EXTRACT

Minutes of the 23rd Meeting of the Academic Council held on 28th July, 2023

Item No.15 – Noting of e-content and Self Learning Material developed by faculty members of CDOE.

Dr. Prashant Sahai Saxena, CDOE briefed the members about the e-content developed by the in-house faculty members for the programmes being offered in Online Mode and ODL Mode.

He further briefed that the CIQA Director has reviewed and approved revised Self Learning Material developed by the in-house faculty members for programmes as given below and as mentioned in the agenda, being offered in Open and Distance Learning (ODL) Mode from the Academic Year 2023-24

1. BBA
2. BCA
3. M.Sc. (Mathematics)
4. MBA
5. MCA

The Members including Director-CIQA reviewed the SLM for accuracy, completeness, relevance to the syllabus and the quality of the writing and presentation. After discussion and deliberations, the members approved the offering of above programmes in ODL Mode and the Self Learning Material (SLM)

For Vivekananda Global University, Jaipur
[illegible]
Registrar
(Dr. Praveen Choudhry)
Registrar

Date: 27-07-2023

Review Report of Self Learning Materials (SLM) for CDOE programmes

In pursuance of the ongoing commitment to ensuring the highest quality of Self Learning Materials (SLM) used in ODL programmes namely BBA, BCA, MCA, MBA and M.Sc. (Mathematics) proposed to be offered by CDOE-VGU, the Centre for Internal Quality Assurance (CIQA) established a dedicated review committee to conduct a comprehensive evaluation. We are pleased to announce that the review has been successfully completed, and we are proud to share the outcome with the members of academic council.

The committee was entrusted with the responsibility of reviewing the SLM for accuracy, completeness, and relevance to the syllabus, assessing the quality of writing and presentation, identifying areas for improvement, and making recommendations to enhance the quality of the SLM.

Director-CIQA pleased to inform you that the Review Committee has diligently carried out this task and has submitted their comprehensive report. The report reflects the hard work, dedication, and commitment of the committee members in ensuring the quality of our self-learning materials.

Director - CIQA, has approved the SLMs after careful consideration of the committee's input. The approval signifies that the SLMs meet the rigorous quality standards set by the CIQA for the benefit of our learners.

[illegible]

VIVEKANANDA GLOBAL UNIVERSITY CIQA JAIPUR

Dr. Devendra Kumar Doda
Director-CIQA

VGU/2023-24/CIQA/00304

Date: 03-07-2023

Office Order

Review Committee for Self-Learning Material (SLM) of CDOE

In order to ensure the quality of the Self Learning Material (SLM) used in CDOE-VGU, the Centre for Internal Quality Assurance (CIQA) has decided to conduct a comprehensive review of all SLM. The review will be conducted by a committee consisting of the following members:

- Prof. Vijay Vir Singh, President, Chairperson
- Prof. (Dr.) Baldev Singh, Director, CDOE, Member
- Mr. Sandeep Jain, CIQA-Coordinator, Member
- Dr. Monu Bhargava, Faculty of Management, Member
- Dr. Surendra Kumar Yadav, Faculty of Computer science & applications, Member
- Dr. Mridula Purohit, Faculty of Basic & applied Science, Member

The review committee will be responsible for the following tasks:

- Review the SLM for accuracy, completeness, and relevance to the syllabus.
- Assess the quality of the writing and presentation of the SLM.
- Identify any areas where the SLM can be improved.
- Make recommendations for improving the quality of the SLM.

The review committee will submit its report to the CIQA Director within one month of the date of this order. All departments are requested to cooperate with the review committee and to provide them with all the necessary assistance.

[illegible]

VIVEKANANDA GLOBAL UNIVERSITY CIQA JAIPUR

Dr. Devendra Kumar Doda

Director, CIQA

Copy to: President/CEO/ Director, FOM/ Director, CDOE/Director, FOCA/ Director, FOBA/Registrar/ All Deans & Associate Deans/ All HODs/ Office File

Self Learning Materials Development Policy

CDOE CENTRE FOR DISTANCE AND ONLINE EDUCATION
Vivekananda Global University

Vivekananda Global University
Jaipur (Rajasthan)

For Vivekananda Global University, Jaipur
[illegible]
Registrar

Self -Learning Materials Policy

I. PURPOSE

The purpose of formulating policies for self-learning materials is to establish a framework for developing self-learning materials tailored to the needs of learners enrolling in Vivekananda Global University's ODL Mode Programs. The guidelines have been formulated to ensure the quality and effectiveness of Self Learning Materials (SLMs) in accordance with the UGC-DEB Regulation-2020 Annexure-VII.

II. OBJECTIVES

The primary objectives of the self-learning materials policy are as follows:

(i) To meet the expectations of individuals residing within the geographical areas under Vivekananda Global University's jurisdiction.

(ii) To facilitate ongoing development of self-learning materials in line with evolving knowledge trends.

(iii) To provide a standardized format for subjects, ensuring maximum accessibility for learners.

III. COVERAGE

Vivekananda Global University's self-learning materials encompass concise yet comprehensive coverage of all subjects offered by the university across undergraduate, postgraduate and PhD levels. This includes regular courses, self-financing courses, and private education courses.

IV. EXPLANATION TO SELF LEARNING MATERIAL

Self-Learning Materials (SLM) possess inherent characteristics such as being self-explanatory, self-contained, self-directed, self-motivating, and self-evaluating. The development of these SLMs involves a meticulous planning process, often beginning with a learning needs assessment that considers learner backgrounds, experiences, and readiness. The adaptable nature of self-learning materials allows for revisions based on hidden talents or potential.

V. ESSENTIALS FOLLOWED TO DESIGN SELF-LEARNING MATERIALS

Self-learning materials are designed with specific characteristics that mimic the functions of an effective teacher. They serve as guides, motivators, explanations, discussion facilitators, question posers, progress assessors, remedial measure suggests, and advisors to learners. The following characteristics ensure the effectiveness of self-learning materials while fostering a sense of interaction with an invisible teacher:

Self-Contained

For Vivekananda Global University, Jaipur
Registrar

[illegible]

Efforts are made to ensure that the content is self-sufficient, eliminating the need for learners to seek additional sources or external guidance, including a teacher. The content within each unit is meticulously detailed, avoiding redundancy and presenting only essential information. This approach ensures that learners have access to all necessary information while avoiding superfluous or redundant details.

Self-Explanatory
Self-learning materials are presented in a manner that allows learners to comprehend the material with minimal external support. Concepts are explained comprehensively, making them accessible to the majority of learners. The content is both self-explanatory and conceptually clear, achieved through a careful analysis and logical presentation tailored to the mental and linguistic backgrounds of the target audience. While some learners may require additional support and guidance, the materials aim to make independent learning feasible.

