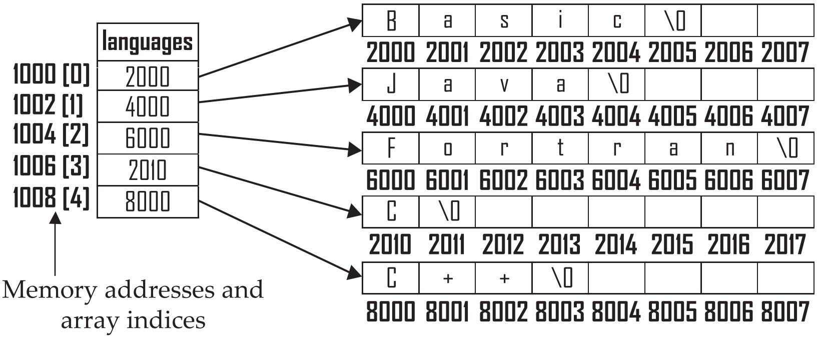
47 Compilation error "L-value required"

Explanation:

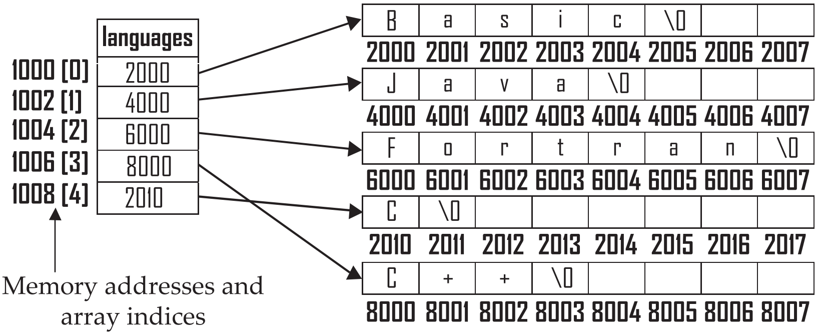
Referring to a two-dimensional array with only one subscript refers to the starting address of the row and is a constant object. The C compiler will not allow its manipulation. Hence, writing languages[3]=languages[4] is not valid and leads to 'l-value required' compilation error.

48.
```
Basic
Java
Fortran
C++
C
```
Explanation:

languages is an array of character pointers and is initialized with the base addresses of the string literal constants "Basic", "Java", "Fortran", "C" and "C++". Contiguous memory (say from the memory address 1000-1009) is allocated to the array languages while the string literal constants are placed randomly in the memory. This is depicted in the figure below:



The statements t=languages[3];, languages[3]=languages[4]; and languages[4]=t; swap the values of languages[3] and language[4]. After the execution of these statements, the contents of array languages are as depicted in the figure below:



Thus, the printing of the strings pointed by the content of the array languages yields the mentioned result.

49. Visual Basic
    Visual Java
    Visual Fortran
    Visual C
    Visual C++

Explanation:

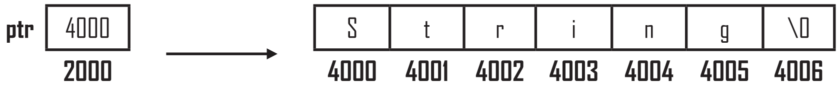
The content of the two-dimensional character array lang is shown below:

lang	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
[0]	V	i	s	u	a	l		B	a	s	i	c	\0
	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012
[1]	J	a	v	a	\0								
	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025
[2]	F	o	r	t	r	a	n	\0					
	2026	2027	2027	2028	2029	2030	2031	2032	2033	2034	2035	2036	2037
[3]	C	\0											
	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047	2048	2049	2050
[4]	C	+	+	\0									
	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063

The character pointer t, say, gets allocated at 4000. After the execution of the assignment statement t=lang[0]; t starts pointing to the starting address of the first row of the character array lang. After the execution of while(*t++!=32); statement, t points to the location next to the blank space (ASCII value 32) in the first row of lang, i.e. 2007. Each iteration of the for loop with the loop counter value i prints the content of lang[0] and copies the strings in the row i+1 at the memory location pointed by t, i.e. 2007.

50.
```
String
tring
ring
ing
ng
g
```
Explanation:

Suppose the character pointer ptr and the character string literal constant "String" are allocated the memory space as shown in the figure below. Since is initialized with the character string literal constant, it points to the starting address of the string literal, i.e. 4000.



Every iteration of the for loop prints a string being pointed by ptr and increments the contents of the pointer ptr.

51. Compilation error

Explanation:

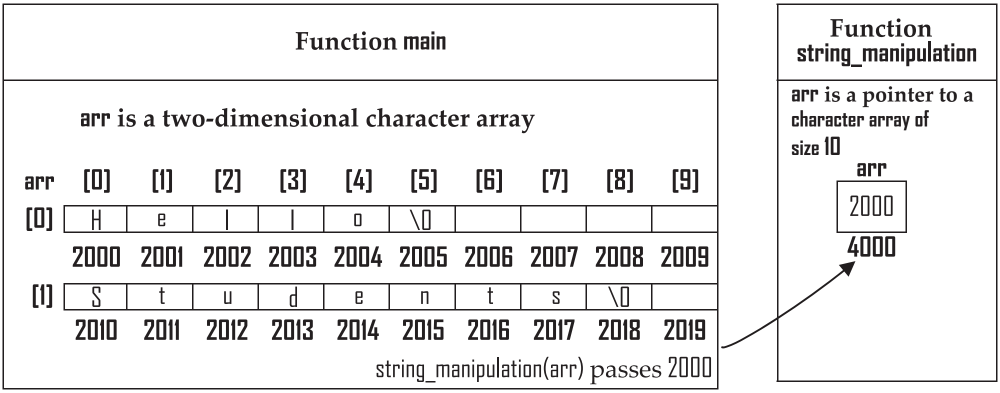
Refer to the explanation given in Answer number 59 (Chapter 6).

While declaring a two-dimensional array, both the row size and column size cannot be left blank. It is mandatory to mention the column size specifier. Hence, the declaration string_manipulation(char [][]); is not valid. It can be rectified by mentioning the column size specifier as string_manipulation(char [][10]);. The same should also be done in the function header.

52. Hello Readers!!

Explanation:

The two-dimensional character array arr is passed by reference to the string_manipulation function. Suppose the array arr (local to the function main) gets allocated at the memory location 2000. The name arr declared in the function header string_manipulation is of type pointer to the character array of size 10 and is local to the function string_manipulation. Suppose it gets allocated at the memory location say 4000. The name of a two-dimensional array refers to the starting address of the first row of the array. Therefore, the function call string_manipulation(arr); passes 2000, i.e. the starting address of the first row of the array to the function string_manipulation. This is shown in the figure below:


Function main
arr is a two-dimensional character array
arr [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
[0] H e l l o \0
2000 2001 2002 2003 2004 2005 2006 2007 2008 2009
[1] S t u d e n t s \0
2010 2011 2012 2013 2014 2015 2016 2017 2018 2019
string_manipulation(arr) passes 2000
Function string_manipulation
arr is a pointer to a character array of size 10
arr
2000
4000

The call to the function strcpy inside the body of function string_manipulation copies the string "Readers!!" at arr[1], i.e., 2010 (because arr is a pointer to a character array of size 10). Thus, the string "Readers!!" overwrites the string "Students" present in the second row of the character array arr. That is why when arr[0] and arr[1] are printed, the output comes out to be Hello Readers!!.

53. Hello Readers!!

Explanation:

The parameter declaration char arr[][10] gets implicitly converted to char(*arr)[10]. Thus, the mentioned code becomes equivalent to the code given in Question number 52.

Refer to the explanation given in Answer number 52 for the output.

54. Compilation error

Explanation:

The return type of a function cannot be an array type. Since the return type of the function print_string is an array type, i.e. char [20], the compiler issues an error message.

55.
```
The argument count is 3
The content of argument vector i.e. array is
c:\ques55.exe
Hello
Readers!!
```

Explanation:

Refer to the explanation given in Section 7.7.

Arguments in command line are separated by blank spaces. Since there are two blank spaces, the total number of command line arguments is three. Note that the name of the program file (actually executable file) is also counted while determining the argument count. argv[0] points to c:\>ques55.exe, argv[1] points to Hello and argv[2] points to Readers!!. Therefore, these strings get printed.

Answers to Multiple-choice Questions

56. b 57. a 58. b 59.c 60. d 61. c 62. c 63. c 64. b 65. c 66. b 67. a 68. c
69. b 70.a 71. c 72. a 73. b 74. c 75. a

Programming Exercises

Program 1 | Input a string and find the number of vowel(s) present in the string

PE 7-1.c

```c
//Number of vowels in a string
#include<stdio.h>
main()
{
char string[200];
int count=0, i=0;
printf("Enter a string:\n");
gets(string);
while(string[i]!='\0')
{
    switch(string[i])
    {
        case 'A':
        case 'E':
        case 'I':
        case 'O':
        case 'U':
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            count++;
        }
    i++;
}
if(count==1)
  printf("One vowel is present in the string");
else
  printf("%d vowels are present in the string", count);
}
```

Output window

```
Enter a string:
There is nothing more beautiful in the world than a healthy wise old man- Yutang
25 vowels are present in the string
```

Program 2 | Input a string and count the number of occurrences of a particular character in the string

PE 7-2.c

```c
//Count number of occurrences of a particular character in the string
#include<stdio.h>
main()
{
    char string[200], ch;
    int count=0, i=0;
    printf("Enter a string:\n");
    gets(string);
    printf("Enter the character:\t");
    scanf("%c",&ch);
    while(string[i]!='\0')
    {
        if(string[i]==ch)
```

Output window

```
Enter a string:
Nature, time and patience are three great physicians- Bohn
Enter the character: e
In the given string, e occurred 8 times
```

(Contd...)

Line | PE 7-2.c | Output window

```
            count++;
        i++;
    }
    printf("In the given string, %c occurred %d times\n",ch, count);
}
```

Program 3 | Input a string and count the number of blank spaces in the string

Line | PE 7-3.c

```
//Number of blank spaces
#include<stdio.h>
main()
{
    char string[200], ch;
    int count=0, i=0;
    printf("Enter a string:\n");
    gets(string);
    while(string[i]!='\0')
    {
        if(string[i]==' ')
            count++;
        i++;
    }
    printf("Number of blank spaces in the given string are %d", count);
}
```

Output window

```
Enter a string:
People resent a joke if there is some truth in it- Tagore
Number of blank spaces in the given string are 11
```

Program 4- Input two strings of equal length from the user and determine how many times the corresponding positions in two strings hold exactly the same characters

Line | PE 7-4.c

```
//Number of same characters at the corresponding positions in two strings
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
main()
{
    char str1[30], str2[30];
    int length1, length2, count=0, i;
    printf("Enter two strings of equal length\n");
    printf("Enter first string:\t");
    gets(str1);
    printf("Enter second string:\t");
    gets(str2);
    length1=strlen(str1);
    length2=strlen(str2);
    if(length1!=length2)
    {
        printf("The entered strings are of different lengths\n");
        exit(1);
    }
```

Output window

```
Enter two strings of equal length
Enter first string:     choice
Enter second string:    chance
Corresponding positions hold same characters 4 times
```

Output window (second execution)

```
Enter two strings of equal length
Enter first string:     very
Enter second string:    much
Corresponding positions hold same characters 0 times
```

Output window (third execution)

```
Enter two strings of equal length
Enter first string:     life
Enter second string:    lovely
The entered strings are of different lengths
```

(Contd...)

```
    else
    {
        for(i=0;i<length1;i++)
            if(str1[i]==str2[i])
                count++;
        printf("Corresponding positions hold same characters %d times", count);
    }
}
```

Program 5 | Input a string and display the alternate characters of the string

Line | **PE 7-5.c**

```
//Printing alternate characters of a string
#include<stdio.h>
main()
{
    char str[200], altchars[200];
    int i=0, length, j=0;
    printf("Enter a string:\n");
    gets(str);
    length=strlen(str);
    while(i<length)
    {
        altchars[j]=str[i];
        i=i+2;
        j=j+1;
    }
    altchars[j]='\0';
    printf("Alternate characters in the string are:\n");
    puts(altchars);
}
```

Output window

```
Enter a string:
Hatred is preferable to the friendship of fools
Alternate characters in the string are:
Hte speeal otefinsi ffos
```

Program 6 | Input a string and display the alternate characters of the string in the reverse order

Line | **PE 7-6.c**

```
//Printing alternate characters of a string in the reverse order
#include<stdio.h>
#include<string.h>
main()
{
    char str[200], altchars[200];
    int i=0, length, j=0;
    printf("Enter a string:\n");
    gets(str);
    length=strlen(str);
    i=length-1;
    while(i>=0)
    {
        altchars[j]=str[i];
        i=i-2;
```

Output window

```
Enter a string:
Harmony in character gains goodwill even from strangers
Alternate characters of the string in reverse order are:
senrsmr eelido na ecrh iyorH
```

(Contd...)

Line | **PE 7-6.c** | **Output window**

```
        j=j+1;
    }
    altchars[j]=‘\0’;
    printf(“Alternate characters of the string in reverse order are:\n”);
    puts(altchars);
}
```

Program 7 | Input a multi-word string and find out the number of words in the string

Line | **PE 7-7.c**

```
//Number of words in a string
#include<stdio.h>
#include<string.h>
main()
{
char str[200];
int i=0, count=0;
printf(“Enter a string:\n”);
gets(str);
while(str[i]!=‘\0’)
{
    if(str[i]==‘ ’)
        count++;
    i++;
}
printf(“Number of words in the string are %d\n”,count+1);
}
```

Output window

```
Enter a string:
A man should be educated enough to know that education alone is not enough
Number of words in the string are 14
```

Program 8 | Input a string and check whether the given string is a palindrome or not

Line | **PE 7-8.c**

```
//To check whether a given string is a palindrome or not
#include<stdio.h>
#include<string.h>
main()
{
    char str[200], rev[200];
    printf(“Enter a string:\t”);
    gets(str);
    strcpy(rev,str);
    rev=strrev(str);
    if(strcmp(str,rev)==0)
        printf(“The given string is a palindrome”);
    else
        printf(“The given string is not a palindrome”);
}
```

Output window

```
Enter a string: NITIN
The given string is a palindrome
```

Output window (second execution)

```
Enter a string: Hello
The given string is not a palindrome
```

Program 9 | Input a string and count the number of occurrences of a particular word in the string

Line **PE 7-9.c**

```
//Counting the number of occurrences of a particular word in a string
#include<stdio.h>
#include<string.h>
main()
{
    char str[200], word[20], temp[20];
    int i=0, j=0, count=0;
    printf(“Enter a string:\n”);
    gets(str);
    printf(“Enter the word:\t”);
    gets(word);
    while(str[i]!=’\0’)
    {
        while(str[i]!=’ ‘ && str[i]!=’\0’)
        {
            temp[j]=str[i];
            j++; i++;
        }
        temp[j]=’\0’;
        if(str[i]!=’\0’)
        {
            i++; j=0;
        }
        if(strcmp(temp,word)==0)
            count++;
    }
  if(count==0)
    printf(“The word \”%s\” does not exist in the string”, word);
  else
   printf(“The word \”%s\“ exists %d times in the string”, word, count);
}
```

Output window

```
Enter a string:
Fools are not aware of their own faults although they are known to all
Enter the word:     are
The word “are” exists 2 times in the string
```

Output window (second execution)

```
Enter a string:
You must not expect everything exactly to your taste
Enter the word:     are
The word “are” does not exist in the string
```

Program 10 | Input a string and count the number of occurrences of a particular string in the string

Line **PE 7-10.c**

```
//Counting the occurrences of a particular string in the string
#include<stdio.h>
#include<string.h>
main()
{
    char str1[200], str2[200], temp[20];
    int i=0, j=0,k=0, count=0;
    printf(“Enter a string:\n”);
    gets(str1);
    printf(“Enter the string to be searched:\t”);
    gets(str2);
    while(str1[i]!=’\0’)
    {
        k=0;
        while(str2[k]!=’\0’)
```

Output window

```
Enter a string:
Try not to become a man of success but rather to be a man of value
Enter the string to be searched:     a man of
String “a man of” exists 2 times
```

Output window (second execution)

```
Enter a string:
Try not to become a man of success but rather to be a man of value
Enter the string to be searched:     civil society
String “civil society” doesnot exist in the given string
```

(Contd...)

Line | **PE 7-10.c** | **Output window**

```c
            {
                if(str1[j]==str2[k])
                    j++, k++;
                else
                {
                    j=i+1;
                    break;
                }
                if(str2[k]==0)
                    count++;
            }
            if(str2[k]==0)
                i=j;
            else
                i++;
        }
        if(count==0)
        printf("String \"%s\" doesnot exist in the given string\n", str2);
        else
        printf("String \"%s\" exists %d times\n", str2, count);
}
```

Program 11 | A class consists of a number of students whose names are entered in a random order. Display the names of all the students that start with a particular character

Line | **PE 7-11.c** | **Output window**

```c
//Displaying the names of students starting with a particular character
 #include<stdio.h>
#include<string.h>
main()
{
    char names[40][30], firstchar;
    int num, i;
    printf("How many students are there in the class:\t");
    scanf("%d",&num);
    printf("Enter the names of students:\n");
    for(i=0;i<num;i++)
    gets(names[i]);
    printf("\nEnter the first character of student's name:\t");
    scanf("%c",&firstchar);
    printf("Students whose names starts with %c are:\n",firstchar);
    for(i=0;i<num;i++)
        if(names[i][0]==firstchar)
            puts(names[i]);
}
```

```
How many students are there in the class:    10
Enter the names of students:
Abhay Singh
Neha Singla
Jasraj Singh
Aditya Raina
Tarun Kumar
Amol Sood
Joydeep Chandra
Tushar Sharma
Rajini Bansal
Sam

Enter the first character of student's name:    A
Students whose names starts with A are:
Abhay Singh
Aditya Raina
Amol Sood
```

Program 12 | A class consists of a number of students whose names are entered in a random order. Display the names in a sorted order

PE 7-12.c

```
//Displaying the names of students in a sorted order
 #include<stdio.h>
#include<string.h>
main()
{
    char names[40][30], current[30];
    int num, i,j;
    printf(“How many students are there in the class:\t”);
    scanf(“%d”,&num);
    printf(“Enter the names of students:\n”);
    for(i=0;i<num;i++)
    gets(names[i]);
    for(i=1;i<num;i++)    //← Insertion Sort
    if(strcmp(names[i],names[i-1])<0) //←equivalent to if(names[i]<names[i-1])
    {
        strcpy(current,names[i]); //←equivalent to current=names[i]
        for(j=i-1;j>=0;j--)
        {

                strcpy(names[j+1],names[j]); //←eq. to names[j+1]=names[j]
                if(j==0||(strcmp(names[j-1],current)<0))
                    break;
        }
    strcpy(names[j],current);
    }
    printf(“\nAfter sorting, names of students are:\n”);
    for(i=0;i<num;i++)          //←Print sorted list
        puts(names[i]);
}
```

Output window

```
How many students are there in the class:     10
Enter the names of students:
Abhay Singh
Neha Singla
Jasraj Singh
Aditya Raina
Tarun Kumar
Amol Sood
Joydeep Chandra
Tushar Sharma
Rajini Bansal
Sam

After sorting, names of students are:
Abhay Singh
Aditya Raina
Amol Sood
Jasraj Singh
Joydeep Chandra
Neha Singla
Rajini Bansal
Sam
Tarun Kumar
Tushar Sharma
```

Program 13 | Chandigarh Housing Board has released a list of successful applicants in the preliminary draw of lots. Find out whether a given name is in the list or not

PE 7-13.c

```
//Searching a name in the list
 #include<stdio.h>
#include<string.h>
main()
{
char applicants[40][30], name[30];
int num, i,found=0;
printf(“The list of draw is of how many applicants?\t”);
```

Output window

```
The list of draw is of how many applicants?    10
Enter the names:
Abhay Singh
Neha Singla
Jasraj Singh
Aditya Raina
Tarun Kumar
Sam
```

(*Contd...*)

Line | **PE 7-13.c**

```
scanf("%d",&num);
printf("Enter the names:\n");
for(i=0;i<num;i++)
gets(applicants[i]);
printf("\nEnter name to be searched:\t");
gets(name)
for(i=1;i<num;i++)     //← Linear search
     if(strcmp(applicants[i],name)==0)
          found=1;
if(found==1)
printf("Name \"%s\" appears in the list of successful applicants");
else
printf("Name \"%s\" does not appear in the list of successful applicants");
}
```

Output window

```
Amol Sood
Joydeep Chandra
Tushar Sharma
Rajini Bansal

Enter the name to be searched:     Sam
Name "Sam" appears in the list of successful applicants
```

Program 14 | Count the number of sentences, words and characters in a given paragraph

Line | **PE 7-14.c**

```
//Counting the number of sentences, words and characters in a given paragraph
 #include<stdio.h>
main()
{
char paragraph[1000];
int i=0, sentence=0, word=0, chs=0;
printf("Enter the text:\n");
scanf("%[^\n]", paragraph);
while(paragraph[i]!='\0')
{
     switch(paragraph[i])
     {
          case '!':
          case '.':
          case '?':
               sentence++;
               chs++;
               break;
          case ' ':
          case '\t':
               chs++;
               word++;
               break;
          default:
               chs++;
     }
     i++;
 }
printf("\nNumber of sentences in paragraph are %d\n", sentence);
printf("Number of words in paragraph are %d\n", word+1);
printf("Number of characters in paragraph are %d\n", chs);
}
```

Output window

```
Enter the text:
Hello! How are you? Where were you? I have been looking
for all these days.

Number of sentences in paragraph are 4
Number of words in paragraph are 15
Number of characters in paragraph are 75
```

Program 15 | Write a C program to read an English sentence and replace lowercase characters by uppercase and vice-versa. Output the given sentence as well as the case converted sentence on two different lines.

PE 7-15.c

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>
Void main()
{
    char sentence[100];
    int count, ch, i;
    clrscr();
    printf(“Enter a sentence\n”);
    for(i=0; (sentence[i] = getchar())!=’\n’; i++)
    {  ; }
    sentence[i]=’\0’;
    count = i; /*shows the number of chars accepted in a sentence*/
    printf(“The given sentence is   : %s”,sentence);
    printf(“\nCase changed sentence is: ”);
    for(i=0; i < count; i++)
    {
        ch = islower(sentence[i]) ? toupper(sentence[i]) : tolower(sentence[i]);
        putchar(ch);
    }
} /*End of main()*/
```

Output window

```
Enter a sentence
Mera Bharat Mahan

The given sentence is: Mera Bharat Mahan
Case changed sentence is: mERA bHARAT mAHAN
```

Program 16 | Write a C program to interchange the main diagonal elements with the secondary diagonal elements.

PE 7-16.c

```
void main ()
{
    int i,j,m,n,a;
    static int ma[10][10];
    printf (“Enter the order of the matrix \n”);
    scanf (“%d%d”,&m,&n);
    if (m==n)
    {
        printf (“Enter the co-efficients of the matrix\n”);
        for (i=0;i<m;++i)
        {    for (j=0;j<n;++j)
             {     scanf (“%d%d”,&ma[i][j]);     }
        }
        printf (“The given matrix is \n”);
        for (i=0;i<m;++i)
        {    for (j=0;j<n;++j)
             {    printf (“ %d”,ma[i][j]);     }
             printf (“\n”);
        }
        for (i=0;i<m;++i)
```

Output window

```
Enter the order of the matrix
3            3
Enter the co-efficients of the matrix
1  2  3
4  5  6
7  8  9
The given matrix is
1 2 3
 4 5 6
 7 8 9
The matrix after changing the
main diagonal & secondary diagonal
 3 2 1
 4 5 6
 9 8 7
```

(Contd...)

Line | PE 7-16.c | Output window

```
        {
            a = ma[i][i];
            ma[i][i]   = ma[i][m-i-1];
            ma[i][m-i-1] = a;
        }
        printf (“The matrix after changing the \n”);
        printf (“main diagonal & secondary diagonal\n”);
    for (i=0;i<m;++i)
    {
        for (j=0;j<n;++j)
        {    printf (“ %d”,ma[i][j]);   }
        printf (“\n”);
    }
}
else
printf (“The given order is not square matrix\n”);
}
}
```

Program 17 | Write a C program to find the sum of the rows and columns of a matrix.

Line | PE 7-17.c | Output window

```
void main ()
{
    int i,j,m,n,sum=0;
    static int m1[10][10];
    printf (“Enter the order of the matrix\n”);
    scanf (“%d %d”, &m,&n);
    printf (“Enter the co-efficients of the matrix\n”);
    for (i=0;i<m;++i)
    {  for (j=0;j<n;++j)
        {  scanf (“%d”,&m1[i][j]);    }
    }
    for (i=0;i<m;++i)
    {   for (j=0;j<n;++j)
        {    sum = sum + m1[i][j] ;   }
        printf (“   Sum of the %d row is = %d\n”,i,sum);
        sum = 0;
    }
    sum=0;
    for (j=0;j<n;++j)
        {    for (i=0;i<m;++i)
            {      sum = sum+m1[i][j];     }
            printf (“Sum of the %d column is = %d\n”, j,sum);
            sum = 0;
        }
}
```

Output window

```
Enter the order of the matrix
3                3

Enter the co-efficients of the matrix
1  2  3
4  5  6
7  8  9
 Sum of the 0 row is = 6
 Sum of the 1 row is = 15
 Sum of the 2 row is = 24
 Sum of the 0 column is = 12
 Sum of the 1 column is = 15
 Sum of the 2 column is = 18
```

Program 18 | Write a C program to display the following Output:

Line **PE 7-18.c**

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

#include<stdio.h>
#include<conio.h>
void main()
{
   int i,j,k;
   clrscr();
   for(i=1;i<=5;i++)
       {
           for(j=5;j>i;j--)
           printf(“ “);
           for(k=1;k<=i;k++)
           printf(“*”);
           printf(“\n”);
       }
   getch();
}
```

Output window

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

Program 19 | Write a C program to accept a set of numbers as string and separate it as integer tokens.

Line **PE 7-19.c**

```
#include <stdio.h>
#include<string.h>
#include<stdlib.h>
void main()
{
    char str[80] ;
    const char s[2] = “ “;
    int a[10],i=0;
    char *token;
    scanf(“%[^\n]s”,str);
    token = strtok(str, s);
    while(token)
    {
        a[i]=atoi(token);
        token = strtok(NULL,” “);
        printf(“%d\n”,a[i]);
        i++;
    }
}
```

Output window

```
10 11 1 4 5

10
11
1
4
5
```

Program 20 | A line of text is given as input, and you need to display in the reverse order.

Line **PE 7-20.c**

```
Note: Reverse the words
Sample Input: india is great
Sample Output: great is india

#include<stdio.h>
#include<string.h>
void main()
{  int l,i,j;
   char string[100];
   printf("enter string\n");
   gets(string);
   l=strlen(string);
   i=l;
   while(i>=-1)
        { if(i<0 || string[i]==' ')
             { for(j=i+1;;j++)
                    { if(string[j]=='\0' || string[j]==' ')
                           { break; }
                                  printf("%c",string[j]);
                    }
               printf(" ");
             }
          i--;
        }
}
```

Output window

```
enter string
india is great
great is india
```

Program 21 | Write a program to print first maximum, first minimum, second maximum, etc., from an array of elements.

Line **PE 7-21.c**

```

Sample input and output is given as follows:
Input
5 3 6 8 1
Output
8 1 6 3 5
Input
67 34 12 6 32 2 7
67 2 34 6 32 7 12

#include <stdio.h>
#include<stdlib.h>
int main()
{
     char str[80];
     int a[10],t,i=0,j,count;
     gets(str);
     int *p1,*p2;
```

Output window

```
67 34 12 6 32 2 7
67 2 34 6 32 7 12
```

(Contd...)

Line	PE 7-21.c	Output window

```
        const char s[2] = " ";
        char *token;
        token = strtok(str, s);
    while( token != NULL )
        {
            a[i]=atoi(token);
            token = strtok(NULL, s);
            i++;
            count=i;
        }
        for(i=0;i<count;i++)
        {
            for(j=0;j<count;j++)
            {   if(a[i]>a[j])
                {
                t=a[i];
                a[i]=a[j];
                a[j]=t;   }
            }  }
        p1=&a[0];
        p2=&a[count-1];
for(i=0;i<count/2;i++)
        {
            printf("%d ",*p1);
            printf("%d ",*p2);
            p1++;
            p2--;
            if(p1==p2)
                {
            printf("%d ",*p1);
                break;
        }
        } return 0; }
```

Test Yourself

1. Fill in the blanks in each of the following:
    a. A string literal constant of zero length is called ____________.
    b. Every string literal in C is terminated by ____________.
    c. The amount of memory taken by an empty string literal is ____________.
    d. The type of string literal is ____________.
    e. A string literal constant is always enclosed within ____________.
    f. Adjacent string literals are ____________.
    g. The scanf function uses ____________ format specification to read a string from the user.
    h. ____________ string library function is used to compare two strings without case sensitivity.
    i. The ____________ character is used to invert the search set.
    j. ____________ function is used to read a character from the keyboard.
    k. Inputs to function main are given by making use of special arguments known as ____________ .

2. State whether each of the following is true or false. If false, explain why.
    a. The length of a string literal constant is equal to the number of characters present in it.
    b. The length of an empty string literal constant is one.
    c. The amount of the memory space required for storing a string literal constant is not fixed and depends upon the number of characters present in the string literal.
    d. The number of bytes required to store a string literal is equal to the number of characters present in it.
    e. It is not mandatory to use ampersand (i.e. address-of operator) with string variable names while reading string using the scanf function.
    f. Unlike the scanf function, the gets function reads the entire line of text until a new line character is encountered and does not stop upon encountering any other white-space character.
    g. The printf function can print a string on the screen without using any format specifier.
    h. If the character array to be printed does not have a terminating null character, the output would be the content of the character array followed by some garbage character.
    i. It is not mandatory to have the first argument of the printf function to be of const char* type.
    j. The string library function strrev reverses all the characters of a string including the null character.
    k. It is not possible to initialize a character array with a string literal constant.
    l. A list of strings can be stored by using a two-dimensional character array.

3. Programming exercises:
    a. A certain piece of text is entered. By mistake, at some places two or more spaces are placed between two words. Write a C program that removes these extra spaces between the words.
    b. Without using inbuilt string library functions, write a C program to check whether a given string is a palindrome or not.
    c. Write a C program to find the longest word in a given string. Also print the length of the word.
    d. Write a C program to read a text and omit all occurrences of a particular word in the text.
    e. Write a C program to read a text and omit all occurrences of a particular string in the text.
    f. Write a C program to read a text. Implement the find and replace functionality. The find functionality will find a given substring in the text, and the replace function will replace the found substring with a given string.

g. Write a C program to display the given series of numbers as given below.
1
2 1
3 2 1
4 3 2 1

4 3 2 1
3 2 1
2 1
1

h. Write a C program to print Pascal triangle.

i. Write a C program to generate anagrams.

j. Write a C program to convert string to integer without using atoi function?

k. Write a C program to find the position of the first occurrence of the substring.
Sample input and output is given as follows:
Input
kilogram
gram
Output
4

Input
mirchihot
hot
Output
6

Input
Mirchi
Red
Output
–1

l. Write a program for merging two sorted arrays by eliminating the repeated elements.
Example:
Input:
2 6 10 20
3 6 15 21 34 36
Output:
2 3 6 10 15 20 21 34 36

m. Write a C program to find the missing number in the given series.
Sample input–output is: Input :12345689
Output: Missing number is 7.

n. Find the reverse of a four-digit number as given in the sample input and output.
Input: 1234
Output: 2143

o. Can you print a staircase as shown in the example?
Input
You are given an integer *N* depicting the height of the staircase.
Output
Print a staircase of height *N* that consists of # symbols and spaces. For example, for *N* = 6, here is a staircase of that height:

```
     #
    ##
   ###
  ####
 #####
######
```

Note: The last line has 0 spaces before it.

p. Write a C program to delete vowels from a string.

q. Write a C program to replace vowels by consonants and consonants by vowels.

r. A line of text is given as input. You need to find the position of the substring, if the substring does not exist display −1.
Input: Sun rises in the east
In
Output: 11
Input: Sun rises in the east
At
Output:-1

s. Write a program to print the first letter of the word in the given string.
Note: If the first letter is a lower case, then convert it to uppercase.

This page is intentionally left blank

PART – IV

FUNCTIONS

This page is intentionally left blank

8

FUNCTIONS

Learning Objectives

In this chapter, you will learn about:

- Functions
- Advantages of using functions
- Classification of functions as user-defined functions and library functions
- User-defined functions
- How to declare, define and call functions
- Way of increasing flexibility of functions
- Different ways of supplying inputs to a function
- `return` statement
- How to provide default inputs to a function
- Recursion and its use to solve problems
- Classification of recursion
- How recursion works
- Tower of Hanoi problem
- Function type and pointers to functions
- Array of function pointers
- Passing arrays and functions to functions
- Commonly used library functions
- Variable argument functions

8.1 Introduction

In the previous chapters, you have seen how to declare identifiers (Chapter 3), how to write expressions (Chapter 4) and how to write statements (Chapter 5). In this chapter, I will tell you how to group these components in a function so that these components can be reused in a program. I will describe the advantages of using functions, how to declare, define and call them. You will be familiarized with the methods of increasing flexibility of a function and different ways of passing inputs to a function. Finally, we will have a discussion about the advanced topics like pointers to functions, arrays of function pointers and passing functions to a function.

8.2 Functions

Most of the computer programs that solve real-world problems are much bigger and complex than the programs presented in the first few chapters. The existing software engineering practices used to develop such complicated programs work on the following principles:

1. **Top-down design, modularization, stepwise refinement and bottom-up development:** According to this principle, a complex problem should be modularized (i.e. divided) into sub-problems that are simpler, manageable and easier to solve as compared to the original problem. If the divided sub-problems are still complex and cannot be easily solved, they are further divided into sub-problems. Each level of division provides a refinement and simplicity to the problem. This process of modularization is carried out till the sub-problems are simple enough and can be easily solved. The solutions for these simple problems are then developed and merged to provide a solution for the overall complex problem. This approach of problem solving is also known as **'divide-and-conquer strategy.'** This strategy is practically followed in real life whereby a senior officer responsible for the execution of a work divides the work among his subordinates. The subordinate officers may further divide the assigned work among their subordinates, get the work done and report back to their senior officer. This hierarchical division of work is shown in Figure 8.1.

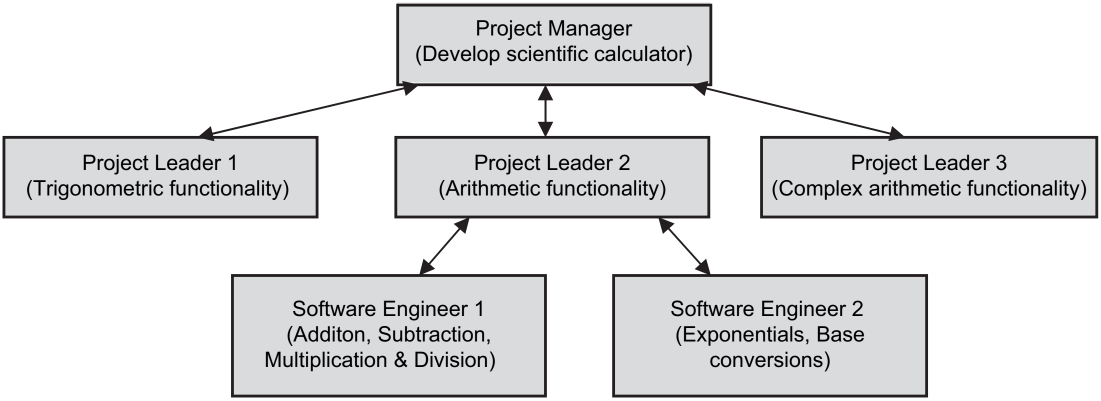


Figure 8.1 | Hierarchical division of work

Thus, in this approach of solution development, a solution to the given problem is thought of at an abstract level. This abstract solution is divided into modules, and each level of division refines the solution by adding details to the divided modules. The process of division is carried out till the divided modules are well defined and simple enough to be generated (i.e. coded). The functionality of each module is kept in a separate function. These functions are relatively independent of each other and interact with each other to provide a solution to the overall problem.

2. **'Don't reinvent the wheel.'** Another important software engineering principle states that **'Don't reinvent the wheel.'** This means that the functionality that has already been developed should be reused instead of being developed again. Functions help a lot in realizing this principle. The commonly required functionality is developed and kept in standard libraries for the use in the form of library functions. In the previous chapters, we have used the input and output functionality by using scanf and printf library functions. The C standard library provides a rich set of functionality for performing the common mathematical calculations, string and character manipulations, input/output and other useful operations.

The above two software engineering principles give a hint about the importance and the need of functions. Several other advantages of modularizing a program into functions include:

1. Reduction in code redundancy
2. Enabling code reuse
3. Better readability
4. Information hiding
5. Improved debugging and testing
6. Improved maintainability

As already described in Chapter 3, a C program is made up of functions. Functions interact with each other to accomplish a particular task. They are classified according to the following criteria:

1. Based upon who develops the function
2. Based upon the number of arguments a function accepts

8.3 Classification of Functions

8.3.1 Based Upon who Develops the Function

Based upon who develops the function, functions are classified as:

1. User-defined functions
2. Library functions

8.4 User-defined Functions

User-defined functions are the functions that are defined (i.e. developed) by the user at the time of writing a program. The user develops the functionality by writing the body of the function. These functions are sometimes referred to as **programmer-defined functions.** Program 8-1 illustrates the use of user-defined functions add, sub and println.

Line | **Prog 8-1.c**

```
 1  //User defined functions
 2  #include<stdio.h>
 3  //Function declarations or function prototypes
 4  println();
 5  int add(int, int);
 6  int sub(int x, int y);
 7  //main function, the master function
 8  main()
 9  {
10      int a,b,sum, diff;
11      printf("Enter the values\t");
12      scanf("%d %d",&a, &b);
13  //Function invocations
14  //Asking the workers to do work
15      sum=add(a,b);
16      diff=sub(a,b);
17      println();
18  //Master presents the results returned by workers
19      printf("Result of addition is %d\n",sum);
20      printf("Result of subtraction is %d\n",diff);
21  }
22  //Function definitions
23  println()
24  {
25  printf("------------------------\n");
26  }
27  int add(int a, int b)
28  {
29  return a+b;
30  }
31  int sub(int a, int b)
32  {
33  return a-b;
34  }
```

Output window

```
Enter the values     4 3
------------------------
Result of addition is 7
Result of subtraction is 1
```

Remarks:

- println, add and sub are user-defined functions
- In line numbers 4, 5 and 6, user-defined functions are declared
- In line numbers 23 to 34, they are defined
- In line numbers 15, 16 and 17, they are called
- Line numbers 23, 27 and 31 consist of headers of the functions println, add and sub
- The variables declared in the function headers or function declarations are known as parameters
- In line numbers 6, x and y are the parameter names
- In line numbers 27 and 31, a and b are the parameter names
- The parameters declared inside the function headers are similar to the variables declared inside the body of the function
- **main is also a user-defined function**

Program 8-1 | A program that illustrates the use of user-defined functions

As you have seen in the code snippet in Program 8-1, there are three aspects of working with user-defined functions:

1. Function declaration, also known as function prototype
2. Function definition
3. Function use, also known as function call or function invocation

8.4.1 Function Declaration

All identifiers (except labels) need to be declared before they are used. As function names are also identifiers, this is true for functions as well. All the functions need to be declared or defined[†] before they are used (i.e. called[‡]). The general form of a **function declaration** is:

```
[return_type] function_name([parameter_list or parameter_type_list]);
```

The important points about the function declaration are as follows:

1. The terms enclosed within the square brackets are optional and might not be present in a function declaration statement. The terms shown in **bold** are the mandatory parts of a function declaration.
2. The function declaration consists of the name of the function along with its return type and parameter list or parameter-type list enclosed within parentheses. Function declaration is also known as **function prototype**. For example, in Program 8-1 the declaration of function add in line number 5 consists of **parameter-type list**, and the declaration of function sub in line number 6 consists of **parameter list**.
3. Function names are identifiers. All syntactic rules discussed in Section 3.5.1 for writing identifier names are applicable for writing the function names as well. The name of a function is also termed as **function designator**.
4. The specification of the return type is optional. If specified, the return type of a function can be any type (e.g. char, int, float, int*, int**, void, etc.) except array type and function type.[§] For example, in Program 8-1 the return type of the function println is not specified and the return type of functions add and sub is int.
5. The syntactic rules for writing a parameter-type list and parameter list in a function declaration are as follows:
    a. The **parameter-type list** is a comma-separated list of parameter types. The parameter type can be any type (e.g. char, int, float, int*, int**, void, etc.) except function type. If only a parameter-type list is mentioned, the function declaration is said to have **abstract parameter declaration**.
    b. A parameter name can optionally follow each parameter type. A parameter name should be a valid variable name. If parameter names follow parameter types in a parameter-type list, it becomes a **parameter list**. If the function declaration consists of a parameter list, it is said to have **complete parameter declaration**. For example, in Program 8-1 (in line number 5) function add has abstract parameter declaration and (in line number 6) function sub has complete parameter declaration.
    c. Using a combination of complete parameter declaration and abstract parameter declaration (i.e. naming some of the parameters and leaving the rest of them unnamed) is also allowed. For example, the following declarations of function add are also allowed:

    ```
    int add(int x, int);
    int add(int, int y);
    ```

    d. No two parameter names appearing in the parameter list can be the same.
    e. The shorthand declaration of parameters in the parameter list is not allowed.
6. Function declaration is a statement, so it must be terminated with a semicolon.

[†] Refer Section 8.4.2 for a description on function definitions.

[‡] Refer Section 8.4.3 for a description on function calls.

[§] Refer Section 8.5.6 for a description on the function type.

7. A function need not be declared, if it is defined before it is called.

The following function declarations are valid:

1. `add();` //←Return type and parameter list are not present
2. `int add(int,int);` //←`int` is the return type and `int, int` is the parameter-type list
3. `int* add(int,float);` //←`int*` is the return type and `int, float` is the parameter-type list
4. `int add(int a, int b);` //←Parameter list contains the names of parameters, i.e. `a` and `b`
5. `int add(int, int b);` //←Combination of abstract and complete parameter declaration

The following function declarations are not valid:

1. `int add(int a, float a);` //←Both the parameter names are the same
2. `int add&sub(int, int);` //←Name of the function is not valid as it contains the special character `&`
3. `int add(int a,b);` //←Shorthand declaration of parameters is not allowed
4. `int add(int a, int b)` //←The declaration is not terminated with a semicolon

Function prototypes (i.e. function declarations) are important and their necessity can be seen from two different perspectives:

1. **User perspective** It tells the user how to use a pre-defined or library function.[¶] It tells the user the number of parameters along with their types that a function expects and its return type. This is necessary and sufficient information for a user to use a function. For example, consider the following function prototype:

   `int add(int,int);`

   It tells the user that function `add` expects two integers and returns the result as an integer. With all this information, the user will be able to use the function `add`. Function prototype does not provide any information about how the functionality is implemented by the function. We have been able to use the `printf` function in the previous chapters because we know its prototype. The prototype of the `printf` function is available in the header file `stdio.h`. We do not know anything about how printing functionality is implemented by the `printf` function.
2. **Compiler perspective** It allows the compiler to perform **type checking**. By type checking the compiler ensures that while making a function call, the user provides the correct number and the correct type of arguments. If the number of arguments is not the same as the number of parameters or if their types are not compatible with the types of parameters provided in the function declaration, the compiler issues an error message.

> *i* If some of the parameters are provided with default arguments,[††] the number of arguments in a function call can be lesser than the number of parameters.

[¶] Refer Section 8.6 for a description on library functions.

[††] Refer Section 8.5.4.3 for a description on default arguments.

8.4.2 Function Definition

Function definition, also known as **function implementation**, means composing a function. Every function definition consists of two parts:

1. Header of the function
2. Body of the function

Thus, defining a function involves composing its header and the body.

8.4.2.1 Header of a Function

The general form of **header of a function** is:

[return_type] **function_name(**[parameter_list]**)**

The important points about the function header are as follows:

1. The terms enclosed within the square brackets are optional and might not be present in a function header. The terms shown in bold are the mandatory part of a function header.
2. Unlike function declaration, the header of a function can only have complete parameter declaration. It cannot have abstract parameter declaration or a combination of abstract and complete parameter declaration. The variables declared in the parameter list will receive the data sent by the calling function.[‡‡] They serve as the inputs to the function.
3. No two parameter names appearing in the parameter list can be the same.
4. The shorthand declaration of parameters in the parameter list is not allowed.
5. The return type and the number and the types of parameters in the function header should exactly match the corresponding return type and the number and types of parameters in the function declaration, if it is present. For example, look at the function declarations in line numbers 4, 5 and 6 and function headers in line numbers 23, 27 and 31 in Program 8-1.
6. It is not mandatory to have the same names for the parameters in the function declaration and function definition. For example, in Program 8-1, the names of parameters in the declaration of function sub in line number 6 are x and y while the names of parameters in the header of the function sub in line number 31 are a and b.
7. The header of a function is not terminated with a semicolon.

8.4.2.2 Body of a Function

The **body of a function** consists of a set of statements enclosed within braces. The body of a function can have non-executable statements and executable statements. The non-executable statements can only come before the executable statements. The non-executable statements declare the local variables in the function and the executable statements determine its functionality, i.e. what the function does. A function can optionally have special executable statement known as the return statement.[§§] The return statement is used to return the result of the computations done in the called function and/or to return the program control back to the calling function.

[‡‡] Refer Section 8.4.3 for a description on calling functions and called functions.

[§§] Refer Section 8.4.3.5 for a description on the return statement.

8.4.3 Function Invocation/Call/Use

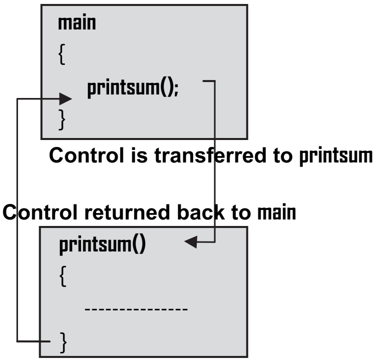
The call to a function can be well described along with the discussion on the classification of functions. Depending upon their inputs (i.e. parameters) and outputs, functions are classified as:

1. Functions with no input–output
2. Functions with inputs and no output
3. Function with inputs and one output
4. Function with inputs and outputs

8.4.3.1 Function with No Input–Output

A **function with no input–output** does not accept any input and does not return any result. Since no input is to be given to the function, the parameter list of such functions is empty. Even if the parameter list is empty, the function header must have the empty set of parentheses or with the keyword void.¶¶ These functions have limited functionality and are not flexible (i.e. they cannot be used in a variety of circumstances). Due to their limited functionality they have limited utility too. Consider the snippet in Program 8-2.

	Trace Col. 2	Prog 8-2.c		Output window

1		//Function with no input-output	 main
			{
			printsum();
			}
			Control is transferred to printsum
			Control returned back to main
			printsum()
			{
			-------------- 	
			}
				Sum of 2 and 3 is 5
				Warnings (2):
				• Function should return a value in function main
				• Function should return a value in function printsum
				Remarks:
				• Ignore the warnings for the time being
				• printsum is a function with no input–output
				• The order of execution of the statements is depicted in the trace column (i.e. column 2)

2		#include<stdio.h>		
3		//Function declaration		
4		printsum();		
5		//main function, the master function		
6	1	main()		
7		{		
8	2	printsum();		
9	6	}		
10		//Definition of function printsum		
11	3	printsum()		
12		{		
13	4	printf("Sum of 2 and 3 is %d",2+3);		
14	5	}		

Program 8-2 | A program that uses a function with no input–output

The important points about functions with no input–output are as follows:

1. In Program 8-2, the function printsum has no input and does not return any result.
2. The function printsum has been invoked, i.e. called in line number 8. A function with no inputs can be **called** by writing a **function designator** (i.e. name of the function) followed by a **function call operator,** i.e. ( ). The function designator followed by the function call operator is known as a **function call**.

¶¶ Refer Section 8.4.3.2 for a description on void functions.

3. The function that calls a function (i.e. which contains a function call) is known as a **calling function**, and the function that has been called is known as a **called function**. In the given code, `main`✍ is the calling function and `printsum` is the called function.
4. A function call terminated with a semicolon is known as a **function call statement**.
5. After the execution of the function call statement, the program control is transferred to the called function. The execution of the calling function is suspended and the called function starts execution. For example, in Program 8-2, after the execution of the function call statement in line number 8, the program control transfers to line number 11. The order of execution of statements in a program can be checked by tracing✍ the program. The program trace is depicted in column 2. Note the position of trace arrows 2 and 3 in Program 8-2.
6. After the execution of the called function (with no output) is complete, the program control returns to the calling function, and the calling function resumes its execution. In Program 8-2, this is depicted by trace steps 5 and 6 in column 2.

> ✍ 1. The **execution of C program** always begins with the function `main`. Function `main` need not be explicitly called.
> 2. Tracing is a debugging technique in which the statements of a program are executed one by one. Non-executable statements are not executed. Hence, during the tracing, the program control does not stop at non-executable statements. Thus, for non-executable statements trace arrows are not shown. The shortcut key for tracing in Borland TC 3.0 and 4.5 is F7. The shortcut key for tracing in MS-Visual C++ 6.0 is F11. Keep on pressing these keys to trace the program.

8.4.3.2 `void` Functions

Program 8-2 on compilation gives a warning message 'Function should return a value.' We have been ignoring this warning since Chapter 3 but now it is the time to know the reason behind this warning and how to remove it.

Every function in C language is supposed to return an integer value. If the return type of a function is not specified, it is assumed to be `int` by default. Thus, in Program 8-2, the return type of the functions `main` and `printsum` is assumed to be `int`. As no `return` statement is used within the body of these functions to return the expected integer value, the compiler gives the warning message 'Function should return a value.'

Removal of warning message

If a function does not return any value, then the return type of the function should be specified as `void` (means nothing). Functions whose return type is `void` are known as **`void` functions.** Reconsider the code snippet mentioned in Program 8-2 with `void` mentioned as the return type of the functions `main` and `printsum`. The modified form of the code listed in Program 8-2 is mentioned in Program 8-3.

Line	Prog 8-3.c	Output window
1	//Function with no input-output	Sum of 2 and 3 is 5
2	#include<stdio.h>	**Remarks:**
3	//Function declaration	• As the functions printsum and main do not return any value,
4	void printsum();	void is specified as their return type
5	//main function, the master function	• The program now on compilation does not give any warn-
6	void main()	ing message
7	{	• **Some compilers (e.g. Borland TC 4.5) do not allow void**
8	printsum();	**to be specified as return type of the function main. They**
9	}	**enforce the return type of the function main to be int**
10	//Function definition	**What to do?**
11	void printsum()	• If Borland TC 4.5 is used, either leave the return type of
12	{	function main unspecified or specify it as int and place
13	printf("Sum of 2 and 3 is %d", 2+3);	return 0; as the last statement of function main. 0 is an arbi-
14	}	trary value. Any integer value can be used instead of 0

Program 8-3 | A program that uses a void function

The important points about void functions are as follows:

1. A void function does not return any value. Either no return statement should be present inside the body of a void function or if it is present, it should be of the form return;. The return statement of the form[†††] return expression; cannot be used inside the body of a void function. When a return statement of the form return; is placed inside the body of a void function, its execution terminates the execution of the void function and returns the program control back to the calling function. The code snippet in Program 8-4 illustrates this fact.

Line	Trace	Prog 8-4.c	Output window
1		//Return statement inside void function	This is a void function
2		#include<stdio.h>	This is a statement before return statement
3		//Function declaration	**Remarks:**
4		void printsum();	• After the function call in line num-
5		//Function definitions	ber 8 gets executed, the program
6	1	void main()	control transfers to line number 10
7		{	• This is depicted by trace arrows 2
8	2	printsum();	and 3
9	7	}	• Execution of the function main is
10	3	void printsum()	suspended and the printsum function
11		{	starts execution
12	4	printf("This is a void function\n");	• After the execution of the return
13	5	printf("This is a statement before return statement\n");	statement in line number 14, the
14	6	return;	program control returns back to the
			main function

(*Contd...*)

[†††] Refer Section 8.4.3.5 for a description on various forms of return statement.

15
16
17
		printf(“This is a statement after return statement\n”);
		printf(“Unreachable code\n”);
		}
			• This is depicted by trace arrows 6 and 7
			• The execution of the function printsum is terminated and the main function resumes its execution
			• printf statements in line numbers 15 and 16 remain unreachable

Program 8-4 | A program that illustrates the use of return statement inside void function

2. A void function call expression evaluates to void. Hence, such expressions cannot be placed on the right side of an assignment operator. For example, the expression a=printsum() is erroneous if printsum is a void function. The code snippet in Program 8-5 illustrates this fact.

Line	Prog 8-5.c	Output window
1	//void function call expression cannot be assigned to a variable	Compilation error “Not an allowed type in function main” **Remarks:** • The return type of the function printsum is void • An expression of type void cannot be assigned to a variable • Hence, the expression a=printsum() in line number 7 is erroneous
2	#include<stdio.h>	
3	void printsum(void);	
4	void main(void)	
5	{	
6	int a;	
7	a=printsum();	
8	printf(“The value of a is %d”,a);	
9	}	
10	void printsum(void)	
11	{	
12	printf(“Sum of 2 and 3 is %d”,2+3);	
13	}	

Program 8-5 | A program that illustrates the void function call expression, which cannot be assigned to a variable

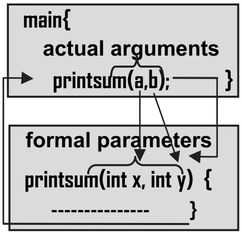
Also, the keyword void is sometimes placed within parentheses in the function header to signify that the function does not have any input. This is depicted in the code snippet in Program 8-5.

8.4.3.3 Function with Inputs and No Output

The function printsum developed in Program 8-2 is rigid. Each invocation of the function printsum prints the sum of 2 and 3. It cannot be used to print the sum of different values. The reason behind this rigidity of the printsum function is the lack of inputs to it. A function can be made flexible by adding inputs to it. The modified flexible form of the code listed in Program 8-2 is mentioned in Program 8-6.

The observable points about the code snippet given in Program 8-6 are as follows:

1. The printsum function developed in Program 8-6 is flexible as compared to the printsum function developed in Program 8-2. It can now be used to print the sum of any two integer values.
2. This flexibility is due to the added inputs. The printsum function now accepts two inputs of the integer type.

	Trace	Prog 8-6.c	(Column 4)	Output window
1		//Function with inputs and no output		Enter values of a & b 4 6 Sum of 4 and 6 is 10 Enter values of a & b again 7 2 Sum of 7 and 2 is 9 **Remarks:** • Function printsum accepts two arguments, i.e. inputs • In line number 11, a and b are known as **actual arguments** • In line number 16, x and y are known as **formal parameters** • The parameters declared in the function header are like other local variables declared inside the body of a function • After execution of the function call in line number 11, the values of a and b are copied into the variables x and y and the control is transferred to the function printsum
2		#include<stdio.h>		
3		//Function declaration		
4		void printsum(int, int);		
5		//Function definitions		
6	1	void main()		
7		{		
8		int a,b;		
9	2	printf(“Enter values of a & b\t”);		
10	3	scanf(“%d %d”,&a,&b);		
11	4	printsum(a,b);		
12	8	printf(“Enter values of a & b again\t”);		
13	9	scanf(“%d %d”,&a,&b);		
14	10	printsum(a,b);		
15	14	}		
16	5, 11	void printsum(int x, int y)		
17		{		
18	6, 12	printf(“Sum of %d and %d is %d\n”,x,y,x+y);		
19	7, 13	}		

Program 8-6 | A program that uses a function with inputs

3. A function with inputs can be called in a similar way as a function without input is called, i.e. by using a function call operator. Inputs to a function are given by providing comma-separated expressions within the parentheses of the function call operator. For example, the printsum function defined in Program 8-6 can be called in the following ways:

```
printsum(2,3);          //←Inputs are constants 2 and 3
printsum(a,b);          //←Inputs are variables a and b
printsum(a+2,b-3);      //←Inputs are expressions a+2 and b-3
```

4. The expressions that appear within the parentheses of a function call are known as **actual arguments**, and the variables declared in the parameter list in the function header are known as **formal parameters**. For example, in Program 8-6, a and b are actual arguments of printsum function and x and y are the formal parameters.
5. The commas separating the actual arguments in a function call are not comma operators. If commas separating arguments in a function call are considered to be comma operators, then no function could have more than one argument. The commas appearing between the arguments in a function call are just separators.
6. The below-mentioned steps are followed when a function with inputs is invoked:
   a. The actual argument expressions are evaluated.
   b. The program control is transferred to the called function and the result of the evaluation of the actual argument expressions are assigned to the formal parameters on one-to-one basis as shown in column 4 of Program 8-6.

   c. The execution of the calling function is suspended and the called function starts the execution.

7. When the execution of the called function (with no output) is complete, the program control returns to the calling function, and the calling function resumes its execution.

Consider the code snippet in Program 8-7, which has a more generalized form of printsum function defined in Program 8-6. The developed printsum function can now print the output in decimal, octal or hexadecimal number system according to the user’s requirement.

Prog 8-7.c

```c
//Further generalization of printsum
#include<stdio.h>
//Function printsum accepts three inputs
void printsum(int, int, char);
//Function definition
void main()
{
    int a,b;
    char base;
    printf(“Enter the values of a & b\t”);
    scanf(“%d %d”,&a,&b);
    printf(“Enter base of output(O, D or H)\t”);
    flushall();
    scanf(“%c”,&base);
    printsum(a,b,base);
}
void printsum(int x, int y, char base)
{
    if(base==’d’||base==’D’)
        printf(“Sum of %d and %d in decimal is %d”,x,y,x+y);
    else if(base==’o’||base==’O’)
        printf(“Sum of %d and %d in octal is %o”,x,y,x+y);
    else if(base==’h’||base==’H’)
        printf(“Sum of %d and %d in hexadecimal is %X”,x,y,x+y);
}
```

Output window

```
Enter the values of a & b     2 10
Enter base of output(O, D or H)     H
Sum of 2 and 10 in hexadecimal is C
```

Remarks:

- The flexibility of the function printsum is increased by providing an additional input, i.e. base
- If flushall function is not used before the use of the scanf function, the scanf function might not prompt the user to enter a character
- The function flushall is used to flush, i.e. empty the streams so that the scanf function prompts the user to enter a character

Program 8-7 | A program that uses a more generalized form of the printsum function developed in Program 8-6

8.4.3.4 Function with Inputs and One Output

The function printsum developed in Programs 8-6 and 8-7 receives inputs but does not return any value, rather it prints the result of the computation. However, the printing of the result of the computation by the called function is not always desired. The result of the computation may be required in the calling function for further processing. The best software engineering practices suggest the following:

1. The developed functions should be kept as general as possible so that they can be used in different situations.
2. Functions should generally be coded without involving any direct I/O operation (i.e. direct use of I/O functions like printf, scanf, getch, etc.). A function should receive inputs in the form of arguments and return the result of computations instead of directly printing it.
3. A function should behave like a 'black box' that receives inputs, and outputs the desired value.

The result of the computations performed inside the called function is returned to the calling function by using the return statement.

8.4.3.5 return Statement

The **return statement** is used to return the result of the computations performed in the called function and/or to transfer the program control back to the calling function. There are two forms of the return statement:

1. return;
2. return expression;

The important points about the return statement are as follows:

1. **First form of the return statement, i.e. return;:**
    a. This form of the return statement is used when a function does not return any value (i.e. inside void functions).
    b. It cannot be used inside a function whose return type is not void.
    c. It terminates the execution of the called function and transfers the program control back to the calling function without returning any value.
2. **Second form of the return statement, i.e. return expression;:**
    a. This form of the return statement returns the function's result along with the program control back to the calling function.
    b. It cannot be placed inside the body of a void function and can only appear inside the body of a function whose return type is not void.
    c. The expression following the keyword return in the return statement is known as the **return expression**.
    d. The return expression can be an arbitrarily complex expression and can even have function calls. For example, in the statement return n*fact(n-1);, the return expression consists of a call to the function fact.
    e. The return expression is evaluated and the result of evaluation of the return expression is returned to the calling function along with the program control.
    f. If the return type of a function and the type of the result of evaluation of a return expression is not the same, the result of evaluation of the return expression is implicitly type casted to the return type of the function, if they are compatible. If they are incompatible, there will be a compilation error. Consider Program 8-8 that makes use of a function to compute the area of a circle.

Line | Prog 8-8.c

```
 1  //Area of a circle
 2  #include<stdio.h>
 3  //Function declaration
 4  circle_area(int);
 5  //Function definitions
 6  void main()
 7  {
 8      int radius;
 9      float area;
10      printf("Enter the radius of circle\t");
11      scanf("%d",&radius);
12      area=circle_area(radius);
13      printf("Area of circle is %f\n",area);
14  }
15  circle_area(int radius)
16  {
17      return 3.1428*radius*radius;
18  }
```

Output window

```
Enter the radius of circle      2
Area of circle is 12.000000
```

Remarks:

- The area of circle that gets printed is 12.000000 instead of the actual value of 12.571200
- This happened because the return type of function circle_area is not mentioned. If the return type of a function is not mentioned, it is assumed to be int by default

Program 8-8 | A program illustrating that the specification of the return type is mandatory if the return type is other than int

The observable points about the code snippet in Program 8-8 are as follows:

i. The area of the circle printed is 12.000000 instead of the actual value 12.571200.
ii. The value of the area actually computed inside the function circle_area is 12.571200 (i.e. a float value) but since the return type of the function is not mentioned, it is assumed to be int (as int is the default return type of a function). The type of result of evaluation of the return expression is not the same as the return type of the function. Thus, as mentioned above, the result of evaluation of the return expression 3.1428*radius*radius, i.e. 12.571200 is type casted (i.e. demoted) to an integer value 12 before being returned. Hence, in the expression area=circle_area(radius), the sub-expression circle_area(radius) evaluates to 12. Since an integer value, i.e. 12 is assigned to a float variable area, it is firstly promoted to 12.000000 and then assigned. This value of area is then printed by the printf function in the next statement, i.e. line number 13.
iii. The precise value of an area can be obtained by specifying the return type of the function circle_area as float. This is shown in the code snippet given in Program 8-9.