Self-Directed
Self-learning materials provide essential guidance, hints, and suggestions at each stage of the learning process. They are designed to facilitate learning through easy-to-follow explanations, sequential development, illustrations, and interactive learning activities. In this way, they fulfill the role of a teacher, guiding, instructing, moderating, and regulating the learning process as if it were occurring in a classroom setting.

Self-Motivating
In distance education, learners spend a significant portion of their study time away from the physical campus. Therefore, study materials, much like a classroom teacher, should be highly motivating. Self-learning materials aim to stimulate curiosity, present challenges, relate knowledge to real-life situations, and make the learning experience meaningful. They provide reinforcement and feedback at every learning stage to keep learners engaged and motivated.

Self-Evaluating
Learners in distance education often remain physically separated from their educational institutions and teachers. To ensure optimal learning, self-learning materials include provisions for feedback. Learners need to assess whether they are on the right track. Self-evaluation tools, such as self-check questions, exercises, and activities, offer learners valuable feedback about their progress. These tools not only reinforce learning but also inspire motivation for self-directed learning. Course writers incorporate built-in evaluation systems by including an appropriate number of self-check exercises, activities, and questions within the course units.

Self-Learning
Self-instructional materials are rooted in the principles of self-learning. Each unit not only provides information but also serves as a study guide, offering directions, hints, and references to facilitate independent learning. The content is structured to be comprehensible, supported by simple explanations, examples, illustrations, and interactive activities. In essence, self-learning materials are designed to empower learners to undertake independent learning, with occasional assistance available

[illegible]

from external sources, including teachers when needed.

VI. PLANNING PROCESS FOR DEVELOPMENT OF SELF LEARNING MATERIAL

Step-1: Learners' need assessment

Step-2: Division and classification of tasks

Step-3: Analyzing the tasks with relevant personnel

Step-4: Assigning the task(s) according to mutual understanding.

Step-5: Preparation and submission of assignment

Step-6: Assessment of self- learning materials by CIQA and expert committee made by the University and decision forpreparation of final self-learning materials is taken by the authority

Step-7: Feed-back and revision

EXPLANATION OF STEPS

Learners' need assessment

Learners' assessment is done by considered level of literacy language proficiency, age group, information communication technology skills, aim of study, personal background and home situation, prior knowledge, prior skills, learning situations, etc.

Division and classification of tasks

All the tasks relating to development of self-learning materials are segregated on the basis of similarities and then are classified according to level of efforts required to make the materials with respect to the level of courses and forms of education.

Analysing the tasks with relevant personnel

The classified tasks are analysed by internal experts and sometimes the suggestions of external experts are taken like the industry practitioners, research consultants, and professionals. By this way the skeleton / structure of the self-learning materials for the concerned subjects are defined keeping in view the levels of courses and forms of educations offered by Vivekananda Global university.

For Vivekananda Global University, Jaipur
Registrar
[illegible]

Assigning the task(s) according to mutual understanding.

After obtaining the output of analyses in the form of structure or skeleton of the self-learning materials, the relevant faculty members are identified. They are briefed by the CIQA of the university and by the board of studiesof the concerned subjects for preparation of the self-learning material.

Preparation and submission of assignment

All three sources of information are used to prepare the self-learning materials like primary sources, secondary sources, and tertiary sources. Primary sources include the first-hand knowledge (by his/her own analytical capabilities) of faculty member(s), who is/are preparing the self-learning material. Secondary sources include the references to the books, journals, magazines, and the like. Tertiary sources include the logical analyses made by the consultants or experts of the industry. Time to time the CIQA of the university is guiding or making follow-up of the materialpreparation till submission.

Assessment of self-learning materials by CIQA of the University and decision for preparation of final self-learning materials is taken by the authority

After submission of the self-learning materials, the CIQA has the important role to play for the finalization of the materials. With subject experts, the CIQA is making a review of the self learning materials. If any changes or modifications are required, the faculty member, who is/are in the responsibility of the preparing the materials are asked to do so within a time frame. Finally, the full-fledged self-learning materials of the specified subjects are printed.

Feed-back and revision

Continuously the feed-backs are taken from the learners and from course instructors in some cases. These feed-backs are analysed by the committee of experts under the supervision of CIQA of the university, so that either the plan will be modified or the structure of the self-learning materials will be modified or the content presentation will be modified for the subsequent academic years.

VII. PRINCIPLES TO BE FOLLOWED TO PREPARE AND EXECUTE THE SELF LEARNING MATERIALS

In addition to the fundamental properties of self-learning materials, the preparation and execution adhere to the following principles-

(i) SLM content should be engaging and maintaining learner attention

(ii) Incorporating previously acquired knowledge

(iii) Providing guidance and hints for independent learning

(iv) Facilitating feedback from learners and instructors

For Vivekananda Global University, Jaipur
Registrar
[illegible]

(v) Establishing appropriate conditions for learning

(vi) Encouraging curiosity through relevant questions

(vii) Structured assessment questions/assignments

(viii) Inclusion of nonverbal aids (e.g., pictures, maps)

(ix) Use of glossaries for better comprehension

(x) Summing up at the end of each unit and the material

(xi) Creating a virtual teacher-student interaction

(xii) Using simple and clear language, avoiding unnecessary jargon

For Vivekananda Global University, Jaipur

Registrar

[illegible]

VIVEKANANDA GLOBAL UNIVERSITY, JAIPUR

(Established by Act 11/2012 of Rajasthan Govt. Covered u/s 2(f) of UGC Act, 1956)

Master of Computer Application

(MCA)

Fundamental of Computer and Programming in C

SEMESTER I

Author: Dr. Pawan Bhambu

Approval Director CIQA: 27th July 2023

Approval of Academic Council: 28th July 2023

[illegible]

For Vivekananda Global University

Registrar

Table of Content

Learning Objectives

After studying this unit, the student will be able to:

- Understand the basic concepts and terminologies related to computers and computing.
- Describe the fundamental components of a computer system and their respective functions.
- Identify different types of computers and their specific applications in various domains.
- Comprehend the characteristics and limitations of computer hardware and software.
- Explore the history and evolution of computer generations and their impact on modern technology.
- Explain the principles of data representation, storage, and retrieval in computer systems.
- Gain insights into computer networks, their importance, and the concept of the internet.
- Develop basic troubleshooting skills to diagnose and solve common computer-related issues.
- Demonstrate proficiency in using operating systems, file management, and software applications.
- Apply knowledge of computer fundamentals in real-world scenarios, including personal and professional computing tasks.

For Vivekananda Global University, Jaipur

[illegible]

Registrar

Introduction to Computer

Computers are electronic devices that process and store data, perform calculations, and execute tasks based on instructions given to them. They are integral to modern life and have become essential tools for various applications, from personal use to complex scientific research and industrial processes.

Components of a Computer:

- Central Processing Unit (CPU): Often referred to as the computer's " brain, " the CPU executes instructions and performs calculations. It carries out the core processing tasks and interacts with other components.
- Memory: Computers have two main types of memory:
    - Random Access Memory (RAM): This is a temporary memory used to hold data and programs that the CPU is currently using. When you turn off the computer, the data in RAM is erased.
    - Storage (Hard Drive, SSD, etc.): This is the long-term memory where files and programs are stored even when the computer is powered off.
- Input Devices allow users to interact with the computer and provide data or instructions. Common input devices include keyboards, mice, touchscreens, and microphones.
- Output Devices: These display or present the results of the computer's processing. Common output devices include monitors, printers, speakers, and headphones.
- Motherboard: The main circuit board that connects and allows communication between various components of the computer.
- Software: This refers to the programs and applications that run on the computer. Software enables users to perform tasks, such as browsing the internet, creating documents, or playing games.
- Operating System (OS): The software that manages the computer's resources and provides a user interface. It acts as an intermediary between the user, applications, and hardware.

How Computers Work:

- Binary Language: Computers communicate using binary code, which consists of only two digits, 0 and 1. Each 0 or 1 is called a "bit." Eight bits make up a "byte," which is a computer's basic unit of storage.
- Machine Language and Instructions: The CPU understands and executes instructions written in machine language, a series of binary codes representing specific operations.
- Fetch-Decode-Execute Cycle: The CPU follows a fundamental process called the Fetch-Decode-Execute cycle. It fetches instructions from memory, decodes them to understand the operation, and executes them.
- Programming: Humans use programming languages to write instructions in a more human-readable form, which are then translated into machine code by software called compilers or interpreters.

Types of Computers:

- Personal Computers (PCs): Designed for individual use, PCs are common in homes, offices, and schools. They come in desktop and laptop form factors.
- Servers: These are computers designed to provide services and resources to other computers on a network. They are commonly used for websites, databases, and file storage.
- Mainframes: Large and powerful computers organizations use to process vast amounts of data and run critical applications.
- Supercomputers: The fastest and most powerful computers for complex scientific simulations, weather forecasting, and other data-intensive tasks.
- Embedded Systems: Computers integrated into other devices or systems to perform specific functions, like in cars, household appliances, and industrial machinery.

For Vivekananda Global University, Jaipur

[illegible]

Registrar

1.1 Characteristics of Computers

Computers possess various characteristics and capabilities that make them powerful tools but also have certain limitations. Let's explore some of the key characteristics and limitations of computers:

Characteristics of Computers:

- Speed: Computers can process vast amounts of data and execute instructions at incredible speeds. They can perform complex calculations and tasks in fractions of a second, making them highly efficient for various applications.
- Accuracy: When programmed correctly, computers are exceptionally accurate and can perform repetitive tasks without errors, reducing human mistakes and ensuring consistent results.
- Storage Capacity: Modern computers can store enormous amounts of data on hard drives, solid-state drives (SSDs), or cloud-based storage solutions. This capacity allows for the retention of vast databases, software, multimedia, and other types of information.
- Versatility: Computers can be programmed to perform a wide range of tasks. From word processing and web browsing to complex simulations and artificial intelligence, their versatility allows them to serve various needs.
- Automation: Computers can automate repetitive tasks, reducing the need for human intervention in certain processes, leading to increased efficiency and productivity.
- Connectivity: Computers can connect to networks, allowing users to share information, access remote resources, and communicate with others worldwide.
- Multitasking: Operating systems enable computers to perform multiple tasks simultaneously, such as running several applications or processes concurrently.

1.2 Limitations of Computers

- Lack of Creativity and Intuition: Computers can only execute tasks based on pre-defined instructions and algorithms. They lack human-like creativity and intuition, making them incapable of understanding context or making decisions beyond their programmed capabilities.
- Dependency on Humans: Despite their capabilities, computers rely on humans to provide instructions and program them for specific tasks. They cannot function autonomously without human intervention.
- Vulnerability to Errors: Computers are precise, but they are also prone to errors caused by hardware malfunctions, software bugs, or incorrect input from users.
- Processing Limitations: While computers are fast, certain complex calculations and simulations may still require significant time and processing power, especially for supercomputing tasks.
- Security Risks: Computers are vulnerable to security threats like viruses, malware, and hacking. Maintaining robust cybersecurity measures is essential to protect sensitive data and systems.
- Environmental Impact: The rapid advancement of technology and the widespread use of computers contribute to electronic waste and energy consumption concerns.
- Lack of Common Sense: Computers lack human-like common sense and understanding of natural language. They interpret data literally, leading to challenges in understanding context or making inferences.

1.3 Block Diagram of Computer

A block diagram of a computer is a visual representation that shows the major components and their interconnections within the computer system. It provides a high-level overview of how different parts of the computer work together to process information and execute tasks.

[illegible]

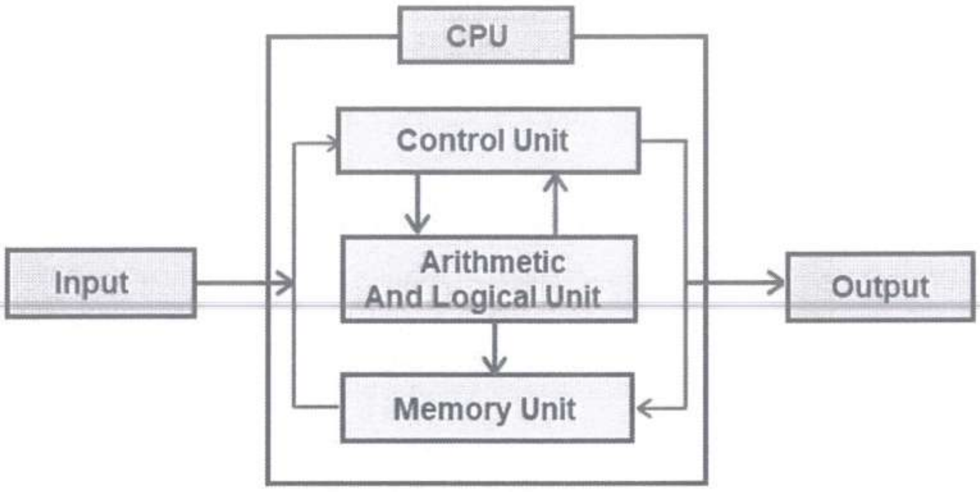


Fig – Block Diagram of Computer

Let's go through the main components typically found in a computer's block diagram:

- Input Devices: These devices allow users to input data and instructions into the computer. Common input devices include the keyboard, mouse, touchpad, touchscreen, microphone, and various sensors.
- Central Processing Unit (CPU): Often referred to as the computer's " brain, " the CPU executes instructions and performs calculations. It consists of three main components:
- Control Unit (CU): It manages the flow of data and instructions within the CPU and coordinates the operations of other components.
- Arithmetic Logic Unit (ALU): This component performs arithmetic operations (e.g., addition, subtraction) and logical operations (e.g., AND, OR) on data.
- Registers: These are small, high-speed storage units within the CPU that temporarily hold data and instructions during processing.
- Memory: Computers have multiple types of memory:
    - Random Access Memory (RAM): This is the main memory used to hold data and programs that the CPU is currently using. It provides fast access but is volatile, meaning its contents are lost when the computer is turned off.