Line | Prog 8-9.c

```
 1  //Area of a circle
 2  #include<stdio.h>
 3  //Function declaration
 4  float circle_area(int);
 5  //Function definitions
 6  void main()
 7  {
 8      int radius;
```

Output window

```
Enter the radius of circle      2
Area of circle is 12.571200
```

Remarks:

- float is specified as the return type of the function circle_area
- The value of area that gets printed is 12.571200 instead of 12.000000 (as printed in Program 8-8)

(Contd...)

Line	Prog 8-9.c	Output window

```
 9      float area;
10      printf("Enter the radius of circle\t");
11      scanf("%d",&radius);
12      area=circle_area(radius);
13      printf("Area of circle is %f\n",area);
14  }
15  float circle_area(int radius)
16  {
17      return 3.1428*radius*radius;
18  }
```

Program 8-9 | A program that illustrates the effect of specification of the return type of a function

3. There is no constraint on the number of return statements that can be placed inside the body of a function. Although, a number of return statements can be placed inside the body of a function, only one of them that appears first in the logical flow of control gets executed. With the execution of this return statement, the program control returns to the calling function and the rest of the statements that appear after this return statement remain unreachable.
4. **'A function can return only one value.'** It is not possible to return more than one value by writing multiple return statements as mentioned in point 3 above or by writing return value1, value2, . . . valueN;. In this statement value1, value2, . . . valueN is the return expression, which is evaluated first and then its outcome is returned. The return expression consists of comma operators. The comma operator guarantees left-to-right evaluation and returns the result of the rightmost sub-expression. Hence, the expression value1, value2, . . . valueN evaluates to valueN and this value is returned. Program 8-10 illustrates this fact.

Line	Prog 8-10.c	Output window

```
 1  //Attempt to return more than one value
 2  #include<stdio.h>
 3  //Function declaration
 4  int sum_diff(int,int);
 5  //Function definitions
 6  void main()
 7  {
 8      int a=10, b=2;
 9      printf("Sum is %d\n",sum_diff(a,b));
10      printf("Difference is %d\n",sum_diff(a,b));
11  }
12  int sum_diff(int a,int b)
13  {
14      int sum=a+b;
15      int diff=a-b;
16      return sum,diff;
17  }
```

```
Sum is 8
Difference is 8
```

Remarks:

- In line number 16, an attempt is made to return values of sum and diff
- However, the return statement can return only one value
- The return statement in line number 16 returns the value of diff, i.e. the value of the rightmost return sub-expression

Program 8-10 | A program illustrating that the return statement cannot return more than one value

As we have seen, it is not possible to return more than one value (without making the use of structures) by making use of the return statement. However, it is possible to indirectly return more than one value to the calling function. This indirect method of returning more than one value to the calling function is discussed in the next section.

8.5 Function with Inputs and Outputs

More than one value can be indirectly returned to the calling function by making the use of pointers. In fact, the pointers can also be used to pass arguments to a function. Depending upon whether the values or addresses (i.e. pointers) are passed as arguments to a function, the argument passing methods in C language are classified as:

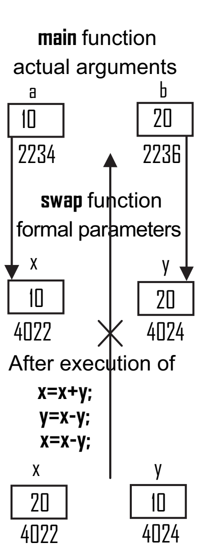
1. Pass by value
2. Pass by address

8.5.1 Passing Arguments by Value

The method of passing arguments by value is also known as **call by value.** In this method, the values of actual arguments are copied to the formal parameters of the function. If the arguments are passed by value, the changes made in the values of formal parameters inside the called function are not reflected back to the calling function. The code snippet listed in Program 8-11 illustrates this concept.

Prog 8-11.c

```
//Use of pass by value in swap function
#include<stdio.h>
//Function declaration
void swap(int,int);
//Function definitions
void main()
{
    int a=10,b=20;
    printf("Before swap values are %d %d\n",a,b);
    swap(a,b);
    printf("After swap values are %d %d\n",a,b);
}
void swap(int x, int y)
{
    x=x+y;
    y=x-y;
    x=x-y;
    printf("In swap function values are %d %d\n",x,y);
}
```



Output window

```
Before swap values are 10 20
In swap function values are 20 10
After swap values are 10 20
```

Remarks:

- On the execution of the function call, i.e. swap(a,b);, the values of actual arguments a and b are copied into the formal parameters x and y
- Formal parameters are allocated at separate memory locations
- A change made in the formal parameters is independent of the actual arguments
- On returning from the called function, the formal parameters are destroyed and the access to the actual arguments gives values that are unchanged

Program 8-11 | A program that illustrates pass by value

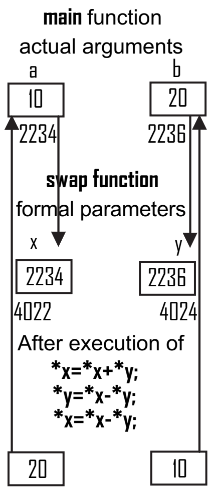
Analogy: The reason why the changes made in the formal parameters in the called function are not reflected back to the calling function can be understood by looking at this analogy. The main function, i.e. the master function wants to get some changes done in a file from its subordinate worker, i.e. the swap function. The main function got the file (i.e. actual arguments) Xeroxed and has handed over the Xeroxed copy of the file (i.e. formal parameters) to the swap function for changes. The swap function has made changes in the Xeroxed copy and has returned the file back to the main function. On getting the control back, the main function is still referring to the original file and finds that no changes have been made in it. The changes have been made in the Xeroxed copy, so how can the main function find changes in the original file?

8.5.2 Passing Arguments by Address/Reference

The method of passing arguments by address or reference is also known as **call by address** or **call by reference**. In this method, the addresses of the actual arguments are passed to the formal parameters of the function. If the arguments are passed by reference, the changes made in the values pointed to by the formal parameters in the called function are reflected back to the calling function. The code snippet listed in Program 8-12 illustrates this concept.

Prog 8-12.c

```
//Use of pass by reference in swap function
#include<stdio.h>
//Function declaration
int swap(int*,int*);
//Function definitions
void main()
{
    int a=10,b=20;
    printf("Before swap values are %d %d\n",a,b);
    swap(&a,&b);
    printf("After swap values are %d %d\n",a,b);
}
int swap(int *x, int *y)
{
  *x=*x+*y;
  *y=*x-*y;
  *x=*x-*y;
  printf("In swap function values are %d %d\n",*x,*y);
}
```



Output window

```
Before swap values are 10 20
In swap function values are 20 10
After swap values are 20 10
```

Remarks:

- Addresses of the actual arguments are passed instead of their values
- Changes made in the called function are actually done in the memory locations of the actual arguments
- On returning from the called function, the formal parameters are destroyed but since the changes were made at the memory locations of the actual arguments, they can still be found there

Program 8-12 | A program that illustrates pass by reference

Analogy: The reason why the changes made in the called function are reflected back to the calling function can be understood by looking at this analogy. The main function, i.e. the master function wants to get some changes done in a file from its subordinate worker, i.e. the swap function. The main function has kept the file (i.e. actual arguments) in a file cabinet (i.e. memory). The main function tells the swap function the changes to be made and the location of the file in the cabinet (i.e. the memory address). The swap function opens up the file cabinet, locates the file, makes changes in it, places it back at the same position in the cabinet and reports to the main function that the work has been done. On getting the control back, the main function opens up the file cabinet, looks at the file and finds the changes made in it.

8.5.3 Returning More Than One Value Indirectly

Consider the code listed in Program 8-10, where we tried to return more than one value by making the use of the return statement and failed. I will now illustrate how to return more than one value to the calling function indirectly by making the use of a call by reference. In the code snippet listed in Program 8-13, the called function indirectly returns more than one value to the calling function.

Prog 8-13.c

```c
//Indirectly returning more than one value
#include<stdio.h>
//Function declaration
void sum_diff(int,int,int*,int*);
//Function definitions
void main()
{
    int a=10, b=2;
    int sum, diff;
    sum_diff(a,b,&sum,&diff);
    printf(“Sum is %d\n”,sum);
    printf(“Difference is %d\n”,diff);
}
void sum_diff(int a,int b, int*sum, int*diff)
{
    *sum=a+b;
    *diff=a-b;
}
```

Output window

```
Sum is 12
Difference is 8
```

Remarks:

- Mixed method of passing arguments is used
- Two arguments, i.e. a and b are passed by value
- Other two arguments, i.e. sum and diff are passed by reference
- The results of the computations made in the called function are stored in the memory locations of the actual arguments (i.e. sum and diff) by making the use of passed addresses
- Actually, sum_diff function does not return any value

Program 8-13 | A program that illustrates the method to indirectly return more than one value by making the use of pass by reference

8.5.4 Passing Arrays to Functions

Like simple variables, arrays can also be passed to functions. There are two ways to pass arrays to functions:

1. Passing individual elements of an array one by one
2. Passing an entire array at a time

Passing individual elements of an array one by one is similar to passing basic variables. The individual elements of an array can be passed either by value or by reference. However, this way of passing an array is not preferred due to the following reasons:

1. If the number of elements in an array is large, passing the entire array will take a large number of function calls, as one element is passed with each function call. As the function calls are time consuming, this method of passing an array to a function will deteriorate the performance of a program.

2. When the individual elements of an array are passed to the function one by one, the complete array will never be available to the called function for processing at a time. The called function will always have a piecemeal array.

The code segments listed in Program 8-14 illustrate the passing of array elements one by one.

Prog 8-14a.c

```
//Individual elements of array passed
//by value
#include<stdio.h>
int sum_array(int,int);
void main()
{
    int arr[10], nele, lc, sum=0;
    printf(“Enter the no. of elements\t”);
    scanf(“%d”,&nele);
    printf(“Enter elements of array\n”);
    for(lc=0;lc<nele;lc++)
        scanf(“%d”,&arr[lc]);
    for(lc=0;lc<nele;lc++)
    {
        sum=sum_array(arr[lc],sum);
    }
    printf(“Sum is %d”,sum);
}
int sum_array(int element, int sum)
{
    return sum+element;
}
```

Prog 8-14b.c

```
//Individual elements of array passed
//by reference
#include<stdio.h>
int sum_array(int*,int);
void main()
{
    int arr[10], nele, lc, sum=0;
    printf(“Enter the no. of elements\t”);
    scanf(“%d”,&nele);
    printf(“Enter elements of array\n”);
    for(lc=0;lc<nele;lc++)
        scanf(“%d”,&arr[lc]);
    for(lc=0;lc<nele;lc++)
    {
        sum=sum_array(&arr[lc],sum);
    }
    printf(“Sum is %d”,sum);
}
int sum_array(int* element, int sum)
{
    return sum+*element;
}
```

Output window

```
Enter the no. of elements 5
Enter elements of array
2
4
5
7
1
Sum is 19
```

Remarks:

- Iteration is used to pass the elements of the array one by one
- Number of iterations required to pass n elements of an array to a function is n

Program 8-14 | A program that illustrates the passing of an array element by element

Passing entire array at a time is a preferred way of passing arrays to functions. The entire array is always passed by reference.

The following sections describe the passing of one-dimensional and multi-dimensional arrays to functions.

8.5.4.1 Passing One-dimensional Arrays to Functions

The syntactic rules to pass one-dimensional arrays to a function are as follows:

1. The actual argument in the function call should only be the name of the array without any subscript.
2. The corresponding formal parameter in the function definition must be of array type or pointer type (i.e. pointer to the first element of the array). If a formal parameter is of array type, it will be implicitly converted to pointer type.
3. The corresponding parameter type in the function declaration should be of array type or pointer type.

The code snippet mentioned in Program 8-15 illustrates the different ways of passing a one-dimensional array to a function.

Line	Prog 8-15a.c (Column 2)	Prog 8-15b.c (Column 3)	Output window
1	//Passing 1-D array	//Passing 1-D array	Enter the no. of elements 5
2	#include<stdio.h>	#include<stdio.h>	Enter elements of array
3	void find_max_min(int[],int);	void find_max_min(int*,int);	2
4	void main()	void main()	4
5	{	{	5
6	int arr[10], nele, lc, sum=0;	int arr[10], nele, lc, sum=0;	7
7	printf(“Enter the no. of elements\t”);	printf(“Enter the no. of elements\t”);	1
8	scanf(“%d”,&nele);	scanf(“%d”,&nele);	Max is 7
9	printf(“Enter elements of array\n”);	printf(“Enter elements of array\n”);	Min is 1
10	for(lc=0;lc<nele;lc++)	for(lc=0;lc<nele;lc++)	**Remarks:** • Passing the entire array at a time is an efficient way of passing a number of values to a function • In column 2, in line number 16, the declared formal parameter arr is of array type • It will be implicitly converted to pointer type • Hence the declaration of arr made in line number 16 in column 2 will be converted to the declaration of arr made in line number 16 in column 3 • The two declarations of arr are equivalent
11	scanf(“%d”,&arr[lc]);	scanf(“%d”,&arr[lc]);	
12	find_max_min(arr, nele);	find_max_min(arr, nele);	
13	printf(“Max is %d\n”,arr[0]);	printf(“Max is %d\n”,arr[0]);	
14	printf(“Min is %d\n”,arr[1]);	printf(“Min is %d\n”,arr[1]);	
15	}	}	
16	void find_max_min(int arr[], int nele)	void find_max_min(int* arr, int nele)	
17	{	{	
18	int lc, max=arr[0], min=arr[0];	int lc, max=arr[0], min=arr[0];	
19	for(lc=1;lc<nele;lc++)	for(lc=1;lc<nele;lc++)	
20	if(arr[lc]>max)	if(arr[lc]>max)	
21	max=arr[lc];	max=arr[lc];	
22	else if(arr[lc]<min)	else if(arr[lc]<min)	
23	min=arr[lc];	min=arr[lc];	
24	arr[0]=max; arr[1]=min;	arr[0]=max; arr[1]=min;	
25	}	}	

Program 8-15 | A program that illustrates the method of passing a one-dimensional array to a function

8.5.4.2 Passing Two-dimensional Arrays to Functions

The syntactic rules to pass two-dimensional arrays to a function are as follows:

1. The actual argument in the function call should be the name of an array.
2. The corresponding formal parameter in the function definition must be of array type or pointer type (i.e. pointer to the first element of the array).
    a. If the formal parameter is of array type, it is mandatory to specify the column specifier. In general, in case of n-D arrays, if the formal parameter is of array type, it is mandatory to specify (n-1) fastest varying specifiers.
    b. If the formal parameter is of pointer type, it must be a pointer to an element of the two-dimensional array (i.e. one-dimensional array having the number of columns same as the number of columns specified for the two-dimensional array). In general, for n-D arrays, if the formal parameter is of pointer type, it must be a pointer to (n-1)-D array having the size specifications same as the (n-1) fastest varying size specifications for the n-D array.

3. The corresponding parameter type in the function declaration should be a matching array type or pointer type.

The code snippet in Program 8-16 illustrates the passing of a two-dimensional array to a function.

Line | Prog 8-16.c

```
 1  //Passing 2-D array
 2  #include<stdio.h>
 3  void largest_ele(int[][10],int*,int*);
 4  void main()
 5  {
 6      int arr[10][10];
 7      int rows, cols, rc, cc;
 8      printf(“Enter no. of rows in array(<10)\t”);
 9      scanf(“%d”,&rows);
10      printf(“Enter no. of cols in array(<10)\t”);
11      scanf(“%d”,&cols);
12      printf(“Enter elements of array:\n”);
13      for(rc=0;rc<rows;rc++)
14          for(cc=0;cc<cols;cc++)
15              scanf(“%d”,&arr[rc][cc]);
16      largest_ele(arr,&rows,&cols);
17      printf(“Largest element is %d\n”,arr[rows][cols]);
18      printf(“Located in row no. %d\n”,rows);
19      printf(“Located in column no. %d\n”,cols);
20  }
21  void largest_ele(int arr[][10],int *rows, int *cols)
22  {
23      int row=0, col=0, rc=0, cc=0, max=arr[0][0];
24      for(rc=0; rc<*rows;rc++)
25          for(cc=0;cc<*cols;cc++)
26              if(arr[rc][cc]>max)
27              {
28                  max=arr[rc][cc];
29                  row=rc; col=cc;
30              }
31      *rows=row; *cols=col;
32  }
```

Output window

```
Enter no. of rows in array(<10)     3
Enter no. of cols in array(<10)      3
Enter elements of array:
8 4 6
7 9 3
2 1  5
Largest element is 9
Located in row no. 1
Located in column no. 1
```

Remarks:

- In line number 21, the declared formal parameter is of array type
- It will be implicitly converted to pointer type
- The equivalent declaration is int(*)[10], i.e. pointer to one-dimensional array of 10 integers
- It is assumed that the row and column number starts with 0

Program 8-16 | A program to illustrate the method of passing of a two-dimensional array to a function

8.5.4.3 Default Arguments

In Section 8.4.3.3, we have seen how functions can be made flexible by adding inputs to them. Each input adds some flexibility to the function and makes the function more general. However, some inputs are the same in majority of the cases and have special values only in rare circumstances. For example, in Program 8-7, the common base input to the function printsum is ‘D’, i.e. decimal number system. In rare circumstances, the user wants the output to be in an octal number system or a hexadecimal number system. These general functions are sometimes unwieldy as the values are to be supplied for each argument.

The C language frees the programmer from this difficulty by providing the concept of **default arguments**. A **default argument** is a value that is an appropriate argument value for a parameter in majority of the cases. Consider the code snippet in Program 8-17 that makes the use of default argument for base input in printsum function discussed in Program 8-7.

Line | **Prog 8-17.c**

```
 1  //Default arguments
 2  #include<stdio.h>
 3  //Function declaration
 4  void printsum(int, int, char base='D');
 5  //Function definition
 6  void main()
 7  {
 8      printf("Use of default arguments:\n");
 9      printf("General conditions:\n");
10      printsum(5,6);
11      printsum(3,4);
12      printf("Rare conditions:\n");
13      printsum(6,9,'H');
14      printsum(6,9,'O');
15  }
16  void printsum(int x, int y, char base)
17  {
18      if(base=='d'||base=='D')
19          printf("Sum of %d and %d in decimal is %d\n",x,y,x+y);
20      else if(base=='o'||base=='O')
21          printf("Sum of %d and %d in octal is %o\n",x,y,x+y);
22      else if(base=='h'||base=='H')
23          printf("Sum of %d and %d in hexadecimal is %X\n",x,y,x+y);
24  }
```

Output window

```
Use of default arguments:
General conditions:
Sum of 5 and 6 in decimal is 11
Sum of 3 and 4 in decimal is 7
Rare conditions:
Sum of 6 and 9 in hexadecimal is F
Sum of 6 and 9 in octal is 17
```

Remarks:

- In line number 4, the parameter base is initialized with the value 'D'
- This initialization makes 'D' as default argument for the parameter base
- A function that provides a default argument for a parameter can be invoked with or without an argument for this parameter
- In line numbers 10 and 11, the function printsum is invoked without specifying an argument for the parameter base
- In line numbers 13 and 14, arguments 'H' and 'O', respectively, are specified as arguments for the parameter base. These values override the default argument value 'D'
- Borland Turbo C 3.0 IDE does not support the use of default arguments

Program 8-17 | A program that illustrates the use of default arguments

The important points about the default arguments are as follows:

1. The arguments can be made default by using initialization syntax within the parameter list during the function declaration. For example, in line number 4 in Program 8-17, the parameter base has been made default by initializing it with 'D'.
2. A function that provides a default argument for a parameter can be invoked with or without an argument for this parameter.
3. However, if an argument is provided, it overrides the default argument value.
4. A function declaration can specify default arguments for all or for a subset of parameters. If the default arguments are specified only for a subset of parameters, then these parameters should be kept on the trailing side. The code snippet in Program 8-18 illustrates this fact.

Line	Prog 8-18.c	Output window
1	//Default arguments for a subset of parameters	Compilation errors
2	#include<stdio.h>	“Default value missing following parameter b”.
3	//Function declaration	“Too few parameters in call to ‘add(int, int, int)’ in
4	int add(int a, int b=12, int c);	function main”
5	//Function definitions	**Remark:**
6	void main()	• In line number 4, the default argument
7	{	for the parameter b cannot be specified
8	add(10,12);	unless and until the default argument
9	}	for parameter c is specified
10	int add(int a, int b, int c)	**What to do?**
11	{	• Either specify the default argument for
12	printf(“The result after addition is %d\n”,a+b+c);	the parameter c or remove the default
13	}	argument value for the parameter b

Program 8-18 | A program illustrating that the specification of default arguments for a subset of parameters

The code snippet in Program 8-19 is the rectified version of the code listed in Program 8-18.

Line	Prog 8-19.c	Output window
1	//Default arguments can be specified for parameters that lie on the	The result after addition is 30
2	//trailing side of the parameter list	The result after addition is 19
3	#include<stdio.h>	**Remarks:**
4	int add(int a, int b=12, int c=8);	• In line number 4, the default argu-
5	void main()	ments are specified for two trailing pa-
6	{	rameters b and c
7	add(10);	• Since, no default argument is specified
8	add(10,1);	for the parameter a, at least one argu-
9	}	ment is required to invoke the function
10	int add(int a, int b, int c)	add
11	{	• In line number 8, the argument value 1
12	printf(“The result after addition is %d\n”,a+b+c);	overrides the default argument value
13	}	for the parameter b

Program 8-19 | A program illustrating that the default arguments can be specified for the parameters that lie on the trailing side of the parameter list

5. The default argument should not be specified in the function definition. If the default argument is provided in the parameter list of function definition as well, there will be ‘Default argument value redeclared error.’ The code snippet in Program 8-20 illustrates this fact.
6. It is not mandatory to have a default argument as a constant expression. Any expression can be used as the default argument. When the default argument is an expression, the expression is evaluated when the function is called. The code snippet in Program 8-21 illustrates this fact.

Line | **Prog 8-20.c**

```
//Redeclaration of default arguments
#include<stdio.h>
//Function declaration along with the specification of the default
//arguments
int add(int a=12, int b=8);
void main()
{
    add();
    add(10);
    add(10,12);
}
//Function definition with re-specification of the default arguments
int add(int a=12, int b=8)
{
    printf("The result after addition is %d\n",a+b);
}
```

Output window

```
Compilation error "Default argument value redeclared"
```

Remark:

- The default arguments are specified in the function declaration, they should not be re-specified in the header of the function definition

What to do?

- Remove default argument values from the header of the function definition

Program 8-20 | A program illustrating that the default arguments should not be re-declared in the header of the function definition

Line | **Prog 8-21.c**

```
 1  //Use of an expression as default argument
 2  #include<stdio.h>
 3  //Function declarations
 4  int sub(int,int);
 5  int add(int a=12,int b=sub(3,1));
 6  //Function definitions
 7  void main()
 8  {
 9      add();
10      add(10);
11      add(10,12);
12  }
13  int add(int a, int b)
14  {
15      printf("The result after addition is %d\n",a+b);
16  }
17  int sub(int a, int b)
18  {
19      return a-b;
20  }
```

Output window

```
The result after addition is 14
The result after addition is 12
The result after addition is 22
```

Remarks:

- In line number 5, the default argument for the parameter b is an expression sub(3,1)
- Carefully note the order of declaration of function sub and function add
- Before specifying sub as the default argument, it should be either declared or defined
- Change the order of declaration of function sub and function add, i.e. interchange the contents of line numbers 4 and 5 and observe the result of compilation

Program 8-21 | A program that illustrates the use of an expression as the default argument

8.5.4.4 Command Line Arguments

We have seen that arguments are given to the functions to increase their flexibility. Since main is also a function, can we give arguments to the function main? The answer to this question is YES! The main function can also accept arguments. The arguments to a called function are supplied from the calling function. However, main is the first function that gets invoked at the program startup. Therefore, how are arguments supplied to the function main? The arguments to the function main are supplied from **command line** and thus, have a special name known as **command line arguments**.

8.5.5 Recursion

Recursion is a powerful programming technique that can be used to solve the problems that can be expressed in terms of similar problems of smaller size. For example, consider a problem to find the factorial of a number n. The problem of finding the factorial of n can be expressed in terms of a similar problem of smaller size as n!=n×(n-1)!. Recursion provides an elegant way of solving such problems.

In recursive programming, a function calls itself. A function that calls itself is known as a **recursive function,** and the phenomenon is known as **recursion**. Recursion is classified according to the following criteria:

1. Whether the function calls itself directly (i.e. **direct recursion**) or indirectly (i.e. **indirect recursion**).
2. Whether there is any pending operation on return from a recursive call. If the recursive call is the last operation of a function, the recursion is known as **tail recursion**.
3. Pattern of recursive calls. According to the pattern of recursive calls, recursion is classified as:
    a. Linear recursion
    b. Binary recursion
    c. n-ary recursion

8.5.5.1 Direct and Indirect Recursion

A function is **directly recursive** if it calls itself, i.e. the function body contains an explicit call to itself. **Indirect recursion** occurs when a function calls another function, which in turn calls another function, eventually resulting in the original function being called again. The functions involved in indirect recursion are known as **mutually recursive functions**. Figure 8.2 illustrates direct and indirect recursion.

Direct recursion	Indirect recursion

A() //←Direct recursive function
{
------------ //←Statements
------------ 	
A(); //←Call to itself
------------ 	
------------ 	
}
	A() //←Mutually recursive function A
	{
	------------ //←Statements
	B(); //←Function A calls function B
	------------ 	
	}
	B() //←Mutually recursive function B
	{
	------------ //←Statements
	A(); //←Function B calls function A
	------------ 	
	}

Figure 8.2 | Direct and indirect recursion

Direct recursive functions are simpler and more elegant as compared to indirectly recursive functions and are most commonly used. The code snippet in Program 8-22 illustrates the use of recursion to find the factorial of a number.

Line	Trace	Prog 8-22.c	Output window
1		//Recursion to find the factorial of a number	Enter the number 3 Factorial of 3 is 6 **Remarks:** • The body of the function fact contains call to itself • Thus, fact is a directly recursive function • Though recursion is very powerful and highly expressive, it is hard to visualize • Trace the program and carefully observe the execution of function calls • Trace arrows in column 2 depicts the order of execution of statements
2		#include<stdio.h>	
3		//Function declaration	
4		int fact(int);	
5		//Function definitions	
6	1	void main()	
7		{	
8		int no, factorial;	
9	2	printf("Enter the number\t");	
10	3	scanf("%d",&no);	
11	4	factorial=fact(no);	
12	15	printf("Factorial of %d is %d", no, factorial);	
13	16	}	
14		//Definition of directly recursive function fact	
15	5,8,11	int fact(int no)	
16		{	
17	6,9,12	if(no==1)	
18	13	return 1;	
19	6, 12	else	
20	7, 10	return no*fact(no-1);	
21	14	}	

Program 8-22 | A program that makes the use of a recursive function to find the factorial of a number

The important points about how to develop recursive functions are as follows:

1. Thinking recursively is the first step to solve a problem using recursion.
2. Every recursive solution consists of two cases:
   a. **Base case:** Base case is the smallest instance of problem, which can be easily solved and there is no need to further express the problem in terms of itself, i.e. in this case no recursive call is given and the recursion terminates. Base case forms the **terminating condition** of the recursion. There may be more than one base case in a recursive solution. Without the base case, the recursion will never terminate and will be known as **infinite recursion.** For example, no==1 is the base case of the recursive function fact listed in Program 8-22.

   b. **Recursive case:** In a recursive case, the problem is defined in terms of itself, while reducing the problem size. For example, when fact(n) is expressed as n×fact(n-1), the size of the problem is reduced from n to n-1.
3. Express the solution in the form of base cases and recursive cases. For example, the factorial problem can be expressed as:

$$\text{fact(no)} = \begin{cases} 1 & \text{when no} = 1 \\ \text{no} \times \text{fact(no} - 1) & \text{when no} > 1 \end{cases}$$

Relation of the above form is known as **recurrence relation**.

4. Code for the recurrence relation.

8.5.5.2 Tail Recursion and Non-tail Recursion

Tail recursion is a special case of recursion in which the last operation of a function is a recursive call. In a tail recursive function, there are no pending operations to be performed on return from a recursive call. Consider the code snippets in Program 8-23 to find the factorial of a number.

Prog 8-23a.c (Column 2)

```
//Non-tail recursive factorial function
#include<stdio.h>
//Function declaration
int fact_norm(int);
void main()
{
    int no, factorial;
    printf("Enter the number\t");
    scanf("%d",&no);
    factorial=fact_norm(no);
    printf("Resultant factorial is %d",factorial);
}
//Non-tail recursive fact function
int fact_norm(int no)
{

    if(no==1)
        return 1;
    else
        return no*fact_norm(no-1);
}
```

Prog 8-23b.c (Column 3)

```
//Tail recursive factorial function
#include<stdio.h>
//Function declaration
int fact_tail(int, int);
void main()
{
    int no, factorial;
    printf("Enter the number\t");
    scanf("%d",&no);
    factorial=fact_tail(no,1);
    printf("Resultant factorial is %d",factorial);
}
//Tail recursive fact function
int fact_tail(int no, int result)
{
    if(no==1)
        return result;
    else
        return fact_tail(no-1,no*result);
}
```

Output window

```
Enter the number 4
Resultant factorial is 24
```

Remarks:

- fact_norm function in column 2 is a non-tail recursive function
- Although the last operation in this function seems to be a recursive function call, it is actually a multiplication operation
- fact_tail function in column 3 is a tail recursive function
- The last operation of this function is a recursive function call

Program 8-23 | Non-tail recursive and tail-recursive versions of function fact

The observable points about the code snippets listed in Program 8-23 are as follows:

1. The function fact_norm listed in Program 8-23a is not tail recursive because there is a pending operation, i.e. multiplication to be performed on return from a recursive call.
2. The function fact_tail listed in Program 8-23b is tail recursive as it has no pending operation on return from a recursive call.
3. Tail recursion is desirable because it eliminates the need to store the result of the computations made in a function before making the tail recursive function call (as there is no operation is to be performed on returning from the tail recursive function). The result of the computations made before tail recursive function call is passed as an argument to the tail recursive function. Due to this, conversion of a non-tail recursive function to a tail recursive function is often required. The method to convert a non-tail recursive function to a tail recursive function is as follows:
    a. A non-tail recursive function can be converted to a tail recursive function by adding one or more auxiliary parameters. For example, result is added as an auxiliary parameter in the definition of function fact_tail.

b. Incorporate the pending operation into the auxiliary parameter in such a way that the non-tail recursive function no longer has a pending operation. For example, the pending operation of multiplication is incorporated into the auxiliary parameter result as no*result.

Consider another application of recursion in finding the terms of a Fibonacci series. In the Fibonacci series, every value is the sum of previous two values. The first two values of the Fibonacci series are 0 and 1. The values 0 1 1 2 3 5 8 13 21 … form the Fibonacci series. The recurrence relation for finding any term in Fibonacci series is:

$$\mathrm{fib}(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{for } n > 2 \end{cases}$$

Program 8-24a lists the code that uses a non-tail recursive function fib_norm to find a Fibonacci term. The conversion of a non-tail recursive function to a tail recursive function is done in Program 8-24b.

Line	Prog 8-24a.c	Prog 8-24b.c	Output window
1	//Non-tail recursive Fibonacci function	//Tail recursive Fibonacci function	Enter term no. 4
2	#include<stdio.h>	#include<stdio.h>	Fibonacci term is 2
3	//Function declaration	//Function declaration	**Remarks:**
4	int fib_norm(int);	int fib_tail(int,int,int);	• fib_norm is a non-tail recursive function as the last operation to be performed in function fib_norm is addition instead of being a recursive call
5	void main()	void main()	
6	{	{	
7	int n, term;	int n, term;	
8	printf("Enter term no.\t");	printf("Enter term no.\t");	
9	scanf("%d",&n);	scanf("%d",&n);	
10	term=fib_norm(n);	term=fib_tail(n,1,0);	• fib_norm has two base cases, i.e. when n=1 and when n=2
11	printf("Fibonacci term is %d",term);	printf("Fibonacci term is %d",term);	
12	}	}	• fib_tail is the corresponding tail recursive version
13	//Non-tail recursive function fib_norm	//Tail recursive version of fib_norm	
14	int fib_norm(int n)	int fib_tail(int n,int next, int result)	• Two auxiliary parameters, i.e. next and result are used
15	{	{	
16			• The pending addition operation in fib_norm is incorporated in the auxiliary parameter of fib_tail as next+result
17	if(n==1) return 0;	if(n==1) return result;	
18	if(n==2) return 1;	return fib_tail(n-1, next+result, next);	
19	return fib_norm(n-1)+fib_norm(n-2);	}	
20	}		

Program 8-24 | Non-tail recursive and tail recursive functions to find a Fibonacci term

4. Tail recursive functions can be easily transformed into iterative functions to improve the efficiency of a program.

8.5.5.3 Pattern of Recursive Calls

Based upon the number of recursive calls within a function, the recursion is classified as:

1. Linear recursion
2. Binary recursion
3. n-ary recursion

8.5.5.3.1 Linear Recursion

The simplest form of recursion is **linear recursion**. A linearly recursive function makes only one recursive call. The function fact discussed in Program 8-22 is a linearly recursive function, as there is only one recursive call within its body. The next section describes how recursion works and how function calls form a linear structure.

8.5.5.3.1.1 How Recursion Works

Consider the code listed in Program 8-22. Figure 8.3 shows how recursion works to compute the value of factorial of 4.

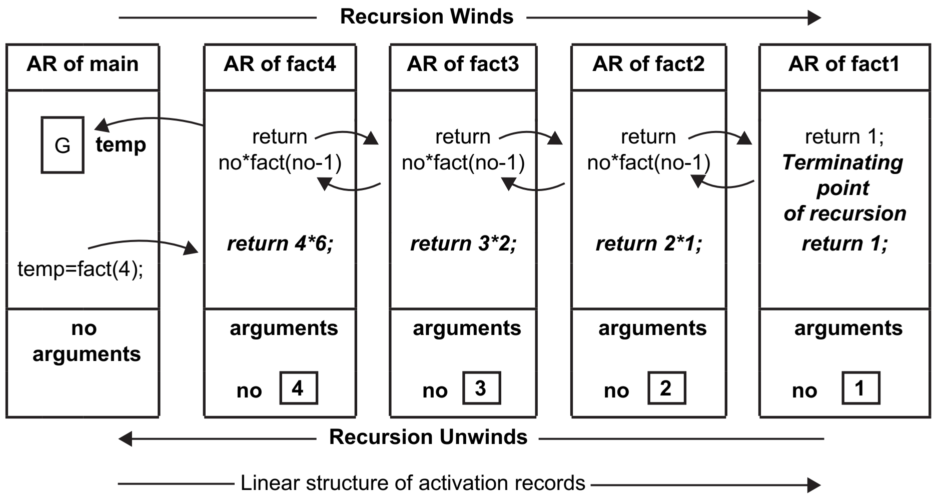


Figure 8.3 | Winding and unwinding of linear recursion

G (in the above figure) signifies garbage value of local variable temp and **AR** stands for activation record.

The function main gives a call to the function fact with 4 as an argument. Execution of this call creates an activation record✍ for the function fact. The activation record of the function main is packed, placed on the run-time stack,✍ and the activation record of the function fact becomes live. The value of no in the live activation record is 4. Since no≠1 in the current activation, the statement return no*fact(no-1); gets executed. The return expression itself contains a call to the function fact with 3 as an argument. The execution of this function call packs the current activation record of fact, places it onto the run-time stack and creates a new activation record with the value of no as 3 and makes it live. The same process is repeated till the activation record with the value of no as 1 gets created. This part of recursion in which a number of activation records are created and piled up on the run-time stack is known as **winding of recursion**. During the winding of recursion, new activation records keep on getting created. As each activation record requires some memory space, the memory requirement of a program increases during the winding of recursion. If there is no spare memory space for creating the new activation records, the recursion terminates abnormally.

When memory space is available, the winding of recursion terminates when the terminating condition of recursion is reached. In the code snippet listed in Program 8-22, the recursion terminates when the value of no becomes 1. From this point onwards, the recursion starts **unwinding**. During the unwinding process, the called activation✍ returns a value to its calling activation. After returning the value, the activation record of the called activation is destroyed and the memory occupied by it is freed. As shown in Figure 8.3, the last activation returns 1 to the second last activation, which in turn returns 2 to the third last activation and so on. In this way, the first activation of the function fact returns 24 to the function main.

✍ The term **activation** means execution of a function. If a function is executing, it is said to be **active**. In a C program, multiple functions can be active at the same time. For example, suppose function main calls a function fun1, which in turn calls another function fun2. While the function fun2 is executing, the functions main, fun1 and fun2 are all **active**. When the function fun2 completes its execution and returns the program control to the function fun1, only the functions main and fun1 remain active and the function fun2 becomes **inactive**.

Activation of each function requires a separate **activation record**. An activation record refers to the chunk of memory, which holds the following:

1. **Dynamic link:** It points to the activation record of the caller.
2. **Saved state:** It refers to the contents of the program counter and registers when the function is called. It is used to restore the context of the caller function when the program control returns.
3. **Parameters:** They refer to the memory space required by the parameters declared within the header of the function.
4. **Local variables:** They refer to the memory space required by the automatic local variables.
5. **Temporary storage:** It refers to the storage used for evaluating the expressions.

Dynamic link
Saved state
Parameters
Local variable
Temporary storage

An activation record is automatically created when a function starts the execution and is automatically destroyed when a function returns the control to its caller. The activation records for all of the active functions are stored in the region of memory called the **stack**.

8.5.5.3.2 Binary Recursion

A **binary recursive function** calls itself twice. The fib_norm function listed in Program 8-24a is a binary recursive function. In the binary recursion, the tree of recursive calls is a binary tree.✍ Figure 8.4 depicts the tree of recursive calls for fib_norm(3).

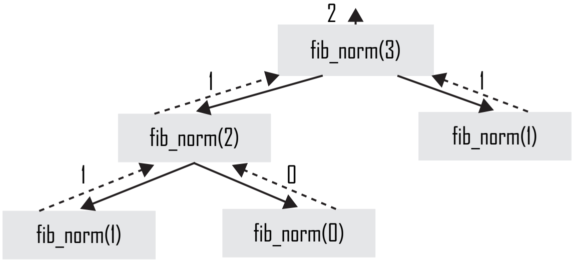


Figure 8.4 | Tree of recursive calls to the function fib_norm

Binary recursion is used in solving some of the important computing problems like:

1. Tower of Hanoi problem
2. Sorting by merge sort
3. Searching by binary search
4. Fibonacci series generation, etc.

Binary tree is a non-linear data structure in which every node of a tree can have at most two children. The tree shown in Figure 8.4 is a binary tree.

8.5.5.3.2.1 Tower of Hanoi Problem

Tower of Hanoi is one of the classical problems of computer science. The problem states that:

1. There are three stands (Stands 1, 2 and 3) on which a set of disks, each with a different diameter, are placed.
2. Initially, the disks are stacked on Stand 1, in order of size, with the largest disk at the bottom.

The initial structure of Tower of Hanoi with three disks is shown in Figure 8.5.

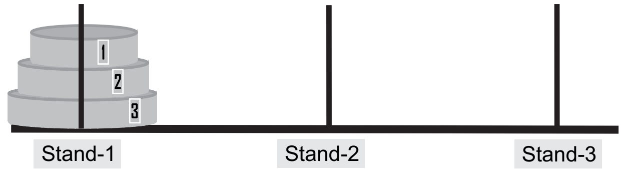


Figure 8.5 | Tower of Hanoi with three disks

The **'Tower of Hanoi problem'** is to find a sequence of disk moves so that all the disks are moved from Stand-1 to Stand-3, adhering to the following rules:

1. Move only one disk at a time.
2. A larger disk cannot be placed on top of a smaller disk.
3. All disks except the one being moved should be on a stand.

'Tower of Hanoi' is tough and computationally expensive. However, the expressive power of recursion can be used to easily formulate a solution to this problem. The general strategy for solving the Tower of Hanoi problem with n disks is shown in Figure 8.6.

1. Move the topmost n-1 disks from Stand-1 to Stand-2.
2. Move the largest disk from Stand-1 to Stand-3.
3. Move n-1 disks from Stand-2 to Stand-3.
4. Final structure.

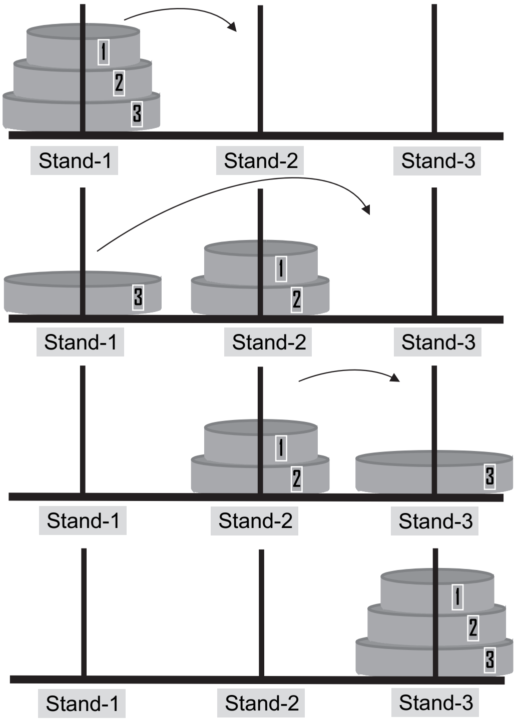


Figure 8.6 | General strategy to solve the Tower of Hanoi problem with three disks

The movement of n-1 disks forms the recursive case of a recursive solution to move n disks. The base case of a solution involves the movement of only one disk. The recurrence relation for solving the Tower of Hanoi problem can be written as:

$$\text{TowerOfHanoi(disks)} = \begin{cases} \text{move the disk} & \text{if disks} = 1 \\ \text{TowerOfHanoi(disks} - 1) & \text{if disks} > 1 \end{cases}$$

The code snippet listed in Program 8-25 solves the Tower of Hanoi problem.

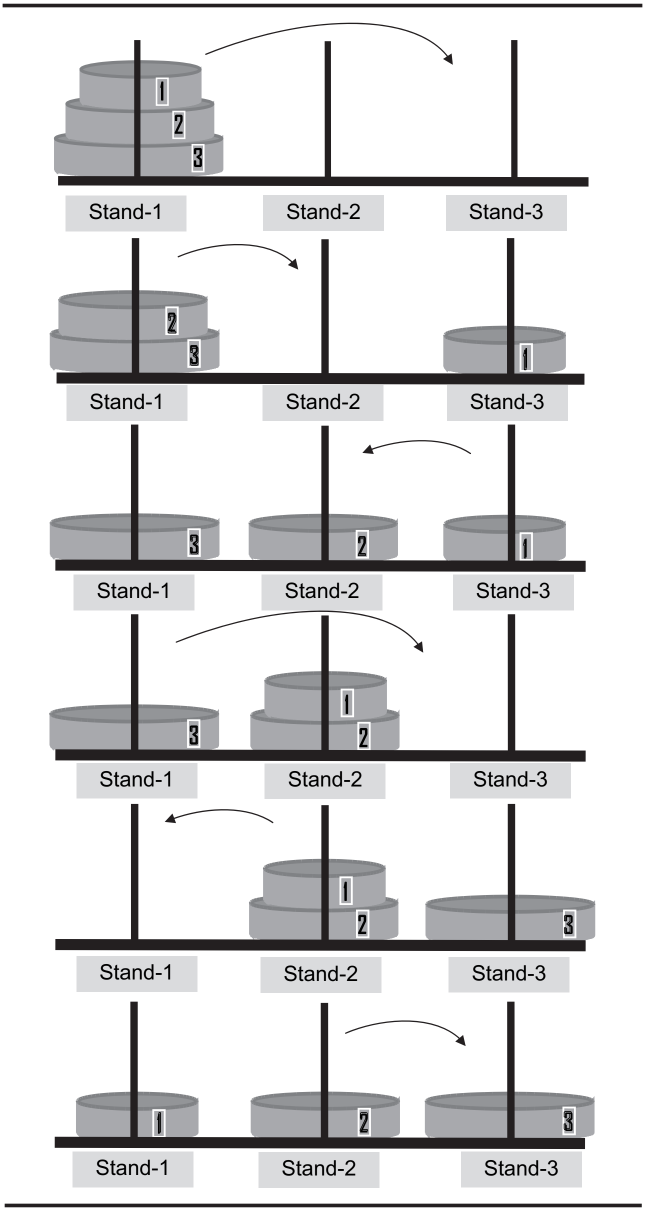
Line	Prog 8-25.c	Output window
1	`#include<stdio.h>`	Follow these moves:
2	`//Function declaration`	Move disk 1 from 1 to 3
3	`void move(int,int,int,int);`	Move disk 2 from 1 to 2
4	`//Function definitions`	Move disk 1 from 3 to 2
5	`void main()`	Move disk 3 from 1 to 3
6	`{`	Move disk 1 from 2 to 1
7	`    int disks=3;`	Move disk 2 from 2 to 3
8	`    printf("Follow these moves:\n");`	Move disk 1 from 1 to 3
9	`    move(disks,1,3,2);`	**Remarks:**
10	`}`	• Line number 15 codes step 1 of the general solution shown in Figure 8.6
11	`void move(int count,int start,int finish,int temp)`	
12	`{`	• Line number 16 is the base case and codes step 2 of the general solution shown in Figure 8.6
13	`    if(count>0)`	

(Contd...)

Line	Prog 8-25.c	Output window
14	{	• Line number 17 codes the step 3 of the general solution shown in Figure 8.6 • How disks will be actually moved can be seen by tracing the program and keeping track of argument values to the recursive calls
15	move(count-1,start,temp,finish);	
16	printf("Move disk %d from %d to %d\n",count,start,finish);	
17	move(count-1,temp,finish,start);	
18	}	
19	}	

Program 8-25 | A program to solve the Tower of Hanoi problem

The actual disk movements are shown in Figure 8.7.



(Contd...)

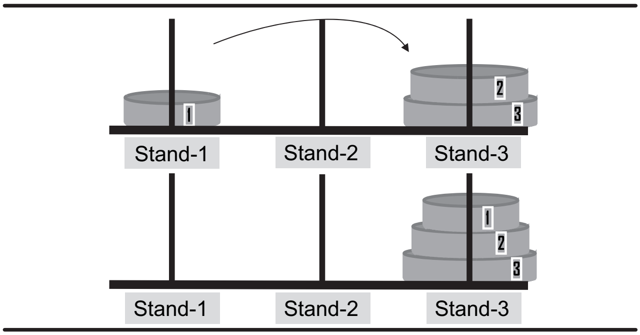
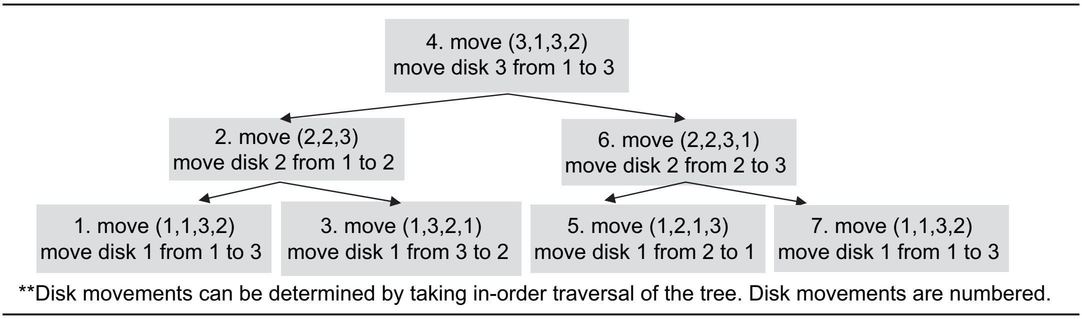


Figure 8.7 | Actual disk movements in solution to the Tower of Hanoi problem with three disks

Binary tree of recursive calls to the `move` function is shown in Figure 8.8.



**Disk movements can be determined by taking in-order traversal of the tree. Disk movements are numbered.

Figure 8.8 | Tree of recursive calls to the function `move`

8.5.5.3.3 n-ary Recursion

The most general form of recursion is **n-ary recursion** where n is not a constant but some parameter of function. **n-ary recursive** functions are used in generating permutations. The permutations of integers 1, 2 and 3 are as follows:

1	*2*	*3*
1	*3*	*2*
2	*1*	*3*
2	*3*	*1*
3	*1*	*2*
3	*2*	*1*

The code snippet listed in Program 8-26 uses n-ary recursion to print the permutations of integers 1, 2 and 3.

Prog 8-26.c

```
//n-ary recursion
#include<stdio.h>
//Definition of n-ary recursive function
permute(int array[], int parray[],int L,int N)
{
int i,j;
//Base case: Processing the permutations
if(L>N)
{
    for(i=1;i<=N;i++)
        printf(“%d “,parray[i]);
    printf(“\n”);
}
//Recursive Case: Number of case depends upon the parameter
//value N. Number of time recursive calls are given is variable.
else
{
    for(i=1;i<=N;i++)
    {
        if(array[i]==0)
        {
            parray[L]=i;
            array[i]=1;
            permute(array,parray,L+1,N);
            array[i]=0;
        }
    }
}
}
main()
{
    int array[10]={0}, parray[10],n;
    printf(“Generating permutations of 1 to n\n”);
    printf(“Enter the value of n(<10)\t”);
    scanf(“%d”,&n);
    permute(array,parray,1,n);
}
```

Output window

```
Generating permutations of 1 to n
Enter the value of n(<10)     3
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Remarks:

- permute is an n-ary recursive function
- The number of time recursive calls are given to permute depends upon the value of the parameter n
- Since the parameter n is a variable, the number of recursive calls in the activation of permute varies
- Trace the program to understand it clearly

Program 8-26 | A program that illustrates the use of recursion to print permutations

8.5.6 Pointers to Functions

Like recursion, pointers to functions provide an extremely interesting, efficient and elegant programming technique. The following concepts allow the creation of a pointer to a function:

1. Like variables, a compiled code upon execution gets some space in the main memory. Thus, a function in the program code is placed at some memory location in the Code Segment.

2. Functions like all other identifiers (except labels) do have a type. **Function type** is one of the derived data types. It consists of return type of the function and types of its parameters. For example, type of a function mult that accepts one integer and one float argument and returns a float value is float(int,float). The construction of a function type from its return type and parameter types is called **'function type derivation'**.
3. It is possible to create a pointer to any type (even void type). Hence, the creation of a pointer to a function type is also possible. A pointer to a function, commonly known as **function pointer,** is a variable that points to the starting address of the function.

Unfortunately, pointers to functions are less frequently used because of their complicated syntax. The following aspects of function pointers must be mastered so that they can be used in a correct way:

1. Declaration of a function pointer
2. Assigning or initializing a function pointer
3. Calling a function using a function pointer

8.5.6.1 Declaration of a Function Pointer

Consider the function fact developed in Program 8-22, which accepts an integer and returns an integer value. The type of function fact is int(int). A pointer to the function type int(int) is declared as:

```
int (*ptr)(int);
```

In the above declaration,✍ ptr is a pointer to a function that accepts an integer and returns an integer value.

> ✍ While reading C declaration, remember that [] and () bind more tightly than *. Hence, in declaration statement int* ptr(int);, the identifier ptr is bound to () instead of * and is read as: **ptr is a function that accepts an integer and returns an integer pointer**. The () can be used to bind ptr with *. In declaration statement int(*ptr)(int);, () is used to bind ptr with *. Hence, this declaration is read as: **ptr is a pointer to a function that accepts an integer and returns an integer value.**

Table 8.1 mentions some of the functions developed in this chapter, their types and pointers to functions of that type.

Table 8.1 | Pointers to function types

S.No	Function name(s)	Program number	Function type	Pointer to function type
1.	println	8-1	int()	int(*)()
2.	add, sub	8-1	int(int,int)	int(*)(int,int)
3.	printsum, main	8-5	void(void)	void(*)(void)
4.	printsum	8-6	void(int,int)	void(*)(int,int)
5.	printsum	8-7	void(int,int,char)	void(*)(int,int,char)

(*Contd...*)

6.	circle_area	8-9	float(int)	float(*)(int)
7.	swap	8-12	int(int*,int*)	int(*)(int*,int*)
8.	sum_diff	8-13	void(int,int,int*,int*)	void(*)(int,int,int*,int*)
9.	find_max_min	8-15a	int(int[],int)	int(*)(int[],int)
10.	find_max_min	8-15b	int(int*,int)	int(*)(int*,int)
11	largest_ele	8-16	void(int[][10],int*,int*)	void(*)(int[][10],int*,int*)
12.	f_calling_fs	8-30	void(int,int,int(*)(int,int))	void(*)(int,int,int(*)(int,int))

8.5.6.2 Assigning or Initializing a Function Pointer

A pointer to a function of type T can be assigned or initialized with the address of a function of type T or with a pointer of the same type. To assign or initialize a function pointer with the address of a function, just place the function designator (i.e. the name of the function) of a suitable and known function on the right side of the assignment operator. In the following statements, the address of the function sub is assigned to the function pointer str:

```
int sub(int,int);
int (*str)(int,int);
str=sub;
```

In the following statements, the function pointer atr is initialized with the address of the function add:

```
int add(int,int);
int(*atr)(int,int)=add;
```

The important points about the function pointer assignment or function pointer initialization are as follows:

1. At the time of function pointer assignment or initialization, the function designator must be known, i.e. declared or defined.
2. The function designator implicitly refers to the starting address of the function. However, the function designator can optionally be preceded by the address-of operator (&) to signify the address of function. The following two statements are equivalent:

```
int (*atr)(int, int)=add;
int (*atr)(int,int)=&add;
```

8.5.6.3 Calling a Function Using Function Pointer

A function pointer can be used to call a function in any of the following two ways:

1. By explicitly dereferencing it using the dereference operator, i.e. *
2. By using its name instead of the function's name

Program 8-27 illustrates the method of calling a function using the function pointers.

Line	Trace	Prog 8-27.c	Output window
1		//Calling functions using function pointers	Calling functions using function pointers:
2		#include<stdio.h>	The result of addition is 22
3		int add(int a,int b);	The result of addition is 5
4	1	main()	**Remarks:**
5		{	• Type of function add is int(int,int)
6		//Assigning address by using function designator only	• ptr1 and ptr2 are pointers to a function
7	2	int (*ptr1)(int,int)=add;	of type int(int,int)
8		//Assigning address by using address-of operator	• ptr1 is assigned an address of the
9	3	int (*ptr2)(int,int)=&add;	function add by using the function
10	4	printf(“Calling functions using function pointers:\n”);	designator only
11		//Calling function by dereferencing function pointer	• ptr2 is assigned an address of the
12	5	(*ptr1)(10,12);	function add by using address-of op-
13		//Calling by using function pointer name	erator and the function designator
14	9	ptr2(2,3);	• ptr1 and ptr2 both point to the function
15	13	}	add
16	6,10	int add(int a, int b)	• In line number 12, ptr1 is dereferenced
17		{	and is used to call the function add
18	7, 11	printf(“The result of addition is %d\n”,a+b);	• In line number 14, ptr2 is used to call
19	8,12	}	the function add without dereferenc-
			ing it
			• Trace the program and note the trace
			arrow numbering

Program 8-27 | A program that illustrates the method of calling function using function pointers

8.5.7 Array of Function Pointers

Like arrays of pointers to other types, it is possible to create array of pointers to function type (i.e. array of function pointers). The following declaration statement declares arr as an array of pointers to functions that accept two integers and returns an integer:

```
int (* arr[4])(int,int);
```

The important points about the above declaration and the array of function pointers are as follows:

1. arr is an array of function pointers. Each pointer takes 2 bytes or 4 bytes in the memory depending upon the compiler and the working environment used. Hence, the total memory space allocated to arr will be 8 bytes or 16 bytes. The code snippet in Program 8-28 illustrates this fact.

Line	Prog 8-28.c	Output window
1	//Size of array of function pointers	Memory allocated to arr is 8 bytes
2	#include<stdio.h>	**Remarks:**
3	main()	• Turbo C 3.0 gives the above-mentioned result.
4	{	If Turbo C 4.5 is used, the result will be 16 bytes
5	int (*arr[4])(int,int);	• The name of an array does not decompose to a
6	printf(“Memory allocated to arr is %d bytes”,sizeof(arr));	pointer type if it is an operand of sizeof operator
7	}	• sizeof operator gives the memory allocated to
		the complete array

Program 8-28 | A program that finds the size of an array of function pointers

2. Like other arrays, arrays of function pointers can also be initialized by providing an initialization list. The initializers in the initialization list should be function designators of the known functions (i.e. declared or defined) of appropriate type. All the initializing functions should have the same type.
3. The array of function pointers can be used to call functions in a generalized way. The code snippet in Program 8-29 illustrates the initialization of an array of function pointers and the method to call functions in a generalized way.

Prog 8-29.c

```
//Array of function pointers
#include<stdio.h>
//Function declarations
int add(int,int);
int sub(int,int);
int mult(int,int);
int div(int,int);
//Function definitions
main()
{
//Array of function pointers initialized with initialization list
    int (*arr[4])(int,int)={add,sub,mult,div};
    int lc;
    printf(“Calling functions using iteration:\n”);
//Functions called in a generalized way by using loop
    for(lc=0;lc<4;lc++)
        arr[lc](6,3);
}
int add(int a,int b)
{
    printf(“Result of addition of %d and %d is %d\n”,a,b,a+b);
}
int sub(int a,int b)
{
    printf(“Result of subtraction of %d and %d is %d\n”,a,b,a-b);
}
int mult(int a,int b)
{
    printf(“Result of multiplication of %d and %d is %d\n”,a,b,a*b);
}
int div(int a,int b)
{
    printf(“Result of division of %d and %d is %d\n”,a,b,a/b);
}
```

Output window

```
Calling functions using iteration:
Result of addition of 6 and 3 is 9
Result of subtraction of 6 and 3 is 3
Result of multiplication of 6 and 3 is 18
Result of division of 6 and 3 is 2
```

Remarks:

- All functions add, sub, mult and div have the same type, i.e. int(int,int)
- These functions accept two integers and return an integer
- arr is an array of 4 function pointers of type int(*)(int,int)
- arr is initialized with an initialization list
- All the initializers are of the same type
- It can also be initialized as: int(*arr[4])(int,int)={&add,&sub,&mult,&div}

Program 8-29 | A program that illustrates the use of array of function pointers

8.5.8 Passing Function to a Function as an Argument

A function can accept arguments of pointer type. We have seen the application of pointers to pass arrays as arguments to the functions. Pointers can also be used to pass functions to a function. The code snippet in Program 8-30 illustrates the use of pointers to pass functions to a function.

Line	Trace	Prog 8-30.c	Output window
1		//Passing function to a function as an argument	Passing functions to a function:
2		#include<stdio.h>	Result of addition of 10 and 20 is 30
3		//Function declarations	Result of subtraction of 10 and 20 is -10
4		int add(int,int);	**Remarks:**
5		int sub(int,int);	• The third argument to the function f_calling_fs is a function pointer of type int(*)(int,int)
6		//Declaration of function whose third parameter is a function ptr	
7		void f_calling_fs(int,int,int(*)(int,int));	
8		//Function definition	
9	1	void main()	• In line number 13, the address of the function add is passed to the function f_calling_fs
10		{	
11	2	printf("Passing functions to a function:\n");	
12		// Third argument in the following function calls is a function designator	
13	3	f_calling_fs(10,20,add);	• In line number 18, the passed argument is used to call the function. Since the address of the function add has been passed, the call fun(a,b) is equivalent to add(a,b)
14	10	f_calling_fs(10,20,sub);	
15	17	}	
16	4,11	void f_calling_fs(int a, int b, int (*fun)(int,int))	
17		{	
18	5,12	fun(a,b);	
19	9,16	}	
20	6	int add(int a,int b)	• Similarly, in line number 14, the function sub is passed to f_calling_fs and on the second execution of line number 18, it is called
21		{	
22	7	printf("Result of addition of %d and %d is %d\n",a,b,a+b);	
23	8	return 0;	
24		}	
25	13	int sub(int a,int b)	
26		{	• Trace the program and note the trace arrow numbering
27	14	printf("Result of subtraction of %d and %d is %d\n",a,b,a-b);	
28	15	return 0;	
29		}	

Program 8-30 | A program that illustrates the passing of functions to a function

8.6 Library Functions

Library functions or pre-defined functions are the functions whose functionality has already been developed by someone and are available to the user for use. For example, printf and scanf are library functions. There are two aspects of working with library functions:

1. Declaration of library functions
2. Use of library functions

8.6.1 Declaration of Library Functions/Role of Header Files

We have seen that the user-defined functions need to be declared before they are called. This is true for library functions as well. A library function needs to be declared before it is called. The declarations of library functions are available in their respective header files. To make these declarations accessible in a program file, the corresponding header files are included. For example, the prototype, i.e. the declaration of printf function is available in the header file stdio.h. That is why stdio.h is included before calling the printf function. If the header file containing the declaration of the library function is not included before its use, there will be a compilation error 'Prototype missing.'

8.6.2 Use of Library Functions

Library functions are used in the same way as user-defined functions, i.e. by using a function call operator. The role and usage of some of the common library functions are listed below.

8.6.2.1 Library of Mathematical Functions

The mathematical library defines some of the common mathematical functions. The declarations of these mathematical functions are available in the header file **math.h.** Table 8.2 lists the commonly used mathematical functions available in the math library.

Table 8.2 | Mathematical functions available in math library

S.No	Function	Function declaration and use	Role
Trigonometric functions			
1.	acos	double acos(double x);	Returns arc cosine of x in radians
2.	asin	double asin(double x);	Returns arc sine of x in radians
3.	atan	double atan(double x);	Returns arc tangent of x in radians
4.	atan2	double atan2(double y, double x);	Returns the arc tangent in radians of y/x based on the signs of both values to determine the correct quadrant
5.	cos	double cos(double x);	Returns the cosine of a radian angle x
6.	cosh	double cosh(double x);	Returns hyperbolic cosine of x
7.	sin	double sin(double x);	Returns the sine of a radian angle x
8.	sinh	double sinh(double x);	Returns hyperbolic sine of x
9.	tan	double tan(double x);	Returns the tangent of a radian angle x
10.	tanh	double tanh(double x);	Returns hyperbolic tangent of x
Exponential, logarithmic and power functions			
11.	exp	double exp(double x)	Returns the value of e raised to the x^{th} power
12.	frexp	double frexp(double x, int *exponent);	frexp splits a double number x into mantissa and exponent. Given x, frexp calculates the mantissa m and exponent n such that $x = m*2^n$
13,	ldexp	double ldexp(double x, int exponent);	Returns x multiplied by 2 raised to the power of exponent, i.e. returns $x*2^n$
14.	log	double log(double x);	Returns the natural logarithm (*base e*) of x
15.	log10	double log10(double x);	Returns the common logarithm (*base 10*) of x
16.	pow	double pow(double x, double y);	Returns x raised to the power of y
17.	sqrt	double sqrt(double x);	Returns the square root of x

(*Contd...*)

Other mathematical functions			
18.	ceil	double ceil(double x);	Returns the smallest integer value greater than or equal to x
19.	fabs	double fabs(double x);	Returns the absolute value of x (a negative value becomes positive, positive value remains unchanged)
20.	floor	double floor(double x);	Returns the largest integer value less than or equal to x
21.	fmod	double fmod(double x, double y);	Calculates x modulo y, i.e. returns the remainder of x divided by y

The return type of every math library function is double.

8.6.2.2 Library of Standard Input/Output Functions

The functionality of standard input and output operations is provided by this library. The declarations of these functions are available in the header file stdio.h. stdio is an acronym for standard input output. The common standard input/output functions are printf, scanf, gets, puts, getch, getchar, putch, putchar, etc.

8.6.2.3 Library of String Processing Functions

This library consists of functions that are used for string processing. The common string library functions are strcpy, strrev, strcat, strcmp, strcmpi, etc. The declarations of these functions are available in the header file string.h. The role and working of string library functions will be discussed in Chapter 7 after the discussion on character arrays.

8.7 Based upon the Number of Arguments a Function Accepts

Based upon the number of arguments a function accepts, functions are classified as follows:

1. Fixed argument functions
2. Variable argument functions

8.7.1 Fixed Argument Functions

A function that accepts a fixed number of arguments is called a **fixed argument function**. If the fixed argument function does not specify any default argument, invoking a fixed argument function with a lesser number of arguments than expected leads to a compilation error. A fixed argument function cannot even be invoked by supplying more number of arguments than expected. For example, pow function listed in Table 8.2 expects two arguments of type double. The following invocations of pow function are erroneous:

```
pow();                  //←Lesser number of arguments supplied than expected
pow(2.0);               //←Lesser number of arguments supplied than expected
pow(2.0,1.5,1.0);       //←More number of arguments supplied than expected
```

8.7.2 Variable Argument Functions

A function that accepts a variable number of arguments is called a **variable argument function**. For example, printf is a variable argument function, which can accept one or more arguments. The type of first argument must be char* and there is no constraint about the type of rest of the arguments. The following calls to printf function are valid:

```
printf("Hello");            //←Only one argument of type char*
printf("%d",2);             //←Two arguments. The type of the first argument is char* and the
                            //    second is int
printf("%s %s","Hi","!!");  //←Three arguments, all of type char*
```

A function that accepts a variable number of arguments✍ can be created by using the macros va_start, va_arg, va_end available in the header file stdarg.h. The piece of code in Program 8-31 illustrates the development of a variable argument function.

Prog 8-31.c

```
//Variable argument functions
//File stdarg.h is to be included for using va_list, va_start, etc.
#include<stdarg.h>
#include<stdio.h>
//Ellipses (i.e. three dots) are used to declare variable argument function
int sum(int no_of_arguments, ...);
//Function definitions
main()
{
    int result;
//Function sum invoked with 4 arguments
    result=sum(3,12,13,14);
    printf("The result of addition of 3 numbers is %d\n",result);
//Function sum invoked with 6 arguments
    result=sum(5,10,20,30,40,50);
    printf("The result of addition of 5 numbers is %d\n",result);
}
//Definition of a variable argument function
int sum(int no_of_arguments,...)
{
    int arg,i=0,total=0;
    va_list ptr;
    va_start(ptr,no_of_arguments);
    arg=va_arg(ptr,int);
    while(i++<no_of_arguments)
    {
        total+=arg;
        arg=va_arg(ptr,int);
    }
    va_end(ptr);
    return total;
}
```

Output window

```
The result of addition of 3 numbers is 39
The result of addition of 5 numbers is 150
```

Program 8-31 | A program that makes use of variable argument functions

The important points about the code listed in Program 8-31 and the variable argument functions are as follows:

1. Since the function sum is ‘a fixed number of argument followed by a variable number of argument’ function, it is declared as int sum(int no_of_arguments,...);. **Ellipses**✍ (...) are used while declaring a variable argument function.
2. **Role of ellipses:** The number of arguments that can be passed to a variable argument function is not fixed. Hence, while declaring a variable argument function, it is not possible to list the types of all the arguments that might be passed to the function during the function call. The solution to this problem is provided by ellipses. While declaring a variable argument function, ellipses (...) are used in the parameter list. **The presence of ellipses (...) tells the compiler that when the function is called, zero or more arguments may follow and that the type of the arguments is not known.** Ellipses (…) used in the declaration of the variable argument function **suspend the type checking**.
   The prototype/declaration of printf function is int printf(const char*,...);. The prototype says that there can be one or more arguments in the printf function call. The type of first argument would be const char* and the latter arguments can be of any type. Due to ellipses (...) the following uses of printf function are valid:
    1. printf(“Hello Readers”);
    2. printf(“%d %d”,2,3);
    3. printf(“%d %s %c”,2,”Hi”,’I’);
3. The variable argument functions are developed with the help of macros va_start, va_arg and va_end, declared in the header file stdarg.h. Therefore, the header file stdarg.h is included so that the macros can be used.✍
4. The header file stdarg.h also declares a type va_list that holds the information needed by the macros va_arg and va_end. A variable ptr of type va_list is declared in the function sum.
5. The macro va_start takes two parameters ptr and lastfix. The type of the first parameter ptr is va_list and lastfix is the last fixed parameter supplied to the variable argument function. The last fixed parameter supplied to the variable argument function sum is no_of_arguments and is of type int. The macro va_start sets ptr to point to the first of the variable arguments being passed to the function.
6. The macro va_arg is used to return the arguments in the variable list. The first time va_arg is used, it returns the first argument in the list. Each successive time va_arg is used, it returns the next argument in the list. The macro va_arg returns the values of type given to it as its second argument (for example, int in the code listed in Program 8-31).
7. The macro va_end should be called after va_arg has read all the arguments. If the macro va_end is not used, the program may show strange and undefined behavior.
8. The order in which the macros va_start, va_arg and va_end should be called is:
   a. va_start must be called before the first call to va_arg or va_end.
   b. va_end should only be called after va_arg has read all the arguments.

✍ **Variable argument functions** actually have a fixed number of arguments followed by a variable number of arguments.

There should be only three dots, i.e. (…) in **ellipses**. Usage of more than three dots in ellipses leads to a compilation error.

The syntax of **using macros** is similar to the syntax of using functions.

8.8 Summary

1. Functions help in modularizing a program into smaller simple parts.
2. Functions are classified based upon: (a) who develops the function and (b) the parameter and the return type of the function.
3. Based upon who developed the function, they are categorized as: (a) user-defined functions and (b) library functions.
4. Based upon the parameter and the return type of the function, they are categorized as: (a) functions with no input and no output, (b) functions with inputs but no output, (c) functions with inputs and a single output and (d) functions with inputs and multiple outputs.
5. User-defined functions are defined by the user at the time of writing a program and are also known as programmer-defined functions.
6. There are three aspects of working with user-defined functions: (a) function declaration, (b) function definition and (c) function call.
7. Function definition, also known as function implementation means composing a function. Every function definition consists of two parts: (a) header of the function and (b) body of the function.
8. A function with no input–output does not accept any input and does not return any result.
9. The execution of a C program always begins with the function `main`. It need not to be called explicitly.
10. Functions whose return type is `void` are known as `void` functions. `void` functions do not return any value.
11. While calling a function, the expressions that appear within the parentheses of a function call are known as actual arguments, and the variables declared in the parameter list in the header of function definition are known as formal parameters.
12. The `return` statement is to return the result of computations done in the called function and/or the program control back to the calling function.
13. There are two forms of `return` statement: (a) `return;` and (b) `return expression;`.
14. Depending upon whether values or addresses are passed as arguments to a function, the argument passing methods in C language are classified as: (a) pass by value and (b) pass by reference/address.
15. If arguments are passed by value, the changes made in the values of formal parameters inside the called function are not reflected back to the calling function.
16. If the arguments are passed by reference/address, the changes made in the values pointed to by the formal parameters in the called function are reflected back to the calling function.
17. A function can return only one value by using the `return` statement but it can indirectly return more than one value using the concept of pass by reference/address.
18. When an array is passed as an argument to a function, it implicitly gets converted to a pointer type.
19. The arguments can be made default by using an initialization syntax within the parameter list during the function declaration.

20. The default argument should not be specified in the function definition.
21. Function calling itself is called recursive function and the process is known as recursion.
22. Recursive functions may be: (a) direct recursive/indirect recursive and (b) tail recursive/non-tail recursive.
23. There are three patterns of recursive calls: (a) linear, (b) binary and (c) n-ary.
24. Like recursion, pointers to functions provide an extremely interesting, efficient and elegant programming technique.
25. A pointer to a function, commonly known as the function pointer, is a variable that points to the address of a function.
26. Library functions or pre-defined functions are the functions whose functionality has already been developed by someone and is available to the user for use.
27. The arguments to the function `main` are supplied at the command line and thus have a special name known as command line arguments.

Exercise Questions

Conceptual Questions and Answers

1. *What is a function? What are the advantages of using functions?*

   A function is a group of statements that performs a specific task and is relatively independent of the remaining code. Functions are used to organize programs into smaller and independent units. Several advantages of modularizing the program into functions include:

   1. Reduction in code redundancy
   2. Enabling code reuse
   3. Better readability
   4. Information hiding
   5. Improved maintainability

2. *Do functions have a type like other identifiers? If yes, how is it derived?*

   Yes, functions do have a type like all other identifiers except labels. Function type is one of the derived types and consists of return type of the function and the types of its parameters. For example, the type of a function `mult` that accepts one integer and one `float` parameter and returns a `float` value is `float(int,float)`. The construction of a function type from its return type and parameter types is called 'function type derivation'.

3. *What are the differences between a function declaration and a function definition?*

   The major differences between a function declaration and a function definition are as follows:

a. A function can only be defined once but can be declared many times.
b. A function can be declared within the body of some other function but cannot be defined within the body of some other function.
c. A function definition can also serve as a function declaration but the vice versa is not true. The function definition serves as a function declaration if it is present before the function call.
d. The function definition can be changed without changing the function declaration but if the function declaration is changed, it becomes necessary to change the function definition.
e. For using (i.e. calling) a function, it is sufficient and necessary to know the function declaration without knowing anything about how it is defined.

4. *What is meant by prototyping a function? Why is a function prototype necessary?*

   The function declaration is also called a function prototype. Hence, function prototyping means declaring a function.
   Refer Section 8.4.1 for a description on function prototype and its necessary.

5. *'C is a strongly typed language'. What does that mean?*

   'C is a strongly typed language' means that the arguments of every function call are type checked during the compilation. If the compiler detects a type mismatch between the type of an argument and the type of corresponding parameter, an implicit-type conversion is applied if possible. If it is not possible to apply implicit-type conversion, the compiler issues an error message. That is why functions cannot be called until they are declared or defined. The declaration or definition of function is necessary for the compiler to perform the type checking on the arguments of the function call against the function parameter list.

6. *Is it mandatory to specify the same name for the parameters in the declaration and definition of a function?*

   No, it is not mandatory to have the same name for the parameters in the function declaration and the function definition. In fact, it is not even compulsory to write names of the parameters in the function declaration.

7. *I want to write a function* `add` *that should add the contents of two integer variables and return their sum. I have made the following declaration for the function:*

   ```
   int add(int v1,v2);
   ```

   The compiler is not accepting it and is showing an error. Why?

   The compiler shows an error due to erroneous parameter list. The shorthand declaration of parameters in the parameter list is not allowed and leads to the compilation error. The rectified declaration for the function can be written as `int add(int v1,int v2);`.

8. *What are user-defined functions and library or pre-defined functions? Is* `main` *a library function or a user-defined function?*

   User-defined functions are defined by the user at the time of writing a program. Library functions are the functions whose functionality has already been developed by someone and are available to the user for use.

   `main` is a user-defined function because the functionality to the `main` function is always added by the user by writing its body.

9. *Why do we include header file(s) in our programs? What is their role?*

   Refer Section 8.6.1.

10. *What is meant by the terms actual arguments and formal parameters?*

    Refer Section 8.4.3.3.

11. *What are the different ways of passing arguments to a function?*

    Refer Sections 8.5.1 and 8.5.2.

12. *Does C actually have a pass by reference?*

    No, the C language actually does not have a pass by reference. The C language always passes the argument by value. The call by reference is artificially simulated by passing addresses by value. In a call by reference, the l-values given as actual arguments are copied into the parameters of pointer type.

13. *How are arrays passed to the functions?*

    Arrays are always passed by reference. The word array here means the entire array and not the individual array elements.

14. *What are the various forms of* `return` *statement? What is the specific use of each form?*

    Refer Section 8.4.3.5.

15. *It is said that 'Function can only return one value'. Can't I return more than one value by writing* `return value1, value2, value3;`?

    Refer Section 8.4.3.5 (Point 4)

16. *Can a function have more than one* `return` *statement within its body?*

    Yes, a function can have more than one `return` statement within its body. There is no constraint about the number of `return` statements that can be placed within a function's body. For example, the following piece of code is valid:

```
int funct()
{
    return 1;                                //←Control returns from this point
    printf ("This can never be executed");   //← This point onwards, code is unreachable
    return 2;
    return 3;
    printf("There are multiple return statements");
}
```

Although, a number of return statements can be placed inside the body of a function, only one of them that appears first in the logical flow of control gets executed. With the execution of this return statement, the program control returns to the calling function and the rest of the statements that appear after this return statement remain unreachable.

On compiling the mentioned code, there will be no error, but the compiler issues a warning message 'Unreachable code in function funct'. This warning is due to the fact that the program control returns to the calling function with the execution of the first return statement and can never reach the latter part of the code.

17. *I have developed the following piece of code to compute the area of a circle. It outputs* 78.000000 *instead of the actual value of the area of circle, i.e.* 78.537498*. Why?*

```
circle_area(int);
main()
{
    int rad=5.5;
    float area;
    area=circle_area(rad);
    printf("The area of circle is %f",area);
}
circle_area(int rad)
{
    float area;
    area=3.1415*rad*rad;
    return area;
}
```

Refer Section 8.4.3.5 (Point 2(f)).

18. *In the programs that I have written till now, I got a warning message 'Function should return a value'. What does this mean?*

This warning message comes if the return type of a given function is not void, and in the body of the function return statement has not been used to return any value. For example, consider the following piece of code:

```
main()
{
    printf("Warning message");
}
```

The mentioned code on compilation gives a warning message: 'Function should return a value'. There can be three different ways to remove this warning:

```
main()
{
    printf(“Warning message”);
    return 0;
}
```

Way I

```
void main()
{
    printf(“Warning message”);
}
```

Way II

```
#pragma warn -rvl
main()
{
    printf(“Warning message”);
}
```

Way III

A compilation of the above-mentioned codes will not generate a warning message because:

1. Way I returns an integer value.
2. Way II mentions the return type as void.
3. Way III configures the compiler using pragma directive in such a way that it does not generate ‘Function should return a value’ warning message.

19. *What is the difference between a warning and an error?*

    Warning is only an indicator that something may go wrong but an error is a notification that some mistake (i.e. syntactic violation) has occurred. The compiler can be configured to turn off the display of warning messages but it cannot be stopped from displaying error messages.

20. *Can a return type of a function be an array type or a function type?*

    No. The return type of a function shall be void or an object type other than an array type and a function type. Arrays and functions are returned to the calling function in the same way as they are passed to the called function, i.e. by the means of pointers. For example, consider the following code snippets:
    In Code I (a), the return type of the function fun_returning_array is an array type. The given code on compilation gives an error. Code 1 (b) shows the rectified version of Code I (a) whereby an array arr is returned by the means of a pointer, i.e. the address of the first element of the array. In Code II (a), the function area_of_square is passed to the function fun by the means of a pointer. In Code II (b), the function fun returns function area_of_square by the means of a pointer.

```
int[3] fun_returning_array();
main()
{
    int *ptr;
    ptr=fun_returning_array();
    printf(“%d %d %d”,ptr[0],ptr[1],ptr[2]);
}
int[3] fun_returning_array()
{
    int arr[3]={1,2,3};
    return arr;
}
```

Code I (a) (Return type is array type)

```
int* fun_returning_array();
main()
{
    int *ptr;
    ptr=fun_returning_array();
    printf(“%d %d %d”,ptr[0],ptr[1],ptr[2]);
}
int* fun_returning_array()
{
    int arr[3]={1,2,3};
    return arr;
}
```

Code I (b) (Return type is a pointer)

(*Contd...*)

```
int area_of_square(int side)
{
    return side*side;
}
int fun(int (*fun_name)(int))
{
//The parameter fun_name contains the address of function
//area_of_square
    return fun_name(2);
//The function area_of_square is called with argment 2. The
//result returned by the function area_of_square is returned
//by the function fun
}
main()
{
//The function fun is called with the name of function
//area_of_square as an argument
    printf("Area is %d",fun(area_of_square));
}
```

Code II (a)(Function passed to a function)

```
int area_of_square(int side)
{
    return side*side;
}
int(*fun())(int)
{
//Function fun accepts no argument. It returns a pointer to
//a function that accepts an integer and returns an integer
    return area_of_square;
}
main()
{
//The function fun is called without any argument. It returns a
//pointer to the function area_of_square. The returned
//pointer is used to call the function area_square with
//argument 2
    printf("Area is %d",fun()(2));
}
```

Code II (b)(Function returned by the means of a pointer)

21. *What are activation records?*

    Refer Section 8.5.5.3.1.1.

22. *What is recursion? What are the advantages and disadvantages of recursion over iteration?*

    Refer Section 8.5.5.

    The merits and demerits of recursion over iteration are listed below:

Iteration	Recursion
1. Performance wise, iteration is superior as compared to recursion	1. Performance of recursion is poor as compared to iteration. Recursion involves calling the same function again and again. The execution of a function call is time consuming as the entire state of a calling function needs to be saved before the control is passed to the called function. Therefore, precious computing time is wasted in book-keeping tasks
2. Memory requirement of an iterative function is less as compared to that of a recursive function	2. Recursion involves function calls. Each function call requires creation of an activation record, which takes some memory. The memory required by an activation record is directly proportional to the number of local variables declared within the recursive function.

(Contd...)

	The total memory required by the recursion is equal to the memory taken by all the activation records that exist at some particular instance
3. Infinite iteration will not terminate	3. Infinite recursion will automatically terminate when there is no memory space left for the creation of the activation records
4. Iteration is diffucult to express in some cases	4. One of the major advantages of the recursion is the **ease of expression**. The tasks that are expressible in terms of themselves can be easily coded by using recursive functions. For example, the computation of the factorial of a number. The factorial of a number is equal to the number multiplied by the factorial of the number minus one, i.e. fact(n)=n*fact(n–1)

23. *What is tail recursion?*

    Refer Section 8.5.5.2.

24. *How do the declaration statements* `int *func(int);`, `int(*func)(int);` and `int(*func())(int);` *differ from each other?*

    While reading C declarations, remember that [] and () bind more tightly than *. In the declaration statement, `int *func(int);` the identifier name `func` is bound to () instead of * and it is read as: **`func` is a function that accepts an integer and returns a pointer to an integer**. In the declaration statement, `int (*func)(int);` () is used to bound `func` to *. Hence, the declaration is read as: **`func` is a pointer to a function that accepts an integer and returns an integer**. In the declaration statement `int(*func())(int);` the identifier name `func` is bound to inner () instead of * and is read as: **`func` is a function that accepts no argument and returns a pointer to a function that accepts an integer and returns an integer.**

25. *If there is a type mismatch between the type of argument and the type of corresponding parameter, will the compiler apply implicit-type conversion? Is the same applicable if there is a mismatch between the type of value returned and the return type of a function?*

    Yes, if the type of the argument and the corresponding parameter do not match or if the type of value returned does not match the return type of the function, an implicit-type conversion is applied, if possible. If it is not possible to apply an implicit-type conversion, the complier issues an error message.

26. *I have encountered the following piece of code:*

```
int add(int v1,int v2=10);
main()
{
    int result;
    result=add(5);
}
int add(int v1,int v2)
{
    return v1+v2;
}
```

There are two parameter names in the parameter list of the function `add`*. I have read that if the number of arguments is incorrect or the types of arguments are not compatible with the types of parameters, the compile time error is issued. In the call to function* `add` *only one argument is given instead of two, but still the code is executing. Why is the compiler not showing an error?*

The compiler does not show an error because `v2` is a default argument. A function that provides a default argument for a parameter can be invoked with or without an argument for that parameter. If an argument is not provided, the default argument value is used, but if it is provided it overrides the default argument value.

27. *What are variable argument functions? How are they created?*

    Refer Section 8.7.2.

28. *A variable argument function can have a variable number of arguments, so it is not possible to list the type and number of all the arguments that might be passed to a function. Therefore, how can I make declaration for a variable argument function, and how is type checking done for a variable argument function?*

    Refer to the role of ellipses mentioned in Section 8.7.2.

29. *Are the following declarations equivalent?*

    1. `void funct();`
    2. `void funct(parameter_list,...);`
    3. `void funct(...);`

    No, the specified declarations are not equivalent. In declaration 1, `funct` is declared as a function that accepts no arguments. In declaration 2, `funct` is declared as a function that at least accepts the arguments of the specific type mentioned in the parameter list. In declaration 3, `funct` is declared as a function that can take zero or more arguments.

30. *Why does the following piece of code not compile successfully?*

```
test_function()
{
    printf("Control is now in test function");
    return;
}
main()
{
    printf("There is a simple call to a test function");
    test_function();
    printf("Control returns to main after executing test function");
}
```

The code does not compile successfully because the return type of the function `test_function` is not specified. By default, it would be considered as `int`. In the body of the function `test_function`, the first form of the `return` statement (i.e. `return;`) is used, but it can only be used if the return type of the function is `void`. That is why the compiler shows an error. There are two ways of removing this error:

1. Specify the return type of the function test_function as void.
2. Use the second form of return statement (i.e. return expression;) in the body of the function test_function. Write return 0; instead of return;.

31. *I have written the following piece of code:*

```
inc_value(int a)
{
5+return a;
}
main()
{
    int a=10,c;
    c=inc_value(a);
    printf("The incremented value of a returned is %d",c);
}
```

Why is the following piece of code not working?

The code is not working because it is not valid to write 5+return a;. return is a statement and cannot be used as an operand of an operator. Only the expressions can form operands of an operator. Instead of 5+return a; it should have been return 5+a; or return a+5;.

32. *What will the output of the following piece of code be?*

```
main()
{
    printf("%d",sizeof(printf("Hello Readers!!")));
}
```

The output of the code **WILL NOT** be Hello Readers!!2. The given piece of code on execution outputs 2. Remember that the operand of sizeof operator is not evaluated. Thus, when the expression printf("Hello Readers!!") is given to it as an operand, it is not evaluated, and the operator operates on its return type, i.e. int. Thus, the output comes out to be 2. Consider another example:

```
main()
{
    int a=2;
    printf("%d ",sizeof(a+=2));
    printf("%d ",a);
}
```

The mentioned piece of code on execution outputs 2 2 instead of 2 4 because the expression a+=2 is not evaluated.

33. *What will the output of the following piece of code be?*

```
main()
{
    printf("goto statement trying to transfer control to other function");
    goto target_pt;
}
other_funct()
{
    target_pt:
    printf("The target label is present in other function");
}
```

The mentioned piece of code on compilation gives an error 'Undefined label target_pt in function main'. The goto statement can only transfer the control from one point to another within the same function. It cannot take the control from one function to another.

34. *Both the function call statement and the* goto *statement can be used to transfer the control from one point to another. Then, why does the* goto *statement cannot be used to transfer control from one function to another?*

    The goto statement cannot be used to take the control from one function to another. Transferring the control from one function to another is not as simple as transferring control within the same function. If a control is to be transferred from one function (i.e. calling function) to another (i.e. called function), the following two additional tasks along with some other activities are to be performed:

    1. **Saving all the computations performed in the calling function prior to the function call:** All the computations performed in the calling function prior to the function call need to be saved so that they need not be carried out again upon returning from the called function. To save all the computations performed, all the local variables declared within the calling function are saved before executing the function call. The stored values of the local variables are restored after returning from the called function.
    2. **Saving the point of function call:** The point from where the function call is given is saved so that the control can return to the same point after executing the called function. The point of the function call can be saved by taking the dump of content of **registers**, specifically IP register. Instruction Pointer (IP) register is a 16-bit register that points to the memory location of the next statement to be executed. When the control returns from the called function, the content of the Instruction Pointer register is restored so that the statement next to the statement containing the function call gets executed.

    Execution of these additional tasks requires some time and that is why function calls are time consuming. Transferring the control within the same function just requires the manipulation of content of the Instruction Pointer and does not require the above tasks to be carried out. Since the goto statement just manipulates the content of the Instruction Pointer and does not carry out the above-mentioned tasks, it cannot be used to transfer the control from one function to another.

35. *Inputs are given to the functions by means of arguments.* main *is also a function. Therefore, can we give inputs to the function* main *by supplying arguments?*

    Yes, inputs to the function main can be given by making use of **command line arguments.**

Code Snippets

Determine the output of the following code snippets. Assume that the inclusion of the required standard header files has been made and there is no prototyping error due to them. Prototypes of user-defined functions are explicitly mentioned, if required.

36.

```
main()
{
    int a;
    a=printf("Hello")+printf("Readers!!");
    printf("\n%d characters printed",a);
}
```

37.
```
main()
{
    int a=10,b=20,c;
    c=add(a,b);
    printf(“The result after addition is %d”,c);
}
int add(int a, int b)
{
    return a+b;
}
```

38.
```
main()
{
    int add(int,int),a,b;
    a= b=10;
    printf(“The result of addition is %d”,add(a,b));
}
int add(int a,int b)
{
    return a+b;
}
```

39.
```
 int add(int,int);
main()
{
    int a=10,b=10,c;
    c=add(a,b);
    printf(“The result after addition is %d”,c);
}
int add(int a, int b)
{
    return a+b;
}
```

40.
```
main()
{
    int add(int,int),a,b,c;
    a=10;b=20;
    c=add(a,b);
    printf(“The result of addition is %d”,add(a,b));
}
int add(int a, b)
{
      return a+b;
}
```

41.
```
main()
{
    int add(int,int),a,b,c;
    a=10; b=20;
    c=add(a,b);
    printf(“The result of addition is %d”,c);
    int add(int a,int b)
```

```
          {
              return a+b;
          }
     }
42.  void fun(int a)
     {
          printf(“The value of a inside fun is %d\n”,a);
     }
     main()
     {
          int a=10,b;
          b=fun(a);
          printf(“The value of b after call to fun is %d”,b);
     }
43.  fun(int a)
     {
          printf(“The value of a inside fun is %d”,a);
     }
     main()
     {
          int a=10,b;
          b=fun(a);
          printf(“\nThe value of b after call to fun is %d”,b);
     }
44.  fun(int a)
     {
          printf(“The value of a inside fun is %d\n”,a);
           a+2;
     }
     main()
     {
          int a=10,b;
          b=fun(a);
          printf(“The value of b after call to fun is %d”,b);
     }
45.  int add(int,int);
     main()
     {
          int a=10,b=20,c;
          c=add(a,b);
          printf(“The result after addition is %d”,c);
     }
     int add(int a, int b)
     {
          a+b;
     }
46.  int add(int,int);
     main()
```

```
    {
        int a=10,b=20,c;
        c=add(a,b);
        printf("The result after addition is %d",c);
    }
    int add(int a, int b)
    {
        a+b;
        return;
    }
47. int add(int a,int b)
    {
        return a+b;
    }
    main()
    {
        int c;
        c=add(10);
        printf("The result after addition is %d",c);
    }
48. int add(int a,int b=12)
    {
        return a+b;
    }
    main()
    {
        int c;
        c=add(10);
        printf("The result after addition is %d",c);
    }
49. int add(int a,int b=12)
    {
        return a+b;
    }
    main()
    {
        int c;
        c=add(10,20);
        printf("The result after addition is %d",c);
    }
50. int add(int a=12,int b)
    {
        return a+b;
    }
    main()
    {
        int c;
        c=add(10,20);
        printf("The result after addition is %d",c);
    }
```

51.
```
int swap(int a,int b)
{
     a^=b^=a^=b;
     printf(“The values of a and b in swap are %d %d\n”,a,b);
}
main()
{
     int a=10,b=20;
     printf(“This is illustration of pass by value\n”);
     printf(“The values of a and b before swap are %d %d\n”,a,b);
     swap(a,b);
     printf(“The values of a and b after swap are %d %d\n”,a,b);
}
```

52.
```
int swap(int *a,int *b)
{
     *a^=*b^=*a^=*b;
     printf(“The values of a and b in swap are %d %d\n”,*a,*b);
}
main()
{
     int a=10,b=20;
     printf(“This is illustration of pass by reference or address\n”);
     printf(“The values of a and b before swap are %d %d\n”,a,b);
     swap(&a,&b);
     printf(“The values of a and b after swap are %d %d\n”,a,b);
}
```

53.
```
int sum_diff(int a,int b)
{
     int sum=a+b;
     int diff=a-b;
     return sum,diff;
}
main()
{
     int a=20,b=10;
     printf(“Sum is %d and Difference is %d\n”,sum_diff(a,b),sum_diff(a,b));
}
```

54.
```
int sum_diff(int a,int b)
{
     int sum=a+b;
     int diff=a-b;
     return sum, return diff;
}
main()
{
     int a=20,b=10;
     printf(“Sum is %d and Difference is %d\n”,sum_diff(a,b),sum_diff(a,b));
}
```

55.

```
int sum_diff(int a,int b)
{
    int sum=a+b;
    int diff=a-b;
    return sum;
    return diff;
}
main()
{
    int a=20,b=10;
    printf(“Sum is %d and Difference is %d\n”,sum_diff(a,b),sum_diff(a,b));
}
```

56.

```
sum_diff(int a,int b,int *sum,int *diff)
{
    *sum=a+b;
    *diff=a-b;
}
main()
{
    int a=20,b=10,sum,diff;
    sum_diff(a,b,&sum,&diff);
    printf(“Sum is %d and Difference is %d\n”,sum,diff);
}
```

57.

```
fun1()
{
    return printf(“Control is in Function1\n”);
}
fun2()
{
    return printf(“Control is in Function2\n”);
}
main()
{
    printf(“%d %d”,fun1(),fun2());
}
```

58.

```
fun1()
{
    return printf(“Control is in Function1\n”);
}
fun2()
{
    return printf(“Control is in Function2\n”);
}
main()
{
    printf(“%d”,fun1()+fun2());
}
```

```
59. int fact(int no)
    {
        if(no==1)
            return 1;
        else
            return no*fact(no-1);
    }
    main()
    {
        int temp;
        temp=fact(4);
        printf(“The value of factorial of 4 is %d”, temp);
    }
60. main()
    {
        printf(“Infinite Recursion\n”);
        main();
    }
61. check_ptr(int [2][3]);
    main()
    {
        int arr[2][3]={1,2,3,4,5,6};
        printf(“Size of arr in function main is %d\n”,sizeof(arr));
        check_ptr(arr);
    }
    check_ptr(int arr[2][3])
    {
        printf(“Size of arr in function check is %d”,sizeof(arr));
    }
62. int add(int a,int b)
    {
        return a+b;
    }
    main()
    {
        int (*ptr)(int,int);
        ptr=add;
        printf(“The result of addition is %d\n”,ptr(2,3));
        printf(“The result of addition is %d”,(*ptr)(2,3));
    }
63. int add(int a,int b)
    {
        return a+b;
    }
    int sub(int a,int b)
    {
        return a-b;
    }
    int mul(int a,int b)
```

```
{
     return a*b;
}
int div(int a,int b)
{
     return a/b;
}
main()
{
     int (*ptr[4])(int,int)={add,sub,mul,div};
     int i;
     for(i=0;i<4;i++)
          printf("The result of called function %d is %d\n",i+1,ptr[i](10,5));
}
```

64.

```
int add(int,int);
int sub(int,int);
fun(int(*)(int,int));
main()
{
     printf("%d\n",fun(add));
     printf("%d",fun(sub));
}
fun(int (*a)(int,int))
{
     return a(2,3);
}
int add(int a,int b)
{
     return a+b;
}
int sub(int a,int b)
{
     return a-b;
}
```

65.

```
int add(int a,int b){return a+b;}
int sub(int a,int b){return a-b;}
int mult(int a,int b){return a*b;}
int div(int a,int b){return a/b;}
int (*f_returning_fps(int))(int,int);
main()
{
     int i=1, j=3, res1,res2;
     res1=f_returning_fps(i)(15,5);
     printf("Result of operation1 is %d\n",res1);
     res2=f_returning_fps(j)(15,5);
     printf("Result of operation2 is %d\n",res2);
}
int(*f_returning_fps(int a))(int,int)
```

```
{
    int (*arr[4])(int,int)={add,sub,mult,div};
    return arr[a];
  }
```

Multiple-choice Questions

66. A function can return

a. No value
b. Only one value
c. Two values
d. As many values as the user likes

67. By default, the return type of a function is

a. `char`
b. `int`
c. `float`
d. `void`

68. A function can be

a. Defined within another function
b. Declared within another function
c. Both defined as well as declared within another function
d. None of these

69. Which of the following can be a possible return type of a function?

a. Array type
b. Function type
c. Pointer type
d. All of these

70. Which of the following is not a valid parameter type for a function?

a. Array type
b. Function type
c. Pointer type
d. None of these

71. A function that calls itself within its own body is called

a. Mutually recursive
b. Indirect recursive
c. Direct recursive
d. None of these

72. The changes made in the parameters in the called function are reflected to the calling function. The probable method of argument passing is:

a. Pass by value
b. Pass by reference
c. Any of pass by value or pass by reference
d. None of these

73. The method used to pass an array to a function is

a. Value
b. Reference
c. Cannot be passed to functions
d. None of these

74. Which of the following is a definite advantage of recursion over iteration?

a. Better execution speed
b. Saving in memory space
c. Ease of expression
d. None of these

75. The declaration statement `int *ptr(int,int);` declares `ptr` to be a

a. Pointer to a function that accepts two integers and returns an integer
b. A function that accepts two integers and returns a pointer to an integer
c. Pointer to an array of two integers
d. None of these

76. The execution of a program

a. Always starts with main function
b. Starts with the function that is defined first
c. Can start from any function
d. None of these

77. The type of a function depends upon

a. Its return type
b. Types of its parameters
c. Its return type and types of its parameters
d. None of these

78. The values given to a function at the time of making the function call are called

a. Actual arguments
b. Formal arguments
c. Formal parameters
d. None of these

79. The statement that is used to terminate the execution of a function is

a. break statement
b. return statement
c. continue statement
d. exit function call statement

80. main is a

a. User-defined function
b. Library function
c. Pre-defined function
d. None of these

81. In the C statement, a=f1(1,2)+f2(2,3)/f3(3,4);, the order in which functions f1, f2 and f3 are called is

a. f1, f2, f3
b. f2, f3, f1
c. f3, f2, f1
d. Random order

82 In the C statement, a=f1(1,2),f2(2,3),f3(3,4);, the order in which functions f1, f2 and f3 are called is

a. f1, f2, f3
b. f2, f3, f1
c. f3, f2, f1
d. Random order

83. In the C statement, printf("%d %d %d",f1(1,2),f2(2,3),f3(3,4));, the order in which functions f1, f2 and f3 are called is

a. f1, f2, f3
b. f3, f2, f1
c. Random order
d. The order is unspecified and is compiler dependent

84. The number of times Infinite recursion is printed by the following C program is

```
main()
{
    printf("Infinite recursion\n");
    main();
}
```

a. Infinite number of times
b. 32767 times
c. Till the run-time stack does not overflow
d. 65535 times

85. Which of the following is a variable argument function?

a. printf
b. puts
c. gets
d. strcpy

Outputs and Explanations to Code Snippets

36.
```
HelloReaders!!
14 characters printed
```

Explanation:

The printf function call is a valid expression. The printf function returns an integer value equal to the number of characters it prints. Hence, printf("Hello") prints Hello and returns 5. Similarly, printf("Readers!!) prints Readers!! and returns 9. The values returned by the printf functions are summed up and the final value is assigned to the integer variable a. The value of a is printed by the next printf statement.

37.
```
Compilation error "Call to undefined function 'add' in function main()"
```

Explanation:

A function needs to be defined or declared before it is called. In the given piece of code, function add is neither defined nor declared before it is called. Hence, the compiler will not be able to perform type-checking and therefore issues an error message.

38.
```
The result of addition is 20
```

Explanation:

Refer the explanation given in Answer number 3.
It is valid to declare a function within the body of some other function. The function add is declared within the body of the function main before its call. Upon invocation, function add returns the result of the addition of the values of a and b, i.e. 20. The returned result is printed by the printf function.

39.
```
The result of addition is 20
```

Explanation:

The only constraint about the place of declaration of a function is that it should be before its call. The declaration can be either in the local scope or in the global scope. In the given piece of code, the function add has been declared in the global scope.

40.
```
Compilation error
```

Explanation:

Shorthand declaration of the parameters in the parameter list is not allowed and this leads to the compilation error. The rectified declaration of the parameter list is as follows:

```
int add(int a, int b)
{......}
```

41.
```
Compilation error
```

Explanation:

Refer the explanation given in Answer number 3.
A function can be declared but cannot be defined within the body of some other function. In the given piece of code, function add is defined within the body of the function main. This is not valid and leads to the compilation error.

42.
```
Compilation error
```

Explanation:

Refer Section 8.4.3.2. (Point 2).
The return type of the function fun is void. It will not return any value. If it does not return any value, how can the returned value be assigned to b? Hence, writing b=fun(a); is erroneous and leads to the compilation error.

43.
```
The value of a inside fun is 10
The value of b after call to fun is 31
```

Explanation:

The return type of the function fun is not specified and by default will be considered as int. The function fun is expected to return an integer value but no return statement is used inside its body to return a value. **If no return statement is used inside the body of a function to return a value, then by default it returns the content of the accumulator register (AX). The content of the accumulator register is the result of the last computation.** The printf function prints a string and returns a value equal to the number of characters it prints. Therfore, after the execution of printf function, the content of the accumulator register will be the value returned by the printf function, i.e. 31. The content of the accumulator register will be returned by the function fun, will be assigned to the variable b and will be printed later.

Try changing the number of characters in the string given to the function printf in the function fun and observe the values of b.

✍ The content of the accumulator register can be observed by **tracing** the program and looking at its content in the **register window**. In Borland TC 3.0, register window can be opened by going to the Window menu and invoking the Register option. In Borland TC 4.5, register window can be opened by going to the View menu and invoking the Register option.

44.
```
The value of a inside fun is 10
The value of b after call to fun is 12
```

Explanation:

Refer the explanation given in Answer number 43.
The last computation performed in the function fun is a+2. After the execution of this computation, content of the accumulator would be 12. As no return statement is used in the function fun, it returns the content of the accumulator register, i.e. 12.

45.
```
The result after addition is 30
```

Explanation:

Since no return statement is present, the result of the last computation that is present in the accumulator register (i.e. result of a+b) is returned.

46. Compilation error

    Explanation:

    The first form of the return statement (i.e. return;) can only be used if the return type of the function is void. In the given code, the return type of the function add is int, so the second form of the return statement, i.e. return expression; should have been used instead of return;.

47. Compilation error "Too few parameter in call to add(int,int) in function main"

    Explanation:

    Function add is a fixed argument function and expects two arguments. As it is called with only one argument, i.e. 10, there is a mismatch in the number of arguments and the number of parameters. Therefore, the compiler issues an error message.

48. The result after addition is 22

    Explanation:

    There will be no compilation error as in Question number 47. If a function provides a default argument for a parameter, then it can be invoked with or without an argument for that parameter. In the given piece of code, the default argument (i.e. 12) is provided for the parameter b. Hence, it is not mandatory to provide an argument for the parameter b.

49. The result after addition is 30

    Explanation:

    If an argument corresponding to the parameter with the default argument is provided in a function call, it overrides the value of the corresponding default argument. In the given piece of code, function add is called with two arguments, i.e. 10 and 20. The value 20 overrides the default argument value. Hence, the value of b in the function add will be 20. Thus, the value returned by the function add will be 30 and it gets printed by the printf function.

50. Compilation error "Default value missing following parameter a"

    Explanation:

    A function declaration can specify default arguments for all or for a subset of parameters. If default arguments are specified only for the subset of parameters, then they should be specified for the parameters that lie on the trailing side. Hence, it is not possible to specify the default argument for the parameter a unless and until the default argument for the parameter b is specified.

51.
    ```
    This is illustration of pass by value
    The values of a and b before swap are 10 20
    The values of a and b in swap are 20 10
    The values of a and b after swap are 10 20
    ```

    Explanation:

    Since the values of a and b are passed by value, the changes made in the values of the parameters inside the called function are not reflected to the calling function.

52.
    ```
    This is illustration of pass by reference or address
    The values of a and b before swap are 10 20
    The values of a and b in swap are 20 10
    The values of a and b after swap are 20 10
    ```

    Explanation:

    Since the values of a and b are passed by reference, the changes made in the values pointed to by the parameters inside the called function are reflected to the calling function.

53. Sum is 10 and Difference is 10

Explanation:

A function can return only one value. It seems that return sum,diff; returns the value of both sum and diff. However, it is not true. In the statement return sum,diff;, the return expression sum,diff is evaluated first and then its outcome is returned. The comma operator involved in the expression guarantees left to right evaluation and returns the result of the rightmost sub-expression. Therefore, the return expression sum,diff evaluates to the result of the evaluation of diff. Hence, both the calls to function sum_diff, returns the value of diff, i.e. 10. That is why the output comes out to be Sum is 10 and Difference is 10.

54. Compilation error "Expression syntax in function main"

Explanation:

return is a statement and not an expression. It cannot be used as an operand of any operator. Writing return sum, return diff; is not valid as return statement is an operand of comma operator. It should be either return sum; return diff; or return sum, diff;.

55. Sum is 30 and Difference is 30

Explanation:

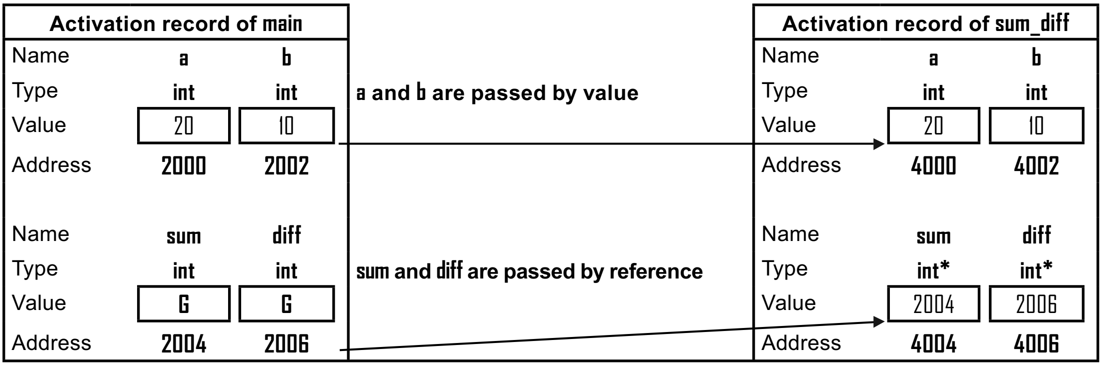
A function can have more than one return statement within its body. If more than one return statement is present inside the body of a function, only the return statement that appears first in the logical flow of control gets executed. In the given piece of code, the statement return sum; appears first in the logical flow of control. Therefore, it gets executed and the control along with the value of sum is returned to the calling function, i.e. main. The statement return diff; will never be executed and forms an unreachable part of the code. Hence, both the calls to the function sum_diff, return the value of sum, i.e. 30. That is why the output comes out to be Sum is 30 and Difference is 30.

56. Sum is 30 and Difference is 10

Explanation:

By making the use of the return statement, a function can return only one value. However, it is possible to indirectly get more than one result from a function either by using global variables or pass by reference. In the given piece of code, pass by reference is used to indirectly get two outputs from the function sum_diff.

Suppose, the variables a, b, sum and diff that are local to the function main are allocated at the memory locations 2000, 2002, 2004 and 2006, respectively. The parameters declared in the header of the function sum_diff are local to the function sum_diff and are allocated at separate memory locations, say 4000, 4002, 4004 and 4006 respectively. Note that the type of the variables sum and diff in the function main is int while the type of variables sum and diff in the function sum_diff is int*. The variables a and b are passed by value while the variables sum and diff are passed by reference. The passed values and the execution of statements are shown in the following figure:



> **i** G (in the above figure) means garbage.

The variables in the statement *sum=a+b; refer to the local variables of the function sum_diff. This statement places the result of addition of a and b, i.e. 30 at the memory location 2004, i.e. in the sum variable of the function main. Similarly, *diff=a+b, places the difference of a and b, i.e. 10 at the memory location 2006, i.e. in the diff variable of the function main. In this way, the function sum_diff has indirectly returned two values to the calling function, i.e. main. Thus, reference to the variables sum and diff in the function main after the execution of the function sum_diff gives 30 and 10, respectively, instead of garbage values.

57.
```
Control is in Function2
Control is in Function1
24 24
```

Explanation:

The comma operator guarantees left-to-right evaluation, but the commas separating the arguments in a function call are not comma operators. If the commas separating the arguments in a function call are considered as comma operators, then no function could have more than one argument. Hence, arguments are not guaranteed to be evaluated from left to right. The order of evaluation of arguments in a function call is unspecified and is compiler dependent. In Borland TC 3.0 and TC 4.5, the evaluation takes place from right to left.

58.
```
Control is in Function1
Control is in Funciton2
48
```

Explanation:

The expression fun1()+fun2() gets evaluated first and the result of its evaluation is printed. The operands of + operator are evaluated from left to right. Hence, the function fun1 is called first and then the function fun2 is called.

59.
```
The value of factorial of 4 is 24
```

Explanation:

Refer Section 8.5.5.3.1.1 for the answer.

60.

```
Infinite Recursion
Infinite Recursion
Infinite Recursion
Infinite Recursion ...
```

Caution:

Keeps on printing 'Infinite Recursion' till the run-time stack does not overflow.

Explanation:

The given piece of code, if executed using Turbo C 3.0, keeps on printing 'Infinite Recursion' till the run-time stack does not overflow. The run-time stack overflows when a large number of activation records are stacked up and there is no memory space left for creating and stacking new activation records. Once the run-time stack overflow occurs, the program will terminate. That is why it is said that **'Infinite recursion will automatically terminate but infinite iteration will not'**. Note that in Turbo C 4.5, it is not allowed to call the function main from within the function main.

61. Size of arr in function main is 12
Size of arr in function check is 2 (In Borland Turbo C 4.5 the output will be 4)

Explanation:

arr declared inside the body of the function main is a two-dimensional array of integers having two rows and three columns. The parameter arr declared in the header of the function check_ptr as int arr[2][3] implicitly gets converted to int (*arr)[3], i.e. pointer to an integer array of size 3. That is why, the size occupied by arr in the function main is 12, and in the function check_ptr is 2 (as a pointer takes two bytes in Borland TC 3.0 irrespective of the data to which it points).

62.

```
The result of addition is 5
The result of addition is 5
```

Explanation:

The declaration statement int(*ptr)(int,int); declares ptr as a pointer to a function that accepts two integers and returns an integer. The assignment statement ptr=add; assigns the starting address of the function add to the pointer ptr. The function add can be invoked by the means of pointer by either writing ptr(2,3); or (*ptr)(2,3);, where 2 and 3 are the values of the arguments to the function add.

63.

```
The result of called function 1 is 15
The result of called function 2 is 5
The result of called function 3 is 50
The result of called function 4 is 2
```

Explanation:

The declaration statement int (*ptr[4])(int,int)={add,sub,mul,div}; declares ptr as an array of pointers to functions that accepts two integers and returns an integer. It also initializes the array locations with the starting addresses of the functions add, sub, mul and div. These functions are called in the loop by writing p[i](10,5), where 10 and 5 are the arguments to the functions. The functions called for the values of i: 0, 1, 2 and 3 are add, sub, mul and div, respectively. The values returned by these functions are then printed.

64.

```
5
-1
```

Explanation:

The declaration fun(int(*)(int,int); declares fun as a function that accepts a pointer to a function that accepts two integers and returns an integer. The return type of fun is not specified and by default would be int. In the function main, fun is called with add as an argument. This means that the starting address of the function add is passed as an argument to the parameter a of the function fun.

Within the body of the function fun, the expression a(2,3), calls the function pointed to by a with 2 and 3 as the arguments. Since a at present points to the function add, the function add is called with the arguments 2 and 3. The value returned by the function add, i.e. 5 is returned by the function fun. Therefore, 5 gets printed. In the next printf statement, the function fun is called with sub as the argument. The starting address of the function sub is passed as an argument to the parameter a of the function fun. The expression a(2,3), calls the function pointed to by a with 2 and 3 as the arguments. Since a now points to the function sub, the function sub is called with the arguments 2 and 3. The value returned by the function sub, i.e. -1 is returned by the function fun and is printed in the function main. Thus, the output.

65.
```
Result of operation1 is 10
Result of operation2 is 3
```

Explanation:

f_returning_fps is a function that takes an integer and returns a pointer to a function that takes two integers and returns an integer. When the function f_returning_fps is invoked with argument values i=1 and j=3, it returns pointers to the functions sub and div, respectively. The returned pointers are used to invoke the respective functions with argument values 15 and 5. The invoked functions sub and div return integer values 10 and 3, respectively. These returned values are assigned to the variables res1 and res2 and are printed by the printf function.

Answers to Multiple-choice Questions

66. b 67. b. 68. b 69. c 70. b 71. c 72. b 73. b 74. c 75. b 76. a 77. c. 78. a 79. b
80. a 81. b 82. a 83. d 84. c 85. a

Programming Exercises

Program 1 | Devise a C function that checks whether a given number is prime or not and illustrate its use

Line PE 8-1.c

```
//Function to check whether a given number is prime or not
#include<stdio.h>
int prime(int no);     //←Function declaration
main()
{
int num;
printf("Enter the number to be checked:\t");
 scanf("%d", &num);
if(prime(num)==0)
    printf("Number is not prime\n");
else
    printf("Number is prime\n");
}
int prime(int no)     //←Function definition
{
    int i;
    for(i=2;i<no;i++)
        if(no%i==0)       //←Is number divisible by any number from 2 to n-1
            return 0;     //←if yes, number is not prime, return 0
    return 1;             //←if no, number is prime, return 1
}
```

Output window

```
Enter the number to be checked:    13
Number is prime
```

Output window (second execution)

```
Enter the number to be checked:    18
Number is not prime
```

Program 2 | Devise a C function that sums all the elements of an array. Illustrate its use

PE 8-2.c

```
//Function that sums all the elements of an array
#include<stdio.h>
int sumall(int array[], int num);      //←Function declaration
main()
{
int num, i, result, elements[20];
printf(“Enter the number of elements in the array (max. 20):\t”);
 scanf(“%d”, &num);
printf(“Enter the elements:\n”);
for(i=0;i<num;i++)
     scanf(“%d”,&elements[i]);
result=sumall(elements, num);
printf(“The sum of all the elements of the array is %d”,result);
}
int sumall(int array[], int num)      //←Function definition
{
     int i,sum=0;
     for(i=0;i<num;i++)
          sum=sum+array[i];
     return sum;
}
```

Output window

```
Enter the number of elements in the array (max. 20)      5
Enter the elements:
10 2 4 7 11
The sum of all the elements of the array is 34
```

Program 3 | Devise a C function that checks whether two matrices can be multiplied or not. If yes, multiply them. Illustrate the use of the developed function

PE 8-3.c

```
//Matrix Multiplication with the help of functions
#include<stdio.h>
#include<stdlib.h>
int mat_multiply(int mx1[][10], int m1, int n1, int mx2[][10], int m2, int n2, int mx3[][10]);
main()
{
int mx1[10][10], mx2[10][10], mx3[10][10]={0};
int m1, n1, m2, n2, i, j, indicator;
printf(“Enter the order of matrix-1 (max. 10 by 10)\t”);
scanf(“%d %d”,&m1, &n1);
printf(“Enter the elements of matrix-1:\n”);
for(i=0;i<m1;i++)
{
     for(j=0;j<n1;j++)
          scanf(“%d”,&mx1[i][j]);
}
printf(“Enter the order of matrix-2 (max. 10 by 10)\t”);
scanf(“%d %d”,&m2, &n2);
printf(“Enter the elements of matrix-2:\n”);
for(i=0;i<m2;i++)
{
     for(j=0;j<n2;j++)
          scanf(“%d”,&mx2[i][j]);
}
```

Output window

```
Enter the order of matrix-1 (max. 10 by 10)    2 3
Enter the elements of matrix-1:
1 2 3
4 5 6
Enter the order of matrix-2 (max. 10 by 10)    3 2
Enter  the elements of matrix-2:
1 2
3 4
5 6
The result of matrix multiplication is:
22 28
49 64
```

Output window (second execution)

```
Enter the order of matrix-1 (max. 10 by 10)    2 3
Enter the elements of matrix-1:
1 2 3
4 5 6
Enter the order of matrix-2 (max. 10 by 10)    2 2
Enter  the elements of matrix-2:
1 2
3 4
Matrices are not compatible for multiplication
```

(*Contd...*)

Line	PE 8-3.c	Output window

```
indicator=mat_multiply(mx1, m1, n1, mx2, m2, n2, mx3);
if(indicator==0)
    printf("Matrices are not compatible for multiplication\n");
else
{
printf("The result of matrix multiplication is:\n");
for(i=0;i<m1;i++)
{
    for(j=0;j<n2;j++)
        printf("%d ",mx3[i][j]);
    printf("\n");
}
}
}
int mat_multiply(int mx1[][10], int m1, int n1, int mx2[][10], int m2, int n2, int mx3[][10])
{
    int i, j, k;
    if(n1!=m2)
        return 0;
    else
    {   for(i=0;i<m1;i++)
            for(j=0;j<n2;j++)
                for(k=0;k<n1;k++)
                    mx3[i][j]=mx3[i][j]+mx1[i][k]*mx2[k][j];
        return 1;
    }
}
```

Program 4 | Merge Sort: Given a list of n elements, arrange them in an ascending order using Merge Sort

Divide-and-conquer is an algorithm design strategy. It works as follows:

1. It checks whether the given instance of problem P is small or not. The given instance is said to be small if it can be easily solved.
2. If the given instance is small, solve it and return the solution. Else, follow the next step.
3. Divide the given instance of problem into smaller sub-problems P_1, P_2, P_3….P_n.
4. Solve the smaller sub-problems recursively by applying divide-and-conquer strategy.
5. Combine the solutions for sub-problems P_1, P_2, P_3….P_n into a solution for P.

Merge Sort is a sorting algorithm that is based on divide-and-conquer strategy. Merge sort works as follows:

1. The size of the given list is determined.
2. If it is 0 or 1 (i.e. it is a small problem), then the list is already sorted. Otherwise, for the lists of the size greater than 1, follow the next step.
3. The unsorted list is divided into two halves of approximately equal size (i.e. division of problem P into P_1 and P_2).
4. The divided sub-lists are recursively sorted by applying Merge Sort.
5. The sorted sub-lists are merged back into one sorted list.

For example, Merge sort sorts the given unsorted list L as follows:

L

[0]	[1]	[2]	[3]	[4]	[5]
12	1	8	10	5	3

//←The list L is divided at midpoint into two halves L1 and L2

(*Contd...*)

L1			L2		
[0]	[1]	[2]	[3]	[4]	[5]
12	1	8	10	5	3

//←The list L1 is further divided at midpoint into two halves L11 and L12

L11		L12	L2		
[0]	[1]	[2]	[3]	[4]	[5]
12	1	8	10	5	3

//←The list L11 is further divided at midpoint into two halves L111 and L112

L111	L112	L12	L2		
[0]	[1]	[2]	[3]	[4]	[5]
12	1	8	10	5	3

//←The lists L111 and L112 are of size 1 and are already sorted. They are merged to form the sorted list L11

L11		L12	L2		
[0]	[1]	[2]	[3]	[4]	[5]
1	12	8	10	5	3

//←List L12 is of size 1 and is already sorted. The list L11 is also sorted. The sorted lists L11 and L12 are merged to form the sorted list L1

L1			L2		
[0]	[1]	[2]	[3]	[4]	[5]
1	8	12	10	5	3

//←The list L2 is divided at midpoint into two halves L21 and L22

L1			L21		L22
[0]	[1]	[2]	[3]	[4]	[5]
1	8	12	10	5	3

//←The list L21 is further divided at midpoint into two halves L211 and L212

L1			L211	L212	L22
[0]	[1]	[2]	[3]	[4]	[5]
1	8	12	5	10	3

//←The lists L211 and L212 are of size 1 and are already sorted. They are merged to form the sorted list L21

L1			L21		L22
[0]	[1]	[2]	[3]	[4]	[5]
1	8	12	5	10	3

//←List L22 is of size 1 and is already sorted. The list L21 is also sorted. The sorted lists L21 and L22 are merged to form the sorted list L2

L1			L2		
[0]	[1]	[2]	[3]	[4]	[5]
1	8	12	3	5	10

//←Both the lists L1 and L2 are sorted. They are merged to form the sorted list L

L					
[0]	[1]	[2]	[3]	[4]	[5]
1	3	5	8	10	12

(*Contd...*)

PE 8-4.c

```c
//Merge Sort
#include<stdio.h>
int mergesort(int list[], int high, int low);
int merge(int num[], int low, int mid, int high);
main()
{
    int list[20], num, i;
    printf("Enter the number of elements (max. 20)\t");
    scanf("%d",&num);
    printf("Enter the elements:\n");
    for(i=0;i<num;i++)
        scanf("%d",&list[i]);
    mergesort(list, 0, num-1);
    printf("After sorting, elements are:\n");
    for(i=0;i<num;i++)
        printf("%d\n",list[i]);
}
int mergesort(int list[], int low, int high)
{
    int mid;
    if(low<high)
    {
        mid=(low+high)/2;
        mergesort(list, low, mid);
        mergesort(list, mid+1, high);
        merge(list, low, mid, high);
    }
}
int merge(int list[], int low, int mid, int high)
{
    int temp[20], k;
    int h=low, i=low, j=mid+1;
    while((h<=mid) && (j<=high))
    {
        if(list[h]<=list[j])
        {
            temp[i]=list[h];
            h=h+1;
        }
        else
        {
            temp[i]=list[j];
            j=j+1;
        }
        i=i+1;
    }
    if(h>mid)
        for(k=j;k<=high;k++)
        {
            temp[i]=list[k];
```

Output window

```
Enter the number of elements(max. 20)    6
Enter the elements:
12
10
5
-3
14
2
After sorting, elements are:
-3
2
5
10
12
14
```

(*Contd...*)

```
            i++;
        }
    else
        for(k=h;k<=mid;k++)
        {
            temp[i]=list[k];
            i++;
        }
    for(k=low;k<=high;k++)
        list[k]=temp[k];
    return 0;
}
```

Program 5 | Quick Sort: Given a list of n elements, arrange them in ascending order using Quick sort

Quick Sort is another efficient sorting algorithm that is based on the divide-and-conquer strategy. In Merge Sort, the list was divided at its midpoint into sub-lists that were independently sorted and later merged. In Quick Sort, the division into two sub-lists is made so that the sorted sub-lists do not need to be merged later. This can be accomplished by picking up an element in the list known as the **pivot element**. The elements of the list are rearranged, so that all the elements that are less than the pivot element come towards the left of the pivot element and all the elements greater than the pivot element come after it (i.e. towards its right). This rearrangement is known as **partitioning**. After partitioning, the pivot element is at its final position. The sub-list of lesser elements (i.e. towards the left of pivot element) and greater elements (i.e. towards the right of pivot element) are recursively sorted by using Quick Sort.

Partitioning:

C.A.R. Hoare, the developer of the Quick Sort algorithm, used the following approach to partition a list:

1. Consider the first element of the list as the pivot element.
2. Rearrange the elements of the list so that the pivot element is moved to its final position. This rearrangement can be done as follows:

a. Suppose the given list is:

[0]	[1]	[2]	[3]	[4]	[5]
12	1	8	10	5	3

b. At the end of the list, append an element that is greater than all the elements present in the list.

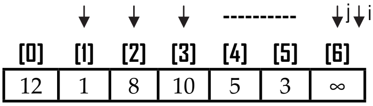
[0]	[1]	[2]	[3]	[4]	[5]	[6]
12	1	8	10	5	3	∞

c. The first element of the unsorted list is the pivot element. Take two pointers, say i and j. The pointer i points to the pivot element and the pointer j points to the appended largest element.

↓i						↓j
[0]	[1]	[2]	[3]	[4]	[5]	[6]
12	1	8	10	5	3	∞

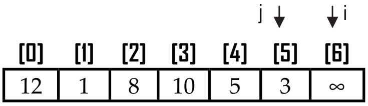
d. Increment the pointer i, till a value greater than the pivot element is encountered. Decrement the pointer j, till a value smaller than the pivot element is encountered. If the pointer i is towards the left of pointer j (i.e. i<j), swap the values pointed to by them else swap the value pointed to by the pointer j with the pivot element. After this process, the pivot element will be at its final position.

(*Contd...*)



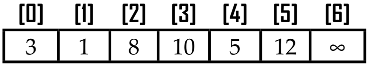
[0]	[1]	[2]	[3]	[4]	[5]	[6]
12	1	8	10	5	3	∞

//←The pointer i is moved till element greater than the pivot element is encountered. Since there is no element greater than the pivot element, the pointer i will stop at the appended largest element. If ∞ would have not been appended, the pointer i would have strayed into garbage field.



[0]	[1]	[2]	[3]	[4]	[5]	[6]
12	1	8	10	5	3	∞

//←The pointer j points to the element lesser than the pivot element.



[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	1	8	10	5	12	∞

//←Since pointer j is towards the left of pointer i, swap the pivot element with the element pointed to by j. The pivot element comes to its final position.

The pivot element 12 has moved to its final position. It divides the list into two sub-lists. One containing the elements lesser than the pivot element and one containing elements greater than the pivot element (empty in this case). This clearly indicates that the divided sub-list may have a significantly different size. The divided sub-lists are recursively sorted by using Quick Sort.

PE 8-5.c

```
//Quick Sort
#include<stdio.h>
int quicksort(int list[], int high, int low);
int partition(int num[], int low, int high);
int swap(int list[], int i, int j);
main()
{
    int list[21], num, i;
    printf("Enter the number of elements (max. 20)\t");
    scanf("%d",&num);
    printf("Enter the elements:\n");
    for(i=0;i<num;i++)
        scanf("%d",&list[i]);
    list[num]=10000;
    quicksort(list, 0, num-1);
    printf("After sorting, elements are:\n");
        for(i=0;i<num;i++)
            printf("%d\n",list[i]);
}
int quicksort(int list[], int low, int high)
{
    int pos;
    if(low<high)
    {
        pos=partition(list, low, high+1);
        quicksort(list, low, pos-1);
        quicksort(list, pos+1, high);
    }
}
int partition(int list[], int low, int high)
```

Output window

```
Enter the number of elements(max. 20)    6
Enter the elements:
12
10
5
-3
14
2
After sorting, elements are:
-3
2
5
10
12
14
```

Remark:

- In the given code it is assumed that the elements entered in the array will be less than 10000

(Contd...)