- Read-Only Memory (ROM): This non-volatile memory stores essential instructions for the computer to start up, like the BIOS (Basic Input/Output System).
- Cache Memory: This small, high-speed memory stores frequently accessed data and instructions to speed up CPU operations.

- Storage Devices: These are used to store data and programs even when the computer is powered off. Common storage devices include Hard Disk Drives (HDDs), Solid-State Drives (SSDs), optical drives (CD/DVD), and USB flash drives.
- Output Devices: These devices display or present the results of the computer's processing. Common output devices include monitors, printers, speakers, and headphones.
- Motherboard: The main circuit board connects and allows communication between various computer components. It contains slots, connectors, and buses to facilitate data transfer.
- Expansion Cards: These optional cards can be added to the motherboard to provide additional functionality, such as graphics cards, sound cards, or network interface cards.
- Bus: The bus is a communication channel that transfers data between computer components. Buses can be classified into several types, including the system bus, data bus, and address bus.
- Power Supply Unit (PSU): The PSU provides electrical power to the computer components and ensures they receive the appropriate voltage and current.

1.4 Types of Computers

Computers come in various types and forms, each designed to serve specific purposes and cater to different user needs. Here are some of the different types of computers:

1. Personal Computers (PCs): These are general-purpose computers designed for individual use. PCs come in two primary form factors:

a. Desktop Computers: These are stationary computers with a separate monitor, keyboard, mouse, and the main processing unit (CPU) housed in a tower or compact form.

b. Laptop Computers: Also known as notebooks, laptops are portable computers with an integrated screen and keyboard. They are convenient for users who need mobility.

2. Workstations: Workstations are powerful computers designed for specialized tasks that require high-performance hardware, such as engineering, 3D modeling, video editing, and scientific simulations.
3. Servers: Servers are computers designed to provide services and resources to other computers on a network. They handle data storage, web hosting, email management, and networking.
4. Mainframes: Mainframes are large and robust computers organizations use for handling massive amounts of data processing and critical business applications. They are known for their high reliability, availability, and scalability.
5. Supercomputers: Supercomputers are the fastest and most powerful computers in the world. They are used for complex scientific simulations, weather forecasting, advanced research, and solving intricate computational problems.
6. Embedded Systems: Embedded computers are integrated into other devices or systems to perform specific functions. They are found in everyday objects, such as cars, household appliances, medical devices, industrial machinery, and IoT (Internet of Things) devices.
7. Tablets: Tablets are portable computers that feature touchscreens and are optimized for ease of use and mobility. They are commonly used for browsing, media consumption, and light productivity tasks.
8. Smartphones: Smartphones are handheld mobile devices that function as powerful computers with communication capabilities, running various applications and connecting to the internet.

For Vivekananda Global University, Jaipur

[illegible]

Registrar

9. Gaming Consoles: Gaming consoles are specialized computers designed for playing video games. They typically connect to televisions or monitors and come with gaming controllers.
10. Raspberry Pi and Single-Board Computers are credit card-sized computers designed for educational and hobbyist purposes. They can be used for programming, DIY projects, and as low-cost, compact computing solutions.

Each type of computer has its strengths and weaknesses, and the choice of which one to use depends on the specific requirements and intended applications. As technology evolves, new types of computers may emerge to cater to emerging needs and trends.

Applications/uses of computers

Computers have countless applications and use across various domains due to their versatility and ability to process and store vast amounts of data. Here are some of the primary applications of computers:

1. Personal Use:
    - Web Browsing: Accessing information and services on the internet.
    - Email: Sending and receiving electronic messages.
    - Word Processing: Creating and editing documents.
    - Multimedia: Viewing and editing photos, videos, and music.
    - Social Media: Connecting and interacting with others online.
    - Entertainment: Playing games, watching movies, and streaming content.
2. Education:
    - Online Learning: Accessing educational resources and courses.
    - Interactive Learning: Using educational software and simulations to enhance learning.
    - Research: Conducting research and accessing vast digital libraries.
3. Business and Productivity:
    - Data Management: Storing, organizing, and analyzing business data.
    - Communication: Facilitating internal and external communication through email, video conferencing, etc.

- Accounting and Finance: Managing financial records, transactions, and payroll.
- Customer Relationship Management (CRM): Tracking and managing customer interactions.
- Project Management: Planning and coordinating tasks and resources.
- Marketing and Advertising: Creating and managing marketing campaigns.
- E-commerce: Selling and purchasing products and services online.

4. Science and Research:
   - Simulation and Modeling: Simulating complex scientific phenomena and experiments.
   - Data Analysis: Analyzing large datasets and conducting statistical studies.
   - Genome Sequencing: Analyzing genetic information for medical research.
   - Weather Forecasting: Predicting weather patterns using advanced simulations.
5. Medicine and Healthcare:
   - Electronic Health Records (EHR): Storing and managing patient medical information.
   - Medical Imaging: Analyzing X-rays, CT scans, MRIs, and other medical images.
   - Drug Discovery: Using computers for virtual screening and drug development.
   - Medical Research: Analyzing biological data for disease research.
6. Engineering and Design:
   - Computer-Aided Design (CAD): Creating detailed engineering and architectural designs.
   - Simulation and Prototyping: Simulating and testing product designs before physical production.
   - 3D Printing: Utilizing computer models to create physical objects.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

7. Entertainment and Media:
   - Video Games: Creating and playing interactive digital games.
   - Animation and Visual Effects: Producing animated films and visual effects for movies and TV.
8. Communication and Social Networking:
   - Social Media Platforms: Connecting with friends and sharing content.
   - Instant Messaging: Communicating with others in real-time.
9. Government and Public Services:
   - Public Administration: Managing government functions and services.
   - E-Government: Providing online services and information to citizens.
10. Security and Defense:
    - Cybersecurity: Protecting computer systems and data from threats.
    - Military Applications: Using computers for defense, surveillance, and simulations.

1.5 Computer Generations

Computers have evolved over time through different generations, each marked by significant technological advancements and changes in design and architecture. These generations are often categorized based on the underlying hardware and the key developments that distinguish one Generation from another. Let's explore the main computer generations:

1.6 First Generation Computers

The First Generation of computers refers to the initial period of electronic computing, characterized by the use of vacuum tubes as the primary electronic component. This Generation spans the 1940s and 1950s and is marked by groundbreaking developments in computer technology.