```
{
    int v=list[low], i=low, j=high;
    do
    {
        do
        {
            i++;
        }while(list[i]<v);
        do
        {
            j--;
        }while(list[j]>v);
        if(i<j)
            swap(list, i, j);
    }while(i<j);
    list[low]=list[j];
    list[j]=v;
    return j;
}
int swap(int list[], int i, int j)
{
    int temp;
    temp=list[i];
    list[i]=list[j];
    list[j]=temp;
    return 0;
}
```

Program 6 | Binary search: Given a list of n elements arranged in ascending order and a key, find whether the given key exists in the list or not. If it exists, print its position in the list

Binary search is an efficient searching algorithm based on the divide-and-conquer strategy. It is based on the assumption that the elements of the list are arranged in an ascending order. Similar to the linear search, it works by comparing the key with the elements of the list, but with a difference in the pattern of making comparisons.

In the binary search, initially the key is compared with the element present at the middle position of the list. If both are equal, the key is found and the search is finished. If the key is less than the middle element, search the key in the list present towards the left of the middle element. If the key is greater than the middle element, search the key in the list present towards the right of the middle element.

Line **PE 8-6.c**

```
//Binary Search
#include<stdio.h>
int binarysearch(int list[], int low, int high, int key);
main()
{
int list[20], num, i, key, low, high, index;
printf("Enter the number of elements (max. 20)\t");
scanf("%d",&num);
printf("Enter the elements in ascending order:\n");
for(i=0;i<num;i++)
```

Output window

```
Enter the number of elements(max. 20)    6
Enter the elements in ascending order:
10
15
32
48
92
128
Enter the key that you want to search   48
48 exists at location no. 4
```

(*Contd...*)

```
  scanf(“%d”,&list[i]);          //←Read elements in the list
printf(“Enter the key that you want to search\t”);
scanf(“%d”,&key);                //←Read the key to be searched
index=binarysearch(list, 0, num-1, key);
if(index==-1)
    printf(“%d does not exist in the list”,key);
else
    printf(“%d exists at location no. %d\n”,key, index+1);
}
int binarysearch(int list[], int low, int high, int key)
{
    int  mid;
    if(low==high)      //←if low==high, there is only one element
    {
        if(list[low]==key)      //←if that element is equal to key
            return low;         //←return its index
        else                    //←else key is not present in the list
            return -1;          //←return -1 as it is  not a valid index value
    }
    else
    {
        mid=(low+high)/2;     //←middle position is found
        if(list[mid]==key)      //←if element at middle position=key
            return mid;         //←return the index of middle location
       else if(list[mid]>key) //←if key<middle element, search left portion of the list
                return binarysearch(list, low, mid-1, key);
             else                  //←search the right portion of the list
                return binarysearch(list, mid+1, high, key);
    }
}
```

Output window (second execution)

```
Enter the number of elements(max. 20)    6
Enter the elements in ascending order:
10
15
32
48
92
128
Enter the key that you want to search    50
50 does not exist in the list
```

Test Yourself

1. Fill in the blanks in each of the following:

a. ____________help in modularizing a program into smaller simple parts.
b. The execution of a C program always begins with function ____________.
c. The expressions that appear within the parentheses of a function call are known as ____________.
d. The two ways of passing arguments to a function are ____________ and ____________.
e. The variables declared in the parameter declaration list in the function header are known as ____________.
f. The first argument to the `printf` function should be of ____________ type.
g. The return type of each math library function is ____________ .
h. The return type of a function cannot be ____________.
i. ____________ is a special case of recursion in which the last operation of a function is a recursive call.
j. By default, the return type of a function is ____________.
k. Execution of each function requires a separate ____________.
l. The activation records for all of the active functions are stored in the region of memory called ____________.
m. The part of recursion in which a number of activation records are created and piled up is known as ____________.

2. State whether each of the following is true or false. If false, explain why.

a. C is a strongly typed language.
b. `main` is a library-defined function.
c. There can be only one `return` statement within a function body.
d. `printf` is an example of a variable argument function.
e. The function designator implicitly refers to the starting address of the function.
f. The `return` statement is used to terminate the execution of a program.
g. A function can be defined within the body of another function, and the function defined within another function is known as nested function.
h. Directly recursive functions are also known as mutually recursive functions.
i. A function need not be declared, if it is defined before it is called.
j. The shorthand declaration of parameters in the parameter list is not allowed.
k. One of the uses of function prototype is in type checking.
l. If the arguments are passed by reference, the changes made in the values pointed to by the formal parameters in the called function are reflected to the calling function.
m. A function can return only one value.

3. Programming exercises:

a. Write a C function that checks whether a given number is even or odd. Illustrate its use.
b. Write a C function that checks whether a given number is perfect or not. Illustrate its use.
c. Write a recursive C function to find the sum of individual digits of a given positive integer number.
d. Write a C function that finds the reverse of a given number.
e. Write a C function that checks whether a given number is a palindrome or not.
f. Write a C function that checks whether a given number is an Armstrong number or not.
g. Write an iterative C function to print the first `n` terms of a Fibonacci series. Get the value of `n` from the user.
h. Write a recursive C function to print the first `n` terms of a Fibonacci series. Get the value of `n` from the user. Illustrate its use.

i. Write an iterative C function that finds the value of x^n. Get the values of x and n from the user. Illustrate its use.

j. Write a recursive C function that finds the value of x^n. Get the values of x and n from the user. Illustrate its use.

k. Write a recursive C function that implements linear search. Given a list of n elements and a key. Using the developed function, check whether the given key exists in the list or not. If yes, print the position at which it exists in the list.

l. Write an iterative C function that finds the factorial of a given integer. Use this function to find ${}^{n}_{r}C = \frac{n!}{r!(n-r)!}$

m. Write a recursive C function that finds the factorial of a given integer. Use this function to find ${}^{n}_{r}C = \frac{n!}{r!(n-r)!}$

n. Write a C function to evaluate the following series. Use a function to compute the factorials. Get the value *x* and the number of terms in the series from the user:

i. $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots \infty$

ii. $\cosh(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots \infty$

iii. $\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots \infty$

iv. $e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{n!}$

o. Write a C function that finds the sum of all the elements of a matrix. Illustrate the use of this function.

p. Write a C function that checks whether a given matrix is symmetric or not. Illustrate its use.

q. Write a C function that finds the sum of elements of the principal diagonal of a matrix. Illustrate the use of this function.

r. Write a C program that extracts the lower-triangular matrix from a square matrix. Illustrate the use of the developed function in a program.

s. Write a C function that finds the largest and the smallest element in a matrix. Illustrate the use of the developed function in a program.

t. Write a C function that swaps the contents of two one-dimensional arrays. Do not use any additional storage space. Illustrate the use of the developed function in a program.

u. Given n boolean variables x_1, x_2, x_3 ….. x_n. We wish to print all the possible combinations of the truth values that they can assume. For instance, if n is equal to 2, there are four possibilities 00, 01, 10 and 11. Write a C program to accomplish this task.

v. Write a C program to implement ternary search. The ternary search works on the following strategy:
Given a sorted list of n elements in ascending order. First, test the element at the location n/3 for equality with the given key x. If they are found to be equal, print that the given key is found at the location n/3, else compare it with the element at the location 2n/3. If they are found to be equal, print that the given key is found at location 2n/3, else reduce the size of the list to one-third and search the given key in the reduced list.

PART – V

STRUCTURES AND UNIONS

This page is intentionally left blank

9

STRUCTURES AND UNIONS

Learning Objectives

In this chapter, you will learn about:

- User-defined data types
- Structures
- How to define new data types using structures
- How to declare objects of the newly created structure type
- Various operations that can be applied on the objects of a structure type
- Arrays, pointers, functions and structures used in conjunction
- Creating syntactically convenient name for user-defined types
- Unions
- Difference between structures and unions
- Application of unions in interrupt programming
- Enumerations
- Storing information less than a byte by making use of bit-fields

9.1 Introduction

In previous chapters, you have seen that C language provides a rich set of primitive and derived data types for the efficient storage and manipulation of data. In case these data types do not suit your requirements, C language also provides the flexibility to create new data types. These data types are known as **user-defined data types** and can be created by using structures, unions and enumerations. In chapter 6, you have learnt that arrays can be used for the storage of homogeneous data. However, they cannot be used for the storage of data of different types. The data of different types can be grouped together and stored by making use of structures. One of the similarities between arrays and structures is that both of them contain a finite number of elements. Thus, array types and structure types are collectively known as aggregate types.

Unions are similar to structures in all aspects except the manner in which their constituent elements are stored. In structures, separate memory is allocated to each element, while in unions all the elements share the same memory.

Enumerations help you in defining a data type whose objects can take a limited set of values. These values are referred to by names, known as enumerators, which are more convenient to handle. In this chapter, I will tell you how to define new data types using structures, unions and enumerations. I will also let you know how to declare and manipulate objects of these newly defined data types.

9.2 Structures

A **structure** is a collection of variables under a single name and provides a convenient way of grouping several pieces of related information together. Unlike arrays, it can be used for the storage of heterogeneous data (i.e. data of different types). There are three aspects of working with structures:

1. Defining a structure type, i.e. creating a new type
2. Declaring variables and constants (i.e. **objects**) of the newly created type
3. Using and performing operations on the objects of the structure type

9.2.1 Defining a Structure

The general form of **structure-type definition** (or just **structure definition**) is:

```
[storage_class_specifier][type_qualifier] struct [structure_tag_name]
    {
        type member_name1[, member_name11, ...];
        [type member_name2[, member_name22, ...]];
         .........
    } [variable_name];
```

The important points about structure definition are as follows:

1. The terms enclosed within the square brackets are optional and might not be present in a structure definition statement. However, the terms shown in **bold** are the mandatory parts of the structure definition.
2. A structure definition consists of the keyword struct followed by an optional identifier name, known as **structure tag-name**, and a **structure declaration-list** enclosed within the braces. The examples of the structure definition given in Table 9.1 are valid.

Table 9.1 | Structure definitions with and without tag-name

```
struct book     //←Structure tag-name is book
{
    char title [25];    //←Structure declaration-list
    char author[20];
    int pages;
    float price;
};
```

(a)

```
struct  //←Structure tag-name not present
{
    char title[25];     //←Structure declaration-list
    char author[20];
    int pages;
    float price;
};
```

(b)

3. The structure definition defines a **new type**, known as **structure type**. For example, in Table 9.1(a) the structure type is struct book. After the definition of the structure type, the keyword struct is used to declare its variables.
4. Since the tag-name of a structure is an identifier, all the rules discussed in Section 3.5.1 for writing an identifier name are applicable for writing the structure tag-name. If the tag-name is present, it will act as a **name** for the **newly created data type**.
5. The newly created type (i.e. tag name of the defined structure) is **visible**, after its definition, only in the scope in which it is defined. Hence, it is not possible to declare objects of the defined structure type outside the scope in which it (i.e. its tag name) is visible. The piece of code in Program 9-1 illustrates this fact.

Prog 9-1.c

```
//The defined structure type is visible only in the scope in which it is defined
#include<stdio.h>
func();             //←Declaration of the function func
main()
{
  struct coord  //←The type struct coord is defined in the scope local to the function main
  {
      int x,y;
  };
  struct coord pt1, pt2;  //←Declaring variables pt1 and pt2 of the created type struct coord
//←Other statements in the function main
//← ...............................
}
func()
{
struct coord pt3;     //←The tag name coord is not visible here
//←Other statements in the function func
//← ...............................
}
```

Output window

```
Compilation errors
"Undefined structure 'coord' in function func()"
"Size of 'pt3' is unknown or zero in function func()"
```

Remarks:

- Since the structure coord is defined in the function main, it is visible only in the function main
- It is possible to declare the variables of this newly created type in the scope local to the function main, but not outside this scope
- Hence, the declaration of the variable pt3 of type struct coord in the scope local to the function func leads to the compilation error

Program 9-1 | A program to illustrate that a defined structure type is visible only in the scope in which it is defined

6. The newly created type is **incomplete**✍ until the closing brace of the structure declaration-list is encountered. The newly created type is **complete** thereafter.
7. The structure declaration-list consists of declarations of one or more variables, possibly of different types. The variable names declared in the structure declaration-list are known as **structure members** or **fields**. Structure members can be variables of the basic types (e.g. char, int, float, etc.), pointer types (e.g. char*, etc.) or **aggregate type**✍ (i.e. arrays or other structure types). They are declared in the same way as normal identifiers are declared.

✍

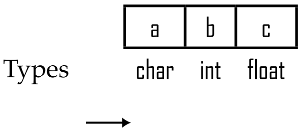
- An **incomplete type** describes an object but lacks the information needed to determine its size. Due to the lack of information about the size, an object of incomplete type cannot be created.
- **Array type and structure type** are collectively known as **aggregate type**.

8. A structure declaration-list cannot contain a member of void type or **incomplete type** or function type. Hence, **a structure definition cannot contain an instance of itself**. However, it may contain a pointer[†] to an instance of itself. Such a structure is known as a **self-referential structure**.
9. In principle, a structure definition can have an infinite number of members. However, practically the number of members in a single structure definition depends upon the translation limits of the compiler.

The interpretation of the above-mentioned rules is shown in Table 9.2.

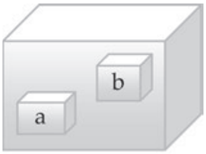
Table 9.2 | Rules regarding the types of structure members

	Form of data	Structure definition

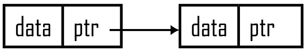
a.		struct record
		{ //← Structure declaration-list consists of variables of different types
		char a;
		int b;
		float c;
		};
		(Valid)
		A structure can have data of different types

(*Contd...*)

[†] Refer Section 9.3 for a description on pointers to structures.

b.	A box contains two boxes
	
	
		struct box
		{
		struct box a; //←Type struct box is incomplete until the closing brace is enco-
		struct box b; // -untered. Hence, a member of type struct box cannot be created
		}; //← Type struct box is complete this point onwards
		(Invalid)
		A structure cannot contain an instance of itself

c.	A name consists of two names: first name and last name.
	first_name last_name
	A phonebook entry consists of the name of a person and his mobile number.
	person_name mobile_no
		struct name
		{
		char first_name[20];
		char last_name[20];
		}; //←Type struct name is complete now onwards
		struct phonebook_entry
		{
		struct name person_name; //←Member of complete type struct name
		char mobile_no[10]; // can be created
		};
		(Valid)
		A structure can contain members of other complete types

d.	A node of a linked list consists of integer data and a pointer to a node.
	
	
	Node 1 Node 2
	Linked list
		struct node
		{
		int data;
		struct node* ptr; //← Structure contains a pointer to an instance of itself.
		}; // This is an example of a self-referential structure
		(Valid)
		A structure can contain a pointer to itself

10. It is possible to use the shorthand declaration to declare two or more structure members of the same type. The examples of the structure definition given in Table 9.3 are valid.

Table 9.3 | Shorthand declaration used to declare structure members of the same type

struct book
{
char title [25], author[20]; //←Shorthand declaration
int pages;
float price;
};
(a)
	struct two_dimensional_coordinate
	{
	int x,y; //← Shorthand declaration
	};
	(b)

11. The name of a structure member can be the same as the structure tag-name without any conflict, since they can always be distinguished from the context. However, the names of two structure members in a structure declaration-list can never be the same.

12. Two different structure types may contain members of the same name without any conflict.
13. It is important to note that a **structure definition does not reserve any space in the memory**.

> A structure definition does not reserve any memory space for the structure members in the **data segment** but since structure definition becomes a part of the program code, it takes some space in the **code segment**.

14. Since structure definition does not reserve any memory space for the structure members, it is not possible to initialize the structure members during the structure definition. The structure definitions in Table 9.4 are not valid.

Table 9.4 | Initialization of structure members is not allowed during the structure definition

```
struct book
{
    char title [30]="India 2020: A Vision for the new millennium ";
    char author[20]="A P J Abdul Kalam";
    int pages=400;
    float price=225.50;
};
```

(a)

```
struct two_dimensional_coordinate
{
    int x=0;
    int y;
};
```

(b)

15. If a structure definition does not contain a structure tag-name, the created structure type is **unnamed**. The unnamed structure type is also known as an **anonymous** structure type. It is not possible to declare its objects (i.e. variables and constants) after its definition. Thus, the objects of unnamed or anonymous structure type should be declared only at the time of structure definition.

    The declaration of the structure variables at the time of unnamed structure definition is given in Table 9.5.

Table 9.5 | Declaration of structure variables at the time of structure definition

```
struct
{
    char title [25];
    char author[20];
    int pages;
    float price;
} book1;
//←Declaration of structure variable book1
```

(a)

```
struct
{
    int x;
    int y;
} pt1, pt2;  //←Declaration of structure variables pt1, pt2
```

(b)

The declaration of structure constants at the time of unnamed structure definition is given in Table 9.6.

Table 9.6 | Declaration of structure constants at the time of structure definition

```
const struct
{
    char title [25];
    char author[20];
    int pages;
    float price;
} book={"Programming C", "Anirudh", 450, 225.50};
//←Creation of qualified constant book
```

(a)

```
struct
{
    char title [25];
    char author[20];
    int pages;
    float price;
} const book={"Programming C", "Anirudh", 450, 225.50};
//←Creation of qualified constant book
```

(b)

i It is always better to provide a structure tag-name while creating a structure type. The tag-name is convenient for declaring the variables and constants of the defined structure type later in the program.

16. A structure-type definition can optionally have a storage class specifier and type qualifiers. However, the type qualifiers and storage class specifier (except typedef[†]) should only be used in a structure definition if the structure objects are also declared at the same time. The piece of code in Program 9-2 illustrates this fact.

Prog 9-2.c

```
//Use of storage class specifier while defining a structure type
#include<stdio.h>
static struct point
{
    int x;
    int y;
};
main()
{
    struct point pt1;
    //←Other statements
 }
```

Output window

Compilation error "Storage class 'static' not allowed here".

Remark:

- The storage class specifiers except typedef should not be used in a structure-type definition if the objects are not declared at the time of structure definition

Program 9-2 | A program illustrating that a storage class specifier except typedef should not be used while defining a structure type if its objects are not declared at the same time

17. Since a structure definition is a statement, it must always be terminated with a semicolon.

9.2.2 Declaring Structure Objects

Variables and constants (i.e. objects) of the created structure type can be declared (actually defined) either at the time of structure definition or after the structure definition. The declaration of variables and constants at the time of structure definition has been discussed in Section 9.2.1. Variables and constants of the created structure type can be created after the structure

[†] Refer Section 9.7 for a description on using typedef storage class specifier in structure definition or with structure object declaration.

definition only if the defined structure type is **named** or **tagged**. The general form of declaring structure objects is:

[storage class specifier] [type_qualifier] **struct named_structuretype identifier_name** [=intialization_list [,...]]**;**

The important points about the structure object declaration are as follows:

1. The terms enclosed within the square brackets are optional and might not be present in a structure object declaration statement. The terms shown in **bold** are the mandatory parts of the structure object declaration.
2. A structure object declaration consists of:

   i. The keyword struct for declaring structure variables. It can also be used in conjunction with const qualifier for declaring structure constants.
   ii. The tag-name of the defined structure type.
   iii. Comma-separated list of identifiers (i.e. variable names or constant names). A variable can optionally be initialized by providing an initializer. However, initialization of a constant is must.
   iv. A terminating semicolon.

   The following structure variable declarations are valid:

   ```
   struct book c_book, algorithm_book;  //←Structure type book defined in Table 9.1(a)
   struct phonebook_entry entry;        //←Structure type phonebook_entry defined in Table 9.2(c)
   struct two_dimensional_coordinate pt1={2,3}, pt2; //← Structure type two_dimensional_coordinate de-
                                                     //  fined in Table 9.3(b). The structure vari-
                                                     //  able pt1 //is initialized.
   ```

   The following structure constant declarations are valid:

   ```
   const struct book c_book, algorithm_book={"C Programming", "Anirudh", 450, 225.50};
   const struct phonebook_entry entry={{"Mohit","Virmani"}, "1234567899"};
   const struct two_dimensional_coordinate pt1={2,3}, pt2={4,5};
   ```

3. Note that, in C language, the objects of the defined structure type cannot be declared without using the keyword struct. However, this rigidity is relaxed in C++ language. If it is inconvenient to use the keyword struct every time to declare an object of the defined structure type, use the storage class specifier typedef[§] to create a syntactically convenient alias name for the defined structure type so that the keyword struct need not be used again and again.
4. Upon the declaration of a structure object, the amount of the memory space allocated to it is equal to the sum of the memory space required by all of its members. For example, the amount of memory allocated to a variable of the structure type struct book defined in Table 9.1 (a) is 51 bytes (if the integer takes 2 bytes) or 53 bytes (if the integer takes 4 bytes). The number of bytes of the memory space occupied by an object of the structure type also depends upon how members of the structure object are stored[¶] in the memory. The memory space allocated to a structure object can be determined by using the sizeof operator.
5. The structure members are assigned memory addresses in increasing order, with the first structure member starting at the beginning address of the structure itself. This can be checked by applying address-of operator on a structure object and its members as

[§] Refer Section 9.7 for a description on the usage of typedef storage class specifier with structures.

[¶] Refer Section 9.2.3.1.3 for a description on the alignment of structure members.

done in Program 9-7 in Section 9.2.3.1.3. Whether structure members are stored in consecutive memory locations or not, depends upon how the members of a structure object are aligned (refer footnote¶ on previous page).

6. **Initializing members of a structure object:** Like variables and array elements, the members of a **structure object** can also be initialized at the compile time. The syntactic rules about structure member initialization are as follows:
   i. The members of a structure object can be initialized by providing an **initialization list**. An initialization list is a comma-separated list of initializers.
   ii. The order of initializers must match the order of structure members in the structure definition.
   iii. The type of each initializer should be the same as the type of corresponding structure member in the structure definition. If the type of an initializer is not the same as the type of the corresponding structure member, implicit type casting will be done if types are compatible. If types are not compatible, there will be a compilation error.
   iv. The number of initializers in an initialization list can be less than the number of members in a structure object and if it happens, the leading structure members (i.e. occurring first) will be initialized with the initializers in the initialization list. The rest of the members will automatically be initialized with 0 (if they are of integer type), 0.0 (if they are of floating point type), '\0' (if they are of char type) and null pointer (if they are of pointer type). This rule is recursively applied to initialize all the elements/members of a structure member (if it is of aggregate type).
   v. Nested structures and arrays can be initialized by using nested braces.

   Examples of structure member initialization are as follows:

```
struct book c_book={"My Life", "C Motilal", 400, 210.50};
struct phonebook_entry entry={{"Rajesh","Kumar"}, "9814000561"};
struct two_dimensional_coordinate pt1={2}, pt2={2,3};
```

i It is important to note that **the structure members cannot be initialized during the structure definition**; however, **the members of a structure object can be initialized by providing an initialization list**.

7. A structure object declaration can optionally have a type qualifier. If the type qualifiers are used while declaring a structure object, they are applied to all the members of the structure object. The piece of code in Program 9-3 illustrates this fact.

Line	Prog 9-3.c	Output window

1
2
3
4
5
6
7
8
9
	//Using type qualifiers while declaring a structure object
	#include<stdio.h>
	struct point
	{
	    int x;
	    int y;
	};
	main()
	{
		Compilation errors "Cannot modify a constant object in function main()"
		Remarks:
		• To access the members of a structure object, the member access operator, i.e. dot operator is used. Refer Section 9.2.3.1.1 for a description on how to access members of a structure object using the dot operator

(*Contd...*)

Prog 9-3.c

```
    const struct point pt={2,3};
    pt.x=20;
    pt.y=40;
    //←Other statements...
}
```

Output window

- The type qualifier const is applied to all the members of the structure object. Hence, the type of pt.x and pt.y is const int and thus, it is not possible to place them on the left side of the assignment operator

Program 9-3 | A program that illustrates the use of type qualifiers while declaring a structure object

8. A structure object declaration can optionally have a storage class specifier. The important points about the usage of a storage class specifier in a structure object declaration are as follows:

    i. If a structure object is declared with a storage class specifier other than typedef, the properties resulting from the storage class specifier except with respect to linkage, also apply to the members of the object, and so on recursively for any aggregate member object present in the structure definition. The piece of code in Program 9-4 illustrates this fact.

Prog 9-4.c

```
//Declaring structure object with a storage class specifier other than
//typedef
#include<stdio.h>
struct point
{
    int x;
    int y;
};
main()
{
    struct point pt1;
    static struct point pt2;
    printf("The coordinates of pt1 are %d,%d\n", pt1.x, pt1.y);
    printf("The coordinates of pt2 are %d,%d\n", pt2.x, pt2.y);
}
```

Output window

```
The coordinates of pt1 are 9495,19125
The coordinates of pt2 are 0,0
```

Remarks:

- Since the structure object pt1 is local to the function main and is not initialized, its members contain garbage values
- Since the structure object pt2 is declared with static storage class qualifier, all the properties resulting from it except linkage, are applicable to all the members of the structure object pt2
- Thus, all the members of the structure object pt2 are initialized to zero, since static storage class specifier has been used
- To access the members of a structure object, the member access operator, i.e. dot operator is used. Refer Section 9.2.3.1.1 for a description on how to access members of a structure object using the dot operator

Program 9-4 | A program that illustrates the declaration of a structure object with a static storage class specifier

    ii. The structure objects declared with register storage class specifier are treated as automatic (i.e. auto) objects.

9.2.3 Operations on Structures

The operations that can be performed on an object (i.e. variable or constant) of a structure type are classified into two categories:

1. Aggregate operations
2. Segregate operations

9.2.3.1 Aggregate Operations

An aggregate operation treats an operand as an entity and operates on the entire operand as a whole instead of operating on its constituent members. The four aggregate operations that can be applied on an object of a structure type are as follows:

1. Accessing members of an object of a structure type
2. Assigning a structure object to a structure variable
3. Address of a structure object
4. Size of a structure (i.e. either structure type or a structure object)

9.2.3.1.1 Accessing Members of an Object of a Structure Type

The members of a structure object can be accessed by using:

1. Direct member access operator (i.e. **.**, also known as **dot operator**).
2. Indirect member access operator[††] (i.e. ->, also known as **arrow operator**).

The important points about the use of a dot operator are as follows:

1. The dot operator accesses a structure member via structure object name while the arrow operator accesses a structure member via a pointer to the structure. The general form of using a dot operator is:

   `structure_object_name.structure_member_name`

2. The dot operator is a binary operator.
3. The first operand of the dot operator should have qualified or unqualified structure type and the second operand should be the name of a member of that type. The piece of code in Program 9-5 illustrates the use of the dot operator.

Line	Prog 9-5.c	Output window
1	`//Use of dot operator`	Enter values of translation vector: 4 2 After translation, coordinates are: Pt1 (8,7) Pt2 (6,5) **Remarks:** • In line number 15, the first operand of each dot operator is of unqualified structure type (i.e. struct coord)
2	`#include<stdio.h>`	
3	`struct coord          //←Definition of type struct coord`	
4	`{                     //← Creation of new type for 2-D coordinate`	
5	`int x,y;`	
6	`};`	
7	`main()`	
8	`{`	
9	`    struct coord pt1={4,5};     //←pt1 is a variable of type struct coord`	
10	`    const struct coord pt2={2,3};  //←pt2 is a qualified constant of type struct coord`	
11	`    int tx, ty;`	

(*Contd...*)

[††] Refer Section 9.3.2 for a description on indirect member access operator.

Line	Prog 9-5.c	Output window
12	printf(“Enter values of translation vector:\n”);	• In line number 16, the first operand of each dot operator is of qualified structure type (i.e. const struct coord)
13	scanf(“%d %d”,&tx, &ty);	
14	printf(“After translation, coordinates are:\n”);	
15	printf(“Pt1 (%d,%d)\n”, pt1.x+tx, pt1.y+ty);	
16	printf(“Pt2 (%d,%d)\n”, pt2.x+tx, pt2.y+ty);	
17	}	

Program 9-5 | A program to illustrate the use of a direct member access operator

9.2.3.1.2 Assigning a Structure Object to a Structure Variable

Like simple variables, a structure variable can be assigned with or initialized with a structure object (i.e. variable or constant) of the same structure type. The piece of code in Program 9-6 illustrates the assignment and initialization of a structure variable.

Line	Prog 9-6.c	Output window
1	//Initialization and assignment of a structure variable	Cutting Stone by Abraham is of Rs. 200
2	#include<stdio.h>	Cutting Stone by Abraham is of Rs. 200
3	struct book //←Structure definition	Cutting Stone by Abraham is of Rs. 200
4	{	**Remarks:**
5	char title[25];	• In line number 12, the structure variable b1 is initialized by providing an initialization list
6	char author[20];	
7	int price;	
8	};	
9	main()	• In line number 14, the structure variable b2 is initialized with the structure variable b1
10	{	
11	//Initializing a structure variable by providing an initialization list	
12	struct book b1={“Cutting Stone”, “Abraham”, 200}:	
13	//Initializing a structure variable with another structure variable	• In line number 17, the structure variable b2 is assigned to the structure variable b3
14	struct book b2=b1;	
15	// Declaring an uninitialized structure variable	
16	struct book b3;	• The assignment operator copies the values of all the members of a structure object present on its right side to the corresponding members of a structure variable present on its left side
17	b3=b2; //←Assigning a structure variable to a structure variable	
18	printf(“%s by %s is of Rs. %d rupees\n”, b1.title, b1.author, b1.price);	
19	printf(“%s by %s is of Rs. %d rupees\n”, b2.title, b2.author, b2.price);	
20	printf(“%s by %s is of Rs. %d rupees\n”, b3.title, b3.author, b3.price);	
21	}	• Hence, printing the values of members of all the three structure variables gives the same result

Program 9-6 | A program that illustrates the initialization and assignment of a structure variable

The important points about the structure variable assignment are as follows:

1. Unlike arrays, a structure variable can be assigned with or initialized with a structure object of the same type. If the type of assigning or initializing structure object is not the

same as the type of structure variable on the left side of the assignment operator, there will be a compilation error. Note that it is not even possible to explicitly type cast a structure type to another structure type.

2. The assignment operator assigns (i.e. copies) values of all the members of the structure object on its right side to the corresponding members of the structure variable on its left side one by one. Hence, the assignment operator, when applied on structure variables performs **member-by-member copy**.
3. The structure assignment does not copy any padding bits.[‡‡]
4. Due to member-by-member copy behavior of the assignment operator on the structure variables, structure objects can be passed to functions[§§] by value and can also be returned from functions.

9.2.3.1.3 Address-of a Structure Object

The address-of operator when applied on a structure object gives its base (i.e. starting) address. It can also be used to find the addresses of the constituting members of a structure object. The piece of code in Program 9-7 illustrates the use of the address-of operator on a structure object and its constituting members.

Prog 9-7.c

```c
//Address-of operator and structures
#include<stdio.h>
struct complex
{
int re, im;
};
main()
{
  struct complex c1={2,3};
  const struct complex c2={4,5};
  printf("Address of c1 is %p\n",&c1);
  printf("Address of its real part is %p\n",&c1.re);
  printf("Address of its imaginary part is %p\n",&c1.im);
  printf("Address of c2 is %p\n",&c2);
  printf("Address of its real part is %p\n",&c2.re);
  printf("Address of its imaginary part is %p\n",&c2.im);
}
```

Memory contents

c1

c1.re	c1.im
2	3
222C	222E

c2

c2.re	c2.im
4	5
223C	223E

Output window

```
Address of c1 is 233F:222C
Address of its real part is 233F:222C
Address of its imaginary part is 233F:222E
Address of c2 is 233F:223C
Address of its real part is 233F:223C
Address of its imaginary part is 233F:223E
```

Remarks:

- The memory allocation is purely random, and the result of the execution may vary for executions at different times or on different machines
- The address of the first structure member is the same as the address of the structure object
- Thus, the first structure member starts at the beginning address of the structure itself

Program 9-7 | A program that illustrates the use of the address-of operator on structures

The output of the address-of operator depends upon how the members of a structure object are stored in the memory. There are two different ways of storing the members of a structure object:

[‡‡] Refer Section 9.2.3.1.3 for a description on structure padding.

[§§] Refer Section 9.6.2 for a description on passing structure objects to functions by value.

a. **Byte aligned:** If the members of a structure object are byte aligned, then every structure member starts from a new byte (i.e. they can appear at any byte boundary). In byte alignment, the data members are stored next to each other. Storage of members of a structure variable using byte alignment is shown in Figure 9.1.

<table>
<tr><th>Definition</th><th>Memory contents</th></tr>
<tr><td><pre>struct type
{
    char a;
    int b;
    char c;
    float d;
}var;</pre></td><td>var<table>
<tr><td>a</td><td colspan="2">b</td><td>c</td><td colspan="4">d</td></tr>
<tr><td>1001</td><td>1011</td><td>1100</td><td>1001</td><td>1111</td><td>1101</td><td>1010</td><td>1000</td></tr>
<tr><td>2000</td><td>2001</td><td>2002</td><td>2003</td><td>2004</td><td>2005</td><td>2006</td><td>2007</td></tr>
</table>
<b>In byte alignment, data members are placed next to each other</b>
char takes 1 byte, int takes 2 bytes and float takes 4 bytes in the memory
Note that only 4 bits are shown in the cells above but actually 8 bits are present in each cell</td></tr>
</table>

Figure 9.1 | Storage of the members of a structure object using byte alignment

b. **Machine-word boundary aligned:** Most of the machines access objects of certain types faster if they are aligned properly. In order to increase the performance of the code on such machines, the compiler aligns the members of a structure object with the storage boundaries whose **addresses are multiple of their respective sizes**. This is shown in Figure 9.2.

<table>
<tr><th>Definition</th><th>Memory contents</th></tr>
<tr><td><pre>struct type
{
    char a;
    int b;
    char c;
    int d;
}var;</pre></td><td>var<table>
<tr><td>a</td><td></td><td colspan="2">b</td><td>c</td><td></td><td></td><td>d</td></tr>
<tr><td>1001</td><td>H</td><td>1011</td><td>1100</td><td>1001</td><td>H</td><td>1111</td><td>1101</td></tr>
<tr><td>2400</td><td>2401</td><td>2402</td><td>2403</td><td>2404</td><td>2405</td><td>2406</td><td>2407</td></tr>
</table>
<b>H represents holes.</b>
char takes 1 byte, int takes 2 bytes and float takes 4 bytes in the memory</td></tr>
</table>

Figure 9.2 | Storage of members of a structure object using machine-word boundary alignment

The character members can appear at any byte boundary (since the size of character type is 1). Let us assume that the structure member a of the type struct type (as shown in Figure 9.2) gets allocated at the memory address 2400. Since the size of the integer type is 2, the member b must appear immediately at the next even-byte boundary. Thus, the memory location 2401 is not a valid start location for the structure member b. Hence, it starts from the storage boundary with the memory address 2402. Similarly, the next two members of the structure object var are stored.

The vacant spaces (as shown in Figure 9.2) in between the members of a structure, if they are machine-word boundary aligned, are known as **holes**. The holes contain random bytes known as **padding bytes**. Thus, the process by which the C compiler inserts unused bytes after the structure members to ensure that each member is appropriately aligned is called **structure padding**. Consider another example given in Figure 9.3.

Definition	Memory contents

struct newtype
{
char a;
double b;
}var;
	var

a								b							
1001	H	H	H	H	H	H	H	1101	0010	1101	1101	0010	1001	1011	1100
2400	**2401**	**2402**	**2403**	**2404**	**2405**	**2406**	**2407**	**2408**	**2409**	**240A**	**240B**	**240C**	**240D**	**240E**	**240F**

H represents holes.
char takes 1 byte and double takes 8 bytes

Figure 9.3 | Another example of storing a structure object using machine-word boundary alignment

Let us assume that the character member a of the type struct newtype is stored at the memory location 2400. The size of the member b is 8 bytes, since it is of type double. Hence, it can only start from a storage boundary whose address is a multiple of 8. Thus, the structure member b is placed at the memory address 2408 and there are seven holes (i.e. padding bytes) between members a and b.

A character object can be allocated at any memory address and have no alignment requirement. Thus, if in Figure 9.3, the character member a gets allocated at the memory address 2405 instead of the memory address 2400, the number of padding bytes required would have been two and if it gets allocated at 2407, no padding byte would have been required. Does it mean that the number of padding bytes required to store objects of a given structure type is variable?
No, for a given compiler and the underlying hardware configuration, the number of padding bytes required to store objects of a given structure type is fixed. A structure member whose address requirement is a higher multiple than another is said to have **stricter alignment**. Thus, in Figure 9.3, the member b has stricter alignment than the member a. Also, **each structure object must be as strictly aligned as its most strictly aligned member**. Thus, an object of the structure type defined in Figure 9.3 should be as strictly aligned as its member b and can only start from the memory locations that are divisible by 8. Therefore, the objects of the structure type defined in Figure 9.3 can start from memory addresses like 2400, 2408, etc. Hence, if a structure object starts from any of these memory locations, the number of padding bytes required would be 7.

Interestingly, if you think that 7 bytes are too much to be wasted for padding, you can place a limit on the amount of padding that can be done by the compiler. The amount of padding can be restricted by setting[¶¶] a **pack size** value. By default, the pack size in Turbo C 3.0 and 4.5 is 2 and is 4 in MS-VC++ 6.0. Thus, **if the members of a structure object are machine-word aligned, they can appear at the storage boundaries that have addresses that are either multiple of their respective sizes or the pack size, whichever is smallest**. Therefore, if the structure object shown in Figure 9.3 is stored using Turbo C 3.0/4.5, there will be two holes between the members a and b and if it is stored using MS-VC++6.0, there will be four holes (since pack size is 4) instead of 7.

[¶¶] Refer Section 9.2.3.1.4 for a description on how to set the pack size.

The important points about structure padding are as follows:

i. The members of a structure object are always stored in the order in which they are declared. They will never be reordered to improve the alignment and save padding.

ii. The padding can only appear in between two structure members (i.e. internal padding) or after the last structure member (i.e. trailing padding). In no case can it appear before the first member of the structure object. The reason behind placing the padding bytes after the last member of the structure object is to enable the alignment in an array of structures. Consider the structure type and an array object defined in Figure 9.4.

<table>
<tr><th>Definition</th><th colspan="9">Memory contents</th></tr>
<tr><td rowspan="6">struct ntype
{
    int a;
    char b;
}var[2];</td><td rowspan="3">var</td><td colspan="4">var[0]</td><td colspan="4">var[1]</td></tr>
<tr><td colspan="2">a</td><td>b</td><td></td><td colspan="2">a</td><td>b</td><td></td></tr>
<tr><td>1001</td><td>1010</td><td>0101</td><td>H</td><td>0100</td><td>1110</td><td>1100</td><td>H</td></tr>
<tr><td></td><td>2400</td><td>2401</td><td>2402</td><td>2403</td><td>2404</td><td>2405</td><td>2406</td><td>2407</td></tr>
<tr><td colspan="9"><b>H represents holes.</b></td></tr>
<tr><td colspan="9">int takes 2 bytes and char takes 1 byte</td></tr>
</table>

Figure 9.4 | Storage of array of structure objects when the members of a structure are machine-word boundary aligned

The member a of the first element of the array var (i.e. first structure object) starts at the even-byte boundary. The member b can be placed at the next byte boundary. Thus, there is no padding between the members a and b of the first structure object. The member a of the second element of the array (i.e. second structure object) must appear at the even-byte boundary. Thus, 2403 is not a valid start location for the member a of the second structure object. Therefore, the compiler places a padding byte at the end of the first structure object so that the second structure object can be aligned properly.

iii. Whether the members of a structure object will be byte aligned or machine-word boundary aligned, depends upon the compiler, its configuration, the working environment and the underlying machine. Some compilers (e.g. Borland TC 3.0 and Borland TC 4.5) use byte alignment by default while some compilers (e.g. MS-VC++ 6.0) by default use machine-word boundary alignment. The pragma directive can also be used to configure[†††] the compiler to use the appropriate alignment scheme for storing the structure members.

9.2.3.1.4 Use of sizeof Operator on Structures

When the sizeof operator is applied to an operand of a structure type, the result is the total number of bytes that an object of such type will occupy in the memory. The important points about the use of a sizeof operator on structures are as follows:

[†††] Refer Section 9.2.3.1.4 for a description on how to configure the compiler to use the appropriate alignment scheme for storing the structure members.

1. The general form of sizeof operator is:
   a. sizeof expression or the sizeof(expression)
   b. sizeof(type i.e structure_type)

The usage of both the forms of sizeof operator on operands of a structure type is given in the code segment listed in Program 9-8.

Prog 9-8.c

```c
//sizeof operator & structures
#include<stdio.h>
struct pad
{
    char a;
    int b;
    char c;
    float d;
};
main()
{
struct pad var;
printf("Objects of type struct pad will take %d bytes\n", sizeof(struct pad));
printf("Structure variable var takes %d bytes\n",sizeof var);
}
```

Output window (Borland TC 3.0)

```
Objects of type struct pad will take 8 bytes
Structure variable var takes 8 bytes
```

Output window (Borland TC 4.5) (second execution)

```
Objects of type struct pad will take 8 bytes
Structure variable var takes 8 bytes
```

Output window (MS-VC++ 6.0) (third execution)

```
Objects of type struct pad will take 16 bytes
Structure variable var takes 16 bytes
```

Remarks:

- In Borland Turbo C 3.0/4.5, character takes 1 byte, integer takes 2 bytes and float takes 4 bytes
- Also, in Borland Turbo C 3.0/4.5, structure members are stored using byte alignment. Hence, there is no padding
- Thus, the sizeof operator gives the output as 1+2+1+4=8 bytes in Borland Turbo C 3.0/4.5
- In Microsoft VC++ 6.0, character takes 1 byte, integer takes 4 bytes and float takes 4 bytes
- Also, in Microsoft VC++ 6.0, the structure members are machine-word boundary aligned and the default pack size is of 4 bytes
- Thus, the sizeof operator outputs 4+4+4+4=16 bytes in Microsoft VC++ 6.0

Program 9-8 | A program that illustrates the use of the sizeof operator on structures

2. The result of the sizeof operator when applied on a structure is equal to the sum of the size of all of its members. It also **includes the space taken by internal and trailing padding.** The pragma directive can be used to turn the structure padding on or off. In Borland Turbo C 3.0 and 4.5, the structure padding can be turned on by using #pragma option –a. Another method to turn on the structure padding in TC 4.5 is by invoking the following menu items:

   options>project>advanced compiler>processor>data alignment>word alignment instead of byte alignment.

The piece of code in Program 9-9 illustrates the use of the pragma directive to configure Borland TC 3.0 and 4.5, so that it stores the structure members using the machine-word boundary alignment.

Line | **Prog 9-9.c**

```
 1  //sizeof operator & structures
 2  #include<stdio.h>
 3  #pragma option –a
 4  struct pad
 5  {
 6      char a;
 7      int b;
 8      char c;
 9      float d;
10  };
11  main()
12  {
13  struct pad var;
14  printf(“Objects of type struct pad will take %d bytes\n”, sizeof(struct pad));
15  printf(“Structure variable var takes %d bytes\n”,sizeof var);
16  }
```

Output window (Borland TC 3.0/4.5)

```
Objects of type struct pad will take 10 bytes
Structure variable var takes 10 bytes
```

Remarks:

- In line number 3, the pragma directive is used to store the structure members using machine-word boundary alignment
- In Borland Turbo C 3.0/4.5, the pack size is 2 bytes
- In Borland Turbo C 3.0/4.5, float takes 4 bytes
- Hence, the sizeof operator gives an output as 2+2+2+4=10 bytes

Program 9-9 | A program to illustrate that the result of the sizeof operator includes internal and trailing padding

The pragma option that can be used to turn off the structure padding in Borland Turbo C 3.0 and 4.5 is #pragma option –a–. The #pragma option -a- is, however, not recognized in MS-VC++ 6.0. To specify the pack size for structures, MS-VC++ 6.0 uses #pragma pack(n) directive, where n is the size according to which the packing will be done. The piece of code in Program 9-10 illustrates the structure packing in MS-VC++ 6.0.

Prog 9-10.c

```
//sizeof operator & structures
#include<stdio.h>
#pragma pack(2)
struct pad
{
    char a;
    int b;
    char c;
    float d;
};
main()
{
struct pad var;
printf(“Objects of type struct pad will take %d bytes\n”, sizeof(struct pad));
printf(“Structure variable var takes %d bytes\n”,sizeof var);
}
```

Output window (MS-VC++ 6.0)

```
Objects of type struct pad will take 12 bytes
Structure variable var takes 12 bytes
```

Remarks:

- In Microsoft VC++ 6.0, #pragma pack(n) is used to specify the pack size
- #pragma pack(2) specifies the pack size to be 2 bytes
- To store the structure members using byte alignment in MS-VC++ 6.0 #pragma pack(1) is used. #pragma pack(1) specifies that members can be placed at any byte boundaries
- #pragma pack(2) specifies that members of size greater than two can be placed at even-byte boundaries
- In Microsoft VC++ 6.0, integer takes 4 bytes
- Thus, the sizeof operator gives output as 2+4+2+4=12 bytes

(*Contd...*)

Memory contents

Alignment with machine-word boundaries, pack size is 2 bytes

var

a		b				c		d			
1001	H	1011	1100	1111	1010	1001	H	1111	1101	1010	1000
2000	2001	2002	2203	2004	2005	2006	2007	2008	2009	2010	2011

(a) #pragma pack(2) used

Alignment with byte boundaries, pack size is 1 byte

var

a	b				c	d			
1001	1011	1100	1111	1010	1001	1111	1101	1010	1000
2000	2001	2002	2003	2004	2005	2006	2007	2008	2009

(b) #pragma pack (1) used

Using MS-VC++ 6.0, char takes 1 byte, int takes 4 bytes and float takes 4 bytes in the memory

Program 9-10 | A program that finds the size of a structure object and a structure type after packing the structure members according to the given pack size

9.2.3.1.5 Equating Structure Objects of the Same Type

The use of an equality operator on the operands of a structure type is not allowed and leads to a compilation error. The piece of code in Program 9-11 illustrates this fact.

Line | **Prog 9-11.c**

```
 1  //Equality operator and structures
 2  #include<stdio.h>
 3  #pragma pack(2)
 4  struct pad
 5  {
 6      char a;
 7      int b;
 8      char c;
 9      float d;
10  };
11  main()
12  {
13  struct pad var1={'A', 2, 'B', 2.5}, var2={'A', 2, 'B', 2.5};
14  if(var1==var2)
15      printf("Structure variables are equal\n");
16  else
17      printf("Structure variables are unequal\n");
18  }
```

Output window (MS-VC++ 6.0)

```
Compilation error "Invalid structure operation in function main"
```

Remarks:

- Since the structure members are not always stored in the contiguous memory locations, the use of the equality operator is not allowed on the objects of a structure type
- Thus, the usage of the equality operator on the structure variables in line number 14 leads to the compilation error

Memory contents

var1

a		b				c		d			
A	1011	2				B	1101	2.5			
2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011

(Contd...)

Memory contents

var2

a		b				c		d			
A	1010	2				B	0001	2.5			
4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011

Program 9-11 | A program to illustrate that the application of the equality operator on the structure objects is not allowed

The important points about the application of an equality operator on the objects of a structure type are as follows:

1. Unlike arrays, the members of a structure object may not be stored in contiguous memory locations. If the members of a structure object are machine-word boundary aligned, there may be some holes in the structure. These holes are filled with **padding**, which is random and undefined. As given in Program 9-11, although the values of all the members of both the structure variables are equal, the structures are not equal because the holes do not contain identical padding.
2. Due to the structure padding, the operation of the equality operator on structures is restricted and this is a general rule. Even if byte alignment is used for storing members of a structure object, in which there are no holes between the structure members, the use of the equality operator on structure objects leads to a compilation error.
3. For similar reasons, the application of relational operators like >=, <=, >, < and != is not allowed on structures.
4. Whether two structure objects are equal or not can be determined by comparing all the members of the structure objects separately. The piece of code in Program 9-12 checks the equality of two structure objects.

Prog 9-12.c

```
//Equality operator and structures
#include<stdio.h>
struct pad
{
    char a;
    int b;
    float c;
};
main()
{
struct pad var1={‘A’, 2, 2.5}, var2={‘A’, 2, 2.5};
const struct pad var3={‘B’,3,5.5}, var4={‘C’,7,9.5};
printf(“Checking equality of structure objects:\n”);
if(var1.a==var2.a && var1.b==var2.b && var1.c==var2.c)
    printf(“Structure variables are equal\n”);
```

Output window (MS-VC++ 6.0)

```
Checking equality of structure objects:
Structure variables are equal
Structure constants are unequal
```

Remarks:

- The equality of structure objects can be checked by equating every member of the structure object
- Specify different initializers in the initialization list of the structure variable var2 and then re-execute the code

(*Contd...*)

```
else
    printf(“Structure variables are unequal\n”);
if(var3.a==var4.a && var3.b==var4.b && var3.c==var4.c)
    printf(“Structure constants are equal\n”);
else
    printf(“Structure constants are unequal\n”);
}
```

Program 9-12 | A program that illustrates a method of determining whether two structure objects are equal

9.2.3.2 Segregate Operations

A segregate operation operates on the individual members of a structure object. The individual members of a structure object are like normal objects (i.e. variables and constants). Therefore, any operation that is applicable on an object of a particular type can be applied on a structure member of that type. The piece of code in Program 9-13 illustrates segregate operations on the members of a structure variable.

Prog 9-13.c

```
//Segregate operations
#include<stdio.h>
struct book
{
    char title[25];
    char author[20];
    int pages;
    float price;
};
main()
{
 struct book book1, book2;
 printf(“Enter title, author name, pages and price of book1:\n”);
 gets(book1.title);
 gets(book1.author);
 scanf(“%d %f”,&book1.pages,& book1.price);
 flushall();
 printf(“Enter title, author name, pages and price of book2:\n”);
 gets(book2.title);
 gets(book2.author);
 scanf(“%d %f”,&book2.pages, &book2.price);
printf(“\nIn second edition, the pages of books are increased by 100\n”);
 printf(“The cost of books is increased by 10%\n\n”);
// Operations on individual members
 book1.pages+=100;
 book2.pages+=100;
 book1.price=book1.price*110/100;
 book2.price=book2.price*110/100;
 printf(“In second edition: book1 has %d pages\n”,book1.pages);
```

Output window

```
Enter title, author name, pages and price of book1:
The Book of Wisdom
Stephen W. K. Tan
480
225
Enter title, author name, pages and price of book2:
Who moved my cheese?
Dr Spencer Johnson
400
210

In second edition, the pages of books are increased by 100
The cost of books is increased by 10%

In second edition: book1 has 580 pages
The second edition of book1 is of Rs. 247.500000
In second edition: book2 has 500 pages
The second edition of book2 is of Rs. 231.000000
```

(*Contd...*)

Prog 9-13.c

```
 printf(“The second edition of book1 is of Rs. %f\n”,book1.price);
 printf(“In second edition: book2 has %d pages\n”,book2.pages);
 printf(“The second edition of book2 is of Rs. %f\n”,book2.price);
}
```

Output window

Program 9-13 | A program that illustrates the operations on the individual members of a structure object

9.3 Pointers to Structures

As pointer to any other type can be created, it is possible to create a pointer to a structure type as well. The pointers to structures have the following advantages:

1. It is easier to manipulate the pointers to structures than manipulating structures themselves.
2. Passing a pointer to a structure as an argument to a function[‡‡‡] is efficient as compared to passing a structure to a function. The size of a pointer to a structure is generally smaller than the size of the structure itself. Thus, passing a pointer to a structure as an argument to a function requires less data movement as compared to passing the structure to a function.
3. Some wondrous data structures (e.g. linked lists, trees, etc.) use the structures containing pointers to structures. Pointers to structures play an important role in their successful implementation.

9.3.1 Declaring Pointer to a Structure

The general form of declaring a pointer to a structure is:

[storage_class_specifier] [type_qualifier] **struct named_structure_type* identifier_name**[=l-value[,...]]**;**

The important points about declaring a pointer to a structure are as follows:

1. The terms enclosed within the square brackets are optional and might not be present in a declaration statement. The terms shown in **bold** are the mandatory parts of a structure pointer declaration statement.
2. A pointer to a structure type can be declared in a separate declaration statement only if the structure type is named. If the structure type is unnamed, the structure pointer should be created at the time of structure definition as shown in Program 9-14.
3. The declared structure pointer can optionally be initialized with an l-value. The initializing l-value should be of appropriate type, else there will be a compilation error.

The piece of code in Program 9-14 illustrates the declaration of a pointer to a structure.

Prog 9-14.c

```
//Pointers to structures
#include<stdio.h>
struct coord
{
    int x, y,z;
```

Output window (Borland TC 4.5)

```
Addresses of pt1 and pt2 are 2397:0076 2397:2292
Addresses of ptr1 and ptr2 are 2397:0D38 2397:228E
ptr1 and ptr2 point to 2397:0076 2397:2292
Size of type (struct coord) is 6
Size of type (struct coord*) is 4
```

(*Contd...*)

[‡‡‡] Refer Section 9.6.3 for a description on passing a pointer to a structure as an argument to a function.

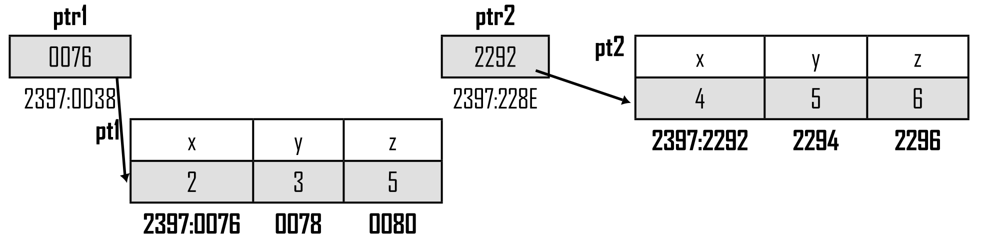
```
}pt1={2,3,5}, *ptr1;     //←Declaration of structure pointer at the
                         //    time of structure definition
main()
{
struct coord pt2={4,5,6};
struct coord *ptr2=&pt2; //←Declaration of structure pointer in a
                             //    separate declaration statement
ptr1=&pt1;
printf("Addresses of pt1 and pt2 are %p %p\n",&pt1,&pt2);
printf("Addresses of ptr1 and ptr2 are %p %p\n",&ptr1,&ptr2);
printf("ptr1 and ptr2 point to %p %p\n",ptr1,ptr2);
printf("Size of type (struct coord) is %d\n", sizeof(struct coord));
printf("Size of type (struct coord*) is %d\n",sizeof(struct coord*));
printf("pt1 and pt2 take %d bytes\n", sizeof(pt1));
printf("ptr1 and ptr2 take %d bytes\n", sizeof(ptr1));
}
```

```
pt1 and pt2 take 6 bytes
ptr1 and ptr2 take 4 bytes
```

Remarks:

- As the memory allocation is purely random, the output may vary for different executions at different times or on different machines
- If executed using Borland Turbo C 3.0, only offset addresses will be printed and the size of struct coord*, ptr1 and ptr2 that gets printed is 2 bytes.

Memory content



Program 9-14 | A program that illustrates the declarations and the use of pointers to a structure type

9.3.2 Accessing Structure Members Via a Pointer to a Structure

The members of a structure object can be accessed via a pointer to a structure object by using one of the following two ways:

1. By using the dereference or indirection operator and the direct member access operator
2. By using the indirect member access operator (i.e. ->, known as the **arrow operator**)

The important points about accessing the structure members, via a pointer to the structure object are as follows:

1. The general form to access a structure member via a pointer to the structure object using the dereference and dot operator is:

   ```
   (*pointer_to_structure_type).structure_member_name
   ```

2. It is mandatory to parenthesize the dereference operator and the structure pointer because the dot operator has a higher precedence than the dereference operator.
3. The members of a structure object can also be accessed via the pointer to the structure object by using only one operator, known as the **indirect member access operator** or **arrow operator**. The general form of such access is:

   ```
   pointer_to_structure_object->structure_member_name
   ```

4. The arrow operator consists of a hyphen (-) followed by a right arrow (>) with no space in between.
5. The expression pointer_to_structure_object->structure_member_name is equivalent to the expression (*pointer_to_structure_object).structure_member_name.

The piece of code in Program 9-15 illustrates the structure member access via the pointer to the structure object.

Line | **Prog 9-15.c**

```
 1  //Accessing structure members via pointer to the structure object
 2  #include<stdio.h>
 3  struct coord
 4  {
 5  int x, y;
 6  };
 7  main()
 8  {
 9  struct coord pt={2,3};
10  struct coord *ptr=&pt;
11  printf("Coordinates of Pt1 are (%d,%d)\n",(*ptr).x, (*ptr).y);
12  printf("Coordinates of Pt1 are (%d,%d)\n",ptr->x, ptr->y);
13  }
```

Output window

```
Coordinates of Pt1 are (2,3)
Coordinates of Pt1 are (2,3)
```

Remarks:

- In line number 11, the structure members are accessed via the pointer to the structure object by using the dereference operator and direct member access operator
- In line number 12, the structure members are accessed by using the indirect member access operator

Program 9-15 | A program that illustrates structure member access via structure pointer

9.4 Array of Structures

It is possible to create an array whose elements are of structure type. Such an array is known as an **array of structures**. Consider the structure type struct book defined in Program 9-13. The information about the title of the book, author's name, number of pages and its price can be stored in a variable of type struct book. We have created the variables book1 and book2 of this type in Program 9-13 to store the specified information about two books. Now, suppose we need to store the information about a number of books available in a library. To store the information about several books, creating a separate variable for each book is not feasible. Here an array of structures provides a convenient way to store the information about the books available in the library. The piece of code in Program 9-16 illustrates the use of array of structures for this purpose.

Line | **Prog 9-16.c**

```
//Array of structures
#include<stdio.h>
#include<conio.h>
#define MAXBOOKS 10
struct book
{
    char title[30];
    char author[30];
    int pages;
```

Output window (Borland Turbo C 4.5)

```
Enter the information of book1:
Enter the title of the book:     The law and the lawyer
Enter the author's name:     M K Gandhi
Enter the number of pages in the book:     200
Enter the price of the book:     125
Do you want to enter more(Y/N):     Y
Enter the information of book2:
Enter the title of the book:     Rise and fall of super powers
Enter the author's name:     Paul
```

(Contd...)