Key characteristics of the First Generation of computers:

- Vacuum Tubes: Vacuum tubes were used as the main switching and amplification devices in these early computers. These glass tubes were large, fragile, and consumed a significant amount of power. They were used for tasks such as computation, memory storage, and input/output operations.

- Size and Scale: First-generation computers were massive machines that filled entire rooms. Their size and complexity made them extremely expensive to build and maintain.

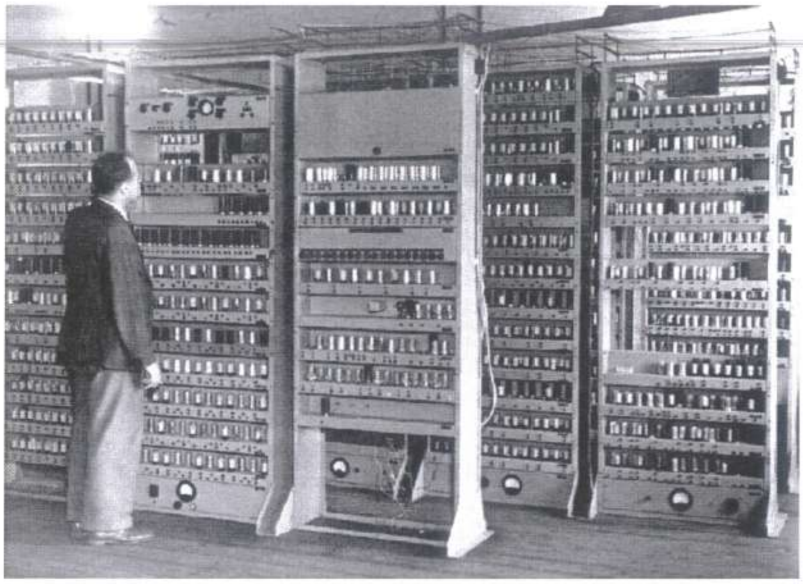


Fig -A First Generation of computers

- Limited Programmability: Programming these computers was a challenging and time-consuming task. Programmers had to manually rewire circuits to change the instructions, making the process highly labor-intensive.
- Batch Processing: First-generation computers primarily used batch processing, where a sequence of jobs was fed into the computer as a batch. Each job would be processed individually without user interaction.
- Magnetic Drum Memory: Early forms of memory included magnetic drum memory, which used a rotating drum coated with magnetic material to store data temporarily.
- Examples of First-Generation Computers:
    - ENIAC (Electronic Numerical Integrator and Computer): Completed in 1945, ENIAC was one of the first electronic general-purpose

For Vivekananda Global University, Jaipur
[illegible]
Registrar

computers. It was designed to calculate artillery firing tables for the United States Army during World War II.

    - UNIVAC I (Universal Automatic Computer I): Developed in the early 1950s, UNIVAC I was the first commercial computer in the United States. It was used for business and scientific applications.

- Limited Applications: First-generation computers were primarily used for scientific calculations, military applications, and specific research tasks.
- Heat and Maintenance: Vacuum tubes generated a considerable amount of heat, leading to frequent failures and the need for constant maintenance.

Fig – Vacuum Tube

Advantages:

- Pioneering Technology: First-generation computers laid the foundation for modern computing and set the stage for future advancements.
- Initial Computations: They were used for essential tasks like calculations for military and scientific purposes.

Disadvantages:

- Size and Cost: First-generation computers were massive and expensive to build and maintain.
- Limited Programmability: Programming involved manual rewiring, making it time-consuming and challenging.
- Reliability Issues: Vacuum tubes were prone to failures, leading to frequent downtime.

1.7 Second Generation Computers

The Second Generation of computers refers to the era of computing from the late 1950s to the mid-1960s. This Generation witnessed significant advancements over the first Generation, primarily due to the replacement of vacuum tubes with transistors as the primary electronic component.

Key characteristics of the Second Generation of computers:

- Transistors: Transistors replaced vacuum tubes as computers' main electronic switching and amplification devices. Transistors were much smaller, more reliable, and consumed less power than vacuum tubes, resulting in more efficient and compact computer designs.
- Smaller Size: Using transistors allowed computers to become smaller and more affordable. This made fitting multiple transistors on a single semiconductor chip, known as a transistor-transistor logic (TTL) chip possible.
- Magnetic Core Memory: Second-generation computers used magnetic core memory as the primary form of RAM. Magnetic cores were small rings of magnetic material threaded with wires, and their magnetization represented the 0s and 1s of binary data.
- Assembly Language and High-Level Programming: Assembler languages and early high-level programming languages (e.g., FORTRAN, COBOL) were developed during this period, making programming more accessible and less time-consuming than the manual rewiring required in the first Generation.
- Batch Processing: Like the first Generation, second-generation computers primarily used batch processing to execute programs. Jobs were submitted as a batch and processed sequentially without user interaction.
- Improved Speed and Reliability: Transistors were faster and more reliable than vacuum tubes, resulting in improved performance and reduced downtime

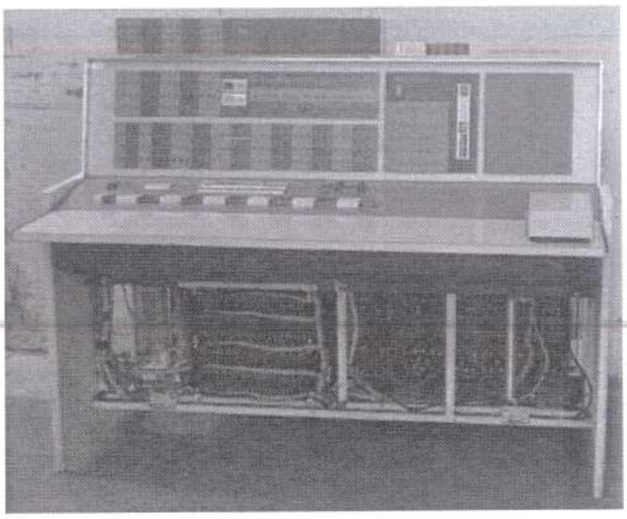


Fig – Second Generation Computer

- Examples of Second-Generation Computers
  - IBM 7090: Introduced in 1959, the IBM 7090 was a significant second-generation computer used for scientific and engineering calculations.
  - UNIVAC II: The successor to the UNIVAC I, UNIVAC II was another commercial computer that gained popularity during this Generation.
- Limited Commercialization: Although second-generation computers were smaller and more practical than their predecessors, they were still costly and primarily used in large organizations, research institutions, and government agencies.

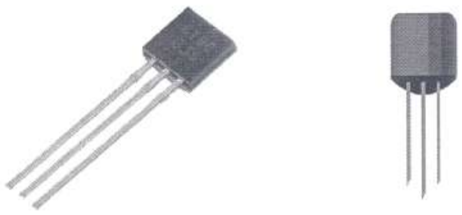


Fig – Transistors

Advantages:

[illegible]

For Vivekananda Global University, Jaipur

Registrar

- Transistors: Transistors were more reliable, smaller, and energy-efficient compared to vacuum tubes.
- Faster and More Compact: Second-generation computers were faster and more compact than their predecessors.
- High-Level Languages: The use of high-level programming languages made programming more accessible.

Disadvantages:

- Still Expensive: Despite advancements, second-generation computers were costly and not widely affordable.
- Limited Memory and Storage: Memory and storage capacities were still relatively limited.

1.8 Third Generation Computers

The Third Generation of computers refers to the computing period from the mid-1960s to the mid-1970s. This Generation saw a significant leap in computer technology with the introduction of integrated circuits (ICs), which replaced individual transistors and other components on a single chip.

Fig – Third Generation Computer

Key characteristics of the Third Generation of computers:

- Integrated Circuits (ICs): The most notable advancement of this Generation was the use of integrated circuits, also known as microchips. Integrated

[illegible]

circuits combined multiple transistors, resistors, capacitors, and diodes onto a single semiconductor chip. This integration led to smaller, faster, and more reliable computers.

- Small Scale Integration (SSI) and Medium Scale Integration (MSI): During the early third Generation, computers used small-scale integration (SSI) with a few transistors on a chip. Later in the Generation, medium-scale integration (MSI) allowed for more complex circuits with hundreds of transistors on a chip.
- Mainframes and Minicomputers: The third Generation saw the rise of mainframe computers and minicomputers. Mainframes were large and powerful computers used by large organizations for heavy data processing and business applications. Minicomputers were smaller and more affordable than mainframes, making them accessible to smaller businesses and institutions.
- High-Level Languages: Third-generation computers continued to support high-level programming languages like FORTRAN, COBOL, and ALGOL. These languages made programming more user-friendly and allowed programmers to write code using English-like syntax.
- Operating Systems: The development of operating systems became more crucial in this Generation to manage computer resources efficiently and allow multiple users to interact with the computer simultaneously.
- Timesharing: Third-generation computers introduced timesharing systems, which allowed multiple users to interact with the same computer concurrently. Each user had dedicated processor time, enabling efficient utilization of computing resources.
- Magnetic Core and Semiconductor RAM: While magnetic core memory continued to be used, semiconductor-based RAM (Random Access Memory) started to gain prominence due to its lower cost and higher capacity.
- Examples of Third-Generation Computers:

[illegible]

    - IBM System/360: Introduced in 1964, the IBM System/360 was a family of mainframe computers that offered a wide range of models with varying processing power, catering to different needs.
    - DEC PDP-8: A popular minicomputer introduced in 1965, known for its affordability and versatility.
- Commercialization and Widespread Use: The third Generation witnessed a broader commercialization of computers, making them more accessible to businesses and institutions, leading to increased adoption and widespread use.

Fig - Integrated Circuit (IC)

Advantages:

- Integrated Circuits: The use of integrated circuits improved performance, reliability, and reduced size.
- Timesharing: Timesharing allowed multiple users to share computer resources simultaneously.
- Commercialization: Third-generation computers saw broader commercialization and widespread use.

Disadvantages:

- Limited Interactive Capabilities: User interfaces were less interactive compared to modern computers.
- Still Relatively Expensive: While more accessible than earlier generations, they were not yet fully affordable for individuals.

For Vivekananda Global University, Jaipur
[illegible]
Registrar

1.9 The Fourth-Generation Computers

The Fourth Generation of computers spans the mid-1970s to the mid-1990s. This Generation of computing is characterized by the development and widespread use of microprocessors, which brought about significant improvements in performance, miniaturization, and energy efficiency.

Fig - Fourth Generation of computers

Key characteristics of the Fourth Generation of computers:

- Microprocessors: The most defining feature of the fourth Generation is the use of microprocessors. Microprocessors are central processing units (CPUs) integrated onto a single chip. This advancement revolutionized computing by enabling a complete computer system on a much smaller scale.
- VLSI Technology: Very Large-Scale Integration (VLSI) technology allows millions of transistors to be placed on a single chip, leading to higher processing power, reduced size, and lower production costs.
- Personal Computers (PCs): The introduction of microprocessors paved the way for developing personal computers (PCs). PCs became more affordable and accessible to individuals, leading to a computing revolution.
- Graphical User Interface (GUI): The fourth Generation saw the development of graphical user interfaces (GUIs), which provided a more user-friendly way

to interact with computers. GUIs featured icons, windows, and a mouse for navigation.

- Operating Systems Advancements: Operating systems became more sophisticated during this Generation, offering multitasking capabilities, better memory management, and improved file systems.
- Storage Advancements: Fourth-generation computers witnessed advancements in storage technology, including the use of hard disk drives (HDDs) for mass storage and floppy disks for portable data storage.
- Networking: Local Area Networks (LANs) and Wide Area Networks (WANs) became more prevalent, allowing computers to communicate and share resources over network connections.
- Software Development: The software industry flourished during this Generation, with various applications developed for business, entertainment, and productivity.
- Examples of Fourth-Generation Computers:
  - IBM PC: Introduced in 1981, the IBM Personal Computer (IBM PC) became a standard for business and personal use.
  - Apple Macintosh: Released in 1984, the Macintosh was one of the first computers to popularize the graphical user interface.
- Integration of Home Computing: Fourth-generation computers brought computing into homes, enabling people to perform various tasks, such as word processing, gaming, and multimedia activities
- High-Performance Computing: Fourth-generation computers also saw the development of high-performance computing systems used in scientific research, simulations, and complex computations.
- End of Vacuum Tubes and Punch Cards: The use of vacuum tubes and punch cards, prominent in earlier generations, became obsolete during the fourth Generation.

Advantages:

For Vivekananda Global University, Jaipur
[illegible]
Registrar

- Microprocessors: Microprocessors revolutionized computing, enabling personal computers and greater accessibility.
- Graphical User Interface (GUI): GUIs made computers more user-friendly and intuitive.
- Faster and More Powerful: Fourth-generation computers exhibited significant performance improvements.

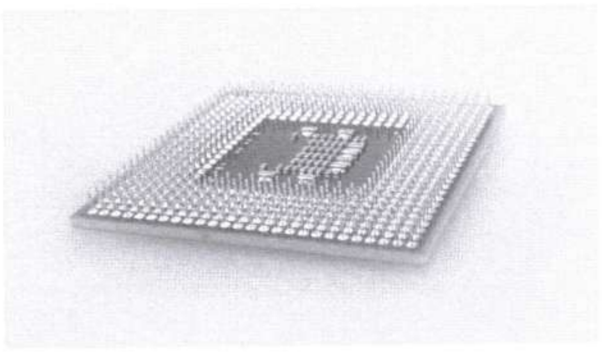


Fig – Microprocessor

Disadvantages:

- Limited Connectivity: The early networking and internet connectivity stages were not as widespread.
- Compatibility Issues: Compatibility between hardware and software posed challenges.