```
10   float price;
11  };
12  main()
13  {
14   struct book library[MAXBOOKS];
15  int count=0,i;
16  char ch;
17  while(1)
18  {
19      printf("Enter the information of book %d:\n", count+1);
20      printf("Enter the title of the book:\t");
21      gets(library[count].title);
22      printf("Enter the author's name:\t");
23      gets(library[count].author);
24      printf("Enter the number of pages in the book:\t");
25      scanf("%d",&library[count].pages);
26      printf("Enter the price of the book:\t");
27      scanf("%f",&library[count].price);
28      flushall();
29      count++;
30      if(count==MAXBOOKS)
31      {
32          printf("Capacity full\n");
33          break;
34      }
35     else
36      {
37          printf("Do you want to enter more(Y/N):\t");
38          ch=getche();
39          printf("\n");
40          if(ch=='y'||ch=='Y')
41              continue;
42          else
43              break;
44      }
45  }
46  printf("\nFollowing are the books in the library:\n\n");
47  for(i=0;i<count;i++)
48  {
49  printf("%s by %s: %d pages is of Rs. %6.2f\n",
50  library[i].title,  library[i].author, library[i].pages,
51  library[i].price);
52  }
53  }
```

```
Enter the number of pages in the book:    250
Enter the price of the book:    150
Do you want to enter more(Y/N):    N

Following are the books in the library:

The law and the lawyer by M K Gandhi: 200 pages is of Rs. 125.00
Rise and fall of great powers by Paul: 250 pages is of Rs. 150.00
```

Remarks:

- In line number 4, macro MAXBOOKS is defined to have value 10
- In line number 5, the structure type struct book is defined
- In line number 14, an array of 10 elements is declared whose element type is struct book
- Elements of this array can be accessed in the same way as the elements of other arrays can be accessed, i.e. by using the subscript operator
- If executed using Borland Turbo C 3.0 or other older compilers, there will be the following error:
  scanf: floating point formats not linked
  Abnormal program termination error

Program 9-16 | A program that illustrates the use of array of structures

The important points about the use of array of structures are as follows:

1. An array of structures is declared in the same way as any other kind of array is declared. The only difference is that the element type of an array of structures is the defined

structure type while the element type of other arrays is either a basic type or a derived type.

2. An array of structures is quite big in size. If it is defined inside the block/local scope without using the static storage class specifier (as in Program 9-16), it is placed on the stack. The **stack** is an area of memory that starts out small and grows automatically up to a predefined limit. It is possible that the default size of a stack is too small to accommodate an array of structures. In such a case, there will be stack overflow run-time error. The following solutions can be used to fix this problem:
   a. Reduce the array size. For example, Program 9-16 executes fine till the size of the array library is kept 60 (in TC 4.5). If the size of the array is further increased, there will be stack overflow run-time error.
   b. Use the storage class specifier static while declaring the array so that it is not stored on the stack.
3. If Program 9-16 is executed using the Borland Turbo C 3.0 compiler, the following error will occur:

```
scanf: floating point formats not linked
Abnormal program termination
```

The reason behind this error is that older Borland compilers like Borland Turbo C 3.0 for DOS attempt at keeping the size of a program compact by using the smaller versions of the scanf function if the program does not use floating point values. However, these compilers get fooled if the floating point values are in an array of structures (as in Program 9-16). The problem can be solved by adding the following lines of code:

```
#include<math.h>
float dummy= cos(0.0);        //←This statement is an executable statement and should
                              //  not be placed in the global scope. It should be placed
                              //  in the local scope after the non-executable statements.
```

The piece of code in Program 9-17 illustrates the usage of the above dummy statement to rectify the problem of the scanf function in Borland Turbo C 3.0 or other older compilers:

Line	Prog 9-17.c	Output window (Borland Turbo C 3.0)
1	//Array of structures	Enter the information of book1:
2	#include<stdio.h>	Enter the title of the book: The law and the lawyer
3	#include<conio.h>	Enter the author's name: M K Gandhi
4	#include<math.h>	Enter the number of pages in the book: 200
5	#define MAXBOOKS 10	Enter the price of the book: 125
6	struct book	Do you want to enter more(Y/N): Y
7	{	Enter the information of book2:
8	char title[30];	Enter the title of the book: Rise and fall of super powers
9	char author[30];	Enter the author's name: Paul Kennedy
10	int pages;	Enter the number of pages in the book: 250
11	float price;	Enter the price of the book: 150
12	};	Do you want to enter more(Y/N): N
13	main	
14	{	Following are the books in the library:
15	struct book library[MAXBOOKS];	

(*Contd...*)

```
int count=0,i;
char ch;
float dummy=cos(0.0);
while(1)
{
    printf(“Enter the information of book %d:\n”, count+1);
    printf(“Enter the title of the book:\t”);
    gets(library[count].title);
    printf(“Enter the author’s name:\t”);
    gets(library[count].author);
    printf(“Enter the number of pages in the book:\t”);
    scanf(“%d”,&library[count].pages);
    printf(“Enter the price of the book:\t”);
    scanf(“%f”,&library[count].price);
    flushall();
    count++;
    if(count==MAXBOOKS)
    {
        printf(“Capacity full\n”);
        break;
    }
   else
    {
        printf(“Do you want to enter more(Y/N):\t”);
        ch=getche();
        printf(“\n”);
        if(ch=='y'||ch=='Y')
            continue;
        else
            break;
    }
}
printf(“\nFollowing are the books in the library:\n\n”);
for(i=0;i<count;i++)
{
printf(“%s by %s: %d pages is of Rs. %6.2f\n”,
library[i].title,  library[i].author, library[i].pages,
library[i].price);
}
}
```

```
The law and the lawyer by M K Gandhi: 200 pages is of Rs. 125.00
Rise and fall of great powers by Paul Kennedy: 250 pages is of Rs. 150.00
```

Remarks:

- The dummy statement float dummy=cos(0.0); is used to load the floating point version of the scanf function
- Addition of this statement removes the problem of the scanf function associated with the usage of floating point values in an array of structures in older compilers like Borland Turbo C 3.0

Program 9-17 | A program that solves the problem of the scanf function associated with the usage of floating values in an array of structures in older compilers like Borland Turbo C 3.0

9.5 Structures within a Structure (Nested Structures)

A structure can be nested within another structure. Nested structures are used to create complex data types. Consider the example of structure type phonebook_entry created in Table 9.2(c). A record in a phone book consists of the fields: name of a person and his mobile number. The field 'name of a person' is a composite field that further consists of a person's first name and his last name. To construct such a type, which consists of composite fields, nested structures are used. The program segment in Program 9-18 illustrates the use of nested structures.

Line | **Prog 9-18.c**

```
 1  //Nested structures
 2  #include<stdio.h>
 3  #include<conio.h>
 4  struct name
 5  {
 6      char first_name[20];
 7      char last_name[20];
 8  };
 9  struct phonebook_entry
10  {
11      struct name person_name;
12      char mobile_no[15];
13  };
14  main()
15  {
16      struct phonebook_entry p1, p2;
17      printf("Enter the details of the first person:\n");
18      printf("Enter the first name of the person:\t");
19      gets(p1.person_name.first_name);
20      printf("Enter the last name of the person:\t");
21      gets(p1.person_name.last_name);
22      printf("Enter the mobile number:\t");
23      gets(p1.mobile_no);
24
25      printf("Enter the details of the second person:\n");
26      printf("Enter the first name of the person:\t");
27      gets(p2.person_name.first_name);
28      printf("Enter the last name of the person:\t");
29      gets(p2.person_name.last_name);
30      printf("Enter the mobile number:\t");
31      gets(p2.mobile_no);
32
33      printf("\nRecords in the phone book are:\n");
34      printf("-------------------------------------\n");
35      printf("%s %s:\t%10s\n",p1.person_name.first_name,
36      p1.person_name.last_name, p1.mobile_no);
37      printf("%s %s:\t%10s\n",p2.person_name.first_name,
38      p2.person_name.last_name, p2.mobile_no);
39  }
```

Output window

```
Enter the details of the first person:
Enter the first name of the person:     Parul
Enter the last name of the person:      Sood
Enter the mobile number:     9882687681
Enter the details of the second person:
Enter the first name of the person:     Rajini
Enter the last name of the person:      Bansal
Enter the mobile number: 9412345980

Records in the phone book are:
-----------------------------------------
Parul Sood :      9882687681
Rajini Bansal:    9412345980
```

Remark:

- As the structure member person_name is of type struct name, the dot operator is applied twice to access its members (as shown in line numbers 35-38)

Program 9-18 | A program that illustrates the use of nested structures

The important points about nested structures are as follows:

1. The nested structures contain the members of other structure types. The structure types used in the structure definition should be complete.

2. It is even possible to define a structure type within the declaration-list of another structure-type definition. The piece of code in Program 9-19 illustrates this fact.

Prog 9-19.c

```
//A structure type defined within another structure type definition
#include<stdio.h>
struct phone_entry
{
    struct name
    {
        char fnam[20];
        char lnam[20];
    } pnam;
    char mno[10];
};
main()
{
struct phone_entry per1={{“Anil”,”Kumar”}}, per2={{“Anand”}};
printf(“Enter the mobile number of %s %s\n”,per1.pnam.fnam, per1.pnam.lnam);
gets(per1.mno);
 printf(“Enter the mobile number of %s %s\n”,per2.pnam.fnam, per2.pnam.lnam);
gets(per2.mno);
printf(“\nPhone book entries are:\n”);
printf(“------------------------\n”);
printf(“%s %s:\t%s\n”, per1.pnam.fnam, per1.pnam.lnam, per1.mno);
printf(“%s %s:\t%s\n”, per2.pnam.fnam, per2.pnam.lnam, per2.mno);
}
```

Output window

```
Enter the mobile number of Anil Kumar
9814456767
Enter the mobile number of Anand
9888852525

Phone book entries are:
------------------------
Anil Kumar     9814456767
Anand     9888852525
```

Remark:

- The structure type struct name is defined within the declaration-list of the structure type struct phone_entry

Program 9-19 | A program illustrating that it is allowed to define a structure type within the structure definition-list of another structure-type definition

3. The member access operator is used to access the members of structure members, e.g. in Program 9-19, the member fnam of the structure member pnam of the structure variable per1 can be assessed by writing per1.pnam.fnam.
4. In principle, structures can be nested infinitely. However, practically the number of levels of nested structure definitions in a single structure definition list depends upon the translation limits of the compiler.

9.6 Functions and Structures

In Chapter 8, you have learnt about functions. You have seen that the flexibility of functions can be increased by passing arguments to functions. In the previous chapters, you have learnt about how to pass variables, arrays and pointers to functions. In this section, I will tell you how to pass structures to a function. The three ways of passing a structure object to a function are as follows:

1. Passing each member of a structure object as a separate argument
2. Passing the entire structure object by value
3. Passing the structure object by address/reference

9.6.1 Passing Each Member of a Structure Object as a Separate Argument

A structure object can be passed to a function by passing each member of the structure object. The members of the structure object can be passed by value or by address/reference. The piece of code in Program 9-20 illustrates the passing of structure objects by the means of passing each of its members by value and by address/reference.

Line | **Prog 9-20.c**

```
 1  //Passing structure objects by passing their structure members
 2  #include<stdio.h>
 3  struct complex
 4  {
 5      int re;
 6      int im;
 7  };
 8  add_complex(int, int, int, int);
 9  mult_complex(int*,int*,int*,int*);
10  main()
11  {
12      struct complex no1, no2;
13      printf(“Enter the real and imaginary parts of 1st number:\t”);
14      scanf(“%d %d”,&no1.re, &no1.im);
15      printf(“Enter the real and imaginary parts of 2nd number:\t”);
16      scanf(“%d %d”,&no2.re, &no2.im);
17      add_complex(no1.re, no1.im, no2.re, no2.im);
18      mult_complex(&no1.re, &no1.im, &no2. re, &no2.im);
19  }
20  add_complex(int a, int b, int c, int d)
21  {
22  if(b+d<0)
23      printf(“The result of their addition is %d%di\n”,a+c,b+d);
24  else
25      printf(“The result of their addition is %d+%di\n”,a+c,b+d);
26  }
27  mult_complex(int* a, int* b, int* c, int*d)
28  {
29  int re, im;
30  re=*a * *c-*b * *d;
31  im=*a * *d + *b * *c;
32  if(im<0)
33      printf(“The result of their multiplication is %d%di\n”,re, im);
34  else
35      printf(“The result of their multiplication is %d+%di\n”,re, im);
36  }
```

Output window

```
Enter the real and imaginary parts of 1st number:   2 –3
Enter the real and imaginary parts of 2nd number:  4   5
The result of their addition is 6+2i
The result of their multiplication is 23–2i
```

Output window (second execution)

```
Enter the real and imaginary parts of 1st number:   –2 –3
Enter the real and imaginary parts of 2nd number:   4 –5
The result of their addition is 2–8i
The result of their multiplication is –23–2i
```

Remarks:

- In line number 17, each member of the structure objects no1 and no2 is passed by value to the function add_complex
- In line number 18, each member of the structure objects no1 and no2 is passed by address/reference to the function mult_complex

Program 9-20 | A program that illustrates the method of passing a structure object by passing its members

The observable points about passing a structure object by the means of passing its members are as follows:

1. This method of passing a structure object to a function is highly inefficient, unmanageable and infeasible if the number of members in a structure object to be passed is large.
2. This method of passing a structure to a function is suited if the structure contains only a few members. Also, the members must be of basic types or derived types but not of structure type (i.e. nested structures).
3. The members of the structure object can be passed by value or by address/reference.

9.6.2 Passing a Structure Object by Value

The member-by-member copy behavior of the assignment operator when applied on structures makes it possible to pass a structure object to, and return a structure object from a function by value. The piece of code in Program 9-21 illustrates the passing of a structure object to a function by value.

Prog 9-21.c

```
//Passing and returning structure objects by value
#include<stdio.h>
struct complex
{
    int re;
    int im;
};
struct complex add_complex(struct complex, struct complex);
main()
{
    struct complex no1, no2, no3;
    printf("Enter the real and imaginary parts of 1st number:\t");
    scanf("%d %d",&no1.re, &no1.im);
    printf("Enter the real and imaginary part of 2nd number:\t");
    scanf("%d %d",&no2.re, &no2.im);
    no3=add_complex(no1,no2);
    if(no3.im<0)
    printf("The result of their addition is %d%di\n",no3.re, no3.im);
    else
    printf("The result of their addition is %d+%di\n",no3.re, no3.im);
}
struct complex add_complex(struct complex a, struct complex b)
{
    struct complex temp;
    temp.re=a.re+b.re;
    temp.im=a.im+b.im;
    //It is invalid to write temp=a+b
    return temp;
}
```

Output window

```
Enter the real and imaginary parts of 1st number:    2 -3
Enter the real and imaginary parts of 2nd number:    4  5
The result of their addition is 6+2i
```

Output window (second execution)

```
Enter the real and imaginary parts of 1st number:   -2 -3
Enter the real and imaginary parts of 2nd number:    4 -5
The result of their addition is 2-8i
```

Remarks:

- The structure objects no1 and no2 are passed to the function add_complex by value
- This passing is possible because of member-by-member copy behavior of the assignment operator when applied on structures

Program 9-21 | A program that illustrates the passing of a structure object to a function by value

The observable points about passing structure objects to a function by value are as follows:

1. This method of passing a structure object to a function is better (i.e. manageable) than the previous method of the structure passing in which each member of the object is passed individually.
2. In this method, all the members of a structure object are passed together instead of being passed individually.
3. If the number of members in a structure object is quite large, this method involves large data movement. In such a case, the method of passing structure object via the pointer (as discussed in Section 9.6.3) will be more efficient.
4. As the structure objects are passed by value, the changes made in the formal parameters in the called function are not reflected back to the calling function. The piece of code in Program 9-22 illustrates this fact.

Line | **Prog 9-22.c**

```c
//Taking reflection of a point about a line inclined at 45° to the x-axis
#include<stdio.h>
struct point
{
    int x;
    int y;
};
reflectpoint(struct point);
main()
{
    struct point pt;
    printf("Enter the x and y coordinates of the point:\t");
    scanf("%d %d",&pt.x, &pt.y);
    reflectpoint(pt);
    printf("The x and y coordinates of the reflected point: (%d,%d)",pt.x, pt.y);
}
reflectpoint(struct point pt)
{
    int temp;
    temp= pt.x;
    pt.x=pt.y;
    pt.y=temp;
}
```

Output window

```
Enter the x and y coordinates of the point:     4 7
The x and y coordinates of the reflected point: (4,7)
```

Remark:

- The changes made in the structure object in the called function reflectpoint are not reflected back to the calling function main

Program 9-22 | A program to illustrate the usage of passing the structure objects by value

9.6.3 Passing a Structure Object by Address/Reference

If the number of members in a structure object is quite large, it is beneficial to pass the structure object by address/reference. This method of passing the structure object requires fixed data (i.e. equal to the size of a pointer) movement irrespective of the size of the structure object. The piece of code in Program 9-23 illustrates the method of passing the structure objects to a function by address/reference.

Line | **Prog 9-23.c**

```
1   //Passing and returning structure objects by address/reference
2   #include<stdio.h>
3   struct complex
4   {
5       int re;
6       int im;
7   };
8   struct complex mult_complex(struct complex*, struct complex*);
9   main()
10  {
11      struct complex no1, no2, no3;
12      printf(“Enter the real and imaginary parts of 1st number:\t”);
13      scanf(“%d %d”,&no1.re, &no1.im);
14      printf(“Enter the real and imaginary parts of 2nd number:\t”);
15      scanf(“%d %d”,&no2.re, &no2.im);
16      no3=mult_complex(&no1,&no2);
17      if(no3.im<0)
18      printf(“The result of their multiplication is %d%di\n”,no3.re, no3.im);
19      else
20      printf(“The result of their multiplication is %d+%di\n”,no3.re, no3.im);
21  }
22  struct complex mult_complex(struct complex* a, struct complex* b)
23  {
24      struct complex temp;
25      temp.re=a->re*b->re-a->im*b->im;
26      temp.im=a->re*b->im+a->im*b->re;
27      //It is invalid to write temp=a*b
28      return temp;
29  }
```

Output window

```
Enter the real and imaginary parts of 1st number:    2 -3
Enter the real and imaginary parts of 2nd number:    4  5
The result of their multiplication is 23–2i
```

Output window (second execution)

```
Enter the real and imaginary parts of 1st number:   –2 –3
Enter the real and imaginary parts of 2nd number:    4 –5
The result of their multiplication is –23–2i
```

Remark:

- In line number 16, the structure objects no1 and no2 are passed by address/reference to the function mult_complex

Program 9-23 | A program that illustrates the passing of a structure object to a function by address/reference

The observable points about passing the structure objects to a function by address/reference are as follows:

1. This method of passing a structure object to a function is better (i.e. efficient) than the previous method of passing a structure object by value.
2. In this method of passing a structure object, instead of passing the entire structure object, only the address of a structure object is passed. Hence, this method of structure passing requires less data movement.
3. Since a structure object is passed by address, the changes made in the objects pointed to by the formal parameters in the called function are reflected back to the calling function. The piece of code in Program 9-24 illustrates this fact.

Line | **Prog 9-24.c**

```
1   //Taking reflection of a point about a line inclined at 45° to the x-axis
2   #include<stdio.h>
3   struct point
4   {
```

Output window

```
Enter the x and y coordinates of the point:     4 7
The x and y coordinates of the reflected point: (7,4)
```

(*Contd...*)

Prog 9-24.c

```
    int x;
    int y;
};
reflectpoint(struct point*);
main()
{
    struct point pt;
    printf("Enter the x and y coordinates of the point:\t");
    scanf("%d %d",&pt.x, &pt.y);
    reflectpoint(&pt);
printf("The x and y coordinates of the reflected point: (%d,%d)",pt.x, pt.y);
}
reflectpoint(struct point* pt)
{
    int temp;
    temp= pt->x;
    pt->x=pt->y;
    pt->y=temp;
}
```

Output window

Remark:

- The changes made in the structure object in the called function reflectpoint are reflected back to the calling function main

Program 9-24 | A program to illustrate the usage of passing the structure objects by address/reference

9.7 typedef and Structures

In Section 9.2.2, we have seen that a structure object can be declared by using the keyword struct followed by the tag-name of the defined structure type and the identifier name of the object to be declared. The usage of the keyword struct while declaring a structure object sometimes proves to be a bit inconvenient. In Chapter 10, we will see that the storage class specifier typedef can be used for creating syntactically convenient names (i.e. aliases). Thus, it can be used to create an alias for the defined structure type so that the keyword struct is not required repeatedly to declare the structure objects. The piece of code in Program 9-25 illustrates the use of the typedef storage class specifier along with structures.

Prog 9-25.c

```
//Use of typedef to create an alias for a structure type
#include<stdio.h>
typedef struct name
{
    char first_name[20];
    char last_name[20];
} NAME;
struct phonebook_entry
{
    NAME person_name;
    char mobile_no[11];
};
typedef struct phonebook_entry PH_ENTRY;
main()
```

Output window

```
Enter the details of the first person:
Enter the first name of the person:    Sam
Enter the last name of the person:    Mine
Enter the mobile number:    9870096971
Enter the details of the second person:
Enter the first name of the person:    Mani
Enter the last name of the person:    Kumar
Enter the mobile number:    9922134654

Records in the phone book are::
--------------------------------------
Sam Mine:    9870096971
Mani Kumar:    9922134654
```

(*Contd...*)

```
15 {
16     PH_ENTRY p1, p2;
17     printf("Enter the details of the first person:\n");
18     printf("Enter the first name of the person:\t");
19     gets(p1.person_name.first_name);
20     printf("Enter the last name of the person:\t");
21     gets(p1.person_name.last_name);
22     printf("Enter the mobile number:\t");
23     gets(p1.mobile_no);
24
25     printf("Enter the details of the second person:\n");
26     printf("Enter the first name of the person:\t");
27     gets(p2.person_name.first_name);
28     printf("Enter the last name of the person:\t");
29     gets(p2.person_name.last_name);
30     printf("Enter the mobile number:\t");
31     gets(p2.mobile_no);
32
33     printf("\nRecords in the phone book are::\n");
34     printf("-------------------------------------\n");
35     printf("%s %s:\t %10s\n",p1.person_name.first_name,
36     p1.person_name.last_name, p1.mobile_no);
37     printf("%s %s:\t %s\n",p2.person_name.first_name,
38     p2.person_name.last_name, p2.mobile_no);
39 }
```

Remarks:

- The storage class specifier typedef is used to name a new type or to rename an old type
- In line number 3, it is used to name a new type
- In line number 13, it is used to rename the already defined type struct phonebook_entry

Program 9-25 | A program that illustrates the use of the storage class specifier typedef to name and rename a structure type

A typedef name, i.e. the created alias name can be the same as the structure name. The piece of code in Program 9-26 illustrates this fact.

Prog 9-26.c

```
//Alias name can be same as structure name
#include<stdio.h>
struct name
{
    char first_name[20];
    char last_name[20];
} ;
typedef struct name name; //←typedef name is same as structure tag-name
main()
{
name person;
printf("Enter the first name and the last name of the person:\n");
scanf("%s %s",person.first_name, person.last_name);
printf("The name of the person is %s %s",person.first_name, person.last_name);
}
```

Output window

```
Enter the first name and the last name of the person:
Arvind Mishra
The name of the person is Arvind Mishra
```

Remarks:

- In C, it is not allowed to declare an object of the defined structure type by using the tag-name of the defined structure type without using the keyword struct
- However, in C++, this rigidity was relaxed

Program 9-26 | A program that illustrates the use of the storage class specifier typedef

9.8 Unions

Just like structures, unions are used to create user-defined types. A **union** is a collection of one or more variables, possibly of different types. All the major aspects of union, like defining a union type, declaring objects of a union type, using and performing operations on objects of a union type are the same as that of structures. The only difference between them is in the terms of storage of their members. In structures, a separate memory is allocated to each member, while in unions, all the members of an object share the same memory.

A union object is used only if one of its constituting members is to be used at a time. In such a situation, it proves to be memory efficient as compared to structures. The important points about unions are as follows:

1. **Defining a union type:** A union type is defined in the same way as a structure type, with the only difference that the keyword union is used instead of the keyword struct to define the union type.
2. **Declaring union objects:** Objects of a union type can be declared either at the time of union type definition or after the union type definition in a separate declaration statement. Objects of the union type can be created after the union definition only if the defined union type is named or tagged.

The general form of declaring a union object is:

[storage_class_specifier] [type_qualifier] **union named_union_type identifier_name** [=intialization_list [,...]]**;**

The important points about a union object declaration are as follows:

1. The terms enclosed with the square brackets are optional and might not be present in a union variable declaration statement. The terms shown in **bold** are the mandatory parts of a union object declaration statement.
2. A union object declaration consists of:
   a. The keyword union for declaring union variables. It can also be used in conjunction with const qualifier for declaring a union constant.
   b. The tag-name of the defined union type.
   c. Comma-separated list of identifiers. The variables can optionally be initialized by providing initialization lists. However, the initialization of constants is must.
   d. A terminating semicolon.
3. **Size of a union object or union type:** Upon the declaration of a union object, the amount of memory allocated to it is the amount necessary to contain its largest member. It can be checked by using the sizeof operator. The piece of code in Program 9-27 illustrates the use of the sizeof operator on a union object.

Line	Prog 9-27.c	Output window

	`//Unions and sizeof operator`
	`#include<stdio.h>`
	`union variables`
	`{`
	`    char a;`
	`    int b;`
		Objects of type union variables will take 8 bytes
		Union variable var takes 8 bytes
		Remarks:
		• The sizeof operator when applied on unions returns the size of its largest member

(*Contd...*)

```
    float c;
    double d;
} ;
main()
{
union variables var;
printf("Objects of type union variables will take %d bytes\n",sizeof(union variables));
printf("Union variable var takes %d bytes\n", sizeof(var));
}
```

- Since in the union type union variables, the size of the largest member (i.e. d of type double) is 8, the sizeof operator when applied on the union type union variables or on the objects of this type, outputs 8

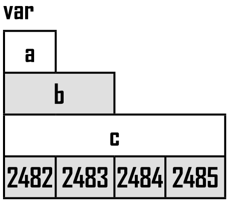
Program 9-27 | A program that illustrates the use of the sizeof operator on unions

4. **Address-of a union object:** The members of a union object are stored in the memory in such a way that they overlap each other. All the members of a union object start from the same memory location, which in fact, is the same as the starting address of the union object. This can be checked by applying the address-of operator to the union object as well as to its constituting members. The application of the address-of operator on a union object and its constituent members is illustrated in Program 9-28.

Prog 9-28.c

```
//Address-of operator and unions
#include<stdio.h>
union variables
{
    char a;
    int b;
    float c;
} ;
main()
{
  union variables var;
  printf("Starting address of var is %p\n",&var);
  printf("Starting address of 1st member is %p\n",&var.a);
  printf("Starting address of 2nd member is %p\n",&var.b);
  printf("Starting address of 3rd member is %p\n",&var.c);
  printf("Starting address of 4th member is %p\n",&var.d);
}
```

Memory contents



Output window

```
Starting address of var is 1A5F:2482
Starting address of 1st member is 1A5F:2482
Starting address of 2nd member is 1A5F:2482
Starting address of 3rd member is 1A5F:2482
```

Remark:

- In the defined type union variables, the first byte (lower order) is shared by all the three members a, b and c. The second byte is shared by the members b and c. The third and the fourth bytes are exclusively owned by the member c

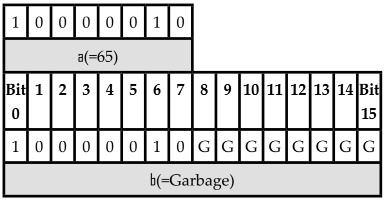
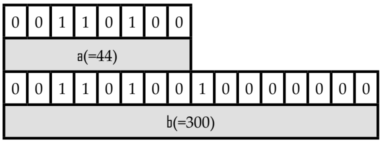
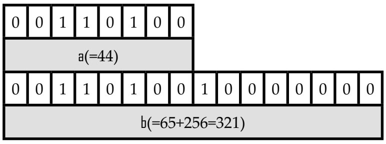
Program 9-28 | A program illustrating that all the members of a union object start from the same memory location

5. **Initialization of a union object:** Since the members of a union object share the same memory, the union object can hold the value of only one of its member at a time. Hence, while initializing a union object, it is allowed to initialize its first member only. The piece of code in Program 9-29 illustrates this fact.

Line	Prog 9-29.c	Output window
1	//Initialization of union objects	Compilation error “Declaration syntax error in function main()” **Remarks:** • The initialization of all the members of the union object var, in line number 11 gives a compilation error • To remove this error, initialize only the first member of the union object var and then re-execute the code
2	#include<stdio.h>	
3	union variables	
4	{	
5	char a;	
6	int b;	
7	float c;	
8	} ;	
9	main()	
10	{	
11	union variables var={‘A’, 2, 2.5};	
12	printf(“The values of the members are %c %d %f”, var.a, var.b, var.c);	
13	}	

Program 9-29 | A program illustrating that only the first member of a union object can be initialized

Since the members of a union object share the memory in an overlapped fashion, only one member at a time can be assigned a value. Accessing the value of this member gives a meaningful result, while accessing the value of any other member gives a garbage value as a result. The piece of code in Program 9-30 illustrates this fact.

	Trace	Prog 9-30.c		Output window
1		//Access only the lastly assigned member of a	**After trace step 2:** **var**  1 0 0 0 0 0 1 0 a(=65) Bit 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 Bit 15 1 0 0 0 0 0 1 0 G G G G G G G G b(=Garbage) **After trace step 4:** **var**  0 0 1 1 0 1 0 0 a(=44) 0 0 1 1 0 1 0 0 1 0 0 0 0 0 0 0 b(=300) **After trace step 7:** **var**  0 0 1 1 0 1 0 0 a(=44) 0 0 1 1 0 1 0 0 1 0 0 0 0 0 0 0 b(=65+256=321) Note that in the above illustrations, the bit 0, i.e., LSB is present on the left-hand side	First member is A First member now is , Second member is 300 First member now is A Second member now is 321 **Remarks:** • After trace step 2, the member a of the union variable var is initialized with the character constant ‘A’, i.e. 65 • This also initializes the lower order byte of the member b, while its upper order byte still contains garbage value • After trace step 4, the value assigned to member b, i.e. 300 completely modifies the content of the member a
2		//union object		
3		#include<stdio.h>		
4		union variables		
5		{		
6		char a;		
7		int b;		
8		} ;		
9	1	main()		
10		{		
11	2	union variables var={‘A’};		
12	3	printf(“First member is %c\n”,var.a);		
13	4	var.b=300;		
14	5	printf(“First member now is %c\n”,var.a);		
15	6	printf(“Second member is %d\n”,var.b);		
16	7	var.a=‘A’.		
17	8	printf(“First member now is %c\n”,var.a);		
18	9	printf(“Second member now is %d\n”,var.b);		
19	10	}		

(*Contd...*)

			• Hence, the value of member a becomes 44 (i.e. 00101100 in binary number system) • After trace step 7, when the value 'A' i.e. 65 is placed in the member a, the lower order byte of the member b is modified and becomes 65 • Hence, the value of the entire member b comes out to be 65+256=321

Program 9-30 | A program illustrating that accessing only the lastly assigned member gives a meaningful result

One of the potential pitfalls in using the unions is the possibility of accidentally retrieving the value currently stored in the union through an inappropriate member. For example, in Program 9-30, if the last assignment is to the member a and the programmer accidentally retrieves the value of the member b, the result will be a garbage value.

3. **Operations on union objects:** All the operations on union objects are applied in the same way as they are applied on the structure objects. For example, the members of a union object can be accessed in the same way as the members of a structure object can be assessed, i.e. by using a member access operator.

   Similar to structures, the following operations are feasible on unions:

   a. Assigning a union object to a union variable of the same type.
   b. Passing a union object or a pointer to a union object as a function argument.
   c. Returning a union object from a function, etc.

9.9 Practical Application of Unions

In the previous chapters, you have used printf, scanf, getch and other library functions a number of times. These library functions perform rudimentary operations like printing an output on the screen, reading input from the keyboard, etc. In other words, these functions interact with the hardware of the machine. However, if you delve deeper into the technical details of how these functions interact with the hardware, you will come to know that these functions do not have any direct interaction with the hardware. The process through which the library functions interact with the hardware of the machine is shown in Figure 9.5.

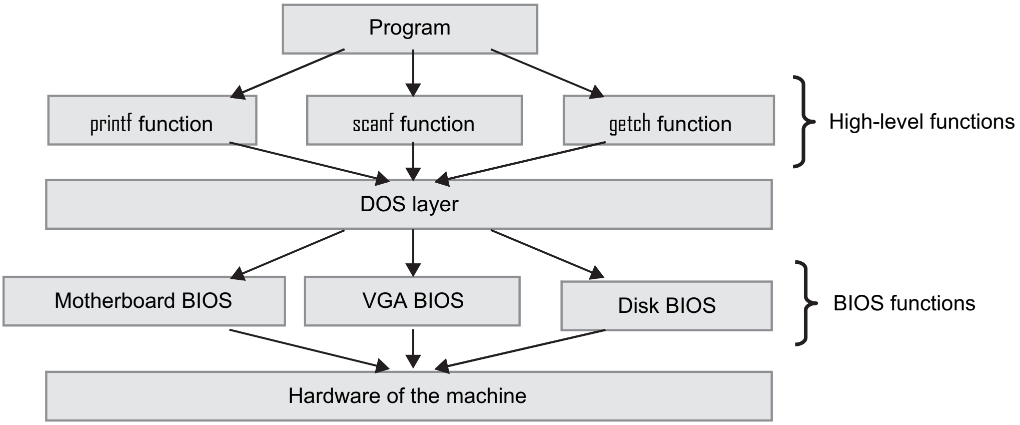


Figure 9.5 | Hardware interaction

The important points about the mechanism through which hardware interaction is performed are as follows:

1. The interaction with the hardware of the machine is done by calling the low-level machine specific code routines, generally provided by the hardware manufacturers. These routines are known as **B**asic **I**nput **O**utput **S**ystem (**BIOS**) routines.
2. There are several different BIOSes. For example, the motherboard BIOS performs the initial hardware detection and system booting. The Video Graphic Adaptor (VGA) BIOS handles all the screen manipulation functions. The Disk BIOS manages disk input–output and other disk operations.
3. These BIOSes are generally placed in the **R**ead **O**nly **M**emory (**ROM**) to ensure that they are always available and are not affected by the disk failures. Thus, they are also known as **ROM-BIOS**es.
4. There is a layer of Disk Operating System (DOS) software, which sits on the top of these lower-level BIOSes and provides a common access to these lower-level BIOSes in a form that is easier to use in the programs.
5. A call to a library function generates a DOS call, which may in turn call an appropriate BIOS routine to interact with the hardware.
6. Thus, a task performed by a library function can also be performed by directly calling DOS or by calling the lower-level BIOS routines to perform the task.

9.9.1 Calling DOS and BIOS Functions

DOS and BIOS functions can be called by generating **interrupts**. An **interrupt** is a signal to the **microprocessor** or just the **processor** of the computer informing that an event has occurred and it needs an immediate attention. When the processor receives an interrupt signal, it suspends the execution of the current program and executes a specific routine (DOS or BIOS routine) known as **Interrupt Service Routine (ISR)**. An interrupt signal can be generated either by hardware or a software function call. When it is generated by hardware (e.g. key press), it is known as **hardware interrupt**. If it is generated by giving a call to the software functions like int86, int86x, intdos, intdosx, etc., it is called **software interrupt**. In this section, we will look at how to generate the software interrupts using the functions int86 and intdos.

The prototype of the function int86 is int int86(int intno, union REGS* inregs, union REG* outregs);. and the function intdos is intdos(union REGS* inregs, union REGS* outregs);.

The important points about the usage of the functions int86 and intdos are as follows:

1. Calling ISRs (DOS or BIOS routines) is not as simple as calling the high-level functions because ISRs are not named. These are stored at the specific locations in the memory and can be executed by transferring the program control to those memory locations. It is very cumbersome to remember the starting address of every ISR. Thus, to abbreviate the problem, an index table, known as **Interrupt Vector Table** (**IVT**) is provided. The IVT is stored in the first 1024 bytes of the memory, i.e. from the memory address 0x0000-0x03FF. There are 256 entries in IVT and each entry is of 4 bytes, which specifies the complete (i.e. segment and offset) address of the interrupt service routine.
2. To call a BIOS service routine, an integer number is provided as an argument to the function int86. This integer argument is an index of an IVT entry, where the address of the specific ISR is stored. From IVT, the address of the ISR is retrieved and the control is transferred to the starting memory location of the ISR. The mentioned procedure is shown in Figure 9.6.

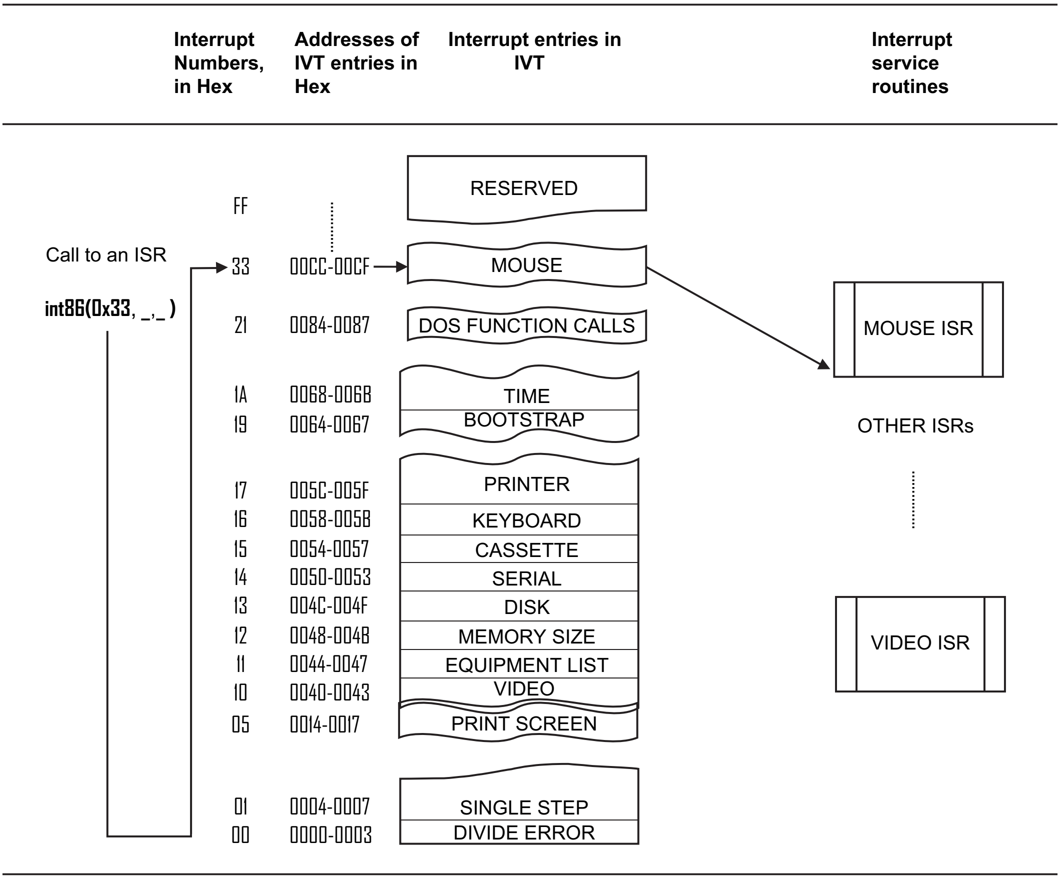


Figure 9.6 | Interrupt vector table and its use

3. To call a DOS service routine, the functions intdos and intdosx are used. The DOS services are grouped together under the interrupt number 0x21. The functions intdos and intdosx execute interrupt 0x21 to invoke the specified DOS function. Since these functions always execute interrupt 0x21, the interrupt number is not given as an input to them.
4. Like arguments are given to the functions, similarly inputs are given to ISRs, which determine their behavior. The inputs to ISRs are given by placing the values in the CPU's (i.e. microprocessor's) registers. The CPU register is a sort of memory, which is internal to it, provides direct and very fast data access as compared to the external memories like cache, **R**andom **A**ccess **M**emory (RAM) and hard disk. The number of registers in a CPU and their size depends upon the architecture of a microprocessor. The registers available in 8086 microprocessor and its family are shown in Figure 9.7.

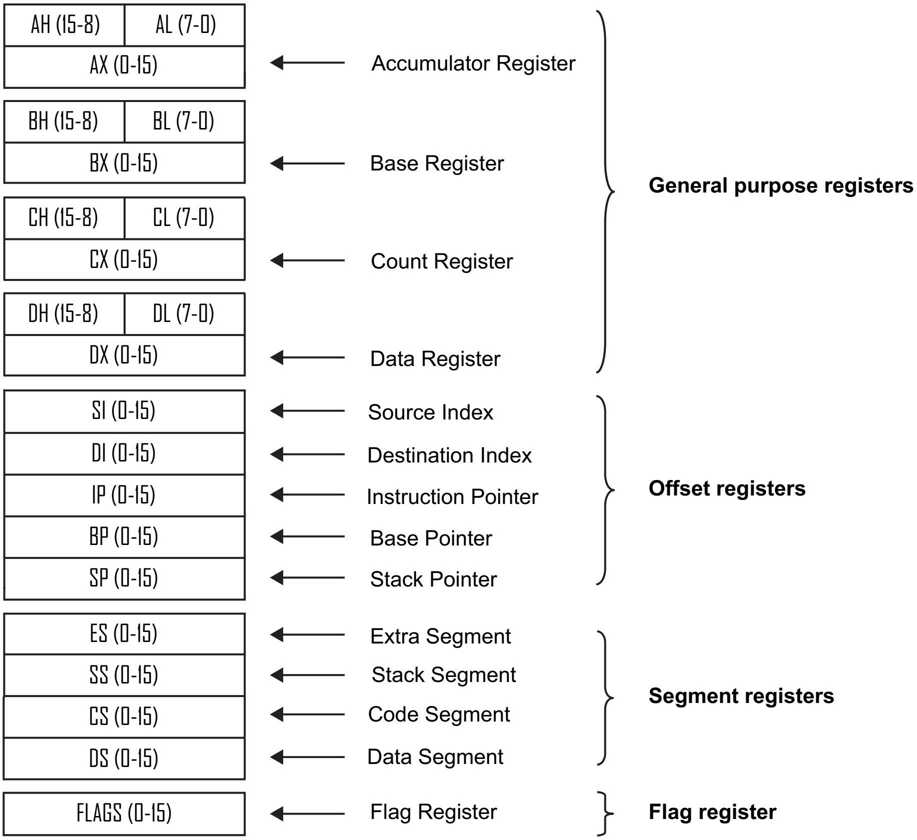


Figure 9.7 | Registers available in 8086 microprocessor and their classification

As shown in Figure 9.7, the registers available in 8086 microprocessor are classified as:

1. General-purpose registers
2. Offset registers
3. Segment registers
4. Flag register

The general-purpose registers, i.e. Accumulator register, Base register, Count register and Data register are 16-bit registers and are referred to as AX, BX, CX and DX, respectively. It is also possible to individually access the lower and the higher bytes of these registers. The lower and the higher bytes of these registers are referred to as AL, AH, BL, BH, CL, CH, DL and DH, respectively. The other registers can be accessed in totality, i.e. all the 16 bits at a time. In Borland Turbo C 3.0/4.5, a type union REGS has been defined in the header file dos.h, which helps in passing the information to and from the functions int86 and intdos.

The important points about the predefined type union REGS are as follows:

1. The type union REGS has been defined in the header file dos.h as:

```
union REGS
{
    struct WORDREGS x;
    struct BYTEREGS h;
};
```

   The types struct WORDREGS and struct BYTEREGS have also been defined in the header file dos.h as:

```
struct WORDREGS
{
    unsigned int ax, bx, cx, dx;
    unsigned int si, di, cflag, flags;
};
struct BYTEREGS
{
    unsigned char al, ah, bl, bh;
    unsigned char cl, ch, dl, dh;
};
```

2. The objects of the type union REGS can be declared as:
   union REGS identifier_names; e.g. union REGS inregs, outregs;
3. The 16-bit members (i.e. ax, bx, cx, etc.) are accessed through the member x, and 8-bit members (i.e. al, ah, bl, bh, etc.) are accessed through the member h of the declared objects of the union type union REGS. For example, if the object iregs is defined to be of the union type union REGS. The 16-bit member bx is accessed as iregs.x.bx while its higher and lower parts, i.e. bh and bl are accessed as iregs.h.bh and iregs.h.bl, respectively.
4. An input to the interrupt service routine can be given by setting the values of the specific members of the declared objects of the type union REGS and by passing them by address/reference to the functions int86 and intdos. The value of a member of a declared object of the type union REGS can be set as:

```
inregs.x.ax=1;    //←Setting the value of the member ax to be 1
inregs.h.ch=2;    //←Setting the value of the member ch to be 2
```

9.9.2 Interrupt Programming

Examples of interrupt programming are given in Programs 9-31 to 9-36. The elaborated interrupt list is given in Appendix D for reference. The exhaustive interrupt list may run up to thousands of pages and such a listing is beyond the scope of this book.

Prog 9-31a.c
Using library function

```
//Printing text at specific location using the function gotoxy
#include<stdio.h>
#include<conio.h>
main()
{
int row, col;
clrscr();
printf(“Enter the row and column number in which you want\
to print the text\t”);
scanf(“%d %d”,&row, &col);
gotoxy(row,col);
printf(“Hello Readers”);
}
```

Prog 9-31b.c
Using interrupt programming

```
//Implementing function gotoxy using interrupt programming
#include<stdio.h>
#include<conio.h>
#include<dos.h>
void mygotoxy(int, int);
main()
{
int row, col;
clrscr();
printf(“Enter the row and column in which you want\
to print the text\t”);
scanf(“%d %d”,&row, &col);
mygotoxy(row,col);
printf(“Hello Readers”);
}
void mygotoxy(int x, int y)
{
union REGS inregs, oregs;
inregs.h.ah=2;
inregs.h.bh=0;
inregs.h.dh=x;
inregs.h.dl=y;
int86(0x10, &inregs, &oregs);
}
```

Output window (Turbo C 3.0)

```
Enter the row and column number in which you want to print the text:     4 5



   Hello Readers
```

Remarks:

- The program calls a ROM-BIOS function that positions the cursor in the desired row and column
- The associated interrupt has number 0x10 in hexadecimal (i.e. 16 in decimal)
- There are a number of services available under this interrupt like positioning the cursor on the screen, changing the size of the cursor, plotting a pixel on the screen, etc
- These service routines have a service number associated with them. The associated service number is to be placed in the AH register before the interrupt is being called
- Refer Appendix D for an elaborated description of ROM-BIOS services
- The function call int86(0x10, &inregs, &oregs); can also be written as int86(16, &inregs, &oregs);
- Generally, interrupts are numbered in hexadecimal and thus, the first method of calling the function is preferred
- All the programs making the use of interrupts work with Turbo C 3.0 compiler for DOS but not with Turbo C 4.5 and MS-VC++ 6.0 compilers for Windows. These compilers work with 32-bit environment (i.e. Windows) and create 32-bit programs, whereas interrupts work only with 16-bit programs

Program 9-31 | A program that illustrates the usage and the implementation of the function gotoxy

Prog 9-32a.c
Using library function

```
//Reading a character using the function getche
#include<stdio.h>
#include<conio.h>
main()
{
char ch;
clrscr();
printf("Enter a character:\t");
ch=getche();
printf("\nThe character that you entered is %c",ch);
}
```

Prog 9-32b.c
Using interrupt programming

```
//Implementing the function getche using the interrupt programming
#include<stdio.h>
#include<conio.h>
#include<dos.h>
char mygetche();
main()
{
char ch;
clrscr();
printf("Enter a character:\t");
ch=mygetche();
printf("\nThe character that you entered is %c",ch);
}
char mygetche()
{
union REGS inregs, oregs;
inregs.h.ah=1;
intdos(&inregs, &oregs);
return oregs.h.al;
}
```

Output window (Turbo C 3.0)

```
Enter a character:      H
The character that you entered is H
```

Remarks:
- The program calls a DOS routine that reads a character from the standard input with echo
- The DOS routines are grouped together under the interrupt number 0x21 in hexadecimal (i.e. 16 in decimal)
- There are a number of services available under this interrupt like read a character, write a character on the screen, write a character to the printer, get machine name, create directory, rename directory, delete file, etc.
- These service routines have a service number associated with them. The associated service number is to be placed in the AH register before the interrupt is called
- Refer Appendix D for an elaborated description of DOS services
- The function call intdos(&inregs, &oregs); can also be written as int86(0x21, &inregs, &oregs);

Program 9-32 | A program that illustrates the usage and the implementation of the function getche

Prog 9-33a.c
Using library function

```
//Printing the character using the function putch
#include<stdio.h>
#include<conio.h>
main()
{
char ch;
```

Prog 9-33b.c
Using interrupt programming

```
//Implementing the function putch using the interrupt programming
#include<stdio.h>
#include<conio.h>
#include<dos.h>
myputch();
main()
```

(Contd...)

Prog 9-33a.c
Using library function

```
clrscr();
printf(“The character is:\t”);
putch(‘A’);
}
```

Prog 9-33b.c
Using interrupt programming

```
{
char ch;
clrscr();
printf(“The character is:\t”);
myputch();
}
myputch()
{
union REGS inregs, oregs;
inregs.h.ah=2;
inregs.h.dl=‘A’;
intdos(&inregs, &oregs);
}
```

Output window (Turbo C 3.0)

```
The character is:      A
```

Remarks:

- The program calls a DOS routine to implement the functionality of the putch function
- The routine that writes a character onto the screen has service number 2
- The service number is placed in the AH register by assigning 2 to inregs.h.ah before the interrupt is called
- Refer Appendix D to see that the character to be written is placed in the DL register
- In the given code, the character ‘A’ is placed in the DL register by assigning ‘A’ to inregs.h.dl

Program 9-33 | A program that illustrates the usage and the implementation of the function putch

Prog 9-34.c
Using library function

```
//Printing a string on the screen by using the interrupt programming
#include<stdio.h>
#include<conio.h>
#include<dos.h>
main()
{
union REGS inregs, oregs;
char *ch="Interrupt Programming$”;
clrscr();
inregs.h.ah=9;
inregs.x.dx=(unsigned int)ch;
intdos(&inregs,&oregs);
}
```

Output window (Turbo C 3.0)

```
Interrupt Programming
```

Remarks:

- The service number of the DOS routine that prints a string on to the screen is 9. It is to be placed in the AH register before the interrupt is called, thus 9 is assigned to inregs.h.ah
- The segment:offset address of the string that is to be printed is to be placed in the DX register
- ch points to the base address of the string that is to be printed
- Thus, ch is assigned to inregs.x.dx. But, since ch is of type char*, it has to be explicitly type casted to unsigned int before assigning to inregs.x.dx
- The program prints a string on the screen without using the printf and puts function

Program 9-34 | A program that illustrates the usage and the implementation of the function puts

Line | **Prog 9-35.c**
Using library function | **Output window (Turbo C 3.0)**

```
//Getting the machine name using the interrupt programming
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
#include<dos.h>
char* machinename();
main()
{
clrscr();
printf("The name of the machine is %s", machinename());
}
char* machinename()
{
union REGS inregs, oregs;
char *ch=(char*)malloc(16);
inregs.h.ah=0x5E;
inregs.x.dx=(unsigned int)ch;
intdos(&inregs,&oregs);
return ch;
}
```

```
The name of the machine is COMP2
```

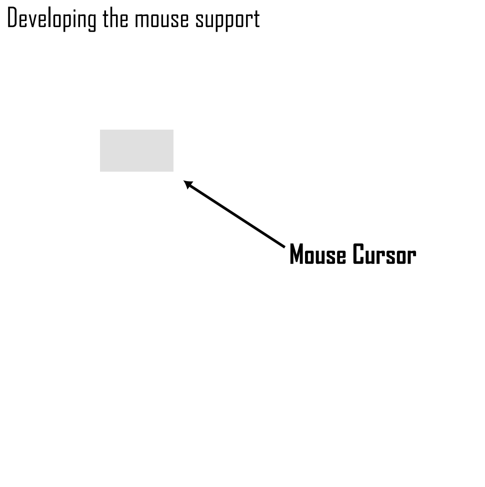
Remarks:

- The service number of the DOS routine that get the machine name is 5E
- Thus, the hexadecimal value 5E is placed in the AH register by assigning the value to inregs.h.ah
- The service routine also expects the base address of a 16-byte buffer (i.e. character array) in which the machine name will be placed to be assigned to the DX register
- Instead of the function call intdos(&inregs, &oregs); the function call int86(0x21, inregs, oregs); can also be used

Program 9-35 | A program that illustrates the usage of the interrupt programming to get the machine name

Line | **Prog 9-36.c**
Using library function | **Output window (Turbo C 3.0)**

```
//Mouse Programming using interrupt programming
#include<stdio.h>
#include<conio.h>
#include<dos.h>
initmouse();
showmouseptr();
union REGS iregs, oregs;
main()
{
    clrscr();
    printf("Developing the mouse support\n");
    initmouse();
    getch();
}
initmouse()
{
    iregs.x.ax=0;
    int86(0x33, &iregs, &oregs);
    if(oregs.x.ax==0)
        printf("Mouse not supported by the system");
    else
        showmouseptr();
}
```



```
Developing the mouse support
```

Remarks:

- The interrupt number for developing the mouse support is 0x33
- The program implements only two services, i.e. initialize the mouse support and show the mouse pointer

(Contd...)

Line	Prog 9-36.c Using library function	Output window (Turbo C 3.0)

24
25
26
27
28
	showmouseptr()
	{
	    iregs.x.ax=1;
	    int86(0x33,&iregs,&oregs);
	}
		• Refer Appendix D and implement other services to read the position and button status of the mouse, to define the horizontal and the vertical cursor range, to change the shape of the cursor, etc

Program 9-36 | A program that illustrates the usage of the interrupt programming to provide mouse support

9.10 Enumerations

In C language, **enumerations** provide another way to create user-defined types. An enumeration type is designed for the objects that can have a limited set of values. For example, consider an application in which we want a variable to hold a Boolean value. We can create an enumeration type BOOL that have two values FALSE and TRUE. The values FALSE and TRUE are known as **enumeration constants**. The variables of type BOOL can either have value FALSE or TRUE (i.e. Boolean value). Note that the values FALSE and TRUE are not the strings, but are integer constants. The compiler internally associates an integer value each with the names FALSE and TRUE. Thus, any operation that is applicable on an integer constant can be applied on them and any operation that is applicable on a variable of integer type can be applied on a variable of type BOOL. Since the integer values are represented by the names, the enumeration type helps in making the programs more readable. The important points about enumerations are as follows:

1. **Definition of an enumeration type:** The general form of an enumeration-type definition is:

   [storage_class_specifier][type_qualifier] **enum** [tag-name] **{enumeration-list}**[identifier=initializer[,...]]**;**

   The important points about the definition of an enumeration type are as follows:

   i. The terms enclosed within the square brackets are optional and might not be present in the definition of an enumeration type. The terms shown in **bold** are mandatory parts of an enumeration definition.
   ii. An enumeration definition consists of a keyword enum, followed by an optional identifier name known as **enumeration tag-name** and a comma-separated list of **enumerators** enclosed within braces. All the enumerators present in the enumeration list forms an **enumeration set**.
   iii. An **enumerator** is an identifier that can hold an integer value. It is also known as the **enumeration constant**. An integer value can optionally be assigned to an enumerator, e.g. in the enumeration-type definition enum BOOLEAN {true=1, false=0};, the integer constants 1 and 0 are assigned to the enumerators true and false, respectively.
   iv. The names of the enumerators in the enumeration list must be unique.
   v. The values assigned to enumerators in the enumeration list need not be unique, e.g. the enumeration definition enum COLORS {red=2, green=1, yellow=1}; is perfectly valid.
   vi. Each enumeration constant has a scope that begins just after its appearance in the enumeration list. Due to this rule, the enumeration definition enum COLORS {red=2, green=red, yellow=green}; is perfectly valid.
   vii. Each enumerator in an enumeration list names a value. The enumeration constants are like symbolic constants except that their values are set automatically. By default, the first enumerator has the value 0. Each subsequent enumerator, if not

explicitly assigned a value, has a value 1 greater than the value of the enumerator that immediately precedes it. The piece of code in Program 9-37 illustrates the interpretation of this rule.

Prog 9-37.c

```
//If not explicitly specified, values to the enumeration constants are
//automatically assigned
#include<stdio.h>
enum CARS {alto, omni, esteem=3, wagonR, swift=1, dzire};
main()
{
 printf(“The value of various enumeration constants are:\n”);
 printf(“%d %d %d %d %d %d”, alto, omni, esteem, wagonR, swift, dzire);
}
```

Output window

```
The values of various enumeration constants are:
0 1 3 4 1 2
```

Remarks:

- The first enumerator, i.e. alto is automatically initialized with 0
- Each subsequent enumerator, if not explicitly assigned a value, has a value 1 greater than the value of the enumerator that immediately precedes it. Thus, the enumeration constant omni will have the value 1
- The enumeration constant esteem is explicitly given a value 3
- The enumeration constant wagonR will have the value 3+1=4
- Similarly, the enumeration constants swift and dzire will have the values 1 and 2, respectively

Program 9-37 | A program to illustrate that the values of enumeration constants are set automatically

viii. The enumeration definition can optionally have the storage class specifier and type qualifiers. However, they should be used in an enumeration-type definition statement only if the objects of the defined enumeration type are declared at the same time.

ix. The enumeration definition is a statement and must be terminated with a semicolon.

2. **Declaring objects of an enumeration type:** There are two ways to declare variables of an enumeration type:

   a. **At the time of enumeration-type definition:** Objects of an enumeration type can be declared at the time of enumeration-type definition. The variable declarations of the defined enumerated type given in Table 9.7 are valid.

Table 9.7 | Declaration of enumeration variables at the time of the enumeration-type definition

enum BOOL {false, true} flag1, flag2; (a)	enum {false, true} flag1, flag2; (b)

The declarations of the constants of the defined enumerated type given in Table 9.8 are valid.

Table 9.8 | Declaration of the enumeration constants at the time of enumeration-type definition

enum BOOL {false, true} const flag1=true, flag2=false; (a)	const enum BOOL {false, true} flag1=true, flag2=false; (b)
enum {false, true} const flag1=true, flag2=false; (c)	const enum {false, true} flag1=true, flag2=false; (d)

b. **After enumeration-type definition in a separate declaration statement**: Objects of an enumeration type can be created after its definition only if it is named or tagged. The keyword enum is used to declare the variables of the defined enumeration type. It is used in conjunction with the const qualifier to create the constants of the newly created type. The general form of declaring the objects of the defined enumeration type is:

[storage_class_specifier][type_qualifier]**enum named_eumeration_type identifier_name**[=initializer[,...]]**;**

The important points about the declaration of an object of the defined enumeration type are as follows:

i. The terms enclosed within the square brackets are optional and might not be present in an enumeration object declaration. The terms shown in **bold** are the mandatory parts of the enumeration object declaration.

ii. An enumeration object declaration consists of:

a. The keyword enum for declaring the enumeration variables. The keyword enum in conjunction with the const qualifier for declaring the constant of the defined enumeration type.

b. The tag name of the defined enumeration type.

c. Comma-separated list of identifiers (i.e. variable names and constant names). A variable can optionally be initialized by providing an initializer. However, the initialization of a constant is must.

d. An object of an enumeration type can be initialized with another object of the same enumeration type or with one of the enumerators present in the enumeration list or with an integer value. The piece of code in Program 9-38 illustrates this fact.

Prog 9-38.c

```c
//Initialization of an object of the enumeration type
#include<stdio.h>
enum SWITCH {off, on};
main()
{
    enum SWITCH s1=on;
    enum SWITCH s2=s1, s3=0;
    printf("The value of enumeration object s1 is %d\n",s1);
    printf("The value of enumeration object s2 is %d\n",s2);
    printf("The value of enumeration object s3 is %d\n",s3);
}
```

Output window

```
The value of enumeration object s1 is 1
The value of enumeration object s2 is 1
The value of enumeration object s3 is 0
```

Warnings(2):

```
Initializing SWITCH with int in function main()
Function should return a value in function main()
```

Remarks:

- Enumerations behave like integers, but it is common for a compiler to issue a warning message when an object of an enumeration type is initialized with something other than one of its constants or an expression of its type
- When the enumeration objects are initialized with integers, the compiler will not check that whether the initialized value is valid for such an enumeration or not
- Thus, it is even possible to initialize s3 with -8, 9 or any other integer value

Program 9-38 | A program that illustrates the initialization of an enumeration object

3. **Operations on the objects of an enumeration type:** The following operations can be performed on the object of an enumeration type:

   a. **Size of an enumeration object or enumeration type**: An enumeration object holds an enumerator, which in fact is an integer constant. Thus, when the sizeof operator is applied on an enumeration object or an enumeration type, it outputs the size of an integer. The piece of code in Program 9-39 illustrates this fact.

Prog 9-39.c

```c
//Size of an enumeration object or enumeration type
#include<stdio.h>
enum SWITCH {off, on};
main()
{
    enum SWITCH s=on;
    printf("The size of the enumeration type SWITCH is %d\n",sizeof(enum SWITCH));
    printf("The size of the enumeration object s is %d\n",sizeof(s1));
}
```

Output window

```
The size of the enumeration type SWITCH is 2
The size of the enumeration object s is 2
```

Remarks:

- The size of the enumeration type or an enumeration object is the same as the size of an integer object
- If executed using MS-VC++ 6.0, it outputs 4

Program 9-39 | A program that illustrates the use of the sizeof operator on an enumeration type and an enumeration object

   b. **Address-of an enumeration object**: The address-of operator can be applied on an enumeration object to find the address of the memory space allocated to it. The piece of code in Program 9-40 illustrates the application of the address-of operator on an enumeration object.

Prog 9-40.c

```c
//Address-of operator and structures
#include<stdio.h>
enum SWITCH {off, on};
main()
{
    enum SWITCH s=on;
    printf("Address of memory space allocated to s is %p\n",&s);
}
```

Output window

```
Address of memory space allocated to s is 249F:220C
```

Remarks:

- As the memory allocation is purely random, the printed address may vary for executions at different times or on different machines
- The definition of an enumeration type does not take any space in the memory, i.e. data segment. However, since it becomes a part of the code, it occupies some space in the code segment
- Hence, it is possible to apply the address-of operator on an enumeration type

Program 9-40 | A program that illustrates the use of the address-of operator on an object of enumeration type

   c. **Assignment of an enumeration object to an enumeration variable:** A variable of an enumeration type can be assigned with another object of the same enumeration type or with one of the enumerators present in the enumeration list or with an integer value.

d. **Behavior of equality operator on the objects of an enumeration type:** The equality operator can be applied to check the equality of two objects of an enumeration type.

The piece of code in Program 9-41 illustrates the use of an assignment operator and the equality operator on the objects of an enumeration type.

Prog 9-41.c

```c
//Equality of enumeration objects
#include<stdio.h>
enum SWITCH {off, on};
main()
{
    enum SWITCH s1=on, s2;
    s2=s1;        //←Assignment to an enumeration variable
    if(s1==s2)    //←Testing the equality of two enumeration variables
        printf("Both the switches are in ON state\n");
    else
        printf("Switches are in different states\n");
}
```

Output window

```
Both the switches are in ON state
```

Remarks:

- An integer can also be assigned to an enumeration type
- When the enumeration objects are assigned with integers, the compiler will not check whether the assigned value is valid for such an enumeration or not. However, it will issue a warning message
- It is possible to equate the enumeration objects of the same enumeration type
- It is even possible to equate the enumeration object with an integer constant or an integer variable

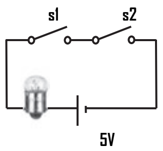
Program 9-41 | A program that illustrates the use of the equality operator on the objects of an enumeration type

e. **Other operators**: All the operators that work on integer objects can be applied on the objects of an enumeration type and those that can be applied on integer constants can be applied on enumerators. The piece of code in Program 9-42 illustrates the use of logical operators on the objects of enumeration types.

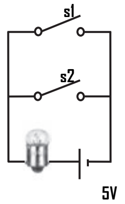
Prog 9-42.c

```c
//Enumerations and logical operators
#include<stdio.h>
enum COMBINATION {series=1, parallels=2};
enum SWITCH {OFF, ON};
main()
{
    enum COMBINATION ckt;
    enum SWITCH s1, s2;
    printf("Enter the configuration of the circuit:\n");
    printf("(Press 1 for series and 2 for parallel)\n");
    scanf("%d",&ckt);
    printf("Enter the status of the switches:\n");
    printf("(Press 0 for OFF state and 1 for ON state)\n");
    scanf("%d %d",&s1,&s2);
    if(ckt==series)
    {
     if(s1==ON && s2==ON)
        printf("The bulb will glow");
```

Series configuration



Parallel configuration



Output window

```
Enter the configuration of the circuit:
(Press 1 for series and 2 for parallel)
1
Enter the status of the switches:
(Press 0 for OFF state and 1 for ON state)
1 1
The bulb will glow
```

Output window (second execution)

```
Enter the configuration of the circuit:
(Press 1 for series and 2 for parallel)
1
Enter the status of the switches:
(Press 0 for OFF state and 1 for ON state)
1 0
Circuit is not complete, bulb will not glow
```

(*Contd...*)

```
    else
        printf("Circuit is not complete, bulb will not glow");
    }
    else
    {
     if(s1==ON || s2==ON)
        printf("The bulb will glow");
     else
        printf("Circuit is not complete, bulb will not glow");
    }
}
```

Output window (third execution)

```
Enter the configuration of the circuit:
(Press 1 for series and 2 for parallel)
2
Enter the status of the switches:
(Press 0 for OFF state and 1 for ON state)
1 0
The bulb will glow
```

Output window (fourth execution)

```
Enter the configuration of the circuit:
(Press 1 for series and 2 for parallel)
2
Enter the status of the switches:
(Press 0 for OFF state and 1 for ON state)
0 0
Circuit is not complete, bulb will not glow
```

Program 9-42 | A program that illustrates the use of logical operators on the objects of the enumeration type

f. **Type conversions**: The objects of the enumeration type can participate in the expressions and can be passed as arguments to functions. Whenever necessary, an enumeration type is automatically promoted to an arithmetic type. The piece of code in Program 9-43 illustrates this fact.

Line | **Prog 9-43.c**