1.10 The Fifth-Generation Computers

The Fifth Generation of computers refers to the current era of computing, starting from the mid-1990s and continuing to the present day. This Generation is characterized by advancements in artificial intelligence (AI), parallel processing, and quantum computing. The Fifth Generation of computers aims to create machines capable of mimicking human-like intelligence and processing vast amounts of data at incredible speeds.

Key characteristics of the Fifth Generation of computers

- Artificial Intelligence (AI): AI is at the forefront of the fifth Generation, focusing on developing computer systems that can learn, reason, and make decisions like humans. Machine learning, deep learning, and neural networks are essential for AI research and application.
- Natural Language Processing (NLP): NLP allows computers to understand, interpret, and respond to human language. Virtual assistants like Siri, Alexa, and Google Assistant are examples of NLP applications.

Fig - Fifth Generation of computers

- Parallel Processing and Supercomputing: The fifth Generation continues to explore parallel processing techniques, where multiple processors work together to solve complex problems simultaneously. Supercomputers use parallel processing to handle extensive scientific simulations, weather forecasting, and other data-intensive tasks.
- Quantum Computing: Quantum computing is an emerging technology that harnesses the principles of quantum mechanics to perform calculations at an unprecedented speed. Quantum computers can potentially revolutionize cryptography, optimization, and drug discovery.
- Big Data: The fifth Generation of computers deals with processing and analyzing massive volumes of data, commonly known as big data. This data

For Vivekananda Global University, Jaipur
[illegible]
Registrar

often comes from diverse sources and requires advanced algorithms to extract meaningful insights.

- Cloud Computing: Cloud computing has become a critical aspect of the fifth Generation, offering on-demand access to computing resources, storage, and applications over the internet.
- Internet of Things (IoT): IoT devices, such as smart home devices and wearable technology, have become prevalent in the fifth Generation. These devices are interconnected and collect data for analysis and automation.
- Advancements in Hardware: Advances in semiconductor technology, such as nanotechnology, have created faster and more energy-efficient processors.
- Examples of Fifth-Generation Technologies:
    - IBM Watson: An AI-powered computer system capable of answering questions and providing insights by analyzing vast amounts of unstructured data.
    - Google's Quantum Computer: Companies like Google are developing quantum computers to explore the potential of quantum computing.
- Ethical and Societal Considerations: The fifth Generation raises ethical concerns surrounding AI and the responsible use of AI systems, as well as privacy and data security issues related to collecting and analyzing vast amounts of data.

Advantages:

- AI and Machine Learning: The development of AI and machine learning brought automation and advanced problem-solving capabilities.
- Quantum Computing: Research in quantum computing holds the promise of solving complex problems exponentially faster.
- Advanced Connectivity: The fifth Generation witnessed the rise of the internet and global connectivity.

Disadvantages:

- Ethical and Privacy Concerns: AI raises ethical questions regarding its impact on society, privacy, and job displacement.
- Complexity: Quantum computing is still in its infancy, and practical implementations remain challenging.

Fig – Artificial Intelligence Technology

The Fifth Generation of computers continues to push the boundaries of what computers can achieve. Researchers and developers are exploring AI, quantum computing, and other cutting-edge technologies to address complex challenges and create innovative solutions for various industries and fields.

However, looking ahead to potential advancements beyond the Fifth Generation, some researchers and experts have discussed various futuristic possibilities for computing technology. While there is no formal consensus on the exact features of a Sixth Generation, here are some theoretical advancements that could be considered as part of a potential Sixth Generation of computers:

- Advancements in Quantum Computing: Quantum computing may see further breakthroughs, enabling more stable and scalable quantum processors, increased qubit counts, and expanded practical applications in cryptography, optimization, and simulations.

For Vivekananda Global University, Jaipur

[illegible]

Registrar

- Neuromorphic Computing: Neuromorphic computing seeks to mimic the human brain's neural architecture, leading to highly efficient and brain-like processors for AI and pattern recognition tasks.
- Brain-Computer Interfaces (BCIs): BCIs could see significant progress, allowing direct communication between the human brain and computers, and enabling new paradigms for human-computer interaction.
- Biological and DNA Computing: Research into biological computing and DNA-based storage could result in novel biochemistry-based computing systems for data processing and storage.
- Hyperconnected Devices: Advancements in the Internet of Things (IoT) may lead to an interconnected network of devices capable of seamless communication and cooperation, facilitating smart cities and smart environments.
- Beyond Silicon: Exploration of alternative materials and technologies, such as graphene, spintronics, or optical computing, could usher in a new era of faster and more energy-efficient computing components.
- AI and Autonomy: AI systems might evolve to achieve higher levels of autonomy, enabling more complex decision-making and learning capabilities.

1.11 Summary

Computer Types:

- Personal Computers (PCs): These are general-purpose computers designed for individual use, commonly found in homes and offices.
- Laptops: Portable PCs with integrated components, including a keyboard, screen, and battery.
- Servers: High-performance computers designed to manage network resources and serve data to multiple clients.
- Mainframes: Powerful computers used for critical applications, large-scale data processing, and transaction handling.

- Supercomputers: High-performance machines designed for complex scientific computations and simulations.

Characteristics:

- Processing Power: Computers vary in processing capabilities, from basic tasks to complex calculations.
- Memory (RAM): Determines how much data the computer can access quickly, affecting performance.
- Storage (Hard Drive/SSD): Refers to the capacity for data storage, from a few gigabytes to several terabytes.
- Connectivity: Computers can connect to networks and the internet to access and share information.
- Operating System: Software that manages computer resources and allows users to interact with the machine.

Limitations:

- Processing Speed: Computers have limitations on how fast they can process data, which can affect performance.
- Storage Capacity: There is a finite limit to how much data a computer can store, although this limit has increased significantly over time.
- Physical Size: Larger computers, like mainframes and supercomputers, require specialized facilities due to their size and power requirements.
- Power Consumption: High-performance computers consume a lot of power, leading to high operating costs.
- Security Vulnerabilities: Computers are susceptible to various forms of cyber threats, requiring robust security measures.

Components:

- Central Processing Unit (CPU): The brain of the computer, responsible for executing instructions and performing calculations.
- Motherboard: The main circuit board that connects all components, allowing them to communicate with each other.

- Memory (RAM): Temporary storage used by the CPU to access data quickly during processing.
- Storage Devices: Hard and solid-state drives (SSDs) are used for long-term data storage.
- Graphics Processing Unit (GPU): Specialized processor for handling graphics-related tasks, crucial for gaming and multimedia applications.
- Input Devices: Keyboards, mice, touchscreens, etc., used to input data and commands.
- Output Devices: Monitors, printers, speakers, etc., used to display or produce the computer's results.
- Power Supply Unit (PSU): Supplies power to all components of the computer.
- Cooling System: Helps dissipate heat generated by the computer's components to prevent overheating.
- Networking Components: Network cards or adapters for connecting to wired or wireless networks.