```
 1  //Enumerations and Type conversion
 2  #include<stdio.h>
 3  enum  shapes {triangle=3, quadrilateral, pentagon, hexagon};
 4  main()
 5  {
 6      enum shapes s1=triangle, s2=quadrilateral, s3;
 7      printf("The number of vertices in s1 are %d\n", s1);
 8      printf("The number of vertices in s2 are %d\n",s2);
 9      printf("Total number of vertices in s1 and s2 are %d\n",s1+s2);
10      printf("\nNo. of vertices in s3 are twice the no. of vertices in s1\n");
11      s3=2*s1;
12      printf("The number of vertices in s3 are %d\n",s3);
13  }
```

Output window

```
The number of vertices in s1 are 3
The number of vertices in s2 are 4
Total number of vertices in s1 and s2 are 7

No. of vertices in s3 are twice of the no. of vertices in s1
The number of vertices in s3 are 6
```

Remarks:

- Whenever objects of an enumeration type participate in an expression, they are promoted to arithmetic type, if required
- In line number 11, the enumeration type `enum shapes` is initially promoted to an integer type and then later demoted back to the enumeration type `enum shapes`

Program 9-43 | A program that illustrates the implicit-type conversion of an enumeration object

g. **Limitation of enumeration type:** The only limitation of an enumeration type is that it is not possible to print the value of an enumeration object in the symbolic form. The value of an enumeration object is always printed in the integer form. However, a debugger may be able to print the values of enumeration objects in the symbolic form. The piece of code in Program 9-44 illustrates this fact.

Line	Trace	Prog 9-44.c	Output window
1		//Limitation of an enumeration type	The value of var is 1
2		#include<stdio.h>	**Watch window**
3	1	main()	**After trace step-3:** a: 1 /*true*/
4		{	**Remarks:**
5	2	enum BOOLEAN {false, true} var;	• It is not possible to print the value of an enumeration object in the symbolic form
6	3	var=true;	• As shown in the watch window, a debugger prints the value of an enumeration object both in the symbolic form and the integer form
7	4	printf("The value of var is %d",value);	
8		//printf("The value of var is %s",value);	
9	5	}	

Program 9-44 | A program illustrating that the value of an enumeration object cannot be printed in the symbolic form

9.11 Bit-fields

From the knowledge that you have imbibed till now, the smallest amount of information that you can store in the memory is 1 byte, i.e. in the form of character objects. However, most of the computer applications need to process the information smaller than a byte. For example, in data communication, the receiver application needs to check the parity✍ of the received data. The parity of the received data can either be even or odd. Only 1 bit of information is sufficient to specify the parity, i.e. bit will be 0 if the parity is even and 1 otherwise. The receiver also needs to know whether the data communication will be synchronous or asynchronous. Again, this information can be stored using 1 bit, i.e. bit will be 0 if data communication is synchronous and 1 otherwise. If the data communication is asynchronous, the receiver has to explicitly synchronize itself with the sender. For this purpose, the sender sends start bits to the receiver, before sending the actual data. The number of start bits that a sender sends can be 0, 1, 2 or 3. The receiver can store this information by using 2 bits.

✍ **Parity** checking is one of the simplest error-checking techniques. Parity refers to the number of bits with the value of one in a given set of bits. Parity can be either even or odd. If the number of 1's in the given set of bits is even, the parity is said to be even. In odd parity, the number of 1's in the given set of bits is odd.

The receiver applications need to be very compact in size (i.e. memory efficient) as they have to be used in mobile devices. When it is required to make smaller applications, where every bit of the memory space is precious, the memory cannot be wasted by storing the information that takes 1 or 2 bits into separate bytes.

Here, the application of bit-fields comes into the real picture. Bit-fields help in packing several objects into a single unit. They can only be declared as a part of a structure or a union.

In a structure or a union declaration-list, it is possible to specify for a member, the number of bits that it will take in the memory. Such a member is called a **bit-field**. The general form of a bit-field declaration is:

```
struct|union [tag_name]
{
    type member_name : integer_constant_expression;
    [type member_name : integer_constant_expression;]
    ..............
}[variable_name];
```

The important points about the bit-field declaration are as follows:

1. The terms shown in square brackets are optional and might not be present. The terms shown in **bold** are mandatory parts of the declaration. The symbol | stands for OR, i.e. either struct or union should be present.
2. A bit-field declaration can only appear within a structure or a union declaration-list.
3. Bit-fields must be of integral type. Thus, the type that can be specified in the bit-field declaration can be char, int or unsigned int.
4. A compile time integer constant expression specifies the width of the bit-field in bits. It must be non-negative and should not be greater than the number of bits available in an object of the type used in the bit-field declaration. The piece of code in Program 9-45 illustrates this fact.

Prog 9-45.c

```
//The size of a bit-field
#include<stdio.h>
struct receiver
{
    unsigned int parity: 22;
    unsigned int mode: 1;
    unsigned int start_bits:2;
    int data;
};
main()
{
//←Statements...
}
```

Output window

```
Compilation error "Bit field too large"
```

Remarks:

- The number of bits specified for a bit-field should not be more than the number of bits available in an object of the type used in the bit-field declaration
- Thus, it is not possible to specify the size of the bit-field parity to be 22, as the number of bits in an object of the type unsigned int is 16

Program 9-45 | A program that demonstrates a constraint about the size of a bit-field

5. If the value of the constant expression specifying the number of bits in a bit-field is 0, then the declaration should have no declarator (i.e. name of the bit-field). A bit-field having 0 width is known as an **unnamed bit-field**. Unnamed bit-fields cannot be referenced as their content at the run time is unpredictable. They are used as dummy fields for the alignment purposes. The piece of code in Program 9-46 illustrates this fact.

Line **Prog 9-46.c**

```
 1  //Unnamed bit-fields
 2  #include<stdio.h>
 3  struct receiver
 4  {
 5      unsigned int parity: 1;
 6      unsigned int mode: 0;
 7      unsigned int start_bits:2;
 8      int data;
 9  };
10  main()
11  {
12  //←Statements...
13  }
```

Output window

```
Compilation error "Bit field must contain at least one bit"
```

Remarks:

- If the value of constant expression specifying the number of bits in a bit-field is 0, then the declaration should have no declarator
- If the bit field is named, then it must contain at least 1 bit
- Thus, the specification of the declarator mode in line number 6 leads to the compilation error
- Remove the declarator name mode and recompile the code

Program 9-46 | A program illustrating that if the size of a bit-field is 0, the bit-field should be unnamed

The operations that can be performed on the bit-fields are as follows:

1. **Referencing a bit-field:** As bit-fields are a part of a structure or a union object, they are referenced in the same way as other structure or union members are referenced. The piece of code in Program 9-47 illustrates this fact.

Prog 9-47.c

```
//Referencing a bit-field
#include<stdio.h>
struct receiver
{
    unsigned int parity: 1;
    unsigned int mode: 1;
    unsigned int start_bits: 2;
    int data;
};
main()
{
struct receiver mobile_receiver={1, 1, 2, 200};
if(mobile_receiver.parity==0)
    printf("The receiver works with even parity\n");
else
    printf("The receiver works with odd parity\n");
if(mobile_receiver.mode==0)
    printf("The receiver supports synchronous data transmission\n");
else
{
    printf("The receiver supports asynchronous data transmission\n");
    printf("There should be %d start bits\n", mobile_receiver.start_bits);
}
}
```

Output window

```
The receiver works with odd parity
The receiver supports asynchronous data transmission
There should be 2 start bits
```

Remark:

- Bit-fields can be referenced in the same way as other members of structure or union type are referenced, i.e. by using a direct member access operator or an indirect member access operator

Program 9-47 | A program that illustrates how to access the bit-fields

2. **Other operations:** Bit-field behave like an integer object and can participate in expressions in exactly the same way as an object of the integer type would do, regardless of how many bits are there in the bit-field.

   The following operations cannot be performed on bit-fields:

   1. **Address-of a bit-field:** It is not possible to obtain the address of a bit-field member. Unary address-of operator cannot be applied to a bit-field object. Thus, it is not possible to have an array of bit-fields or pointers to bit-fields. The piece of code in Program 9-48 illustrates this fact.

Line | **Prog 9-48.c**

```
 1  //Address-of a bit-field member
 2  #include<stdio.h>
 3  struct receiver
 4  {
 5      unsigned int parity: 1;
 6      unsigned int mode: 1;
 7      unsigned int start_bits: 2;
 8      int data;
 9  };
10  main()
11  {
12  struct receiver mobile_receiver={1, 1, 2, 200};
13  printf("The memory address of object mobile_receiver is %p\n", &mobile_receiver);
14  printf("The memory address of bit-field parity is %p\n",&mobile_receiver.parity);
15  //←Other statements...
16  }
```

Output window

```
Compilation error "illegal to take address of bit-field in function main()"
```

Remarks:

- It is not allowed to take the address of a bit-field member
- Hence, line number 14 is erroneous and leads to a compilation error
- Comment line number 14 and re-execute the code to see that it is possible to take the address of a structure object that contains bit-fields

Program 9-48 | A program to illustrate that it is not possible to take the address of a bit-field

   2. **Size-of a bit-field:** Like the address-of operator, it is not possible to apply the sizeof operator on a bit-field object. The piece of code in Program 9-49 illustrates this fact.

Prog 9-49.c

```
 1  //sizeof a bit-field member
 2  #include<stdio.h>
 3  struct receiver
 4  {
 5      unsigned int parity: 1;
 6      unsigned int mode: 1;
 7      unsigned int start_bits: 2;
 8      int data;
 9  };
10  main()
11  {
12  struct receiver mobile_receiver={1, 1, 2, 200};
13  printf("The size of bit-field object parity is %d\n",sizeof(mobile_receiver.parity));
14  //←Other statements...
15  }
```

Output window

```
Compilation error "sizeof may not be applied to a bit-field in function main()"
```

Remarks:

- It is not allowed to apply the sizeof operator to a bit-field object
- Hence, line number 13 is erroneous and leads to a compilation error

Program 9-49 | A program illustrating that it is not possible to apply the sizeof operator on bit-fields

The important points about the application of the sizeof operator on bit-fields are as follows:

1. An implementation may allocate an addressable storage unit large enough to hold a bit-field. If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation defined.
2. An unnamed bit-field (i.e. bit-field with width 0) indicates that no further bit-field is to be packed in the unit in which the previous bit-field (if any) is placed. The piece of code in Program 9-50 illustrates this fact.

Prog 9-50.c

```
//sizeof structure having unnamed bit-field
#include<stdio.h>
struct receiver
{
    unsigned int parity: 1;
    int :0;                     //←Unnamed bit-field
    unsigned int mode: 1;
    unsigned int start_bits: 2;
    int data;
};
main()
{
struct receiver mobile_receiver;
printf("The size of object mobile_receiver is %d\n", sizeof(mobile_receiver));
//←Other statements...
}
```

Output window (Turbo C 4.5)

```
The size of object mobile_receiver is 6
```

Remarks:

- The unnamed bit-field is used for alignment purposes
- An unnamed bit-field indicates that no further bit-field is to be packed in the unit in which the previous bit-field is placed
- Thus, the bit-fields, i.e. mode and start_bits are placed in the unit next to the unit in which the bit-field parity is placed
- The structure member data takes 2 bytes
- Thus, the total size occupied by the structure object mobile_receiver is 2+2+2= 6 bytes

Program 9-50 | A program that illustrates the use of an unnamed bit-field for alignment purpose

9.12 Summary

1. C language also provides the flexibility to create new types, known as user-defined types.
2. User-defined types can be created by using structures, unions and enumerations.
3. Unlike arrays, the data of different types can be grouped together and stored by making use of structures.
4. A structure is a collection of variables under a single name and provides a convenient way of grouping several pieces of related information together.
5. The structure definition defines a new type, known as structure type.
6. The structure declaration-list in a structure definition consists of declarations of one or more variables, possibly of different types.
7. A structure declaration-list cannot contain a member of void type or incomplete type or function type.
8. A structure definition cannot contain an instance of itself.
9. A structure definition may contain a pointer to an instance of itself. Such a structure is known as self-referential structure.

10. Structure definition does not reserve any space in memory.
11. It is not possible to initialize the structure members during the structure definition.
12. The structure members cannot be initialized during the structure definition, but the members of a structure object can be initialized by providing an initialization list.
13. An unnamed structure type is also known as an anonymous structure type.
14. The member of a structure object can be accessed by using: direct member access operator or indirect member access operator.
15. A structure object can be assigned to a structure variable of the same type.
16. An assignment operator when applied on structure variables performs member-by-member copy.
17. The members of a structure object can be byte aligned or machine-word boundary aligned.
18. If the members of a structure object are machine-word boundary aligned, the padding bytes can appear in between two structure members or after the last structure member.
19. The sizeof operator when applied on a structure object includes the space taken by internal and trailing padding.
20. The use of the equality operator on operands of a structure type is not allowed.
21. An operation that is applicable on an object of a particular type can be applied on a structure member of that type.
22. Like a pointer to any other type, it is possible to create a pointer to a structure type as well.
23. It is possible to define a structure type within the declaration-list of another structure-type definition.
24. Unions are similar to structures except that memory is shared among all the members.
25. The amount of memory allocated to a union object is the amount necessary to contain its largest member.
26. Only the first member of a union object can be initialized.
27. Unions are extensively used in interrupt programming.
28. Enumerations provide another way to create a user-defined type. An enumeration type is designed for variables that can have a limited set of values.
29. In a structure or a union declaration-list, it is possible to specify for a member, the number of bits that it will take in the memory. Such a member is called a bit-field.
30. Bit-fields help in packing several objects into a single unit.

Exercise Questions

Conceptual Questions and Answers

1. *I know that C language provides a rich set of primitive and derived data types for the efficient storage and manipulation of the data. Unfortunately, none of the primitive or derived data types suit my requirements. What should I do so that I can efficiently store and manipulate the data?*

   C language provides a rich set of primitive and derived data types for the efficient storage and manipulation of the data. Even then, in case these data types do not suit your requirements, you can define new data types. These new data types are known as user-defined data types. In C language, the user-defined data types can be created by using structures, unions and enumerations.

Functions are created for the operations allowed on these data types. These user-defined data types along with the defined functions can be used for the efficient storage and manipulation of the data.

2. *What are aggregate types?*

   Aggregate types represent multiple values of the same type or of different types. Aggregate types include arrays and structures. Arrays are used to represent multiple values of the same type, while structures are used to represent multiple values of the same type or of different types.

3. *Like structure type, union type also contains a number of members, possibly of different types. Then, why does the aggregate type not include union type?*

   Aggregate type does not include a union type because an object of a union type can contain only one member at a time.

4. *What are anonymous structures?*

   Unnamed structures are known as anonymous structures. The tag-names are not specified while defining anonymous structures.

5. *Consider the following piece of code:*

   ```
   struct complex
   {
       int re;
       int im;
   }
   main()
   {
       struct complex no1={2,3}, no2={4,5};
       printf("The sum of complex numbers is %d+%di",no1.re+no2.re, no1.im+no2.im);
   }
   ```

   We know that the structure-type definition is a statement and must be terminated with a semicolon. However, Turbo C 3.0 compiler on compiling the above-mentioned code gives no error, although the structure-type definition is not terminated with a semicolon. Why? Can it be avoided to terminate a structure definition with a semicolon?

   The Turbo C 3.0 compiler does not show any error because it interprets the structure-type definition as a return type of the function main. This does not mean that the structure definition should not be terminated with a semicolon. The missing semicolon at the end of a structure definition would lead to a compilation error in the following cases:

   1. Some compilers (e.g. Borland Turbo C 4.5) do not allow the return type of the function main to be any other type except int. In such cases, the mentioned piece of code on compilation gives an error. Hence, the mentioned piece of code will not work with all the compilers.
   2. If there is some declaration statement present after the structure definition with a missing terminating semicolon (as shown below), there will be a compilation error even if it is compiled with a Borland Turbo C 3.0 compiler.

      ```
      struct complex
      {
          int re;
          int im;
      }                   //←The missing semicolon will lead to a compilation error
      int somevariable;   //←Declaration statement
      main()
      ```

```
{
...//←Statements
}
```

6. *I have written the following piece of code:*

```
struct type
{
    char a;
    int b;
    float c;
};
type variable;
```

The mentioned piece of code does not work and gives a compilation error. Why? How can I rectify it?

In C language it is not allowed to declare an object of the defined structure type only by using its tag-name without using the keyword struct. Hence, the statement type variable; is erroneous. There are two ways to rectify this problem:

1. **Using the keyword struct:** Use the keyword struct to declare the object variable of the defined structure type. Hence, the statement type variable; should be written as struct type variable;.
2. **Using the storage class specifier typedef:** Use the storage class specifier typedef to construct a syntactically convenient alias name for the defined structure type and then declare an object using the alias name. The storage class specifier typedef can be used either at the time of the structure definition or after the structure definition in a separate statement as shown below:

```
typedef struct type
{
    char a;
    int b;
    float c;
}type;
type variable; //←Object declaration
```

(a) typedef used at the time of structure definition

```
struct type
{
    char a;
    int b;
    float c;
};
typedef struct type type;
type variable;      //←Object declaration
```

(b) typedef used after the structure definition in a separate statement

7. *I know that a function cannot be defined within the body of another function. However, can I define a structure type within another structure-type definition?*

Yes, a structure type can be defined within another structure-type definition. For example, the structure definitions struct registers, struct word_registers, struct byte_registers shown below are perfectly valid:

```
struct registers
{
    struct word_registers {unsigned int ax, bx, cx, dx, si, di, cflags, flags;} x;
    struct byte_registers {unsigned char al, ah, bl, bh, cl, ch, dl, dh;} h;
};
```

8. *Why does a structure not have an instance of itself?*

Refer Section 9.2.7 to answer this question.

9. *Can a structure have a pointer to itself?*

   Yes, a structure can have a pointer to an instance of itself. Such a structure is known as a self-referential structure.

10. *I know that the* sizeof *operator when applied on the structures returns the total memory space required by all of its members. I have applied the* sizeof *operator on an object of the following structure type:*

```
struct fields
{
    char a;
    int b;
    char c;
    float d;
};
```

   The sizeof *operator is returning a size larger than the sum of size of all the fields. Why? How can I rectify this problem?*

   A structure may have internal and trailing padding to align the structure members with the machine-word boundaries. The sizeof operator counts these internal and trailing padding bytes as well. Thus, the sizeof operator returns a size larger than the sum of size of all the fields of the structure. This problem can be rectified by byte aligning of the members of the structure so that there is no padding. The members of a structure can be byte aligned by using #pragma option -a- (if working with Borland Turbo C 3.0/4.5) or #pragma pack(1) (if working with MS-VC++ 6.0) .

11. *I have defined two structure types* struct type1 *and* struct type2 *as given below.*

```
struct type1                    struct type2
{                                {
    long a;                          char c;
    short b;                         long a;
    char c;                          short b;
};                               };
```

   Both the types have the same members but listed in a different order. Would the sizeof *operator return the size of both the types to be same?*

   No, the sizeof operator would not necessarily return the same size for both the defined structure types. If the members of the structure types are machine-word boundary aligned, the structure members may have the padding in between or at the end. The padding depends upon the order in which of the members of a structure are placed in the structure-type definition. For example, if MS-VC++ 6.0 compiler is used and the pack size is 4 bytes, an object of the type struct type1 will be stored in the memory as:

a				b		c	▨
1001	1000	0101	0010	0000	1111	1011	H
2400	2401	2402	2403	2404	2405	2406	2407

   An object of the structure type struct type2 will be stored in the memory as:

c	▨	▨	▨	a				b		▨	▨
1011	H	H	H	1001	1000	0101	0010	0000	1111	H	H
2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411

The structure member c of the structure type struct type2 can start from any byte boundary. The structure member a can only start from a storage boundary that has an address, which is multiple of 4, i.e. size of type long. Hence, the structure member a cannot start from the memory location 2401, since it is not multiple of 4. It can be placed at memory address 2404 and thus there are 3 padding bytes in between the member c and a. The structure member b can start from memory location 2408, since the memory address is a multiple of 2, i.e. size of type short.

The compiler also places 2 padding bytes after the member b because it wants to ensure the alignment constraints on the next structure object, if there is any. Suppose the compiler does not pad at the end and the member c of the next structure object starts from the memory location 2410. It is a valid start location for the member c since it is of char type. Then, the compiler places 3 padding bytes as it did in the previous object and places the member a at the memory location 2414. However, the memory address 2414 is not divisible by 4, i.e. size of the member a. Hence, it is not a valid start location for the member a. Thus, the compiler cannot enforce alignment constraints by starting the second structure object from the memory location 2410.

Now, suppose the compiler pad places 2 bytes at the end of the first structure object and starts placing the members of the next structure object from the memory location 2412. The member c of the second structure object is placed at the memory location 2412. There will be 3 padding bytes in between the structure members c and a and the member a starts from the memory location 2416. It is a valid start location for the member a, since the address is divisible by 4. Thus, the compiler is able to enforce alignment constraints.

Thus, if the members of the structure objects are machine-word boundary aligned, an object of the type struct type1 will take 8 bytes while an object of the type struct type2 will take 12 bytes. If the members of the structure objects are byte aligned, objects of both the types struct type1 and struct type2 will take the same number of bytes because in byte alignment there is no padding.

12. *Some of the precious memory space can be saved if the members of a structure type are judiciously arranged. Would the compiler do this task for me and rearrange the members of the defined structure type in a manner that requires less padding?*

    No, the members of a structure object are always stored in the order in which they are declared in the structure-type definition. The compiler will never reorder them to improve the alignment and save padding.

13. *Is the following definition of the structure type syntactically correct?*

```
struct car
{
    char* make;
    char* model;
    enum color colour;
};
enum color {red, green, blue};
```

    No, the given definition of the structure type struct car is erroneous. The scope of the enumeration tags begins just after the appearance of the tag in a type specifier that declares the tag (i.e. enumeration tags cannot be used before they are defined). Thus, the usage of the enumeration tag color in the declaration-list of the struct car leads to the compilation error ‘‘color' must be a previously defined enumeration tag’. The code can be rectified by defining enumeration type enum color before the definition of the structure type struct car.

14. *Is the following piece of code syntactically correct? If yes, what would its output be?*

```
struct complex
{
    int re;
    int im;
};
struct complex con(struct complex);
main()
{
    struct complex num={2,3};
    printf("After conjugation, the real and imaginary parts are %d and %d",con(num).re, con(num).im);
}
struct complex con(struct complex num)
{
    num.im=-num.im;
    return num;
}
```

Yes, the given piece of code is syntactically correct and on execution outputs:

```
    After conjugation, the real and imaginary parts are 2 and -3
```

If f is a function returning a structure or a union, and x is a member of that structure or union, then f().x is a valid expression. Thus, con(num).re and con(num).im are valid expressions and evaluates to 2 and -3, respectively.

15. *Like array name and function name, does a structure name point to the base address of the structure?*

    No, like array name and function name (i.e. function designator), the structure name does not point to the base address of the structure. A structure name refers to the entire structure.

16. *Like for arrays, can it be said with certainty that the members of a structure object are stored in contiguous memory locations?*

    No, like arrays it cannot be said with certainty that the members of a structure object are stored in contiguous memory locations. Members of a structure object may have padding in between.

17. *Given the following type definition and object declarations:*

    ```
    struct t { int i; const int ci;};
    struct t t;
    const struct t ct;
    ```

    What would be the type of the following expressions?

    1. t.i
    2. t.ci
    3. ct.i
    4. ct.ci

    The given expressions are of the following types:

1.	t.i	int
2.	t.ci	const int
3.	ct.i	const int
4.	ct.ci	const int

18. *Why does the equality operator (==), inequality operator (!=) and other relational operators not work on structures?*

    The equality operator, inequality operator and relational operators do not work on structures because there is no way for a compiler to implement structure comparison. A simple byte-by-byte comparison would fail while comparing the random bits present in the internal padding. A member-by-member comparison might require unacceptable amounts of repetitive code for large structures. Also, any compiler-generated comparison would not compare the members appropriately in all cases, e.g. the members of the type char* should be compared with the strcmp function instead of being compared with equality (==) operator.

19. *How can I find the byte offset of a member within a structure?*

The byte offset of a member within a structure can be found by using offsetof macro defined in the header file stddef.h. The offsetof macro accepts the name of the structure type as the first argument and the name of member whose offset is to be found as the second argument. It returns the byte offset of the member as an integer value. The following piece of code illustrates the use of the offsetof macro to find the offset of a member:

```
#include<stddef.h>
struct type
{
    char a;
    int b;
    char c;
    float d;
};
main()
{
printf("The offset of member a is %d\n",offsetof(struct type, a));
printf("The offset of member b is %d\n",offsetof(struct type, b));
printf("The offset of member c is %d\n",offsetof(struct type, c));
printf("The offset of member d is %d\n",offsetof(struct type, d));
}
```

The mentioned piece of code on execution using Borland Turbo C 3.0/4.5 outputs:

```
The offset of member a is 0
The offset of member b is 1
The offset of member c is 3
The offset of member d is 4
```

The important points about the usage of the offsetof macro are as follows:

1. The output of the offsetof macro depends upon how the structure members are aligned (i.e. byte aligned or machine-word boundary aligned). Make the structure members machine-word boundary aligned in the above-mentioned code by using #pragma option -a and then re-execute the above-mentioned code and look at the output.
2. If the macro is not previously defined, define it as:

```
#define offsetof(s_name, m_name) (size_t)&(((s_name*)0)->m_name)
```

20. *How is the declaration* struct type {.......}; *different from* typedef struct {........} type;*?*

The differences between the two declaration statements are as follows:

struct type{.....};	typedef struct {......} type;
1. This declaration statement declares a structure tag name (i.e. type) 2. The keyword struct is to be used while declaring objects of the defined type (e.g. struct type objects;) 3. The requirement of using the keyword struct to declare the instances of the defined structure type is a bit inconvenient	1. This declaration statement declares a typedef name (i.e. an alias name) 2. The objects can be declared just by using the typedef name (e.g. type objects;). There is no need to use the keyword struct 3. This form of declaration is slightly more abstract. For example, from the declaration statement type objects; the user does not come to know that type refers to a structure type as the keyword struct is not used

21. *I have heard that a structure can have a pointer to itself but the mentioned piece of code is not working and is giving a compilation error. Why?*

```
typedef struct
{
    int data;
    NODE* link;
} NODE;
```

The storage class specifier typedef creates a new name (i.e. an alias name) for a type. We can define a new structure type and create a typedef name (i.e. alias name) for it at the same time. However, a typedef name cannot be used until it is defined. In the given question, the typedef name NODE is used before it is defined. This leads to a compilation error. There are two ways to rectify this problem:

1. Instead of defining an unnamed type, define a named type by giving a tag name to the structure (e.g. struct node). Then declare the field link as struct node* link;. The rectified code is mentioned below:

```
typedef struct node
{
    int data;
    struct node* link;
} NODE;
```

2. Disentangle the typedef definition from the structure definition and place it before the structure definition as shown below:

```
typedef struct node NODE;
struct node
{
    int data;
    NODE *link;
};
```

22. *Is the definition of the following union type syntactically correct? If yes, what would be the size of an object of the following union type?*

```
union numbers
{
    struct {char a[10];} one;
    struct {int a[10];} two;
    struct {float a[10];} three;
};
```

Yes, the definition of the union type union numbers is syntactically valid. The amount of the memory allocated to a union object is equal to the size of its largest member. Since the largest member in the given union type union numbers is three (i.e. of 4×10=40 bytes), the size of an object of the union type union numbers would be 40.

23. *Is the following piece of code syntactically correct? If yes, what would its output be?*

```
typedef struct error
{
    int warning, error, exception;
} error;
main()
{
    error err;
    err.error=2;
    switch(err.error)
    {
        case 1:  printf("Some warnings are there\n"); break;
        case 2: printf("Some error occurred\n"); break;
        case 3: printf("Some exception is there\n");
    }
}
```

Yes, the following piece of code is syntactically correct and on execution outputs 'Some error occurred'. Based upon the context, the compiler can distinguish between the different usages of the name error. For example, in the statement error err; the usage of error is treated as the typedef name. In the statement err.error=2; the usage of error is treated as the member name. If there would have been a statement like struct error err; the usage of error would have been treated as the structure tag name.

24. *How can I keep track of which field of a union is in use?*

There is no automatic way to keep a track of which union field is in use. However, we can create a type with an additional member, which keeps a record of the union field currently in use. The following code segment illustrates the definition of such type:

```
struct trackedunion
{
    enum {UNKNOWN, CHAR, INT, FLOAT, LONG, DOUBLE} code;
    union
    {
        char a;
        int b;
        float c;
        long d;
        double e;
    } u;
};
```

Initially the value of code is set to UNKNOWN, because it is not known that which union field is in use. After that, whenever a value is assigned to a union field, the code field is set appropriately. Thus, the code field keeps a track of which union field is being last written to.

25. *What are the differences between a symbolic constant and an enumeration constant?*

The important differences between symbolic constants and enumeration constants are as follows:

Symbolic constants	Enumeration constants
1. Symbolic constants are created with the help of define directive 2. The symbolic constants have global scope. They can be used throughout the translation unit (i.e. file) after their definition 3. Symbolic constants do not have any type associated with them 4. The values of the symbolic constants are to be mentioned explicitly	1. Enumeration constants are created as a part of enumeration type definition 2. The enumeration constants have the local scope. They can only be used in the scope in which the enumeration type has been defined 3. Enumeration constants are of integer type 4. The values of the enumeration constants are set automatically. By default, the first enumeration constant has value 0

26. *Instead of printing the values of the enumeration constants, I want to print them symbolically. How can I do that?*

    The only limitation of an enumeration type is that it is not possible to print the value of an enumeration object in their symbolic form. By default, the value of an enumeration object is always printed in the integer form. However, you can write your own function to map an enumeration value into a string. The following piece of code illustrates one such function map that maps an enumeration value into a string:

```
#include<stdio.h>
enum BOOLEAN {FALSE, TRUE};
char* map(enum BOOLEAN);
main()
{
    enum BOOLEAN a, b;
    a=FALSE;
    b=TRUE;
    printf("The values of a and b in integer form are %d and %d\n", a, b);
    printf("The values of a and b in symbolic form are %s and %s",map(a), map(b));
}
char* map(enum BOOLEAN a)
{
    switch(a)
    {
        case 0:
            return "FALSE";
        case 1:
            return "TRUE";
    }
}
```

The above-mentioned piece of code on execution outputs:

```
The values of a and b in integer form are 0 and 1
The values of a and b in symbolic form are FALSE and TRUE
```

27. *Is the following piece of code syntactically correct? If yes, what would its output be?*

```
typedef enum error {warning, error, exception} error;
main()
```

```
{
    error err;
    err=error;
    switch(err)
    {
        case 1:  printf("Some warnings are there\n"); break;
        case 2: printf("Some error occurred\n"); break;
        case 3: printf("Some exception is there\n");
    }
}
```

No, the mentioned piece of code is syntactically incorrect and on compilation leads to 'Multiple declarations for 'error'' error. The compiler shows an error because based upon the context, it is not able to distinguish between the use of error as an alias name in the declaration statement error err; and error as an enumeration constant in the assignment statement err=error;.

28. *What is the efficient way to store flag values?*
    Flag refers to one or more bits that are used to store a value that has an assigned meaning. Flags are generally used to control or indicate the outcome of different operations. For example, the carry flag of microprocessor is set to 1 if an addition operation generates a carry out of the most significant bit position. Similarly, the zero flag is set to 1 when the result of an operation is zero (e.g. subtraction of two equal numbers). Flags can be efficiently stored by making the use of bit fields.

29. *What are unnamed bit-fields? Why are they used?*

    Bit-fields with length 0 are known as unnamed bit-fields. Unnamed bit-fields are used for the alignment purposes. An unnamed bit-field indicates that the next field should be placed in a separate unit and not with the previous field in the same unit.

30. *'Use of standard library functions increase the size of the executable file but the use of interrupt functions does not increase the size of the executable file'. Is this statement true?*

    No. This statement is false. The Turbo C library functions also use the interrupts and were written by programmers. The only difference between the library functions and the interrupts is in the ease of usage. The library functions are easier to use and are more flexible as compared to interrupts.

Code Snippets

Determine the output of the following code snippets. Assume that the inclusion of the required header files has been made and there is no prototyping error due to them.

31.
```
struct book
{
    char title[20];
    char author[20];
    int pages;
    float price;
};
main()
```

```
{
    book Cbook;
    printf("The size of object Cbook is %d bytes", sizeof(Cbook));
}
```

32.
```
struct book
{
    char title[20];
    char author[20];
    int pages;
    float price;
};
main()
{
    struct book Cbook;
    Cbook.title="The power of positive attitude";
    Cbook.author="P Subramanyam";
    Cbook.pages=400;
    Cbook.price=225.50;
    printf("%s by %s is of %f rupees", Cbook.title, Cbook.author, Cbook.price);
}
```

33.
```
struct book
{
    char *title;
    char *author;
    int pages;
    float price;
};
main()
{
    struct book Cbook;
    Cbook.title="The power of positive attitude";
    Cbook.author="P Subramanyam";
    Cbook.pages=400;
    Cbook.price=225.50;
    printf("%s by %s is of %f rupees", Cbook.title, Cbook.author, Cbook.price);
}
```

34.
```
struct car
{
    struct engine e;
    struct chassis c;
};
struct engine
{
    int hp;
    int cc;
};
struct chassis
```

```
{
    int length;
    int width;
};
main()
{
    struct car yourcar={{68, 1400}, {3200, 2358}};
    if(yourcar.e.cc<1400)
        printf(“It is a small segment car\n”);
    else if(yourcar.e.cc>=1400 && yourcar.e.cc<1600)
        printf(“It is a middle segment car\n”);
    else
        printf(“It is a big segment car\n”);
}
```

35.

```
struct car
{
    struct engine {int hp; int cc;} e;
    struct chassis {int length; int width;} c;
};
main()
{
    struct car yourcar={{68, 1400}, {3200, 2358}}, mycar={{52, 1000},{3500,2500}};
    if(yourcar.c.length>mycar.c.length)
        printf(“Your car is lengthier than mine\n”);
    else
        printf(“My car is lengthier than yours\n”);
}
```

36.

```
//Assuming the compiler used is Borland Turbo C 4.5
struct car
{
    char *make;
    char *model;
    char *reg_no;
    struct {int hp; int cc;} e;
    struct {int length; int width; char* color;} c;
    float cost;
};
main()
{
    struct car mycar;
    printf(“The size of type struct car is %d\n”,sizeof(struct car));
    printf(“The objects of type struct car will take %d bytes of memory\n”, sizeof(mycar));
}
```

37.

```
struct car
{
    char *manufacturer=”Maruti”;
    char *make;
};
```

```
    main()
    {
        struct car mycar, yourcar;
        mycar.make=”Swift”;
        yourcar.make=”Dzire”;
        printf(“We own %s and %s cars manufactured by %s”, mycar.make, yourcar.make, mycar.manufacturer);
    }
```

```
38. struct car
    {
        char *make;
        char *model;
    };
    main()
    {
        struct car mycar={“Maruti”, ”Dzire”};
        struct car yourcar=mycar;
        strupr(yourcar.make);
        strupr(yourcar.model);
        printf(“Your car is %s-%s\n”,yourcar.make, yourcar.model);
        printf(“My car is %s-%s\n”,mycar.make, mycar.model);
    }
```

```
39. struct car
    {
        char *make;
        char *model;
    };
    main()
    {
        struct car mycar={“Maruti”, ”Dzire”}, yourcar={“Maruti”, ”Dzire”};
        if(mycar==yourcar)
            printf(“Both of us own car of same make and model”);
        else
            printf(“We own different types of cars”);
    }
```

```
40. struct car
    {
        char *make;
        char *model;
    };
    main()
    {
        struct car mycar={“Maruti”, ”Dzire”}, yourcar={“Maruti”, ”Dzire”};
        if(mycar.make==yourcar.make && mycar.model==yourcar.model)
            printf(“Both of us own car of same make and model”);
        else
            printf(“We own different types of cars”);
    }
```

41.
```
struct car
{
    char *make;
    char *model;
};
main()
{
    struct car mycar={“Maruti”, ”Dzire”}, yourcar={“Maruti”, ”Dzire”};
    if(strcmp(mycar.make,yourcar.make)==0 && strcmp(mycar.model, yourcar.model)==0)
        printf(“Both of us own cars of same make and model”);
    else
        printf(“We own different types of cars”);
}
```

42.
```
struct 3Dpoints
{
    int x;
    int y;
    int z;
};
main()
{
    struct 3Dpoints pt1, pt2;
    pt1.x=pt2.x=20;
    pt1.y=10; pt2.y=30;
    printf(“Points in xy plane are:\n”);
    printf(“Pt1(%d %d)\n”, pt1.x, pt1.y);
    printf(“Pt2(%d %d)\n”,pt2.x,pt2.y);
}
```

43.
```
struct complex
{
    int re;
    int im;
};
main()
{
    struct complex number={2,3};
    int *ptr1=&number.re, *ptr2=&number.im;
    if(ptr2>ptr1)
        printf(“The imaginary part is stored towards the right of real part in the number object\n”);
    else if(ptr1>ptr2)
        printf(“The real part is stored towards the right of imaginary part in the number object\n”);
    else
        printf(“Both the real part and imaginary part overlap\n”);
}
```

44.
```
struct point
{
    int x, y;
};
```

```
main()
{
     struct point origin;
     printf("The coordinates of origin are %d,%d", origin.x, origin.y);
}
```

45.
```
struct point
{
     int x, y;
};
main()
{
     struct point origin={0};
     printf("The coordinates of origin are %d,%d", origin.x, origin.y);
}
```

46.
```
struct point
{
     int x, y;
};
main()
{
     static struct point origin;
     printf("The coordinates of origin are %d,%d", origin.x, origin.y);
}
```

47.
```
#include<alloc.h>
struct node
{
     int data;
     struct node *link;
};
main()
{
     struct node* ptr, *temp;
     ptr=(struct node*)malloc(sizeof(struct node));
     ptr->data=10;
     temp=(struct node*)malloc(sizeof(struct node));
     ptr->link=temp;      temp->data=20;
     temp=(struct node*)malloc(sizeof(struct node));
     ptr->link->link=temp;      temp->data=30;
     temp->link=NULL;
     temp=ptr;
     while(temp!=NULL)
     {
          printf("%d\t",temp->data);
          temp=temp->link;
     }
}
```

48.
```
struct complex
{
    int re;
    int im;
};
main()
{
    struct complex no={2,3};
    struct complex* cptr=&no;
    printf("The real and imaginary parts of complex number are %d and %d", *cptr.re, *cptr.im);
}
```

49.
```
struct complex
{
    int re;
    int im;
};
main()
{
    struct complex no={2,3};
    struct complex* cptr=&no;
    printf("The real and imaginary parts of complex number are %d and %d\n", (*cptr).re, (*cptr).im);
    printf("The real and imaginary parts of complex number are %d and %d", cptr->re, cptr->im);
}
```

50.
```
union contactno
{
    char mobileno[10];
    char landlineno[10];
    char pagerno[10];
};
main()
{
    union contactno electrician={"9416234213", "5356785", "941-998856"};
    printf("The mobile number of my electrician is %s\n", electrician.mobileno);
    printf("You can also contact him on his landline number %s", electrician.landlineno);
}
```

51.
```
union coordinates
{
    int x;
    int y;
};
main()
{
    union coordinates point;
    point.x=20;
    point.y=30;
    printf("The coordinates of point are %d,%d", point.x, point.y);
}
```

52.

```
#define struct union
struct type
{
    char a;
    int b;
    float c;
};
main()
{
    printf(“The size of defined structure type is %d”, sizeof(struct type));
}
```

53.

```
typedef struct union;
struct type
{
    char a;
    int b;
    float c;
};
main()
{
    printf(“The size of defined structure type is %d”, sizeof(struct type));
}
```

54.

```
enum color {red, green, blue};
main()
{
    printf(“The values of enumeration constants are %d %d %d”, red, green, blue);
}
```

55.

```
enum color {red, green=red, blue=green};
main()
{
    printf(“The values of enumerations constants are %d %d %d”, red, green, blue);
}
```

56.

```
enum values {a, b=32767, c};
main()
{
    printf(“Values of enumeration constants are %d %d %d”, a, b, c);
}
```

57.

```
enum values {a=2, b=3, c};
main()
{
    int var=7, res1, res2;
    res1=var%b;
    res2=res1%res2;
    printf(“The values of res1 and res2 are %d and %d”,res1, res2);
}
```

58.

```
main()
{
    int bitfield: 2;
    bitfield=3;
    printf(“The value of bitfield is %d”,bitfield);
}
```

59.

```
int parity=1;
struct dataobject
{
    int paritybits: parity;
    int data;
};
main()
{
    int i, count=0;
    struct dataobject obj={0, 2, 23};
    while(obj.data!=0)
    {
        if(obj.data%2==1)
            count++;
        obj.data=obj.data>>1;
    }
    if(count%2==0)
    {
        obj.paritybits=0;
        printf(“The data has even parity”);
    }
    else
    {
        obj.paritybits=1;
        printf(“The data has odd parity”);
    }
}
```

60.

```
struct dataobject
{
    int paritybits: 1;
    int data;
};
main()
{
    int i, count=0;
    struct dataobject obj={0, 2, 23};
    while(obj.data!=0)
    {
        if(obj.data%2==1)
            count++;
        obj.data=obj.data>>1;
    }
```

```
        if(count%2==0)
        {
            obj.paritybits=0;
            printf("The data has even parity");
        }
        else
        {
            obj.paritybits=1;
            printf("The data has odd parity");
        }
    }
```

Multiple-choice Questions

61. User-defined types can be created by using
    a. Structures
    b. Unions
    c. Enumerations
    d. All of these

62. A structure declaration-list cannot contain a member of
    a. void type
    b. Incomplete type
    c. Function type
    d. All of these

63. Objects of the defined structure type can be created
    a. At the time of structure declaration
    b. After the structure declaration
    c. Either at the time of structure declaration or after the structure declaration
    d. None of these

64. A member of a structure object can be accessed through the structure object name by using
    a. Direct member access operator
    b. Indirect member access operator
    c. Arrow operator
    d. None of these

65. A member of a structure object can be accessed through a pointer to the structure object by using
    a. Direct member access operator
    b. Indirect member access operator
    c. Dereference operator
    d. None of these

66. Which of the following operators is not applicable on an object of a structure type?
    a. Equality operator
    b. Assignment operator
    c. Address-of operator
    d. sizeof operator

67. Nested structure contains members of
    a. Same structure type
    b. Other defined structure types
    c. Incomplete structure types
    d. None of these

68. The maximum number of members in a structure declaration-list
    a. Can be two
    b. Can be infinite
    c. Depends upon the translation limits of the compiler
    d. None of these

69. Which of the following method of passing a structure object to a function is most efficient?
    a. Passing each member of a structure object as a separate argument
    b. Passing a structure object by value
    c. Passing a structure object by address/ reference
    d. None of these

70. The amount of the memory allocated to a union object is
    a. The amount of memory necessary to contain its largest member
    b. The amount of memory necessary to contain its smallest member
    c. The sum of memory requirement of all of its members
    d. None of these

71. Which member(s) of a union object can be initialized?
    a. Only first member
    b. Only last member
    c. All members
    d. None of these

72. An enumeration constant is of
    a. char type
    b. int type
    c. float type
    d. None of these

73. The width specifier of a bit field should be a
    a. Variable
    b. Constant
    c. Compile time constant expression of integer type
    d. None of these

74. The value of a constant expression specifying the width of a bit field cannot be
    a. 0
    b. 1
    c. Greater than the number of bits available in an object of the type used in bit field declaration
    d. None of these

75. A bit-field of which of the following types cannot be created
    a. int
    b. unsigned int
    c. char
    d. float

Outputs and Explanations to Code Snippets

31. Compilation error "Undefined symbol 'book' in function main"

    Explanation:

    In C language, it is not allowed to declare an object of the defined structure type by using its tag-name without using the keyword struct. Hence, the declaration statement book Cbook; is erroneous and leads to the compilation error. To rectify the code, use the keyword struct in the declaration statement and write it as struct book Cbook; or use the storage class specifier typedef to create book as an alias name for the structure type struct book.

32. Compilation error "L-value required in function main"

    Explanation:

    Both the expressions Cbook.title and Cbook.author are of type char[20] (i.e. array type) and do not have an l-value. Hence, they cannot be placed on the left side of the assignment operator. Placement of these expressions on the left side of the assignment operator leads to the specified compilation error.

33. The power of positive attitude by P Subramanyam is of 225.500000 rupees

    Explanation:

    The expressions Cbook.title and Cbook.author are of type char* and have l-values. Hence, they can be assigned the base addresses of the strings.

34. Compilation errors
    "Undefined structure 'engine'"
    "Undefined structure 'chassis'"
    "Size of the type is unknown or zero"

    Explanation:

    Structure tags have the scope that begins just after the appearance of the tag in a type specifier that declares the tag. The usage of the structure tag-names engine and chassis in the declaration-list of the structure type struct car leads to 'Undefined structure 'engine'' and 'Undefined structure 'chassis' errors because the structure tags have not yet been defined. Also, a structure definition cannot contain a member of the incomplete type. A structure type is said to be incomplete until the closing brace of its declaration-list is encountered. An incomplete type lacks the information needed to determine the size of its object. Hence, the usage of incomplete types struct engine and struct chassis in the declaration-list of struct car leads to the 'Size of the type is unknown' error. To rectify the code, define the structure types struct engine and struct chassis before the definition of the structure type struct car.

35. My car is lengthier than yours

    Explanation:

    It is allowed to define a structure type within another structure-type definition. Hence, the definitions of the structure types struct engine and struct chassis in the declaration-list of struct car are perfectly valid. Also, the members e and c are of the complete type because before their occurrence the closing brace of their respective structure types, i.e. struct engine and struct chassis has already been seen by the compiler. The length member of the member c of the objects yourcar and mycar is initialized with the values 3200 and 3500, respectively. Hence, the expression yourcar.c.length>mycar.c.length evaluates to false and "My car is lengthier than yours" gets printed.

36. The size of type struct car is 28
    The objects of type struct car will take 28 bytes of memory

    Explanation:

    The specified result is the result of the execution in Turbo C 4.5.

    Refer Section 9.2.3.1.4 to answer this question.

37. Compilation error

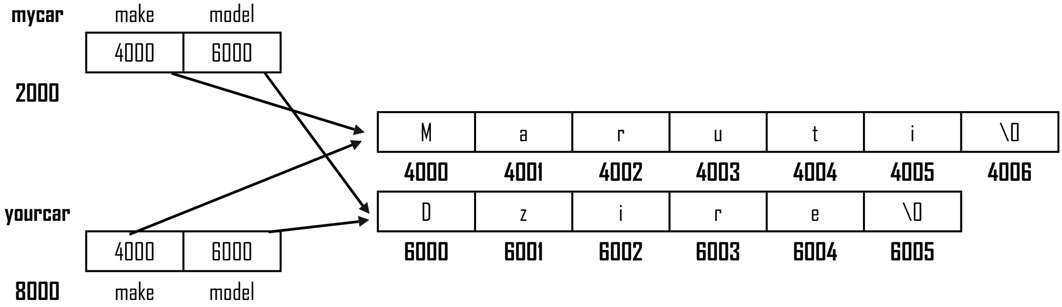
    Explanation:

    Since the structure definition does not reserve any memory space for the structure members, it is not possible to initialize the structure members during the structure definition. Hence, the initialization of the structure member manufacturer with the string literal "Maruti" during the structure definition is erroneous and leads to the compilation error.

38. Your car is MARUTI-DZIRE
    My car is MARUTI-DZIRE

    Explanation:

    Suppose the structure object mycar gets allocated at the memory location 2000. The make and the model members of the structure object mycar are initialized with the string literals "Maruti" and "Dzire" (say located at memory locations 4000 and 6000, respectively). Thus, they point to the base addresses of the strings. Another structure object yourcar, say gets allocated at the memory location 8000. Since the structure object yourcar is initialized with the structure object mycar, all the members of mycar are copied one by one to the corresponding members of yourcar. Thus, the make and model members of the structure object yourcar also start pointing to the strings located at the memory locations 4000 and 6000, respectively. This is shown in the figure below:



As the corresponding members of the structure objects mycar and yourcar point to the same memory locations (i.e. same strings), the changes made in the strings through yourcar.make and yourcar.model will also be available through mycar.make and mycar.model.

39. Compilation error "Illegal structure operation in function main"

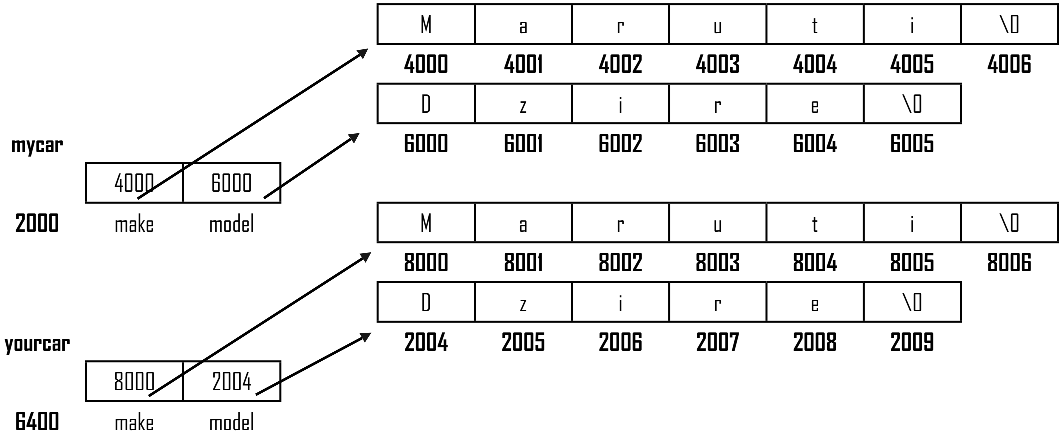
Explanation:

The use of the equality operators on the structures is not allowed. Hence, the expression mycar==yourcar is erroneous and leads to a compilation error.

40. We own different types of cars

Explanation:

Suppose the structure objects mycar and yourcar get allocated at the memory locations 2000 and 6400, respectively. The members of these structure objects point to the string literals as shown in the figure given below:



The sub-expression mycar.make==yourcar.make compares the value 4000 with 8000 and hence evaluates to false. Similarly, the sub-expression mycar.model==yourcar.model also evaluates to false. Thus, the if expression evaluates to false and the printf statement present in the else body gets executed.

The specified code gives the unexpected output because the equality operator compares the pointers instead of the strings pointed to by the pointers.

41. Both of us own cars of same make and model

    Explanation:

    In the given code, the strcmp function is used to compare the strings pointed to by the pointers. Since the strings compare equal, the if expression evaluates to true and the printf statement present in the if body gets executed.

42. Compilation error

    Explanation:

    The tag-name of a structure is an identifier and must start with a letter or an underscore. 3Dpoints is not a valid identifier name and hence cannot form a structure tag-name.

43. The imaginary part is stored towards the right of real part in the number object

    Explanation:

    If the objects pointed are the members of the same structure object, pointers to the structure members declared later compare greater than the pointers to the members declared earlier in the structure. Thus, ptr2>ptr1 evaluates to true.

44. The coordinates of origin are 7903,19125

    Explanation:

    Since the structure object origin is defined inside the body of the function main, it has local scope. Thus, its members will not be automatically initialized and will contain garbage values.

45. The coordinates of origin are 0,0

    Explanation:

    If the number of initializers in the initialization list is less than the number of structure members in a structure object, the leading structure members (equal to the number of initializers in the initialization list) are initialized with the initializers in the initialization list and the rest of the structure members will automatically be initialized with 0. Thus, in the given code, the member y of the structure object origin automatically gets initialized to 0.

46. The coordinates of origin are 0,0

    Explanation:

    If a structure object is declared with a storage class specifier, the properties resulting from the storage class specifier (except with respect to linkage) apply to all the members of the object. Thus, as the structure object origin is declared with the storage class specifier static, all the members of the structure object will automatically be initialized to 0.

47. 10 20 30

    Explanation:

    The code creates a linked list of three nodes. The data fields of the nodes are assigned the values 10, 20 and 30, respectively. The while loop is used to traverse the list and print the values of the data field.

48. Compilation error "Structure required on the left side of . in function main"

    Explanation:

    The dot operator has higher precedence than the dereference operator. Hence, the expression *cptr.re is interpreted as *(cptr.re). The dot operator on its left side expects a structure name. Since in the interpreted expression pointer to a structure is present on the left side of the dot operator instead of a structure name, there is a compilation error.

49.
```
The real and imaginary parts of complex number are 2 and 3
The real and imaginary parts of complex number are 2 and 3
```

Explanation:

The members of a structure object can be accessed via the pointer to the structure object by using one of the following two ways:

1. By using a dereference or indirection operator and dot operator
2. By using an arrow operator

50.
```
Compilation error
```

Explanation:

It is not allowed to initialize all the members of a union object. Only the first member of the union object can be initialized.

51.
```
The coordinates of point are 30,30
```

Explanation:

In the union object point, both the members x and y share the memory locations. Changing the value of a member will change the value of the other member too. Thus, assignment of the value 30 to the member y will also change the value of the member x from 20 to 30.

52.
```
The size of defined structure type is 4
```

Explanation:

During the preprocessing stage, the macro struct is text replaced by the replacement string union wherever it appears in the program code. Thus, after the preprocessing stage, the code becomes

```
union type
{
    char a;
    int b;
    float c;
};
main()
{
    printf("The size of defined structure type is %d", sizeof(union type));
}
```

When the sizeof operator is applied on a union type, it outputs the size of its largest member. Thus, sizeof(union type) returns 4.

53.
```
Compilation error
```

Explanation:

The storage class specifier typedef is used for creating a synonym name or alias for a known type. The syntax of the typedef declaration is:

```
typedef type_name synonym_name;
```

The type_name should be a defined type. Since in the given declaration, struct is not a defined type, the statement is erroneous and on compilation shows an error '{ expected'. The compiler expects structure declaration-list after the keyword struct. Also, the synonym_name should be a valid identifier name. In the given declaration statement, synonym name is union, which is a keyword and not a valid identifier name. This also leads to an error.

54. The values of enumeration constants are 0 1 2

    Explanation:

    The values of the enumeration constants are set automatically. The first enumerator has the value 0. Each subsequent enumerator, if not explicitly assigned a value, has a value 1 greater than the value of the enumerator that immediately precedes it. Thus, the enumeration constant red will have the value 0, green will have the value 1 and blue will have the value 2.

55. The values of enumeration constants are 0 0 0

    Explanation:

    Each enumeration constant has a scope that begins just after its appearance in an enumeration list. Thus, it is possible to initialize the enumerator green with the enumerator red and the enumerator blue with the enumerator green.

56. Compilation error "The value for 'c' is not within the range of an int"

    Explanation:

    An enumerator can hold an integer value. Also, each subsequent enumerator in an enumeration list, if not explicitly assigned with a value, has a value 1 greater than the value of the enumerator that immediately precedes it. Thus, the enumerator c will have the value 32767+1, i.e. 32768. Since the value of the enumerator c falls outside the range of the integer type, there will be a compilation error.

57. The value of res1 and res2 are 1 and 1

    Explanation:

    All the operators that work on an integer type can be applied on objects of an enumeration type and the operators applicable on integer constants can be applied on enumerators. Thus, the application of the modulus operator on the objects of enumeration type res1 and res2 and enumerator b is perfectly valid.

58. Compilation error

    Explanation:

    A bit-field declaration can only appear within a structure or a union declaration-list.

59. Compilation error "Constant expression required"

    Explanation:

    The width specifier of a bit-field can be a constant expression of the integer type. In the given piece of code, the variable parity is used to specify the width of the bit-field paritybits. This is erroneous and leads to the specified compilation error.

60. The data has even parity

    Explanation:

    The while loop is used to count the number of 1's in the data member of the structure object obj. After the termination of the while loop, the value of the variable count is equal to the number of 1's in the data member. If the value of variable count is even, the bit-field paritybits of the object obj is set to 0 else it is set to 1. A message indicating the parity of data is also printed.

Answers to Multiple-choice Questions

61. d 62. d 63. c 64. a 65. b 66. a 67. b 68. c 69. c 70. a 71. a 72. b 73. c 74. c 75. d

Programming Exercises

Program 1 | Define a data type for storing complex numbers and implement addition, subtraction, multiplication, conjugate and modulus operations for the defined type

PE 9-1.c

```
//Definition of complex data type and various operations applicable on it
#include<stdio.h>
#include<math.h>
#include<string.h>
struct complex
{
    int re;
    int im;
};
typedef struct complex COMP;
COMP add(COMP, COMP);
COMP sub(COMP*, COMP*);
COMP mult(COMP, COMP);
COMP conjugate(COMP);
float modulus(COMP);
void print(char* opr, COMP result, char* no='\0');
void printmod( char*, float);
main()
{
    COMP no1, no2, result;
    float mod;
    printf("Enter the real and imaginary part of first complex number:\n");
    scanf("%d %d", &no1.re, &no1.im);
    printf("Enter the real and imaginary part of second complex number:\n");
    scanf("%d %d", &no2.re, &no2.im);
    result=add(no1, no2);
    print("addition", result);
    result=sub(&no1, &no2);
    print("subtraction", result);
    result=mult(no1, no2);
    print("multiplication", result);
    result=conjugate(no1);
    print("conjugate", result, "no1");
    mod=modulus(no1);
    printmod("no1",mod);
}
COMP add(COMP no1, COMP no2)
{
    COMP result;
    result.re=no1.re+no2.re;
    result.im=no1.im+no2.im;
    return result;
}
```

Output window

```
Enter the real and imaginary part of first complex number:
2 3
Enter the real and imaginary part of second complex number:
4 5
The result of addition is 6+8i
The result of subtraction is −2−2i
The result of multiplication is −7+19i
The result of conjugate of no1 is 2−3i
The result of modulus of no1 is 3.605551
```

(*Contd...*)

Line	PE 9-1.c	Output window

```
COMP sub(COMP* no1, COMP* no2)
{
    COMP result;
    result.re=no1->re-no2->re;
    result.im=no1->im-no2->im;
    return result;
}
COMP mult(COMP no1, COMP no2)
{
    COMP result;
    result.re=no1.re*no2.re – no1.im*no2.im;
    result.im=no1.re+no2.im + no1.im* no2.re;
    return result;
}
COMP conjugate(COMP no)
{
    COMP result;
    result.re=no.re;
    result.im=-no.im;
    return result;
}
float modulus(COMP no)
{
    float result;
    result=pow((no.re*no.re+no.im*no.im), 0.5);
    return result;
}
void print(char* opr, COMP res, char* no)
{
    if(strcmp(opr, ”conjugate”)==0)
    {
       if(res.im<0)
         printf(“The result of conjugate of %s is %d%di\n”,no,res.re,res.im);
       else
         printf(“The result of conjugate of %s is %d+%di\n”,no,res.re,res.im);
    }
    else
    {
         if(res.im<0)
              printf(“The result of %s is %d%di\n”,opr, res.re,res.im);
         else
              printf(“The result of %s is %d+%di\n”,opr, res.re,res.im);
    }
}
void printmod(char* no, float result)
{
    printf(“The result of modulus of %s is %f\n”, no, result);
}
```

Program 2 | Develop a phonebook application. It should be able to store, modify and list entries present in the phonebook. A phonebook entry consists of the name of a person and his contact information. The name of a person consists of his first name and family name. The contact information consists of the landline number and the mobile number of the person

Line **PE 9-2.c**

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<conio.h>
typedef struct name          //←Definition of struct type name
{
    char fname[20];
    char lname[20];
}NAM;                        //←struct type name is aliased as NAM
typedef struct contact       //← Definition of struct type contact
{
    char landline[12];
    char mobile[12];
}CON;                        //←struct type contact is aliased as CON
typedef struct phoneentry    //←Definition of struct type phoneentry
{
    NAM pname;
    CON pcontact;
}PENT;                       //←struct type phoneentry is aliased as PENT

void printmenu()             //←Function printmenu prints various options
{
    printf("****************************************\n");
    printf("1. Press 1 to add records in phone book\n");
    printf("2. Press 2 to delete a record\n");
    printf("3. Press 3 to list available records\n");
    printf("4. Press 4 to search a record\n");
    printf("5. Press 5 to exit\n");
    printf("*****************************************\n\n");
}

void addrecord(PENT book[], int* count)     //←Function addrecord adds a record in phone book and increments the count
{
    char ch;
    clrscr();
    printf("                ****************\n");
    printf("                  ADD RECORDS\n");
    printf("                 ****************\n");
    printf("Enter the first name of the person:\t");
    gets(book[*count].pname.fname);
    printf("Enter the last name of the person:\t");
    gets(book[*count].pname.lname);
    printf("Enter the landline number:\t");
    gets(book[*count].pcontact.landline);
    printf("Enter the mobile number:\t");
    gets(book[*count].pcontact.mobile);
    (*count)++;
```

(*Contd...*)

PE 9-2.c

```
 printf(“Record entered successfully\n\n”);
    flushall();
    printf(“Do you want to enter more records(Y/N):\t”);
    scanf(“%c”,&ch);
    flushall();
    if(ch==‘Y’||ch==‘y’)
        addrecord(book, count);
    else
        return;
}

void listrecords(PENT book[],int count)      //←Function listrecords lists all the records available in the phone book
{
    int i=0;
    clrscr();
    printf(“                    ********************\n”);
    printf(“                      LISTING RECORDS\n”);
    printf(“                    ********************\n”);
    printf(“\n%-4s %-20s%-20s%-12s %-12s\n”,”S.No”,”First name”,”Last name”,”Landline No.”, “Mobile No.”);
    printf(“---------------------------------------------------------------------------\n”);
    while(i<count)
    {
        printf(“%4d. %-20s%-20s%-12s %-12s\n” ,i+1,book[i].pname.fname,book[i].pname.lname, book[i].pcontact.landline,
        book[i].pcontact.mobile);
            i++;
    }
    printf(“----------------------------------------------------------------------------\n”);
    printf(“\n%d record(s) available\n”,count);
    printf(“Press any key to return to main menu...\n”);
    getch();
}

void searchrecord(PENT book[], int count)      //←Function searchrecord searches a record according to various criteria
{
    int ch,i=0, found=0, no=0;
    char key[25];
    clrscr();
    printf(“                    ****************\n”);
    printf(“                      SEARCH RECORDS\n”);
    printf(“                    ****************\n”);
    printf(“1. Press 1 to search by first name\n”);
    printf(“2. Press 2 to search by last name\n”);
    printf(“3. Press 3 to search by mobile number\n”);
    printf(“4. Press any other key to return to main menu\n”);
    flushall();
    printf(“Enter your choice:\t”);
    scanf(“%d”,&ch);
    switch(ch)
    {
        case 1:
            printf(“\n\nEnter the first name of the person\n”);
```

(Contd...)

```
            flushall();
            gets(key);
            while(i<count)
            {
                if(strcmp(book[i].pname.fname,key)==0)
                {
                    if(no==0)
                        printf(“\n%-4s %-20s%-20s%-12s %-12s\n”,”S.No”,”First name”,”Last name”,”Landline No.”, “Mobile No.”);
                    found=1; no++;
                    printf(“%4d.  %-20s%-20s%-12s  %-12s\n”,  no,  book[i].pname.fname,  book[i].pname.lname,  book[i].pcontact.landline,
                    book[i].pcontact.mobile);
                }
                i++;
            }
            if(found==0)
                printf(“No record found\n”);
            else
            printf(“\n%d record(s) found\n”,no);
            printf(“Press any key to continue...\n”);
            getch();
            break;
        case 2:
            printf(“\n\nEnter the last name of the person\n”);
            flushall();
            gets(key);
            while(i<count)
            {
                if(strcmp(book[i].pname.lname,key)==0)
                {
                    if(no==0)
                        printf(“\n%-4s %-20s%-20s%-12s %-12s\n”,”S.No”,”First name”,”Last name”,”Landline No.”, “Mobile No.”);
                    found=1; no++;
                        printf(“%4d. %-20s%-20s%-12s %-12s\n”, no,book[i].pname.fname, book[i].pname.lname,
                        book[i].pcontact.landline, book[i].pcontact.mobile);
                }
                i++;
            }
            if(found==0)
                printf(“No record found\n”);
            else
                printf(“\n%d record(s) found\n”,no);
            printf(“Press any key to continue...\n”);
            getch();
            break;
        case 3:
            printf(“\n\nEnter the mobile number of the person\n”);
            flushall();
            gets(key);
            while(i<count)
            {
                if(strcmp(book[i].pcontact.mobile,key)==0)
                {
                    if(no==0)
```

(*Contd...*)

PE 9-2.c

```
                        printf(“\n%-4s %-20s%-20s%-12s %-12s\n”,”S.No”,”First name”,”Last name”,”Landline No.”, “Mobile No.”);
                    found=1; no++;
                printf(“%4d.%-20s%-20s%-12s %-12s\n”, no, book[i].pname.fname, book[i].pname.lname, book[i].pcontact.landline,
                    book[i].pcontact.mobile);
                }
            i++;
            }
            if(found==0)
                printf(“No record found\n”);
            else
            printf(“\n%d record(s) found\n”,no);
    }
    printf(“Press any key to continue....\n”);
    getch();
}

deleterecord(PENT book[], int* count)      //←Function deleterecords deletes a record with particular S.NO in the list
{
    int sno, i;
    clrscr();
    printf(“                *****************\n”);
    printf(“                    RECORD DELETION\n”);
    printf(“                *****************\n”);
    printf(“\n\nEnter the S.No of the record that you want to delete:\t”);
    scanf(“%d”,&sno);
    i=sno-1;
    if(sno<=0||sno>*count)
        printf(“Not a valid S.No\n”);
    else
    {
        while(i<*count)
        {
             book[i]=book[i+1];
             i++;
         }
        *count=*count-1;
        printf(“Record successfully deleted\n”);
    }
    printf(“Press any key to return to main menu...\n”);
    getch();
}

main()
{
    int ch, count=0;
    PENT book[50];
    clrscr();
    while(1)
    {
        printf(“      PHONE BOOK          \n”);
        printmenu();
        printf(“Enter the choice:\t”);
```

(Contd...)

```
        scanf("%d",&ch);
        flushall();
        switch(ch)
        {
            case 1:
                addrecord(book,&count);
                break;
            case 2:
                deleterecord(book, &count);
                break;
            case 3:
                listrecords(book, count);
                break;
            case 4:
                searchrecord(book, count);
                break;
            case 5:
                exit(1);
                break;
            default:
                printf("Invalid option\n");
                getch();
                exit(1);
        }
        clrscr();
    }
}
```

Output window (screen 1)

```
        PHONE BOOK
*******************************************
1. Press 1 to add records in phone book
2. Press 2 to delete a record
3. Press 3 to list available records
4. Press 4 to search a record
5. Press 5 to exit
********************************************


Enter the choice:    1
```

Output window (screen 2)

```
          ****************
              ADD RECORDS
          ****************
Enter the first name of the person:    Arvind
Enter the last name of the person:    Kakria
Enter the landline number:    2576898
Enter the mobile number:    9878776856

Do you want to enter more records(Y/N): y
```

(*Contd...*)

Output window (screen 3)

```
          ****************
             ADD RECORDS
          ****************
Enter the first name of the person:     Mohit
Enter the last name of the person:     Bansal
Enter the landline number:     2576897
Enter the mobile number:     9888566892

Do you want to enter more records(Y/N): n
```

Output window (screen 4)

```
        PHONE BOOK
*******************************************
1. Press 1 to add records in phone book
2. Press 2 to delete a record
3. Press 3 to list available records
4. Press 4 to search a record
5. Press 5 to exit
********************************************


Enter the choice:     3
```

Output window (screen 5)

```
          *****************
            LISTING RECORDS
          *****************
S.No  First Name                 Last Name                        Landline No.        Mobile No.
----------------------------------------------------------------------------------------------------
   1.  Arvind                     Kakria                           2576898             9878776856
   2.  Mohit                      Bansal                           2576897             9888566892
----------------------------------------------------------------------------------------------------


2 record(s) available
Press any key to return to main menu...
```

Output window (screen 6)

```
        PHONE BOOK
*******************************************
1. Press 1 to add records in phone book
2. Press 2 to delete a record
3. Press 3 to list available records
4. Press 4 to search a record
5. Press 5 to exit
********************************************


Enter the choice:     2
```

(Contd...)

Output window (screen 7)

```
            *****************
              RECORD DELETION
            *****************


Enter the S.NO of the record that you want to delete:    1
Record successfully deleted
Press any key to return to main menu...
```

Output window (screen 8)

```
         PHONE BOOK
*******************************************
1. Press 1 to add records in phone book
2. Press 2 to delete a record
3. Press 3 to list available records
4. Press 4 to search a record
5. Press 5 to exit
********************************************


Enter the choice:    3
```

Output window (screen 9)

```
            *****************
              LISTING RECORDS
            *****************
S.No  First Name              Last Name                 Landline No.     Mobile No.
----------------------------------------------------------------------------------------------------
   1.  Mohit                   Bansal                    2576897          9888566892
----------------------------------------------------------------------------------------------------


1 record(s) available
Press any key to return to main menu...
```

Output window (screen 10)

```
         PHONE BOOK
*******************************************
1. Press 1 to add records in phone book
2. Press 2 to delete a record
3. Press 3 to list available records
4. Press 4 to search a record
5. Press 5 to exit
********************************************


Enter the choice:    5
```

Test Yourself

1. Fill in the blanks in each of the following:
   a. Structures can be used for the storage of data of ____________ type.
   b. A structure that contains a pointer to an instance of itself is known as ____________.
   c. ____________ and ____________ are collectively known as aggregate type.
   d. Unnamed structure types are also known as ________________________.
   e. Like elements of an array are accessed by their indices, the elements of a structure are accessed by their ____________.
   f. Elements of a structure type can be accessed faster if they are ____________ aligned.
   g. The members of a structure object can be accessed via a pointer to the structure object by using ____________ operator.
   h. The precedence of the direct member access operator is ____________ than the dereference operator.
   i. The memory allocated to a union object is the amount necessary to contain its ____________ member.
   j. The ____________ can be used to create an alias for a previously defined type.

2. State whether each of the following is true or false. If false, explain why.
   a. The variables of the defined structure type can only be declared in the scope in which the defined structure type is visible.
   b. A structure declaration-list cannot contain a member of void, function or incomplete type.
   c. Structure, unions and enumerations are collectively known as aggregate type.
   d. A structure that contains an instance of itself is known as a self-referential structure.
   e. The name of a structure member can be the same as the structure tag-name.
   f. A structure definition does not reserve any space in the memory.
   g. Structure members can be initialized during the structure definition.
   h. In C language, an object of a structure type can be created by just using its tag-name.
   i. Like an array name, the name of a structure refers to its base address.
   j. The assignment operator copies all the members of a structure object to a structure variable along with the padding bytes.
   k. Structure padding can appear anywhere within a structure object.
   l. Unlike functions, a structure can be defined within another structure-type definition.
   m. The keyword typedef is used to create a new data type.
   n. Unions can be initialized in the same way as structures are initialized.

3. Programming exercise:
   a. Define a structure data type called DATE for storing dates. The type contains three integer members: day, month and year. Implement the following operations for the defined data type:
      i. Isvalid: Checks whether the entered date is valid or not, e.g. 31-2-2009 is not a valid date since February does not have 31 days.
      ii. Nextdate: Finds the next date, e.g. if the current data is 31-1-2009, then the result of Nextdate operation is 1-2-2009.
      iii. Datediff: Finds the difference between two dates.
   b. Define a structure data type TRAIN_INFO. The type contains:
      i. Train No: integer type
      ii. Train name: string
      iii. Departure time: aggregate type TIME
      iv. Arrival time: aggregate type TIME

v.	Start station:	string
vi.	End station:	string

The structure type `TIME` contains two integer members: hour and minute. Maintain a train timetable and implement the following operations:

1. List all the trains (sorted according to train number) that depart from a particular station.
2. List all the trains that depart from a particular station at a particular time.
3. List all the trains that depart from a particular station within the next one hour of a given time.
4. List all the trains between a pair of start station and end station.

This page is intentionally left blank

10

STORAGE CLASS AND PREPROCESSOR DIRECTIVES

Learning Objectives

In this chapter, you will learn about:

- Storage duration/lifetime of an object
- Storage classes
- Translators and their classification
- Phases of translation
- Trigraph replacement, line splicing and tokenization
- Macros and its types
- Token replacement and token pasting
- Predefined macros
- Source file inclusion and line control directive
- `error` directive
- `pragma` directive
- Null directive

10.1 Storage Duration/Lifetime of an Object

An identifier denotes an object. Whenever an identifier is declared (actually defined), some storage space depending upon the type of an identifier is reserved by the compiler. For example, upon encountering the declaration statement `int variable=20;`, the compiler reserves 2 bytes (or 4 bytes in Turbo C 4.5) of storage space. Upon execution, the reserved storage space is allocated. The allocated memory space is denoted as an **object** (specifically data object). This is shown in Figure 10.1.

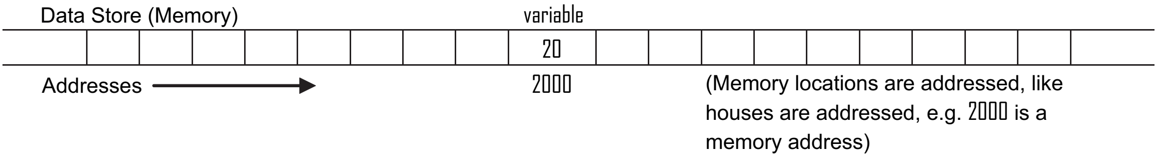

Data Store (Memory)
variable
20
Addresses
2000
(Memory locations are addressed, like houses are addressed, e.g. 2000 is a memory address)

Figure 10.1 | Data object `variable` allocated at the memory location `2000`

Object exists only at the run time, i.e. at the time of execution of the program.

The duration for which the storage space is reserved depends upon the **storage duration** of the object. The storage duration of an object determines its **lifetime**. Thus, the **lifetime of an object** is a portion of the program execution during which the memory space is guaranteed to be reserved for it. Throughout the lifetime of an object, it has a constant address and it retains its last stored value. The lifetime of an object and the scope of an identifier are related but are entirely different concepts. Ideally, **the scope of an identifier should be a subset of the lifetime of an object** it denotes; otherwise, it would be possible to refer to an identifier even after its denoted storage space goes away. If an object is referred outside of its lifetime, its behavior would be undefined.

In C language, there are three types of lifetime:

1. **Static (or global):** An object (i.e. a function or a variable) with static or global lifetime exists and has a value throughout the execution of a program. All the objects associated with the functions and global identifiers have static or global lifetime.
2. **Automatic (or local):** Objects with automatic or local lifetime are allocated new storage space each time the execution control passes to the block in which their associated identifiers are defined. When the program control moves out of the block, the objects associated with the identifiers defined within the block cease to exist and no longer have meaningful values. All the objects associated with the local identifiers by default have automatic or local lifetime.
3. **Allocated:** The lifetime of an allocated object extends from the time of their allocation until deallocation. The allocation is done with the help of memory allocation functions

like **malloc**, **calloc** or **realloc**. The deallocation can be done by calling the library function named **free**. The malloc, calloc or realloc functions allocate the memory at the run time. The allocation of memory made at the run time is known as **dynamic memory allocation**, in contrast to the **static memory allocation**, in which the memory is allocated (actually reserved) at the compile time.

10.2 Storage Classes

Every identifier not only has a data type but also has a storage class. To fully define an identifier, one needs to mention not only its data type, but also its **storage class**. If any storage class is not specified in a declaration statement, the compiler assumes the default storage class depending upon the scope in which the declaration is made. The **storage class** of an identifier determines:

1. Where the object associated with the identifier would be stored (in the memory or CPU registers).
2. What the initial value of the object associated with the identifier would be (if the identifier is not initialized in the declaration statement).
3. Whether the object associated with the identifier would have static (global) or automatic (local) lifetime.
4. What the linkage of a function or an identifier would be.

The storage class of an identifier can be specified with the help of a **storage class specifier**. The storage class specifier is prefixed in a declaration statement declaring an identifier associated with the object. The C language provides the following storage class specifiers:

1. auto
2. register
3. static
4. extern
5. typedef

The general syntax of a declaration statement is:

```
[storage_class_specifier][type_qualifier | type_modifier] datatype identifiername [=value[, ...]];
```

For example, the declaration statement static int a;, associates static storage class with the object identified by a.

The important points about the usage of storage class specifiers in a declaration statement are as follows:

1. At most one storage class specifier can be specified in a declaration statement. For example, the declaration statement auto register int a; is erroneous as two storage class specifiers, i.e. auto and register have been used in the declaration statement.
2. The storage class specifier that can be used in a declaration statement depends upon the scope in which the declaration is made. The exact meaning of each storage class specifier depends upon:
    a. Whether the declaration appears in the global scope or local scope.
    b. Whether the identifier being declared is a variable or a function.

The following sections present the use of various storage class specifiers in detail.

10.2.1 The auto Storage Class

The important points about the **auto** storage class are as follows:

1. By default, an object whose identifier has block scope or local scope (i.e. declared within a block) has auto storage class.
2. The storage class specifier auto specifies that the declared data object (i.e. variable) will be stored in the main memory.
3. It specifies that the declared object will have automatic (local) lifetime. The object will come into existence from the point of its declaration and remains into existence till the program control remains within the block in which it is declared. The code snippet in Program 10-1 illustrates this fact.

Line	Trace	Prog 10-1.c	Output window
1		//Existence of auto variables	The value of a is 10 The value of b is 20 Here b is not visible The value of a is 10 **Remarks:** • To look at the value of the variable a and b at various trace steps, add watch on a and b • The procedure to add watch in Turbo C 3.0 is: o Go to Debug Menu by pressing ‘Alt+d’ o Go to watch option by pressing ‘w’ o Press ‘Enter’ to add watch on variable a o Repeat the entire procedure to add watch on variable b o The shortcut key to add watch is ‘Ctrl+F7’ • The procedure to add watch in Turbo C 4.5 is: o Go to Debug Menu by pressing ‘Alt+d’ o Go to watch option by pressing ‘w’ o The shortcut for the first two steps (i.e. for opening watch option directly) is Ctrl+F5 o Enter the expression on which watch is to be placed i.e. variable a o Repeat the entire procedure to add watch on variable b • After adding watch, start tracing and open watch window to observe the value of a and b. • To open the watch window, go to the window menu by pressing ‘Alt+w’ and then select the watch option by pressing ‘w’
2		#include<stdio.h>	
3	⇨ 1	main()	
4		{	
5	⇨ 2	auto int a=10	
6	⇨ 3	printf(“The value of a is %d\n”,a);	
7		{	
8	⇨ 4	int b=20;	
9	⇨ 5	printf(“The value of b is %d\n”,b);	
10		}	
11	⇨ 6	printf(“Here b is not visible\n”);	
12	⇨ 7	printf(“The value of a is %d”,a);	
13	⇨ 8	}	
			Watch window
			At trace step 1: Undefined symbol ‘a’ Undefined symbol ‘b’

(*Contd...*)

Line	Trace	Prog 10-1.c	Output window
			At trace step 2: a=-29011 (i.e. Garbage value as a is yet not initialized) Undefined symbol 'b' (b is not yet defined as the program control has not yet entered the block in which it is declared)
			At trace step 3: a=10 Undefined symbol 'b'
			After trace step 3, i.e. At trace step 4: a=10; b=657 (i.e. Garbage value as b is not yet initialized)

			At trace step 5:
			a=10
			b=20

			At trace step 6: a=10 Undefined symbol 'b' (Now b does not exist as the program control came out of the block in which it is declared)
			After trace step 8: Undefined symbol 'a' Undefined symbol 'b'

Program 10-1 | A program illustrating that the auto objects have automatic lifetime

4. The variables declared with auto storage class specification are not implicitly initialized. A variable declared with auto storage class specification has to be explicitly initialized, otherwise it will have a garbage value.
5. It is not possible to specify auto storage class specifier in the declarations that are made in the global scope. The piece of code in Program 10-2 illustrates this fact.

Line	Prog 10-2.c	Output window

1
2
3
4
5
6
7
8
9
	//auto storage class specifier
	#include<stdio.h>
	auto int a;
	main()
	{
	printf("Enter the value of a");
	scanf("%d",&a);
	printf("The entered value is %d",a);
	}
		Compilation error "Storage class 'auto' is not allowed here"
		Remark:
		• Since the scope of an identifier should be a subset of the lifetime of its object, auto (i.e. local) cannot be the lifetime of an object that has a global scope

Program 10-2 | A program illustrating that auto storage class specifier cannot be used in the declarations made in the global scope

6. The variables declared with auto storage class specification have no linkage.

10.2.2 The register Storage Class

The important points about the register storage class are as follows:

1. The register storage class suggests that the access to the declared object should be as fast as possible.
2. The object of an identifier for which the register storage class has been specified is stored in central processing unit (CPU) register instead of being stored in random access memory (RAM) or the main memory, if possible. The CPU register is a scarce resource and provides faster access than memory. If it is not possible to spare a CPU register to store an identifier; the identifier will be stored in RAM and the register specification is simply treated as auto specification.
3. The storage class specifier register specifies that the declared object will have automatic (i.e. local) lifetime. Hence, the register storage class specifier cannot be used in the declarations made in the global scope.
4. The variables declared with register storage class specification are not implicitly initialized. A variable declared with register storage class specification has to be explicitly initialized, otherwise it will have a garbage value.
5. The variables declared with register storage class specification have no linkage.
6. It is not possible to compute the address of an object whose identifier is declared with register storage class specifier. If address-of operator (i.e. &) is applied to an object declared with storage class register, the compiler will issue an error message. The piece of code in Program 10-3 illustrates this fact.

Prog 10-3.c

```c
//register storage class specifier and the address of an identifier
#include<stdio.h>
main()
{
    register int a=200;
    printf("The value of a is %d\n", a);
    printf("The address of variable a is %p",&a);
}
```

Output window

Compilation error "Must take address of a memory location"

Remarks:

- It is not possible to compute the address of a variable declared with register storage class specification
- Note that some compilers ignore the register storage class specifier and store objects in the memory as an auto object. In such a case, there will be no compilation error and the address of the allocated memory space will be printed

Program 10-3 | A program illustrating that it is not possible to compute the address of an object whose identifier is declared with a register storage class specifier

8. The register storage class is commonly used for loop counters to improve the performance of a program.

10.2.3 The static Storage Class

The important points about the static storage class are as follows:

1. The storage class specifier static specifies that the declared object will have static (i.e. global) lifetime.

2. It specifies that the declared object will be stored in the main memory.
3. It can be used both with the identifiers declared in the local scope (i.e. local identifiers) as well as in the global scope (i.e. global identifiers).
4. The variables declared with static storage class specification are implicitly initialized. If a variable declared with static storage class specification is not explicitly initialized, its object will be implicitly initialized to 0 if it is of int type, 0.0 if it is of float type and '\0' if it is of char type.
5. If a static variable is present inside the local scope, the associated object is initialized only once. The object will not be reinitialized even if the program control re-enters the block in which the variable is declared. Thus, the value of static variables persists between the function calls. The piece of code in Program 10-4 illustrates this fact.

Prog 10-4.c

```
//The value of static variables persists between the function calls
#include<stdio.h>
fun(int i);
main()
{
int i=0;
for(i=0;i<5;)
    fun(++i);
}
fun(int i)
{
    static int a=10;
    printf("The value of a on entry to fun on execution no. %d is %d\n", i, a);
    printf("The value of a after increment is %d\n",++a);
}
```

Output window

```
The value of a on entry to fun on execution no. 1 is 10
The value of a after increment is 11
The value of a on entry to fun on execution no. 2 is 11
The value of a after increment is 12
The value of a on entry to fun on execution no. 3 is 12
The value of a after increment is 13
The value of a on entry to fun on execution no. 4 is 13
The value of a after increment is 14
The value of a on entry to fun on execution no. 5 is 14
The value of a after increment is 15
```

Remarks:

- The value of variable a persists between the function calls
- The variable a is initialized only once, i.e. when the function fun is called for the first time

Program 10-4 | A program illustrating that the value of static variables persists between the function calls

6. The static storage class specifier can also be used to modify the linkage of an identifier:
   a. The global identifiers by default have external linkage. If static specifier is used in the declaration of a global identifier, the identifier will have internal linkage instead of external linkage.
   b. When static storage class specifier is used with the local identifiers, the local identifiers will have internal linkage instead of no linkage.
7. The static storage class specifier cannot be used in parameter declaration either in the function declaration or in the function definition. The piece of code in Program 10-5 illustrates this fact.

Prog 10-5.c

```
//static storage class specifier
#include<stdio.h>
int add(static int a, static int b)
```

Output window

```
Compilation error "Storage class static is not allowed
here"
```

(*Contd...*)

Prog 10-5.c	Output window

{
return a+b;
}
main()
{
int c=add(2,3);
printf("The value of c is %d",c);
}
	Remark:
	• The usage of static storage class specifier is not allowed in the parameter declaration either in the function declaration or in the function definition

Program 10-5 | A program illustrating that a static storage class specifier cannot be used in the parameter declaration either in the function declaration or in the function definition

10.2.4 The extern Storage Class

The important points about the **extern** storage class are as follows:

1. Identifiers declared in the global scope, by default, have extern storage class.
2. The storage class specifier extern is used to declare a variable without defining it.
3. However, if a variable is initialized, the extern declaration becomes a definition. For example, extern int a; is a declaration but extern int a=200; is a definition. The initialization is possible only if the declaration is done in the file or global scope.
4. An extern variable cannot be initialized if a declaration statement is written within the block or local scope. The piece of code in Program 10-6 illustrates this fact.

Prog 10-6.c	Output window

//extern storage class specifier
#include<stdio.h>
main()
{
    extern int a=200;
    printf("The value of a is %d",a);
}
	Compilation error "extern variable cannot be initialized in function main"
	"Undefined symbol 'a' in function main"
	Remarks:
	• The extern storage class can only be used with the objects that have external linkage
	• Since local variables have no linkage, extern cannot be used in the declaration statement present in the local scope

Program 10-6 | A program illustrating that an extern variable cannot be initialized if the declaration is done in the block scope or local scope

5. The extern storage class specifier is used to specify that an object is defined with external linkage elsewhere in a program.
6. The extern storage class specifier cannot be used in the parameter declaration either in the function declaration or in the function definition. The piece of code in Program 10-7 illustrates this fact.

Prog 10-7.c	Output window

//extern storage class specifier
#include<stdio.h>
int add(extern int a, extern int b)
{
	Compilation error "Storage class extern is not allowed here"
	Remark:
	• The function parameters can only have auto or register storage class

(*Contd...*)

Line	Prog 10-7.c	Output window

5
6
7
8
9
10
11
	return a+b;
	}
	main()
	{
	int c=add(2,3);
	printf("The value of c is %d",c);
	}	

Program 10-7 | A program illustrating that extern storage class specifier cannot be used in the parameter declaration either in the function definition or declaration

10.2.5 The typedef Storage Class

The important points about the typedef storage class are as follows:

1. The typedef storage class specifier is used for syntactic convenience only.
2. typedef is used for creating a synonym or an alias for a known type.
3. The syntax of writing a typedef declaration is:

   ```
   typedef known-type-T synonym-name;
   ```

   where T is a generic term and can be int, float, char or any other type.

The code snippet in Program 10-8 illustrates the use of typedef storage class.

Line	Prog 10-8.c	Output window

1
2
3
4
5
6
7
8
	//typedef storage class specifier
	#include<stdio.h>
	main()
	{
	typedef char* cp;
	cp c;
	printf("The size of character pointer c is %d bytes",sizeof(c));
	}
		The size of character pointer c is 2 bytes
		Remarks:
		• After creating a synonym name cp using typedef, it is possible to refer to the type char* by writing cp
		• The declaration in line number 6, declares a variable c of type char*
		• If executed using Borland TC 4.5, the size of the character pointer would be 4 bytes

Program 10-8 | A program that illustrates the use of typedef storage class specifier

4. Note that typedef does not introduce a new type. It only creates a synonym for the known type.

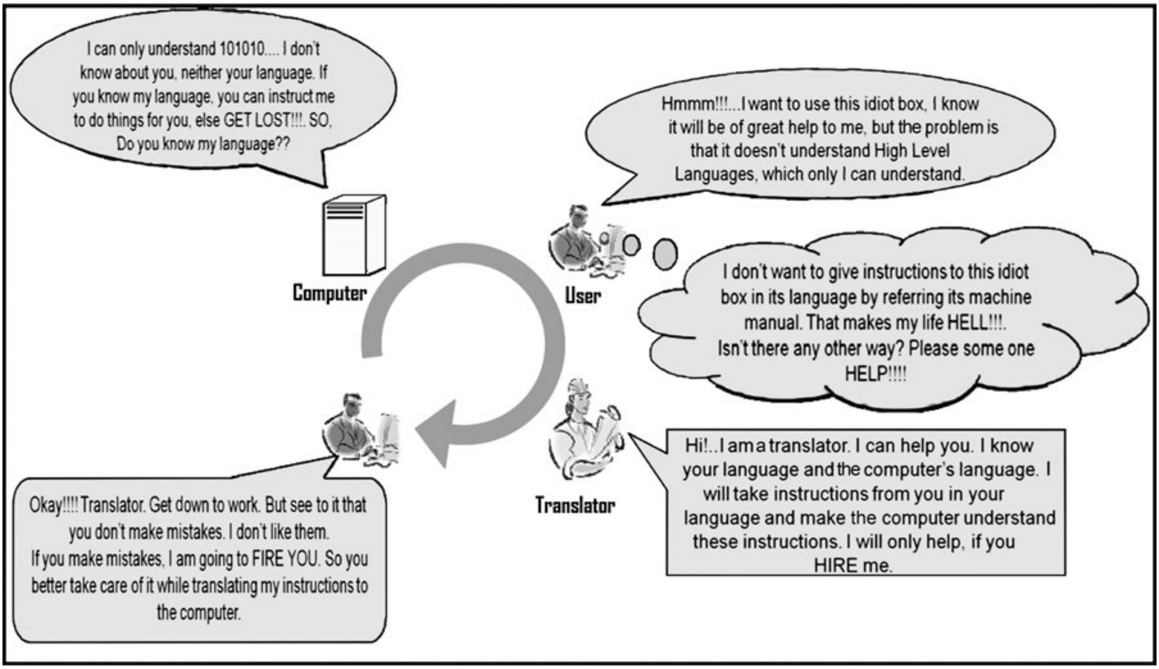
Table 10.1 summarizes the features of a variable defined with the described storage class specifications.

Table 10.1 | Summary of storage classes

S.No	Storage class	Storage	Initial value	Lifetime	Linkage
1.	auto	Memory	Garbage	Automatic	No
2.	register	CPU registers	Garbage	Automatic	No
3.	static	Memory	Zero	Static	Internal
4.	extern	Memory	Zero	Static	External
5.	typedef	Used for syntactic convenience only			

10.3 The C Preprocessor

In the previous chapters, you have developed several programs using C language, which is a high-level language. However, you will be surprised to know that the computer (i.e. the machine) cannot understand high-level languages. It can only understand machine-level languages, which are in the form of 1's and 0's. Humans do not want to write programs in machine-level languages because they are difficult to read and modify, more error prone and difficult to debug. Therefore, **translators**, which convert a high-level language program into an equivalent machine-level language program, are used to enable humans to write programs in high-level languages and at the same time make it possible to execute them on machines. The concept of translators can be understood by looking at the story board given below:


I can only understand 101010.... I don't know about you, neither your language. If you know my language, you can instruct me to do things for you, else GET LOST!!!. SO, Do you know my language??
Hmmm!!!...I want to use this idiot box, I know it will be of great help to me, but the problem is that it doesn't understand High Level Languages, which only I can understand.
Computer
User
I don't want to give instructions to this idiot box in its language by referring its machine manual. That makes my life HELL!!!. Isn't there any other way? Please some one HELP!!!!
Hi!..I am a translator. I can help you. I know your language and the computer's language. I will take instructions from you in your language and make the computer understand these instructions. I will only help, if you HIRE me.
Translator
Okay!!!! Translator. Get down to work. But see to it that you don't make mistakes. I don't like them. If you make mistakes, I am going to FIRE YOU. So you better take care of it while translating my instructions to the computer.

You should also know that **compiler** is not the only translator that works before the execution of a program. The **preprocessor** is another translator that works and processes the source code before it is given to the compiler. It operates under the control of commands known as **preprocessor directives**. In this chapter, I will tell you how the preprocessor directives are written, various preprocessor directives and the precautions one must take while using them.

10.4 Translators

A **translator** is a program that takes a program written in a language called the **source language** as an input and converts it into an equivalent program in another language called the **target language**. Translators are classified according to the classes of their source and target languages. The classification of translators according to the classes of their source and target languages is shown in Table 10.2.

Table 10.2 | Classification of translators according to their source and target languages

S.No	Source language Input →	Name of Translator Output →	Target language
1.	High-level language	Preprocessor	High-level language
2.	High-level language	Compiler	Low-level language (i.e. assembly-level language or machine-level language)
3.	Assembly-level language	Assembler	Machine-level language
4.	High-level language	Interpreter	Machine-level language

The **preprocessor** is a translator that converts a program written in one high-level language into an **equivalent** program written in another high-level language. For example, a preprocessor converts the code written in C into an equivalent program in simplified C language. The **compiler** converts a program written in a high-level language into an equivalent program either in an assembly-level language or a machine-level language. If the output of the compiler is an assembly language program, an **assembler** is required to further convert it into the machine code. An **interpreter** is a translator that converts the statements written in a high-level language program into equivalent statements in a machine-level language one by one on the fly.

10.5 Phases of Translation

The conversion of a source program file into an executable file is done in eight conceptual steps known as **phases of translation**. The eight phases of translation are:

1. Trigraph sequences are replaced by their single character equivalents. This phase is carried out by the preprocessor and is called **trigraph replacement**.
2. Each instance of a backslash character (i.e. \) immediately followed by a new-line character is deleted by the preprocessor. This process is known as **line splicing**.
3. The source file is decomposed into preprocessing tokens and a sequence of white-space characters. Each comment is replaced by a single-space character and new-line characters are retained. Whether a sequence of white-space characters other than a new-line character is to be replaced by a single-space character or not is implementation defined. This phase is carried out by the preprocessor and is called **tokenization**.
4. The preprocessor directives are executed and macros are expanded. This is known as **directive handling** and **macro expansion**. After their execution, all the preprocessor directives are then deleted.
5. Escape sequences in character constants and string literals are converted to their character equivalents.
6. Adjacent string literals are concatenated.
7. Each preprocessing token is converted into a token. White-space characters separating tokens are no longer significant and are removed. The resulting tokens are syntactically and semantically analyzed and translated into an object code by the compiler.

8. All external object and function references are resolved. All the required libraries are linked together to satisfy an external reference not defined in the current program. This phase is carried out by the linker, and the output of this phase is an executable file ready for the execution.

The first four phases of translation need an explicit description and are described in the following sections in detail. The working of the rest of the phases is clear from the above-mentioned text and will be clearer in the further course of discussion.

10.5.1 Trigraph Replacement

A **character set** defines the valid characters that can be used in a source program or interpreted when a program is running. The set of characters that can be used to write a source program is called a **source character set**, and the set of characters available when the program is executing is called an **execution character set**. It is possible that the source character set is different from the execution character set.

There are a number of character sets that exist. For example, ISO 646, ASCII, EBCDIC, ISO8859, ISO8859-1, ISO8859-2,…, ISO8859-16, etc. A character that exists in one character set might not exist in some other character set.

To write C programs using character sets that do not contain all of C's punctuation characters, ANSI allows the use of nine trigraph sequences in the source file. A **trigraph sequence** is a sequence of three characters, the first two of which are question marks and the third character should belong to the given set of characters {=, (, /,), ', <, !, >, -}. Trigraph sequences are replaced by their corresponding character equivalents during the first phase of translation (i.e. **trigraph replacement**). Table 10.3 lists the valid trigraph sequences and their character equivalents.

Table 10.3 | Trigraph sequences and their character equivalents

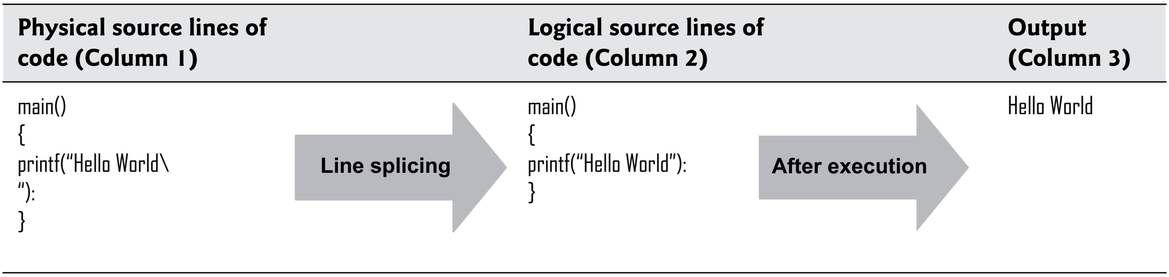
S.No	Trigraph sequence	Character equivalent
1.	??=	#
2.	??(	[
3.	?? /	\
4.	??)	]
5.	??'	^
6.	??<	{
7.	??!	|
8.	??>	}
9.	??-	~

No other trigraph sequence is recognized. A question mark (?) that does not begin the above-mentioned trigraph sequences remains unchanged during the translation.

Some compilers support an option to turn the recognition of trigraphs off or disable the trigraphs by default, and they require an option to turn them on. Some issue warning messages when they encounter trigraph sequences in the source files. Borland supplies a separate trigraph processor (TRIGRAPH.EXE) with Turbo C 3.0 and 4.5. This file is present in the BIN folder of the Turbo C installation and is only used when the trigraph processing is desired. The objective behind supplying a separate trigraph processor is to maximize the speed of compilation.

10.5.2 Line Splicing

During the preprocessing stage, each instance of a backslash character (i.e. \) immediately followed by a new-line character is deleted. This process is known as **splicing**. Physical source lines present in the source program are spliced to form logical source lines. Only the last backslash on any physical source line is eligible for being a part of such a splice. Consider Figure 10.2.



Physical source lines of code (Column 1)

```
main()
{
printf(“Hello World\
”):
}
```

Logical source lines of code (Column 2)

```
main()
{
printf(“Hello World”):
}
```

Output (Column 3)

```
Hello World
```

Figure 10.2 | Line splicing

Column 1 contains the physical source lines of the code. After the preprocessing stage, the physical source lines are spliced to form logical source lines of the code, as mentioned in column 2 in Figure 10.2. Logical source lines are processed by the compiler during phase 7 of translation. The output produced on the execution of the logical source lines of the code listed in column 2 is shown in column 3 in Figure 10.2.

10.5.3 Tokenization

A **preprocessing token** is the smallest indivisible element of C language in the translation phases from 3 to 6. The categories of the preprocessing tokens are: header names, identifiers, preprocessing numbers, character constants, string literals, punctuators and a single non-white-space character. A **token** is the smallest indivisible element of C language in the translation phases 7 and 8. The categories of tokens are: keywords, identifiers, constants, string literals and punctuators (i.e. operators, separators or terminator). For example, the operator `+=` is one token.

C’s **tokenizer** is greedy in nature. It always tries to create the biggest possible token. If an input stream of characters has been parsed into tokens up to a given character, the next token is the longest sequence of the characters that could constitute a token. For example, the program fragment `x+++++y` is parsed as `x ++ ++ + y`, which violates a constraint on the increment operator and leads to a compilation error. If the tokenizer would have been intelligent instead of being greedy and parses the mentioned fragment as `x ++ + ++ y`, it would have been a valid expression.

10.5.4 Preprocessor Directive Handling

The preprocessor is controlled by directives known as **preprocessor directives**, which are not a part of C language. A **preprocessor directive** consists of various preprocessing tokens and begins with a # (pound) symbol. The important points for writing a preprocessor directive are as follows:

1. The pound symbol (#) should either be the first character in a source file or the first non-white-space character in a line.
2. A new-line character ends the preprocessor directive.
3. The white-space characters that can appear between the preprocessing tokens within a preprocessing directive are a single-space character or a horizontal tab-space character (i.e. white-space characters like new-line, vertical tab and form feed are not allowed).
4. The preprocessor directives can appear anywhere in a program but are generally placed at the beginning of a program before the function main or before the beginning of a particular function.

Table 10.4 illustrates the application of the rules mentioned above for writing an include directive.

Table 10.4 | Rules for writing the preprocessor directives

S.No	Preprocessor directive	Valid or invalid?
1.	#include<stdio.h>	**Valid**, pound symbol is the first character in the source file
2.	#include <stdio.h>	**Valid,** white-space characters (only space and horizontal tab) can appear within a preprocessor directive
3.	#include<conio.h>	**Valid**, pound symbol is the first non-white-space character in a line
4.	a#include<string.h>	**Invalid**, as pound symbol is not the first non-white-space character in a line
5.	#include<math.h> #include<stdarg.h>	**Invalid**, as the first preprocessor directive is not terminated with a new-line character and the second preprocessor directive's pound symbol is not the first non-white-space character
6.	#include <dos.h>	**Invalid**, as a white-space character between preprocessing tokens within a preprocessing directive cannot be a new-line character

The various preprocessor directives available in C language are as follows:

1. Macro replacement directive (#define, #undef)
2. Source file inclusion directive (#include)
3. Line directive (#line)
4. Error directive (#error)
5. Pragma directive (#pragma)
6. Conditional compilation directives (#if, #else, #elif, #endif, #ifdef, #ifndef)
7. Null directive (# new-line)

10.5.4.1 Macro Replacement Directives

A **macro** is a facility provided by the C preprocessor, by which a token can be replaced by the user-defined sequence of characters. Macros are defined with the help of the define directive. The identifier name immediately following the define directive is called the **macro name**. Macro names are generally written in upper case.

10.5.4.1.1 Types of Macro

There are two types of macros:

1. Macro without arguments, also called **object-like macros**.
2. Macro with arguments, also called **function-like macros**.

10.5.4.1.1.1 Object-like Macros

An **object-like macro** is also known as a **symbolic constant**. It is defined as:

```
#define macro-name replacement-list
```

The important points about object-like macros are as follows:

1. The define directive causes each subsequent instance of the macro name to be replaced by the replacement list of preprocessing tokens present in the definition of the macro.
2. The replacement list can even be empty.
3. The object-like macro name must be a preprocessing identifier. During the translation phases 3 to 6, keywords are not recognized separately and are treated as identifiers. Hence, they can also be used as a macro name, e.g. the following object-like macro definition is perfectly valid:

   ```
   #define int char
   ```

4. There shall be a white-space character (blank-space character or horizontal tab space character) between the macro name and the replacement list in the definition of an object-like macro.

The piece of code in Program 10-9 illustrates the use of an object-like macro.

Line	Prog 10-9.c	After the preprocessing stage	Output window
1	//Object-like macro	//The content of the header file stdio.h	Area of circle is 78.550000 **Remarks:** • PI is an object-like macro • During the preprocessing stage, each subsequent instance of PI is replaced by its replacement list (i.e. 3.142)
2	#include<stdio.h>	//replaces the include directive and is	
3	#define PI 3.142	//placed here	
4	main()	main()	
5	{	{	
6	int rad=5;	int rad=5;	
7	printf(“Area of circle is %f”,PI*rad*rad);	printf(“Area of circle is %f”,3.142*rad*rad);	
8	}	}	

Program 10-9 | A program that illustrates the definition and the use of an object-like macro

10.5.4.1.1.2 Function-like Macros

A macro with arguments is called a **function-like macro**. Its usage is syntactically similar to a function call and it can be defined as:

```
#define macro-name(parameter-list) replacement-list
```

The piece of code in Program 10-10 illustrates the use of a function-like macro.

Line	Prog 10-10.c	After the preprocessing stage	Output window
1	//Function-like macro	//The content of the header file stdio.h	Area of square is 25 **Remarks:** • Each time a function-like macro name is encountered, the macro name is replaced by the replacement list • The parameters present in the replacement list of macro definition are replaced by the actual arguments present in the macro invocation
2	#include<stdio.h>	//replaces the include directive and is	
3	#define SQR(x) (x*x)	//placed here	
4	main()	main()	
5	{	{	
6	int side=5;	int side=5;	
7	printf("Area of square is %d",SQR(side));	printf("Area of square is %d",(side*side));	
8	}	}	

Program 10-10 | A program that illustrates the definition and the use of a function-like macro

> The white-space characters preceding or following the replacement list are not considered as a part of the replacement list for either form of macro (i.e. object-like or function-like).

During the preprocessing stage, the macro names are expanded and are replaced by their replacement lists. This process is known as **macro expansion**. Macro expansion is purely textual. If proper care is not taken while defining macros, they might lead to unexpected results.

10.5.4.1.2 Common Macro Pitfalls

Macros can create problems if they are not defined and used carefully. The common macro pitfalls are described in subsequent sections.

10.5.4.1.2.1 Magical White Space

1. There should be a white-space character (blank-space character or horizontal tab-space character) between the macro name and the replacement list in the definition of an object-like macro. The piece of code in Program 10-11 illustrates the effect of the violation of the above-mentioned rule.

Line	Prog 10-11.c	Output window
1	//Magical white-space character	Compilation error "Expression syntax in function main" **Remarks:** • The code is not working due to the erroneous definition of the object-like macro PI
2	#include<stdio.h>	
3	#define PI=3.1428	
4	main()	

(*Contd...*)

5
6
7
	{
	printf("The value of constant PI is %f",PI);
	}
		• There should be a white-space character between the macro name and the replacement list in the definition of an object-like macro instead of the character '='
		What to do?
		• Rectify the macro definition as #define PI 3.1428

Program 10-11 | A program that illustrates the significance of a white-space character between the macro name and its replacement list

2. There should be no white-space character between the macro name and the left parenthesis of parameter list in the definition of a function-like macro. The piece of code in Program 10-12 illustrates the effect of the violation of the above-mentioned rule.

Line	Prog 10-12.c	Output window

1
2
3
4
5
6
7
	//Magical white-space character
	#include<stdio.h>
	#define CUBE (x) x*x*x
	main()
	{
	printf("Cube of 5 is %d",CUBE(5));
	}
		Compilation error "Undefined symbol x in function main"
		Remarks:
		• Due to a white-space character between the macro name CUBE and the left parenthesis of the parameter list in the macro definition, CUBE will be treated as an object-like macro and not as a function-like macro
		• After the macro expansion, the expression CUBE(5) will become (x) x*x*x(5)
		• The preprocessed code on compilation gives the specified error
		What to do?
		• Remove the white-space character between the macro name and the left parenthesis in the macro definition. Re-execute the code and check the result

Program 10-12 | A program illustrating that there should be no white-space character between the macro name and the left parenthesis of the parameter list

10.5.4.1.2.2 Operator Precedence Problems

1. In the definition of a macro, the replacement list must always be parenthesized to protect any lower precedence operator in it from a higher precedence operator in the surrounding expression. The piece of code in Program 10-13 illustrates the effect of the violation of the above-mentioned rule.

Line	Prog 10-13.c	Output window

1
2
3
4
5
6
7
8
	//Operator precedence problem-I
	#include<stdio.h>
	#define DOUBLE(x) x+x
	main()
	{
	int result, x;
	printf("Enter the value of x\t");
	scanf("%d",&x);
		Enter the value of x 3
		Value of result is 18
		Expected result:
		Value of result is 30
		Remarks:
		• Macro expansion is purely textual
		• Macros are expanded during the preprocessing stage before the compilation stage

(*Contd...*)

Line	Prog 10-13.c	Output window

9
10
11
	result=5*DOUBLE(x);
	printf("Value of result is %d",result);
	}
		• Thus, after the preprocessing stage, the expression result=5*DOUBLE(x) becomes result=5*x+x
		• Since the multiplication operator has a higher precedence than the addition operator, it will operate first
		• Thus, the result of the expression comes out to be 18 instead of 30
		What to do?
		• Parenthesize the replacement list to protect the lower precedence operator (i.e. addition operator) in it from the surrounding higher precedence operator (i.e. multiplication operator)
		• Re-define the macro as: #define DOUBLE(x) (x+x) and re-execute the code

Program 10-13 | A program that illustrates an operator precedence pitfall in the macro definition

2. In the definition of a function-like macro, all the occurrences of parameters in the replacement list must be parenthesized to protect any low precedence operator in the actual arguments from the rest of the macro expansion. The piece of code in Program 10-14 illustrates the effect of the violation of the above-mentioned rule.

Line	Prog 10-14.c	Output window

1
2
3
4
5
6
7
8
9
	//Operator precedence problem-II
	#include<stdio.h>
	#define SQR(x) (x*x)
	main()
	{
	int val=2, result;
	result=SQR(val+1);
	printf("Result is %d",result);
	}
		Result is 5
		Expected result:
		Result is 9
		Remarks:
		• After the preprocessing stage, the expression result=SQR(val+1) becomes result=val+1*val+1 (i.e. result=2+1*2+1)
		• Since the multiplication operator has a higher precedence than the addition operator, it will get evaluated first. Thus, the expression evaluates to 5
		• Since the lower precedence operators in the actual arguments are not protected from the rest of the macro expansion, the program gives an unexpected result
		What to do?
		• Parenthesize all the parameters in the replacement list
		• Redefine the macro as: #define SQR(x) ((x)*(x)) and re-execute the code

Program 10-14 | A program that illustrates an operator precedence pitfall in the macro definition

10.5.4.1.2.3 Arguments with a Side-effect

1. While calling a function-like macro, the argument should not be an expression with a side-effect.✍

✍ A **side-effect** is a modification of a data object or a file. Modifying an object, modifying a file or calling a function that does any of these operations are all side-effects. The evaluation of an expression may also produce side-effects. For example, the evaluation of the expression result=value++ has side-effects as it modifies the data objects, namely result and value. The assignment operator,

(Contd...)

increment operator and decrement operator have side-effects. The side-effects of evaluations should be complete at certain specified points in the execution sequence known as **sequence points**.

A **sequence point** is a point in the program execution sequence at which all the side-effects of the previous evaluations are complete and no side-effects of subsequent evaluations have taken place. The semicolon marks a sequence point, i.e. all the changes made by assignment operators, increment operators and decrement operators in a statement must take place before the program control proceeds to the next statement.

The piece of code in Program 10-15 illustrates the call to a function-like macro whose argument is an expression with a side-effect.

Line	Prog 10-15.c	Output window

1
2
3
4
5
6
7
8
9
	//Arguments with side-effect
	#include<stdio.h>
	#define SQR(x) (x*x)
	main()
	{
	int val=2, result;
	result=SQR(++val);
	printf("Result is %d",result);
	}
		Result is 16
		Expected result:
		Result is 9
		Remarks:
		• Function-like macro call text replaces all the occurrences of the parameters in the replacement list with the actual arguments. **The actual arguments are not evaluated before being replaced**
		• Thus, after the preprocessing stage, the expression result=SQR(++val) becomes result=++val*++val and evaluates to 16
		• If SQR would have been defined as a function, the result would have been 9 because **in a function call the actual arguments are evaluated before being passed to the function**
		What to do?
		• Eliminate the side effect from the argument of SQR and write the statement in line number 6 as:
		++val;
		result=SQR(val);

Program 10-15 | A program to illustrate that an argument to a function-like macro should not be an expression with a side-effect

10.5.4.1.2.4 Undesirable Semicolon

1. Avoid the use of a semicolon in and at the end of a macro definition. The code snippet in Program 10-16 illustrates the effect of the presence of a semicolon in a macro definition.

Line	Prog 10-16.c	Output window

1
2
3
4
5
6
	//Effect of the use of a semicolon in the macro definition
	#include<stdio.h>
	#define SWAP(a,b) a=a+b; b=a-b; a=a-b
	main()
	{
	int a=20, b=10;
		Compilation error "Misplaced else in function main"
		Remarks:
		• During the preprocessing stage, the macro SWAP is expanded and is replaced by multiple statements (i.e. a=a+b; b=a-b; a=a-b;)

(*Contd...*)

Line	Prog 10-16.c	Output window
7	printf(“Swap the values of a and b only if a is greater\n”);	• Only the first statement (i.e. a=a+b;) forms the if body. The other two statements will be considered as the statements next to the if statement • The else clause remains unmatched and leads to “Misplaced else error” • It is recommended to use commas instead of using semicolons in the macro definition **What to do?** • Redefine the macro as: #define SWAP(a,b) a=a+b, b=a-b, a=a-b
8	if(a>b)	
9	SWAP(a,b);	
10	else	
11	printf(“Values are not swapped\n”);	
12	printf(“Resultant values of a and b are %d %d”,a,b);	
13	}	

Program 10-16 | A program that illustrates the effect of the use of a semicolon in a macro definition

The code snippet in Program 10-17 illustrates the effect of the presence of a semicolon at the end of a macro definition.

Line	Prog 10-17.c	Output window
1	//Effect of the use of semicolon at the end of macro definition	Compilation error **Remark:** • The semicolon at the end of the macro definition after the macro expansion forms an ill-formed expression and leads to a compilation error **What to do?** • Remove the semicolon present at the end of the macro definition • Re-execute the code and check the result
2	#include<stdio.h>	
3	#define CUBE(x) ((x)*(x)*(x));	
4	main()	
5	{	
6	int a=2, b=8;	
7	if(CUBE(a)==b)	
8	printf(“Cube of a is equal to b\n”);	
9	else	
10	printf(“Cube of a is not equal to b\n”);	
11	}	

Program 10-17 | A program that illustrates the effect of the use of a semicolon at the end of a macro definition

10.5.4.1.3 Stringification/Token Replacement

In a function-like macro definition, if the replacement list consists of a parameter immediately preceded by a ‘#’ preprocessing token, then during the preprocessing stage, the preprocessor replaces both the ‘#’ preprocessing token and the parameter with a single character string literal (which contains the spelling of the argument corresponding to the parameter). Since # and parameter are replaced by a single character string literal, it is known as **token replacement**. In addition, as # preprocessing token converts the argument corresponding to a parameter into a string literal, # is known as **stringizing operator** and the operation is known as **stringification**. The code snippets in Programs 10-18 and 10-19 illustrate the use of a stringizing operator.

Line	Prog 10-18.c	After the preprocessing stage	Output window
1	//Token replacement or stringification	//The content of the header file stdio.h	Token replacement
2	#include<stdio.h>	//replaces the include directive and is	
3	#define STR(x) #x	//placed here	
4	main()	main()	
5	{	{	
6	printf(STR(Token replacement));	printf(“Token replacement”);	
7	}	}	

Program 10-18 | A program that illustrates the use of a stringizing operator

Line	Prog 10-19.c	After the preprocessing stage	Output window
1	//Token replacement or stringification	//The content of the header file stdio.h	Token replacement
2	#include<stdio.h>	//replaces the include directive and is	
3	#define STR(x) #x	//placed here	
4	main()	main()	
5	{	{	
6	char str[20]=STR(Token replacement);	char str[20]="Token replacement";	
7	puts(str);	puts(str);	
8	}	}	

Program 10-19 | A program that illustrates the use of a stringizing operator

The important points about token replacement are as follows:

1. White-space characters between the argument's preprocessing tokens become a single-space character in the replaced character string literal constant. The piece of code in Program 10-20 illustrates this fact.

Line	Prog 10-20.c	After the preprocessing stage	Output window
1	//Token replacement or stringification	//The content of the header file stdio.h	Token replacement
2	#include<stdio.h>	//replaces the include directive and is	
3	#define STR(x) #x	//placed here	
4	main()	main()	
5	{	{	
6	char str[20]=STR(Token replacement);	char str[20]="Token replacement";	
7	puts(str);	puts(str);	
8	}	}	

Program 10-20 | A program illustrating that during stringification, white-space characters between the argument's preprocessing token are replaced by a single-space character

2. White-space characters before the first preprocessing token and after the last preprocessing token composing the macro's argument are deleted. The piece of code in Program 10-21 illustrates this fact.

Line	Prog 10-21.c	After the preprocessing stage	Output window
1	//Token replacement or stringification	//The content of the header file stdio.h	Token replacement
2	#include<stdio.h>	//replaces the include directive and is	
3	#define STR(x) #x	//placed here	
4	main()	main()	
5	{	{	
6	char str[20]=STR(Token replacement);	char str[20]="Token replacement";	
7	puts(str);	puts(str);	
8	}	}	

Program 10-21 | A program illustrating that during stringification, white-space characters at the start and at the end of an argument's preprocessing token are deleted

3. The original spelling of each preprocessing token in the argument is retained in the character string literal constant, except a '\' character is inserted before each '"' and '\' character. The piece of code in Program 10-22 illustrates this fact.

Line	Prog 10-22.c	After the preprocessing stage	Output window
1	//Token replacement or stringification	//The content of the header file stdio.h	White "space" character
2	#include<stdio.h>	//replaces the include directive and is	
3	#define STR(x) #x	//placed here	
4	main()	main()	
5	{	{	
6	char str[30]=STR(White "space" character);	char str[30]="White \"space\" character";	
7	puts(str);	puts(str);	
8	}	}	

Program 10-22 | A program that illustrates the insertion of '\' character during stringification

10.5.4.1.4 Concatenation/Token Pasting

In an object-like macro definition, if in the replacement list, a ## preprocessing token appears between two tokens, both the tokens are pasted to form one token. In a function-like macro definition, if in the replacement list, a ## preprocessing token appears between two parameters, the parameters will be replaced by the corresponding arguments, and the arguments will be glued and pasted to form one token. Since, two tokens are pasted (or concatenated) to create one token, it is known as **token pasting** or **token concatenation** or just **concatenation**, and the operator ## is known as the **concatenation operator**. The code snippets in Program 10-23 illustrate token pasting in an object-like macro.

Line	Prog 10-23.c	After the preprocessing stage	Output window
1	//Token pasting in an object-like macro	//The content of the header file stdio.h	Value of xy is 10
2	#include<stdio.h>	//replaces the include directive and is	
3	#define var x##y	//placed here	
4	main()	main()	
5	{	{	
6	int var=10;	int xy=10;	
7	printf("Value of xy is %d",xy);	printf("Value of xy is %d",xy);	
8	}	}	

Program 10-23 | A program that illustrates token pasting in an object-like macro

The code snippets in Program 10-24 illustrate token pasting in a function-like macro.

Line	Prog 10-24.c	After the preprocessing stage	Output window
1	//Token pasting in a function-like macro	//The content of the header file stdio.h	Value of var1 is 10
2	#include<stdio.h>	//replaces the include directive and is	
3	#define PASTE(x,y) x##y	//placed here	
4	main()	main()	
5	{	{	
6	int PASTE(var,1)=10;	int var1=10;	
7	printf("Value of var1 is %d",var1);	printf("Value of var1 is %d",var1);	
8	}	}	

Program 10-24 | A program that illustrates token pasting in a function-like macro

The following points must be remembered while using token pasting:

1. A ## preprocessing token shall not occur at the beginning or at the end of the replacement list for either form of the macro definition (i.e. object-like macro or function-like macro).
2. ## is one token. There should be no white-space character between two # characters.

10.5.4.1.5 Predefined Macros

The ANSI C standard defines several macros for the use in C language. The macros that are already defined in C language are known as **predefined macros**. These macros can be used without defining them. They cannot be redefined and hence, these macro names cannot appear immediately after define and undef directive.

The predefined macros recognized by ANSI-compliant compilers are as follows:

1. **__FILE__**: The __FILE__ macro expands to the name of the current file in the form of a string constant. The piece of code in Program 10-25 illustrates the use of __FILE__ macro.

Prog 10-25.c

```
//__FILE__ macro
#include<stdio.h>
main()
{
printf("The name of current file is %s", __FILE__);
}
```

Output window

```
The name of current file is 10-25.c
```

Program 10-25 | A program that illustrates the application of the predefined __FILE__macro

2. **__LINE__**: The __LINE__ macro expands to the current line number in the source file. The expanded line number is a decimal integer constant. The line number can be altered with the help of the line directive.† The piece of code in Program 10-26 illustrates the use of __LINE__ macro.

Prog 10-26.c

```
//__LINE__ macro
#include<stdio.h>
main()
{
printf("Current line number is %d", __LINE__);
}
```

Output window

```
Current line number is 5
```

Remark:

- Place two blank lines before the printf statement and re-execute the code to notice the change in the output

Program 10-26 | A program that illustrates the application of the predefined __LINE__ macro

3. **__DATE__**: The __DATE__ macro expands to the compilation date of the source file in the form of a string constant. The expanded string constant is 11 characters long and is of the form "Mmm dd yyyy". The important points about __DATE__ macro are as follows:

   a. The name of the month will be three characters long with the first character being in uppercase.
   b. The name of the month is the same as generated by the asctime library function declared in the header file time.h.
   c. If the value of day of the month is less than 10, it is padded with space on the left (i.e. the first character of dd is a space character). Some of the compilers, e.g. Turbo C 3.0 output zero padded value of the day, if it is less than 10.

† Refer Section 10.5.4.3 for a description on line directive.

The piece of code in Program 10-27 illustrates the use of __DATE__ macro.

Prog 10-27.c

```c
//__DATE__ macro
#include<stdio.h>
main()
{
printf(“Date of compilation is %s”, __DATE__);
}
```

Output window (using Turbo C 3.0)

```
Date of compilation is Apr 02 2009
```

Output window (using Turbo C 4.5)

```
Date of compilation is Apr  2 2009
```

Program 10-27 | A program that illustrates the application of the predefined __DATE__ macro

4. **__TIME__**: This macro expands to a string constant that describes the time at which the C preprocessor is being invoked. The expanded string constant is eight characters long and is of the form "hh:mm:ss". The piece of code in Program 10-28 illustrates the use of __TIME__ macro.

Prog 10-28.c

```c
//__TIME__ macro
#include<stdio.h>
main()
{
printf(“Time of preprocessing is %s”, __TIME__);
}
```

Output window (using Turbo C 4.5)

```
Time of preprocessing is 18:56:55
```

Program 10-28 | A program that illustrates the application of the predefined __TIME__ macro

5. **__STDC__**: This macro expands to 1, if the compiler conforms to ANSI C and ISO C standards. Some compilers may not support this macro. For example, this macro is not supported by Turbo C 3.0. The piece of code in Program 10-29 illustrates the use of __STDC__ macro.

Prog 10-29.c

```c
//__STDC__ macro
#include<stdio.h>
main()
{
if(__STDC__==1)
printf(“This compiler conforms to ANSI and ISO C standards”);
else
printf(“This compiler does not comply with ANSI and ISO C standards”);
}
```

Output window (using Turbo C 4.5)

```
This compiler conforms to ANSI and ISO C standards
```

Program 10-29 | A program that illustrates the use of the predefined __STDC__ macro

The important points about the predefined macros are as follows:

1. The ANSI predefined macros start and end with two underscores. There should not be a white-space character between the underscores.
2. The predefined macro name cannot appear immediately following a define directive. Also, a predefined macro cannot be undefined using an undef directive.‡ The piece of code in Program 10-30 illustrates this fact.

‡ Refer Section 10.5.4.1.6 for a description on the undef directive.

Line	Prog 10-30.c	Output window (Turbo C 3.0)
1	//Predefined macros	Compilation errors
2	#include<stdio.h>	"Define directive needs an identifier"
3	#define __TIME__ 10	"Bad undef directive syntax"
4	#undef __DATE__	
5	main()	
6	{	
7	printf("define and undefine directives cannot be used with predefined macros");	
8	}	

Program 10-30 | A program to illustrate that it is not allowed to redefine and undefine a predefined macro

Some implementations provide additional predefined macros. Whether a predefined macro is supported by a specific implementation or not can be checked by referring to its documentation. The common implementation defined macros are:

1. **__cplusplus**: This macro is defined when C++ compiler is in use. It can be used to test whether C compiler or C++ compiler is used.
2. **NULL**: The NULL macro is defined in the header files stdio.h and stddef.h. It represents a null pointer value. The NULL pointer is defined as (void*)0. The null pointer created with the help of NULL does not point to any object or function and is not the same as the uninitialized pointer, which might point anywhere.
3. **EOF**: The EOF macro is defined in the header file stdio.h. This macro represents an integer value that is returned when end-of-file is encountered.

10.5.4.1.6 undef Directive

The undef preprocessor directive causes the specified identifier to be no longer defined as a macro name. The general form of the undef preprocessor directive is:

```
#undef identifier
```

The piece of code in Program 10-31 illustrates the use of the undef directive.

Line	Prog 10-31.c	Output window
1	//undef directive	Compilation error "Undefined symbol 'VER' in function main"
2	#include<stdio.h>	
3	#define VER 2.2	
4	#undef VER	
5	main()	
6	{	
7	printf("Current version of software is %f",VER);	
8	}	

Program 10-31 | A program that illustrates the use of the undef directive

The important points about the undef directive are as follows:

1. If the identifier specified with the undef directive is not currently defined as a macro name, it is ignored.
2. The identifier specified with the undef directive cannot be the name of a predefined macro.
3. A macro can be redefined anywhere in the program. The most recent definition of the macro is considered while expanding the macro. If the redefinition of the macro is not identical (i.e. the redefined macro definition is not exactly the same as the first definition of the macro), the compiler will issue a warning message 'Redefinition of 'macroname' is not identical'. It is not compulsory to undefine a macro before redefining it. The code snippet in Program 10-32 illustrates the above-mentioned facts.

Prog 10-32.c	Output window

//Redefining a macro
#include<stdio.h>
#define DOUBLE 2
#define DOUBLE(x) (2*x)
main()
{
printf("Double of 2 is %d", DOUBLE(2));
}
	Double of 2 is 4
	Warning:
	Redefinition of 'DOUBLE' is not identical
	Remarks:
	• The macro DOUBLE is redefined without undefining it
	• It is not mandatory to undefine a macro before redefining it
	• After the redefinition of DOUBLE as a function-like macro, it is not possible to use it as an object-like macro, i.e. as a symbolic constant
	• Usage of DOUBLE as a symbolic constant instead of a function-like macro leads to a compilation error
	• The macro definitions will not be considered identical if:
	o one of the macro is an object-like macro and the other is a function-like macro
	o both are object-like macros but they have different replacement lists
	o both are function-like macros but they have different parameter lists or replacement lists

Program 10-32 | A program that illustrates the redefinition of a macro

10.5.4.1.7 Scope of Macro Definitions

The identifier defined as a macro can be used from the point of its definition till a corresponding undef directive is encountered or (if it is not encountered) till the end of the translation unit (i.e. file). Unlike the scope of other identifiers (i.e. variables, labels, etc.), the scope of a macro name is independent of the block structure. The piece of code in Program 10-33 illustrates this fact.

Line | **Prog 10-33.c**

```
 1  //Scope of macro definitions
 2  #include<stdio.h>
 3  main()
 4  {
 5  printf("Macro MAC and variable var are not yet defined\n");
 6  printf("They cannot be used here\n");
 7  {
 8  #define MAC 10
 9  int var=10;
10  printf("Value of MAC=%d, var=%d\n",MAC, var);
11  }
12   //Here variable var is inaccessible
13  printf("Macro MAC can be used here but variable var cannot\n");
14  printf("Value of MAC outside the block is %d\n",MAC);
15  #undef MAC
16  printf("Macro MAC cannot be used now onwards");
17  }
```

Output window

```
Macro MAC and variable var are not yet defined
They cannot be used here
Value of MAC=10, var=10
Macro MAC can be used here but variable var cannot
Value of MAC outside the block is 10
Macro MAC cannot be used now onwards
```

Remarks:

- Macro defined inside the block (line number 8) is used outside the block (line number 14). It shows that the scope of macro definition is independent of the block structure
- Usage of macro name after it has been undefined (using #undef) leads to a compilation error

Program 10-33 | A program that illustrates the concept of scope of macro definitions

10.5.4.2 Source File Inclusion Directive

The source file inclusion directive include tells the preprocessor to replace the directive with the content of the file specified in the directive. The include directive is generally used to include the header files, which contain the prototypes of the library functions and the definitions of the predefined constants. The source file inclusion directive include can be written in three different ways:

1. #include <name-of-file>: #include<name-of-file> searches the prespecified list of directories (names of include directories can be set in IDE settings) for the source file (whose name is given within angular brackets), and text embeds the entire content of the source file in place of itself. If the file is not found there, it will show the error 'Unable to include 'name-of-file''.
2. #include "name-of-file": #include"name-of-file" first searches the file in the current working directory. If this search is not supported or if the search fails, this directive is reprocessed as if it reads #include<name-of-file>, i.e. the search will be carried out in the prespecified list of directories. If the search still fails, it will show the error 'Unable to include 'name-of-file''.
3. #include token-sequence: #include token-sequence searches the file as in Point 1 or in Point 2 depending upon the form of the directive to which it matches after the preprocessing token sequence is processed.

The piece of code in Program 10-34 illustrates the use of the third form of the include directive.

Line | **Prog 10-34.c**

```
//Source file inclusion directive
#define STR(x) #x
#include STR(stdio.h)
```

Output window

```
Third form of source file inclusion directive
```

(Contd...)

Line	Prog 10-34.c	Output window
4	main()	
5	{	
6	printf(“Third form of source file inclusion directive”);	
7	}	

Program 10-34 | A program that illustrates the use of the source file inclusion directive

10.5.4.3 line Directive

The line directive is used to reset the line number and the file name as reported by __LINE__ and __FILE__ macros. The line directive is used for the purpose of error diagnostics. The line directive has two forms:

1. #line constant: The line directive of this form causes the compiler to ascertain that the line number of the next source line is equal to the decimal integer constant specified in the directive. It has no effect on the file name as reported by the __FILE__ macro.
2. #line constant “filename”: The line directive of this form causes the compiler to ascertain that the line number of the next source line is given by the decimal integer constant and the current file is named by the identifier filename specified in the directive.

The piece of code in Program 10-35 illustrates the use of the line directive.

Line	Prog 10-35.c	Output window
1	//line directive	Line no. is 5, Filename is 10-35.c
2	#include <stdio.h>	Now, Line no. is 200, Filename is 10-35.c
3	main()	Atlast, Line no. is 100, Filename is Abc.c
4	{	**Remarks:**
5	printf(“Line no. is %d, Filename is %s\n”, __LINE__, __FILE__);	• In line number 6, the line directive assigns
6	#line 200	200 as the line number to the next line
7	printf(“Now, Line no. is %d, Filename is %s\n”, __LINE__,__FILE__);	• In line number 8, the line directive assigns
8	#line 100 “Abc.c”	100 as the line number to the next line and
9	printf(“Atlast, Line no. is %d, Filename is %s\n”, __LINE__, __FILE__);	changes the file name to “Abc.c”
10	}	

Program 10-35 | A program that illustrates the use of the line directive

10.5.4.4 error Directive

The error directive causes the preprocessor to generate the customized diagnostic messages and causes the compilation to fail. The error directive has the following forms:

1. #error: This directive causes the preprocessor to issue an error without any message.
2. #error token-sequence: This directive causes the preprocessor to issue an error message that includes the text specified by the token sequence.

The error directive is often used with conditional compilation directives.[§] The code segment in Program 10-36 illustrates the use of the error directive.

Prog 10-36.c

```
//error directive
#include <stdio.h>
#error This is a customized error message
main()
{
printf("Use of error directive cause the compilation to fail");
}
```

Output window

```
Compilation error
Fatal 10-36.C 3: Error directive: This is a customized error
message
```

Program 10-36 | A program that illustrates the use of the error directive

10.5.4.5 pragma Directive

The pragma directive is used to specify diverse options to the compiler. The options are specific for the compiler and the platform used. The pragma directive configures some of the compiler options that can otherwise be configured from the command line. Note that all options of the compiler cannot be configured using the pragma directive. An unrecognized pragma directive is ignored without an error or a warning message. It is strongly recommended to use the pragma directive after referring to the compiler documentation. The pragma directive is written as:

```
#pragma token-sequence
```

The commonly used forms of the pragma directive are as follows:

1. #pragma option: It is written as #pragma option [options...]. The common options that can be used with Turbo C 3.0 and the DOS environment are given in Table 10.5.

Table 10.5 | Some of the pragma options available with Turbo C 3.0

S.No	Option	Role
1.	-C	Allows nested comments
2.	-C-	(Default) Does not allow the nesting of comments
3.	-G	Causes the compiler to bias its optimization in favor of speed over size
4.	-G-	(Default) Causes the compiler to bias its optimization in favor of size over speed
5.	-r	(Default) Enables the use of register variables
6.	-r-	Suppresses the use of register variables
7.	-a	Forces structure members to be aligned on machine-word boundary.
8.	-a-	(Default) Results in byte alignment

[§] Refer Section 10.5.4.6 for a description on conditional compilation directives.

Program 10-37 illustrates the use of the pragma option –C.

Prog 10-37.c

```
//Nested multi-line comments
#include <stdio.h>
main()
{
/*Start of Outer Comment
  /*Inner Comment*/
End of Outer Comment*/
printf("By default nested comments are not allowed");
 }
```

Output window

```
Compilation error
```

Rectified code

```
//Nested multi-line comments
#include <stdio.h>
#pragma option -C
main()
{
/*Start of Outer Comment
  /*Inner Comment*/
End of Outer Comment*/
printf("By default nested comments are not allowed\n");
printf("pragma option –C makes them allowed");
}
```

Output window

```
By default nested comments are not allowed
pragma option –C makes them allowed
```

Program 10-37 | A program that illustrates the use of pragma option –C to allow nested comments

2. #pragma warn: The #pragma warn can be used to turn on, off or toggle the state of warnings. The #pragma warn can be written as:

```
#pragma warn +www     (Turns on the warning with character code www)
#pragma warn –www     (Turns off the warning with character code www)
#pragma warn .www     (Toggles the state of warning with character code www)
```

The character codes for specific warnings can be determined by referring to the compiler documentation. The common warning character codes that can be used with the pragma directive in Turbo C 3.0 are given in Table 10.6.

Table 10.6 | Some of the warning codes that can be used with the pragma directive in Turbo C 3.0

S.No	Warning code	Warning
1.	dup	Redefinition of 'macro' is not identical
2.	voi	void functions may not return a value
3.	rvl	Function should return a value
4.	par	Parameter 'parameter' is never used
5.	pia	Possibly incorrect assignment
6.	rch	Unreachable code
7.	aus	'Identifier' is assigned a value that is never used

The code snippets in Program 10-38 illustrate the use of #pragma warn to suppress the common warnings mentioned above.

Prog 10-38.c

```
//Suppression of warning messages
#include <stdio.h>
#define PI 2
#define PI 4
main()
{
int a=10;
if(a=PI)
    printf("The value of PI is %d",PI);
return 1;
    printf("This is unreachable statement");
}
```

Modified code

```
//Suppression of warning messages
#include <stdio.h>
#pragma warn –dup
#pragma warn –pia
#pragma warn –rch
#pragma warn –rvl
#pragma warn -aus
#define PI 2
#define PI 4
main()
{
int a=10;
if(a=PI)
    printf("The value of PI is %d",PI);
return 1;
    printf("This is unreachable statement");
}
```

Output window (Prog 10-38.c)

```
The value of PI is 4
Warnings(5):
Redefinition of PI is not identical
Possibly incorrect assignment in function main()
Unreachable code in function main()
Function should return a value in function main()
'a' is assigned a value that is never used in function main()
```

Output window (Modified code)

```
The value of PI is 4
Warnings(0)
```

Remark:

- All the warnings are suppressed by using the pragma directive

Program 10-38 | A program that illustrates the use of pragma directive to suppress various warnings

3. #pragma startup and #pragma exit: The #pragma startup and #pragma exit directives can be used to execute a function before and after the execution of the function main. These directives can be written as:

```
#pragma startup function-name
#pragma exit function-name
```

The piece of code in Program 10-39 illustrates the use of #pragma startup and #pragma exit directives.

Prog 10-39.c

```
//pragma startup and pragma exit directives
#include <stdio.h>
function_before_main()
{
printf("This will be executed before main\n");
}
function_after_main()
{
printf("This will be executed after main\n");
}
```

Output window

```
This will be executed before main
This is main function
This will be executed after main
```

Remarks:

- The output indicates that the function function_before_main is executed before, and the function function_after_main is executed after, the execution of the function main

(Contd...)

Line	Prog 10-39.c	Output window

11
12
13
14
15
16
	#pragma startup function_before_main
	#pragma exit function_after_main
	main()
	{
	printf("This is main function\n");
	}
		• The functions function_before_main and function_after_main must be defined before being used with the pragma directives
		• The function function_before_main can set up some prerequisites for the function main, and function_after_main can perform some clear up tasks

Program 10-39 | A program that illustrates the use of pragma startup and pragma exit

10.5.4.6 Conditional Compilation Directives

Conditional compilation means that a part of a program is compiled only if a certain condition comes out to be true. The available conditional compilation directives are as follows:

#if, #ifdef, #ifndef, #else, #elif, #endif

The syntax of using the conditional compilation directives is listed in Table 10.7.

Table 10.7 | Syntax and semantics of the conditional compilation directives

S.No	Conditional compilation directive	Syntax	Semantics
1.	#if-#endif	#if constant-exp statements-set #endif	The compiler compiles the statements-set only if the constant expression evaluates to true
2.	#if-#else-#endif	#if constant-exp statements-set1 #else statements-set2 #endif	If the constant expression evaluates to true, the statements-set1 will be compiled, else the statements-set2 will be compiled
3.	#if-#elif-#endif	#if constant-exp1 statements-set1 #elif constant-exp2 statements-set2 #endif	Statements-set1 will be compiled if the constant expression1 evaluates to true. The statements-set2 will be compiled only if the constant expression1 evaluates to false and constant expression2 evaluates to true
4.	#ifdef-#endif	#ifdef identifier statements-set #endif	Statements-set will be compiled only if the identifier is a predefined macro name or has been previously defined as a macro with define preprocessor directive without an intervening undef directive with the same identifier name
5.	#ifdef-#else-#endif	#ifdef identifier statements-set1 #else statements-set2 #endif	Statements-set1 will be compiled only if the identifier is a predefined macro name or has been previously defined as a macro name with define preprocessor directive without an intervening undef directive with the same identifier name. Otherwise, statements-set2 will be compiled

(Contd...)

6.	#ifdef-#elif-#endif	#ifdef identifier statements-set1 #elif constant-exp statements-set2 #endif	Statements-set1 will be compiled only if the identifier is a predefined macro or has been previously defined as a macro with define preprocessor directive without an intervening undef directive with the same identifier name. Statements-set2 will be compiled if no macro with the name as specified by the identifier has been defined, and the constant expression evaluates to true. If the macro has not been previously defined and the constant expression evaluates to false, no statements-set will be compiled
7.	#ifndef-#endif	#ifndef identifier statements-set #endif	Statements-set will be compiled if no macro with the name as specified by the identifier has been previously defined
8.	#ifndef-#else-#endif	#ifndef identifier statements-set1 #else statements-set2 #endif	Statements-set1 will be compiled if no macro with the name as specified by the identifier has been previously defined. Otherwise, statements-set2 will be compiled
9.	#ifndef-#elif-#endif	#ifndef identifier statements-set1 #elif constant-exp statements-set2 #endif	Statements-set1 will be compiled only if no macro with a name as specified by the identifier has been previously defined. Statements-set2 will be compiled if a macro with the name as specified by the identifier is a predefined macro or has been defined with define directive without an intervening undef directive with the same identifier name and the constant expression evaluates to true. If the macro has been defined and the constant expression evaluates to false, no statements-set will be compiled

The important points about the use of conditional compilation directives are as follows:

1. The conditional compilation preprocessor directives can appear anywhere in the program.
2. The statements set can be empty, can have preprocessor directives and/or C statements.

The piece of code in Program 10-40 illustrates the use of conditional compilation directives.

Line	Prog 10-40.c	Output window
1	//Conditional compilation directives	Embedded systems are used in real time applications
2	#include <stdio.h>	
3	#define EMBEDDED	
4	#ifndef EMBEDDED	
5	#error This code is meant for embedded systems only	
6	#endif	
7	main()	

(*Contd...*)

Line	Prog 10-40.c	Output window
8	{	
9	#ifdef EMBEDDED	
10	printf("Embedded systems are used in real time applications\n");	
11	#else	
12	This part of program will not be compiled	
13	Put code meant for Non-embedded systems	
14	#endif	
15	}	

Program 10-40 | A program that illustrates the use of conditional compilation directives

10.5.4.7 Null Directive

A null directive is of the form:

```
# new-line
```

The null directive has no effect.

10.6 Summary

1. A translator is a program that converts a program written in a source language to an equivalent program in a target language.
2. Translators are classified according to the classes of their source and target languages.
3. According to classes of their source and target languages, translators are classified as preprocessors, compilers, assemblers and interpreters.
4. The preprocessor is a translator that is invoked prior to the compiler.
5. The preprocessor is controlled by the commands known as preprocessor directives, which are not a part of C language.
6. There are eight phases of translation to convert a source program file into an executable file.
7. Trigraph replacement is the first phase of translation. During this phase, the trigraph sequences are replaced by their single-character equivalents.
8. During the second phase of translation, known as line splicing, each an instance of a backslash character immediately followed by a new-line character is deleted by the preprocessor.
9. The third phase of translation is tokenization, during which the source file is decomposed into the preprocessing tokens and a sequence of white-space characters.
10. During the fourth phase of translation, the preprocessor directives are executed and the macros are expanded.
11. A preprocessor directive always begins with a # (pound) symbol.
12. A macro is a facility provided by a C preprocessor, by which a token can be replaced by the user-defined sequence of characters.
13. Two types of macros can be created: object-like macros and function-like macros.

14. Object-like macro is also known as a symbolic constant.
15. Function-like macro is a macro with arguments.
16. A function-like macro is said to be safe, if it behaves like a function call.
17. Macros can create problems if they are not defined and used carefully.
18. The preprocessing token # is known as a stringizing operator.
19. The preprocessing token ## is used for token pasting.
20. Conditional compilation directives are used for conditional compilation. It means that a part of a program is compiled only if a certain condition comes out to be true.
21. Conditional compilation directives are: #if, #ifdef, #ifndef, #else, #elif, #endif.

Exercise Questions

Conceptual Questions and Answers

1. *What are translators and how are they classified?*

   Refer Section 10.4.

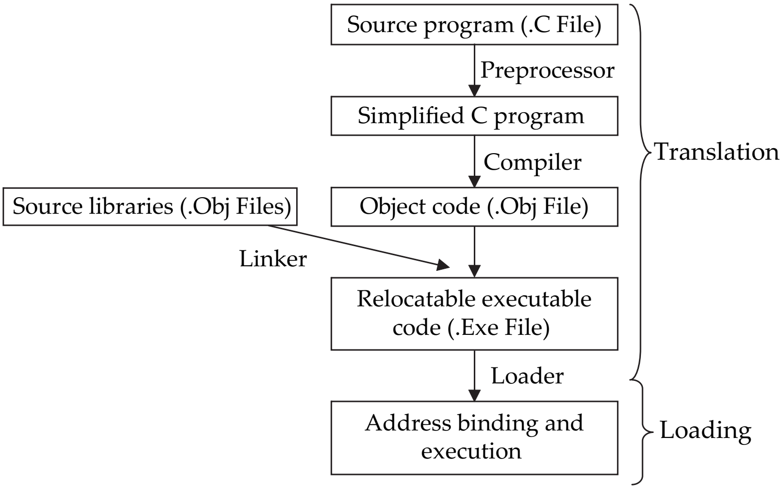
2. *What are the various stages a program undergoes before execution?*

   The various stages a program undergoes before execution are:

   1. Translation
   2. Loading

   The various parts of translation are:

   1. Preprocessing
   2. Compilation
   3. Linking



3. *What are the various phases of translation?*

   Refer Section 10.5.

4. *What is a trigraph sequence and a digraph sequence?*

   Refer Section 10.5.1 on trigraph sequence.

Digraph sequences are a pair of characters that get replaced by their character equivalent. Similar to a trigraph processor, a separate digraph processor is required for processing the digraph sequences. The following table lists the valid digraph sequences and their character equivalents:

S.No	Digraph sequence	Character equivalent
1.	<:	[
2.	:>	]
3.	<%	{
4.	%>	}
5.	%:	#
6.	%:%:	##

The major difference between trigraph sequences and digraph sequences is that trigraphs are replaced within string literals but digraph sequences are not.

5. *What is line splicing?*

   Refer Section 10.5.2.

6. *What is the difference between a token and a processing token?*

   Refer Section 10.5.3. Also refer question number 1 (Chapter 4).

7. *How are preprocessor directives written? List the various preprocessor directives available in C.*

   Refer Section 10.5.4.

8. *What is a macro? What are object-like macros and function-like macros? How are they defined?*

9. *The following lines of code are written in a source file:*

```
#define EMPTY              //←line number 1
EMPTY #include <stdio.h>  //←line number 2
```

   Can you say that line number 2 is a preprocessor directive?

   Line number 2 begins with a macro name EMPTY. Since line number 2 does not begin with a pound symbol (#), it will not be said as a preprocessor directive

10. *Why is the following piece of code not working?*

```
#define PI=3.1417
main()
{
    printf("The value of constant PI is %f",PI);
}
```

   The following piece of code is not working due to the erroneous definition of the object-like macro PI. There should be a white-space character between the macro name and the replacement list in the definition of the object-like macro PI instead of the character '='. The rectified piece of code is written as

```
#define PI 3.1428571
main()
{
    printf("The value of constant PI is",PI);
}
```

11. *I have read that 'An identifier should be declared before it is used, else there will be a compilation error'. The identifier* PI *has not been declared in the following piece of code but still the code gets executed and the compiler does not show any error. Why?*

```
#define PI 3.1417
main()
{
    printf("The value is %f", PI);
}
```

The compiler does not show any error because the compiler does not find any token PI as it has already been replaced by the replacement list 3.1417 during the preprocessing stage. After the preprocessing stage and macro expansion, the processed code handed over to the compiler will be

```
main()
{
    printf("The value is %f", 3.1417);
}
```

Since this code does not contain any instance of the token PI, there is no requirement to declare it.

12. *I have written the following piece of code:*

```
#define square(x) x*x
main()
{
    float result;
    result=1.0/square(2);
    printf("Result is %f",result);
}
```

I was expecting the output of the code to be 0.250000*, but on execution, the code outputs* 1.000000*. Why? How can I rectify it?*

Remember that macro expansion is purely textual. Macros are expanded during the preprocessing stage before the compilation stage. This fact is illustrated by the code segments listed below:

Before the preprocessing stage	After the preprocessing stage
#define square(x) x*x	
main()	main()
{	{
float result;	float result;
result=1.0/square(2);	result=1.0/2*2; //←Macro expanded
printf("Result is %f",result);	printf("Result is %f",result);
}	}

After the macro expansion is carried out during the preprocessing stage, the expression `result=1.0/square(2);` becomes `result=1.0/2*2;`. Since the division operator and the multiplication operator have the same precedence and are left-to-right associative, the division is carried out first and then the multiplication is done. Thus, the result comes out to be 1.000000. The given piece of code can be rectified by **parenthesizing the macro's replacement list to protect any lower precedence operators present in it from the higher precedence operators present in the surrounding expression.** The rectified code is given below:

```
#define square(x) (x*x)
main()
{
    float result;
    result=1.0/square(2);
    printf("Result is %f",result);
}
```

The mentioned rectified code on execution outputs: `Result is 0.250000`

13. *I have defined the macro in the way suggested in the answer of the previous question and have written the following piece of code:*

```
#define square(x) (x*x)
main()
{
    int number=2,result;
    result=square(number+1);
    printf("Square of 3 is %d",result);
}
```

Still the code does not work as intended and outputs 5 *instead of* 9. *Why? How can I rectify it?*

After macro expansion is done during the preprocessing stage, the given piece of code becomes:

```
main()
{
    int number=2,result;
    result=(number+1*number+1);
    printf("Square of 3 is %d",result);
}
```

The expression `result=(number+1*number+1);` evaluates to 5 instead of the expected value 9 because the multiplication operator has a higher precedence than the addition operator. The given piece of code can be rectified **by parenthesizing all the occurrences of the parameters in the macro's replacement list to protect any low precedence operators in the actual arguments from the rest of the macro expansion.** The rectified code is given below:

```
#define square(x) ((x)*(x))
main()
{
    int number=2,result;
    result=square(number+1);
    printf("Square of 3 is %d",result);
}
```

The mentioned rectified code on execution outputs: `Square of 3 is 9`

14. *If I define the macro* `square` *as suggested in the answer of the previous question and call it in an expression, can I safely assume that my code will work correctly as if it were an expression statement consisting of a function call?*

If the macro square is defined in the way suggested in Answer number 13, still it cannot be safely assumed that an expression containing a call to the macro square is the same as if it is an expression containing a call to a function that returns the squared value of its input parameter. Consider the following pieces of code and the differences in the results of their executions:

Code-I Macro version	Code-II Function version

`#define square(x) ((x)*(x))`
`main()`
`{`
`int i=2,result;`
`result=square(++i);`
`printf("The value of result is %d",result);`
`}`
	`int square(int x){`
	`return x*x; }`
	`main()`
	`{`
	`int i=2,result;`
	`result=square(++i);`
	`printf("The value of result is %d",result);`
	`}`

The execution of code-I, i.e. macro version outputs: `The value of result is 16.`
The execution of code-II, i.e. function version outputs: `The value of result is 9.`
The macro square exhibited this type of behavior because its argument is an expression (i.e. ++i) with a side-effect.

15. *What are the points that one should keep in mind while defining macros?*

    Refer Section 10.5.4.1.2.

16. *What are the differences between function-like macros and functions?*

    Although function-like macros and functions appear to be the same, they are actually not. The major differences between function-like macros and functions are as follows:

Function-like macros	Functions
1. The replacement list of function-like macros is just text replaced during the preprocessing stage every time the macro name is encountered. There is no argument passing and no control is transferred.	1. In a function call, the control is passed to the called function along with the arguments, the calculations are made in the called function and their value is returned to the calling function.
2. Since the control is not actually transferred, the time required in making a function call is saved. Thus, the use of function-like macros provides a better performance as compared to functions.	2. As the control transfers to and fro between the called function and the calling function, some of the time gets wasted in making the function call. Thus, the use of functions and their calls slow down the program.
3. Since the macro name is text replaced by the replacement list during the preprocessing stage, the use of macros will increase the program size. This increases the code redundancy.	3. Functions use the same piece of code again and again. Hence, they avoid code redundancy and this is the main benefit of using functions.
4. Thus, the use of macros makes the program run faster but increases the program size.	4. Thus, the use of the function makes the program smaller and compact but it deteriorates the program's speed.

17. *I have encountered the following piece of code that makes use of an object-like macro* PI. *When I try to execute the code, it gives an error* 'Undefined symbol PI'. *Why?*

```
#define PI 3.141
#undef PI
main()
{
    int rad=2;
    printf("Area of circle is %f",PI*rad*rad);
}
```

The given piece of code on compilation gives an error 'Undefined symbol PI' due to the usage of undef directive. The symbol PI has been defined as an object-like macro but as it has been undefined with the undef directive before its use; therefore, the preprocessor will not be able to make the macro replacement. That is why, the compiler shows the error.

18. *What is meant by token replacement and token pasting?*

Refer Sections 10.5.4.1.3 and 10.5.4.1.4.

19. *Is macro replacement carried out within a string literal constant?*

No replacement is carried out if a name identical to the macro name appears as a part of a string literal constant or as a part of some other name. For example, consider the following piece of code:

```
#define LINE 100
main()
{
    int MAXLINE=25;
    printf("The length of LINE is %d", MAXLINE);
}
```

The mentioned piece of code on execution prints: The length of LINE is 25. No replacement is carried out for the name LINE that appears as a part of the string literal or as a part of the name MAXLINE.

20. *What are the various ways in which a source file inclusion directive can be written?*

Refer Section 10.5.4.2.

21. *What method is adopted for locating the includable source files in ANSI specifications?*

According to ANSI specifications:

(1) **#include<name-of-file>** searches a prespecified list of directories (names of include directories can be set in IDE settings) for the source file (whose name is given within angular brackets), and text embeds the entire content of the source file in place of itself. If the file is not found there, it will show error 'Unable to include name-of-file'.

(2) **#include"name-of-file"** first searches the file in the current working directory. If this search is not supported or if the search fails, this directive is reprocessed as if it reads #include<name-of-file>, i.e. search will be carried out in a prespecified list of directories. If the search still fails, it will show the error 'Unable to include name-of-file'.

(3) **#include token-sequence** searches the file as in (1) or (2) depending upon the form of directive to which it matches after the token sequence is processed.

22. *Is there any difference that arises if double quotes, instead of angular brackets are used for including the standard header files?*

If double quotes are used for the inclusion of standard header files instead of angular brackets, the search space unnecessarily increases (in addition to the list of prespecified directories, search will unnecessarily be carried out first in the current working directory), and the time required for the inclusion will be more.

23. *Under what circumstances should the use of quotes be preferred over the use of angular brackets for the inclusion of header files and under what circumstances is the use of angular brackets beneficial?*

    Self-created or user-defined header files should be included with double quotes because the inclusion with double quotes makes the files to be searched first in the current working directory (where the user has kept self-created header files) and then in the prespecified list of directories. If the standard header files are to be included, angular brackets should be used because the standard header files are present in the prespecified list of directories and there is no use in searching them in the current working directory. Usage of double quotes for including the standard header files will work but will take more time.

24. *What is conditional compilation?*

    Refer Section 10.5.4.6.

25. *What is the role of the* error *directive?*

    Refer Section 10.5.4.4.

    Suppose the user wants to develop some functionality that is very specific to some applications like Video Graphic Adaptors (VGAs), etc. The user has written the following piece of code:

```
main()
{
    #ifndef VGA
        #error This code is for Video Graphic Adaptors only
    #else
        int hresolution=640, vresolution=480;
        //....code specific to VGA follows
    #endif
}
```

    The code on compilation gives 'Fatal error: This code is for Video Graphic Adaptors only' as VGA is not previously defined. If VGA is previously defined using the define directive, the code sets the horizontal and vertical resolution to be 640 and 480, respectively, and the other code statements specific to VGA will be processed.

26. *What is the role of the* pragma *directive?*

    Refer Section 10.5.4.5.

27. *Are nested multi-line comments by default allowed in C? If no, how can the* pragma *directive be used to allow them?*

    No, by default the nested multi-line comments are not allowed. Use #pragma option –C to make the nested multi-line comments allowed.

    Refer Section 10.5.4.5.

28. *How can the* pragma *directive be used to suppress* 'Function should return a value' *warning?*

    Refer Section 10.5.4.5.

29. *By default, a program execution always starts with and terminates with the function* main. *Can I make some other function to execute before or after the execution of the function* main*? If yes, how?*

    Yes, the #pragma startup and #pragma exit directives can be used to execute a function before and after the execution of the function main.

    Refer Section 10.5.4.5.

30. *A compiler can translate a high-level language program into an equivalent low-level language program, i.e. assembly-level language or machine-level language program. Till now, the compiler has been producing a machine-level code. How can I configure the compiler so that it starts producing an assembly-level code?*

Assembly-level code can be generated by using -S option of the Turbo C 3.0 compiler. It should be noted that this option cannot be used with the pragma directive. It should be invoked from the command line only.

Code Snippets

Determine the output of the following code snippets. Assume that the inclusion of the required header files has been made and there is no prototyping error due to them. Also, the trigraph processor is available and is invoked first.

31.
```
??=include<stdio.h>
main()
{
    int arr??(5??)=??<1,2,3,4,5??>;
    printf(“The first three elements are: %d %d %d”,arr[0],arr[1],arr[2]);
}
```

32.
```
%:include<stdio.h>
main()
<%
    int arr<:5:>=<%1,2,3,4,5%>;
    printf(“The first three elements are:\n%d %d %d”,arr[0],arr[1],arr[2]);
%>
```

33.
```
main()
{
    printf(“Trigraph??/tsequences??/nin string literal”);
}
```

34.
```
main()
{
    printf(“Digraph<:sequences:>”);
}
```

35.
```
main()
{
    printf(“Will it be replaced???/tYes/No?”);
}
```

36.
```
main()
{
    printf(“Hello
Readers!!”);
}
```

37
```
main()
{
    printf(“Hello \
Readers!!”);
}
```

38.
```
main()
{
    printf(“Hello ””Readers!!”);
}
```

```
39. main()
    {
        printf(“Hello ” ”Readers!!”);
    }

40. main()
    {
        char *str=”Hello”;
        printf(str”Readers!!”);
    }

41. #define PI=3.14
    main()
    {
        int rad=2;
        printf(“Circumference of the circle is %f”,2*PI*rad);
    }

42. #define PI 3.14
    main()
    {
        int rad=2;
        printf(“Circumference of the circle is %f”,2*PI*rad);
    }

43. #define PI 3.14;
    main()
    {
        int rad=2;
        printf(“Circumference of the circle is %f”,2*PI*rad);
    }

44. #define int char
    main()
    {
        int var;
        printf(“The size of var is %d”,sizeof(var));
    }

45. #define + -
    #define * /
    main()
    {
        int a;
        a=2+3*5;
        printf(“The value of a is %d”,a);
    }

46. #define clrscr() 200
    main()
    {
        printf(“This will be printed\n”);
        clrscr();
    printf(“The value is %d”,clrscr());
}
```

47.
```
#define SQUARE(x) x*x
main()
{
    printf(“The square value of 2 is %d”, SQUARE(2));
}
```

48.
```
#define SQUARE (x) x*x
main()
{
    printf(“The square value of 2 is %d”,SQUARE(2));
}
```

49.
```
#define SQUARE(x) x*x
main()
{
    int a=20,b;
    b=a/SQUARE(2);
    printf(“The value of b is %d”,b);
}
```

50.
```
#define SQUARE(x) (x*x)
main()
{
    int a=20,b;
    b=a/SQUARE(2);
    printf(“The value of b is %d”,b);
}
```

51.
```
#define SQUARE(x) (x*x)
main()
{
    int a=5,b;
    b=SQUARE(a+2);
    printf(“The value of b is %d”,b);
}
```

52.
```
#define SQUARE(x) ((x)*(x))
main()
{
    int a=5,b;
    b=SQUARE(a+2);
    printf(“The value of b is %d”,b);
}
```

53.
```
#define SQUARE(x) ((x)*(x))
main()
{
    int a=2,b;
    b=SQUARE(++a);
    printf(“The value of b is %d”,b);
}
```

54.
```
#define SQUARE(x) ((x)*(x));
main()
{
    int a=2,b=4;
    if(SQUARE(a)==b)
        printf("Square of a is equal to b");
    else
        printf("Square of a is not equal to b");
}
```

55.
```
#define SWAP(a,b) a^=b; b^=a; a^=b;
main()
{
    int a=20,b=10;
    printf("The values of a and b before swap are %d %d\n",a,b);
    SWAP(a,b)
    printf("The values of a and b after swap are %d %d\n",a,b);
}
```

56.
```
#define SWAP(a,b) a^=b; b^=a; a^=b;
main()
{
    int a=20,b=10;
    printf("Swap the values of a and b only if a is greater than b");
    if(a>b)
        SWAP(a,b)
    else
        printf("Values are not swapped");
    printf("Resultant values of a and b are %d %d",a,b);
}
```

57.
```
#define SWAP(a,b) a^=b, b^=a, a^=b
main()
{
    int a=20,b=10;
    printf("Swap the values of a and b only if a is greater than b\n");
    if(a>b)
        SWAP(a,b);
    else
        printf("Values are not swapped\n");
    printf("Resultant values of a and b are %d %d",a,b);
}
```

58.
```
#define SWAP(a,b) a^=b^=a^=b
main()
{
    int a=20,b=10;
    printf("Swap the values of a and b only if a is greater than b\n");
    if(a>b)
        SWAP(a,b);
    else
        printf("Values are not swapped\n");
    printf(“Resultant values of a and b are %d %d”,a,b);
}
```

59.
    ```
    #define VALUE 100
    main()
    {
        int MAXVALUE=1000;
        printf(“The VALUE is %d”,MAXVALUE);
    }
    ```

60.
    ```
    #define STR(x) #x
    main()
    {
        printf(STR(Hello Readers!!));
    }
    ```

61.
    ```
    #define STR(x) #x
    main()
    {
        printf(STR(Hello                    Readers!!));
    }
    ```

62.
    ```
    #define STR(x) #x
    main()
    {
        printf(STR(                 Hello Readers!!             ));
    }
    ```

63.
    ```
    #define STR(x) #x
    main()
    {
        printf(STR(Hello “Read”ers!!));
    }
    ```

64.
    ```
    #define STR(x,y,z) #x#y#z
    main()
    {
        char str1[30]=STR(THE,C,PREPROCESSOR);
        char str2[30]=STR(THE,C,COMPILER);
        puts(str1);
        puts(str2);
    }
    ```

65.
    ```
    #define STR(x) #x
    #include STR(stdio.h)
    main()
    {
        printf(“Third form of include directive”);
    }
    ```

66.
    ```
    #define PASTE(tk1,tk2) tk1##tk2
    main()
    {
        int var1=100;
        printf(“The value of var1 is %d”,PASTE(var,1));
    }
    ```

67.
```
#define PASTE(tk1,tk2) tk1##tk2
main()
{
     int var1=100,var2=200,var3=300;
     int i;
     for(i=1;i<=3;i++)
          printf(“The value of var%d is %d\n”,i,PASTE(var,i));
}
```

68.
```
#define PASTE(tk1,tk2) tk1##tk2
main()
{
     int var[]={100,200,300};
     int i;
     for(i=0;i<=2;i++)
          printf(“The value of var%d is %d\n”,i,PASTE(var,[i]));
}
```

69.
```
#define p(x,y,z) x##y##z
main()
{
     int arr[]={ p(2,3,4), p(,5,6), p(6,,7), p(8,9,), p(10,,), p(,11,)},i;
     for(i=0;i<6;i++)
          printf(“%d ”,arr[i]);
}
```

70.
```
#define CONST 100
#undef CONST
main()
{
     printf(“The value of CONST is %d”,CONST);
}
```

71.
```
#define CONST 100
#undef VAR
main()
{
     printf(“The value of CONST is %d”,CONST);
}
```

72.
```
#define CONST 100
main()
{
     printf(“The value of CONST is %d”,CONST);
     #undef CONST
}
```

73.
```
#define CONST 100
main()
{
     printf(“The value of CONST is %d”,CONST);
}
#undef CONST
```

74.
```
#define CONST 100
main()
{
     #define CONST 10
     printf("The value of CONST is %d",CONST);
}
```

75.
```
#define VER 1
main()
{
     #ifdef VER
          printf("Place code corresponding to version 1");
     #else
         printf("Place code corresponding to version other than 1");
     #endif
}
```

76.
```
#define VER 1
main()
{
     #ifdef VER
          printf("Place code corresponding to version 1");
     #else
          WILL IT BE A COMPILATION ERROR??
     #endif
}
```

77.
```
#define VER 1
main()
{
     #if VER==1
          printf("Place code corresponding to version 1");
     #else
         printf("Place code corresponding to version other than 1");
     #endif
}
```

78.
```
#define VER 1
main()
{
     #if VER=1
          printf("Place code corresponding to version 1");
     #else
         printf("Place code corresponding to version other than 1");
     #endif
}
```

79.
```
#define VER 1
main()
{
     int a=1;
     #if a==VER
```

```
        printf("Place code corresponding to version 1");
    #else
        printf("Place code corresponding to version other than 1");
    #endif
}
```

80.
```
#define VER 1
main()
{
    const int a=1;
    #if a==VER
        printf("Place code corresponding to version 1");
    #else
        printf("Place code corresponding to version other than 1");
    #endif
}
```

81.
```
main()
{
    #ifdef WINDOWS
       PLACE CODE FOR WINDOWS OPERATING SYSTEM
    #elif defined(LINUX)
        PLACE CODE FOR LINUX OPERATING SYSTEM
    #else
        #error OPERATING SYSTEM IS NOT KNOWN
    #endif
}
```

82.
```
main()
{
/* The C PREPROCESSOR
/* THERE ARE VARIOUS DIRECTIVES*/
PRAGMA IS ONE OF THEM*/
printf("THE C PREPROCESSOR");
}
```

83.
```
#pragma option –C
main()
{
/* The C PREPROCESSOR
/* THERE ARE VARIOUS DIRECTIVES*/
PRAGMA IS ONE OF THEM*/
printf("THE C PREPROCESSOR");
}
```

84.
```
int req_var_value;
func()
{
    printf("This function setups the prerequisites of function main\n");
    req_var_value=200;
}
#pragma startup func
main()
```

```
{
    printf("This is function main\n");
    printf("The requisite value of variable is %d",req_var_value);
}
```

85.
```
#define size_of(data) ((char *)(&data+1)-(char *)(&data))
main()
{
    int INT;
    char CHAR;
    float FLOAT;
    double DOUBLE;
    printf("Size of int: %d\n",size_of(INT));
    printf("Size of char: %d\n",size_of(CHAR));
    printf("Size of float: %d\n",size_of(FLOAT));
    printf("Size of double: %d\n",size_of(DOUBLE));
 }
```

Multiple-choice Questions

86. A translator that converts a program written in a high-level language into an equivalent program written in some other high-level language is

a. Interpreter
b. Compiler
c. Assembler
d. Preprocessor

87. Preprocessing is a phase of translation, which occurs

a. Before compilation
b. After compilation
c. After compilation but before linking
d. None of these

88. Which of the following are replaced even within string literals?

a. Macro names
b. Digraph sequences
c. Trigraph sequences
d. None of these

89. Among function-like macro call and function call, which one is efficient time-wise?

a. Function-like macro call
b. Function call
c. Both take equal time
d. None of these

90. The following piece of code on execution leads to:
```
main(){
puts("Hello","Readers!!"); }
```

a. Compilation error
b. Hello
c. Readers!!
d. None of these

91. The following piece of code on execution leads to:
```
#define puts printf
main(){
puts("Hello","Readers!!"); }
```

a. Compilation error
b. Hello
c. Readers!!
d. None of these

92. The following piece of code on execution leads to:

```
#define int char
main(){
int a=4;
printf("%d",sizeof(a)); }
```

a. Compilation error
b. 1
c. 2
d. None of these

93. The following piece of code on execution leads to:

```
#define sizeof
main(){
int a=4;
printf("%d",sizeof(a)); }
```

a. Compilation error
b. 1
c. 2
d. 4

94. The following piece of code on execution leads to:

```
#define a 10
void fun();
main()
{
    fun();
    printf("%d",a); }
fun()
{
    #undef a
    #define a 50
}
```

a. Compilation error
b. 10
c. 50
d. None of these

95. If the following piece of code is executed on a 16-bit DOS environment, the output will be

```
#define cp_d char*
typedef char* cp_t;
main(){
cp_t p1,p2;
cp_d p3,p4;
printf("%d %d\n",sizeof(p1), sizeof(p2));
printf("%d %d",sizeof(p3), sizeof(p4));
}
```

a.
```
2 2
2 2
```
b.
```
2 2
2 1
```
c.
```
2 1
2 1
```
d.
```
2 1
2 2
```

Outputs and Explanations to Code Snippets

31.
    ```
    The first three elements are: 1 2 3
    ```

    Explanation:

    The source file contains the trigraph sequences like ??=, ??(, ??), ??< and ??>. During the first phase of translation, these trigraph sequences are replaced by their character equivalents #, [,], { and }, respectively, by the Borland trigraph processor TRIGRAPH.EXE.

32.
    ```
    The first three elements are: 1 2 3
    ```

    Explanation:

    The digraph sequences are replaced by their character equivalents. The digraph sequences %:, <%, %>, <: and :> are replaced by #, {, }, [and], respectively. Some of the IDEs like GNU provides an integrated digraph processor with a GNU GCC compiler while some of them like Turbo C require a separate digraph processor.

33.
    ```
    Trigraph        sequences
    in string literal
    ```

    Explanation:

    Trigraph sequences are replaced even within string literals. The trigraph sequence ??/ in the string literal "Trigraph??/tsequences??/nin string literal" is replaced by the character equivalent \. Hence, the string literal after the trigraph replacement becomes "Trigraph\tsequences\nin string literal". The resultant string when printed produces the mentioned output.

34.
    ```
    Digraph<:sequences:>
    ```

    Explanation:

    Refer to the explanation given in Answer number 4.

    The digraph sequences within string literals are not replaced.

35.
    ```
    Will it be replaced?        Yes/No?
    ```

    Explanation:

    Trigraph sequences are replaced within string literals. After processing, the trigraph processor outputs:

    ```
    main
    {
        printf("Will it be replaced?\tYes/No?");
    }
    ```

    The processed code on execution outputs the above-mentioned result.

36.
    ```
    Compilation error "Unterminated string or character constant"
    ```

    Explanation:

    String literals cannot span multiple lines in this way.

37.
    ```
    Hello Readers!!
    ```

    Explanation:

    Refer to the explanation given in Section 10.5.2.

    During phase 2 of translation, the physical source lines in

    ```
    main()
    {
        printf("Hello \
    ```

```
    Readers!!");
}
```

are spliced to form the following logical source lines:

```
main()
{
    printf("Hello Readers!!");
}
```

Logical source lines are processed by the compiler. Hence, on execution, Hello Readers!! is the output.

38. Hello Readers!!

Explanation:

Refer Section 10.5.

During phase 6 of translation, adjacent string literal constants are concatenated.

39. Hello Readers!!

Explanation:

Refer Section 10.5.

During phase 7 of translation, the white-space characters between two tokens are removed. After the execution of this phase, the white space between the string literal tokens "Hello " and "Readers!!" is removed and they become adjacent to each other. During rescanning and further replacement, these adjacent string literals are concatenated to form "Hello Readers!!".

40. Compilation error

Explanation:

During translation, only the string literals are concatenated. str is not a string literal. A try to concatenate the string pointed to by str with the string literal "Readers!!" leads to the compilation error.

41. Compilation error

Explanation:

Refer Section 10.5.4.1.2.

The compilation error is due to the erroneous definition of object-like macro PI.
There shall be a white-space character (blank-space character or horizontal tab-space character) between the macro name and the replacement list in the definition of the object-like macro instead of the character '='.

42. Circumference of the circle is 12.560000

Explanation:

During phase 4 of translation, macro names are replaced by their replacement list. Thus, after phase 4 of translation, the source code becomes:

```
main()
{
    int rad=2;
    printf("Circumference of the circle is %f", 2*3.14*rad);
}
```

The above code on execution outputs the above-mentioned result.

43. Compilation error

Explanation:

After macro expansion, the statement printf("Circumference of the circle is %f",2*PI *rad); becomes printf("Circumference of the circle is %f",2*3.14;*rad);, which is not valid due to the occurrence of the semicolon after 3.14. It is always recommended to avoid the use of semicolon in or at the end of a macro definition.

44. The size of var is 1

    Explanation:

    It is legal to use a reserve word as a macro name. However, this should be done with utmost care. After the preprocessing stage, the declaration int var; becomes char var;. Since the memory allocation is done by the compiler, to which the type of identifier var is char, it allocates 1 byte to it. Hence, the size of var comes out to be 1.

45. Compilation error "Define directive needs an identifier"

    Explanation:

    Macro name should be identifiers. Since + and – are not valid identifiers, they cannot be used as macro names.

46. This will be printed
    The value is 200

    Explanation:

    After the macro expansion, the code becomes:

    ```
    main()
    {
        printf("This will be printed\n");
        200;
        printf("The value is %d",200);
    }
    ```

    The above code is free from any compilation error and on execution gives the above-mentioned result.

47. The square value of 2 is 4

    Explanation:

    Refer Section 10.5.4.1.

48. Compilation error "Undefined symbol x in function main"

    Explanation:

    SQUARE in the given piece of code does not become a function-like macro. It becomes an object-like macro due to the white-space character between the macro name SQUARE and left parenthesis. After macro expansion, the given piece of code becomes:

    ```
    main()
    {
        printf("The square value of 2 is %d",(x) x*x(2));
    }
    ```

    The preprocessed code on compilation gives 'Undefined symbol x' error because the symbol x has not been declared. Even if x would have been declared, there would still be an error because expression (x) x*x(2) is not well formed.

49. The value of b is 20

    Explanation:

    After the macro expansion, the expression b=a/SQUARE(2) becomes b=a/2*2. Since the division and the multiplication operators have the same precedence and are left-to-right associative, in the given expression division is carried out first and then multiplication is done.

50. The value of b is 5

    Explanation:

    After the macro expansion, the expression b=a/SQUARE(2) becomes b=a/(2*2), which on evaluation assigns 5 to the variable b. The result is different from the result of the execution in Answer number 49 because the replacement list of macro SQUARE has been parenthesized.

51. The value of b is 17

    Explanation:

    After the macro expansion, the expression b=SQUARE(a+2) becomes b=a+2*a+2. Since the multiplication operator has higher precedence as compared to the addition operator, multiplication is carried out first. Hence, the right side of the expression b=5+2*5+2 evaluates to 17, which is then assigned to b.

52. The value of b is 49

    Explanation:

    After the macro expansion, the expression b=SQUARE(a+2) becomes b=(a+2)*(a+2), which on evaluation assigns 49 to the variable b. The result is different from the result of the execution in Answer number 51 because all the occurrences of the parameters in the replacement list of macro SQUARE has been parenthesized.

53. The value of b is 16

    Explanation:

    Refer Section 10.5.4.1.2.3.

54. Compilation error

    Explanation:

    The semicolon at the end of the macro definition is the cause of the compilation error. After the macro expansion, the if controlling expression becomes ((a)*(a));==b. The expression is ill-formed and on compilation leads to an error.

55. The values of a and b before swap are 20 10
    The values of a and b after swap are 10 20

    Explanation:

    After the expansion of the macro SWAP, the given piece of code becomes:

    ```
    main()
    {
       int a=20,b=10;
       printf("The values of a and b before swap is %d %d\n",a,b);
       a^=b; b^=a; a^=b;
       printf("The values of a and b after swap is %d %d\n",a,b);
    }
    ```

    The above code swaps the values of a and b.

56. Compilation error "Misplaced else in function main"

    Explanation:

    When the macro SWAP is replaced by the multiple statements (i.e. a^=b; b^=a; a^=b;), only the first statement (i.e. a^=b;) forms the if body. The other two statements will be considered as the statements next to the if statement. The else clause remains unmatched and leads to 'Misplaced else' error.

57. Swap the values of a and b only if a is greater than b
    Resultant values of a and b are 10 20

Explanation:

After the expansion of the macro SWAP, there will be a single statement in the if body. Hence, there will be no error as in Answer number 56.

58. Swap the values of a and b only if a is greater than b
    Resultant values of a and b are 10 20

    Explanation:

    After the expansion of the macro SWAP, there will be a single statement in the if body, which swaps the value of the variables a and b.

59. The VALUE is 1000

    Explanation:

    No replacement is carried out if a name same as the macro name appears as a part of a string literal constant or as a part of some other name.

60. Hello Readers!!

    Explanation:

    The stringizing operator # preceding a parameter of a function-like macro converts an argument corresponding to the parameter into a string literal. In the given piece of code, STR(Hello Readers!!) gets converted to a string literal “Hello Readers!!” and is printed.

61. Hello Readers!!

    Explanation:

    The stringizing operator # converts a sequence of white-space characters between the argument’s preprocessing tokens into a single white-space character in the replaced string literal. In the given piece of code, STR(Hello   Readers!!) gets converted to “Hello Readers!!”.

62. Hello Readers!!

    Explanation:

    The stringizing operator # deletes the white-space characters before the first preprocessing token and after the last preprocessing token of the argument. In the given piece of code, STR(Hello Readers!!   ) gets converted to “Hello Readers!!” and is printed.

63. Hello “Read”ers!!

    Explanation:

    The stringizing operator # inserts backslash character (i.e. \) before every instance of ” and \ characters that appears in the argument while converting it into string literal. In the given piece of code, STR(Hello “Read”ers!!) gets converted to “Hello \“Read\”ers!!”. This string is printed by the printf function. Hence, the output is Hello "Read"ers!!.

64. THECPREPROCESSOR
    THECCOMPILER

    Explanation:

    The stringizing operator converts each argument corresponding to a parameter into a string literal, and the adjacent string literals get concatenated.

65. Third form of include directive

    Explanation:

    The stringizing operator converts stdio.h into “stdio.h”. After replacement, the source file inclusion directive becomes #include“stdio.h”. This form of include directive is valid and searches the file stdio.h firstly in the current working directory and then in the prespecified list of directories.

66. The value of var1 is 100

Explanation:

In a function-like macro definition, if in the replacement list, a ## preprocessing token appears between two parameters, the parameters are replaced by the corresponding arguments and the arguments are glued and pasted to form one token. In the given piece of code, the arguments var and 1 corresponding to the parameters tk1 and tk2, respectively, are pasted to create one token, i.e. var1. Hence, after preprocessing, the given piece of code becomes:

```
main()
{
    int var1=100;
    printf("The value of var1 is %d",var1);
}
```

This code on execution outputs the mentioned result.

67. Compilation error "Undefined symbol vari in function main"

Explanation:

During the preprocessing stage, the macro PASTE performs the token pasting and gets replaced by vari. During the compilation stage, the name vari is found to be undefined and a compile time error is raised.

68. The value of var0 is 100
The value of var1 is 200
The value of var2 is 300

Explanation:

During the preprocessing stage, the macro PASTE performs the token pasting and gets replaced by var[i]. To C compiler var[i] is a well-formed expression having a subscript operator whose operands are of array type and integer type. Hence, on execution the given code outputs the mentioned result.

69. 234 56 67 89 10 11

Explanation:

The preprocessing tokens ## paste the arguments corresponding to the parameters x, y and z. If any of the argument corresponding to the parameter x, y or z is missing, it will be ignored. After token pasting and macro expansion, p(2,3,4) will be replaced by one token, i.e. 234. Similarly, p(,5,6) will be replaced by 56 as the missing argument corresponding to the parameter x is ignored.

70. Compilation error "Undefined symbol CONST in function main"

Explanation:

The undef directive causes the CONST preprocessor definition to be no longer defined as a macro name. Hence, during the preprocessing stage, no macro expansion is carried out for CONST. After the preprocessing stage, during the compilation stage, there will be a compilation error since the name CONST has not been declared.

71. The value of CONST is 100

Explanation:

It is not erroneous to apply undef to an unknown identifier. Hence, #undef VAR is perfectly valid. Since, VAR has not been previously defined using the define directive, this directive will be ignored without any error or warning message.

72. The value of CONST is 100

    Explanation:

    At the point of usage of CONST, CONST is defined as a macro with 100 as its replacement list. During the preprocessing stage, macro CONST will be replaced by 100. And when the undef directive is encountered, it causes CONST to be no longer defined as a macro name.

73. The value of CONST is 100

    Explanation:

    A preprocessor directive can appear anywhere within a program.

74. The value of CONST is 10

    Explanation:

    A macro can be redefined anywhere in the program. The most recent definition of the macro is considered while expanding the macro. If the redefinition of the macro is not identical, the compiler will issue a warning 'Redefinition of 'macroname' is not identical'.

75. Place code corresponding to version 1

    Explanation:

    The ifdef directive tests whether a name has been defined as a macro or not. Since VER has already been defined using the define directive, the printf statement that lies between #ifdef-#else will be compiled and later on executed.

76. Place code corresponding to version 1

    Explanation:

    #ifdef-#else-#endif is a condition compilation directive. The ifdef directive tests whether a name has been defined using the define directive or not. Since VER has already been defined using the define directive, the printf statement that lies between #ifdef-#else will be compiled and later on executed. The text that lies between #else-#endif will not be compiled. Hence, there will be no compilation error.

77. Place code corresponding to version 1

    Explanation:

    Since, the constant expression (i.e. VER==1) of the if directive evaluates to true, the statements that lie between #if-#else will be compiled and later on executed.

78. Compilation error "L-value required in function main"

    Explanation:

    The constant expression of the if directive is erroneous. A symbolic constant VER is placed on the left side of the assignment operator, and this leads to a compilation error.

79. Compilation error "Constant expression required in function main"

    Explanation:

    Only a constant expression can be used with the if directive. Since a is a variable, a==VER is not a constant expression and cannot be used with the if directive.

80. Place code corresponding to version 1

    Explanation:

    The const qualifier has been used to make a as a qualified constant. Hence, a==VER forms a constant expression and can be used with the if directive.

81. Fatal: Error directive: OPERATING SYSTEM IS NOT KNOWN in function main

    Explanation:

    The defined operator checks whether a given identifier has been defined as a macro or not. It evaluates to 1 if identifier has been defined. Since WINDOWS and LINUX have not been defined, the error directive produces the customized error OPERATING SYSTEM IS NOT KNOWN.

82. Compilation error

    Explanation:

    By default, nested multi-line comments are not allowed in C language.

83. THE C PREPROCESSOR

    Explanation:

    The #pragma option -C has been used to make the nested multi-line comments allowed.

84. This function setups the prerequisites of function main
    This is function main
    The requisite value of variable is 200

    Explanation:

    The #pragma startup is used to make the function func execute before the function main. The function func sets the value of global variable req_var_value to be 200. This value of global variable req_var_value is accessed inside the function main.

85. Size of int: 2
    Size of char: 1
    Size of float: 4
    Size of double: 8

    Explanation:

    The macro size_of implements the functionality of the sizeof operator.

Answers to Multiple-choice Questions

86. d 87. a 88. c 89. a 90. a 91. b 92. b 93. d 94. b 95. b

Programming Exercises

Program 1 | Define a macro to find the greatest of the two given numbers. Illustrate the use of this macro in a program

PE 10-1.c

```c
//Macro to find greatest of the two numbers
#include<stdio.h>
#define GREATEST(a,b) (a>b?a:b)
main()
{
int num1, num2;
printf("Enter two numbers:\t");
scanf("%d %d", &num1, &num2);
printf("The greatest of two numbers is %d",GREATEST(num1,num2));
}
```

Output window

```
Enter two numbers:      12 10
The greatest of two numbers is 12
```

Program 2 | Define a macro to check whether a given number is even or odd. Illustrate the use of this macro in a program

PE 10-2.c

```c
//Macro to check whether a given number is even or odd
#include<stdio.h>
#define EVENODD(a) ((a)%2==0?”even”:”odd”)
main()
{
int num;
printf(“Enter a number to be checked:\t”);
scanf(“%d”, &num);
printf(“%d is an %s number”, num, EVENODD(num));
}
```

Output window

```
Enter a number to be checked: 12
12 is an even number
```

Output window (second execution)

```
Enter a number to be checked: 5
5 is an odd number
```

Program 3 | Define a macro to find the harmonic mean of two numbers. Illustrate the use of this macro in a program

PE 10-3.c

```c
//Macro to find the harmonic mean of two numbers
#include<stdio.h>
#define HMEAN(a,b)  ((float)(2*(a)*(b))/((a)+(b)))
main()
{
int num1, num2;
printf(“Enter two numbers:\t”);
scanf(“%d %d”, &num1, &num2);
printf(“Harmonic mean of %d and %d is %f”,num1, num2, HMEAN(num1,num2));
}
```

Output window

```
Enter two numbers:     4 6
Harmonic mean of 4 and 6 is 4.800000
```

Program 4 | Define a macro to swap the contents of two variables. Illustrate the use of this macro in a program

PE 10-4.c

```c
//Macro to swap the contents of two variables
#include<stdio.h>
#define SWAP(a,b)  (a^=b^=a^=b)
main()
{
int num1, num2;
printf(“Enter two numbers:\t”);
scanf(“%d %d”, &num1, &num2);
printf(“Before swapping, the value of num1=%d and num2=%d\n”,num1,num2);
SWAP(num1, num2);
printf(“After swapping, the value of num1=%d and num2=%d”,num1,num2);
}
```

Output window

```
Enter two numbers:     4 6
Before swapping, the value of num1=4 and num2=6
After swapping, the value of num1=6 and num2=4
```

Program 5 | Define a nested macro to find the minimum of three integers. Illustrate the use of this macro in a program

PE 10-5.c

```
//Nested macro to find the minimum of three integers
#include<stdio.h>
#define MIN2(a,b) (a<b?a:b)
#define MIN3(a,b,c) (MIN2(a,b)<c?MIN2(a,b):c)
main()
{
int a, b, c;
printf(“Enter three numbers:\t”);
scanf(“%d %d %d”, &a, &b, &c);
printf(“Minimum of %d, %d and %d is %d”, a, b, c, MIN3(a,b,c));
}
```

Output window

```
Enter three numbers:     4 1 6
Minimum of 4, 1 and 6 is 1
```

Program 6 | Define a macro to check whether a given three-digit number is an Armstrong number or not. Illustrate the use of this macro in a program

PE 10-6.c

```
//Nested macro to check whether a given three digit number is an Armstrong number or not
#include<stdio.h>
#define POW3(x) ((x)*(x)*(x))
#define ARM(n) ((n==POW3(n%10)+POW3(n/10%10)+POW3(n/100%10) ) ? “is”:”is not”)
main()
{
int num;
printf(“Enter a three digit number:\t”);
scanf(“%d”, &num);
printf(“%d %s an Armstrong number”, num, ARM(num));
}
```

Output window

```
Enter a three digit number:     153
153 is an Armstrong number
```

Output window (second execution)

```
Enter a three digit number:     127
127 is not an Armstrong number
```

Test Yourself

1. Fill in the blanks in each of the following:

   a. A translator that converts a program written in a high-level language into an equivalent program in a machine-level language is known as ____________.

   b. The set of characters available when the source program file is executing is called ____________.

   c. The first two characters of a trigraph sequence are ____________.

   d. The input character sequence x+++++y is divided into the following stream of tokens ________________________.

   e. ____________ is a facility provided by a C preprocessor, by which a token can be replaced by the user-defined sequence of characters.

   f. Object-like macros are also known as ____________.

   g. The ____________ directive is used to configure some of the compiler options.

   h. The only directive that has no effect is ____________.

   i. ____________ is the smallest element of the language during the third to sixth phase of translation.

   j. The C tokenizer always tries to create ____________ possible token.

2. State whether each of the following is true or false. If false, explain why.

   a. During the preprocessing stage, each instance of backslash character immediately followed by a new-line character is deleted.

   b. The keywords are not preprocessing tokens. Thus, it is possible to use a keyword as an identifier name in a preprocessor directive, e.g. #define int char.

   c. The preprocessor directives are terminated with a semicolon.

   d. The preprocessor directives can only appear before the function main.

   e. The concatenation operator (i.e. ##) can appear at the beginning or at the end of the replacement list in a macro definition.

   f. ## is one token and there should be no white-space character between two ## characters.

   g. A predefined macro cannot appear immediately following a define directive.

   h. A predefined macro can be undefined using an undef directive.

   i. If the identifier specified with the undef directive is not currently defined as a macro, there will be a compilation error.

   j. The scope of a macro is the block in which it is defined.

   k. Macro replacement is carried out even within a string literal constant.

3. Programming exercises:

   a. Define a macro to check whether a given year is a leap year or not. Illustrate the use of this macro in a program.

   b. Define a macro to find the sum of digits of a three-digit number. Illustrate the use of this macro in a program.

   c. Define a macro to check whether a given three-digit number is perfect or not. Illustrate the use of this macro in a program.

   d. Define a macro that does not make use of a modulus operator to check whether a given number is even or not. Illustrate the use of this macro in a program.

   e. Define a macro to find the maximum of three integers. Illustrate its use.

   f. Define a macro that does not make use of a bitwise XOR operator to swap the contents of two variables. Illustrate its use.

Index