Summary of Generations

1. First Generation (1940s-1950s):
   - Vacuum Tubes: The first computers used vacuum tubes as the primary electronic component for processing and memory.
   - ENIAC: The Electronic Numerical Integrator and Computer (ENIAC) was one of the earliest general-purpose electronic computers and was completed in 1945.
   - Limited Programmability: Programming these computers involved manually rewiring circuits, making them labor-intensive and challenging to program.
2. Second Generation (1950s-1960s):
   - Transistors: Vacuum tubes were replaced by transistors, which were smaller, more reliable, and consumed less power.

   - High-Level Programming Languages: Assembly languages and early high-level programming languages (e.g., FORTRAN, COBOL) were developed, making programming more accessible.
   - Batch Processing: Computers started using batch processing to process multiple jobs simultaneously.
3. Third Generation (1960s-1970s):
   - Integrated Circuits (ICs) were introduced, packing multiple transistors onto a single chip. This made computers smaller, more powerful, and more energy-efficient.
   - Operating Systems: The development of operating systems allowed for better resource management and multi-programming.
   - Timesharing: Multiple users could interact with the computer simultaneously through timesharing systems.
4. Fourth Generation (1970s-1980s):
   - Microprocessors: The invention of microprocessors led to the development of smaller, cheaper, and more powerful computers. Microprocessors combine the CPU and other components on a single chip.
   - Personal Computers (PCs): The emergence of PCs revolutionized computing, making computers accessible to individuals and businesses.
   - Graphical User Interface (GUI): GUIs made computers more user-friendly, with icons, windows, and a mouse for navigation.
5. Fifth Generation (1980s-Present):
   - VLSI Technology: Very Large-Scale Integration (VLSI) enabled even more components to be packed onto a single chip, further increasing computing power and efficiency.
   - Parallel Processing: Computers began using parallel processing, with multiple processors working together to perform tasks faster.
   - Artificial Intelligence (AI): This Generation saw significant advancements in AI, including expert systems, natural language processing, and machine learning.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

1.12 Review Questions

1 What are the different types of computers, and how do they differ in terms of usage?

2 Explain the main characteristics that define a computer's performance and capabilities.

3 What are the limitations of modern-day computers, and how might these limitations be addressed in the future?

4 Describe the essential components that make up a standard desktop computer and their respective functions.

5 Compare and contrast laptops and desktop computers, highlighting their advantages and disadvantages.

6 Discuss the significance of input and output devices in facilitating human-computer interaction.

7 What are the key factors to consider when selecting a computer for gaming purposes?

8 Describe the importance of cooling systems in computers and how they prevent overheating.

9 Discuss the advantages and disadvantages of each computer generation (first to fifth Generation)

10 Outline the major characteristics of each computer generation (first to fifth Generation) and their technological advancements.

1.13 Keywords

- Personal Computers (PCs) - General-purpose computers for individual use.
- Laptops - Portable computers with integrated components and batteries.
- Servers - High-performance computers for network resource management.
- Mainframes - Powerful computers for critical applications and large-scale data processing.
- Supercomputers - High-performance machines for complex scientific computations.
- Processing Power - Ability to execute tasks and calculations efficiently.

- Memory (RAM) - Temporary storage for quick data access during processing.
- Storage - Capacity for long-term data retention (HDD, SSD).
- Connectivity - Ability to connect to networks and the internet.
- Graphics Capability - Handling and rendering of visual elements.
- Processing Speed - The maximum speed at which a computer can perform tasks.
- Storage Capacity - Limited amount of data a computer can store.
- Physical Size - The space and form factor constraints of the computer.
- Power Consumption - The amount of energy a computer requires to operate.
- Security Vulnerabilities - Weaknesses that malicious actors can exploit.
- Central Processing Unit (CPU) - The main processing unit responsible for executing instructions.
- Motherboard - The main circuit board connecting all components.
- Memory (RAM) - Temporary storage used for data during processing.
- Storage Devices - Hard drives (HDD) or solid-state drives (SSD) for long-term data storage.
- Graphics Processing Unit (GPU) - Specialized processor for handling graphics-related tasks.

1.14 References

1. E.Balaguruswamy, "Computing Fundamentals & C Programming", Tata McGraw Hill, 2008.
2. B. A. Forouzan & R. F. Gilberg "Computer Science – A structured programming Approach Using C", 2011
3. E. Bala Guruswamy, "Programming in ANSI" Tata McGraw Hill, 2011.

Table of Content

[illegible]

Learning Objectives

After studying this unit, the student will be able to:

- Understand the distinction between system software and application software.
- Describe the functions and components of an operating system.
- Identify different types of application software and their specific uses.
- Explain the concept of open-source software and its benefits.
- Discuss the significance of software updates and security patches.
- Identify various input devices and their respective roles in interacting with computers.
- Describe the functions of common input devices, such as keyboards, mice, and touchscreens.
- Explain the purpose of output devices and how they present information to users.
- Recognize the importance of device drivers in enabling communication between devices and the operating system.
- Understand how input/output devices enhance user experiences in different computing contexts.

Introduction

Input and output devices are essential components of a computer system that enable communication between the user and the computer. These devices play a crucial role in facilitating the exchange of information and instructions, allowing users to interact with the digital world and the computer to respond accordingly. Let's take a closer look at input and output devices:

Input Devices:

Input devices are hardware components that allow users to input data, commands, and instructions into the computer system. These devices convert human-readable information into a machine-readable format. Here are some common examples of input devices:

a. Keyboard: The keyboard is perhaps the most widely used input device. It consists of a set of keys, including letters, numbers, and special characters. Users can type text and commands using the keyboard.

b. Mouse: The mouse is a pointing device that lets users control the cursor on the computer screen. It typically has buttons and a scrolling wheel, enabling users to select, drag, and interact with graphical elements.

c. Touchpad and Trackpad: These are alternative pointing devices commonly found on laptops. Users can move the cursor by sliding their fingers on the touch-sensitive surface.

d. Graphics Tablet: Also known as a digitizer or drawing tablet, this device allows artists and designers to draw directly on the tablet's surface with a stylus, which is then translated into digital artwork on the computer.

e. Scanner: A scanner captures physical documents or images and converts them into digital formats that can be stored or edited on the computer.

f. Microphone: This device captures audio and converts it into digital data, enabling users to record audio, make voice commands, or participate in voice calls.

g. Webcam: A webcam captures video and transmits it in real-time, allowing users to engage in video conferencing, record videos, or take pictures.

Output Devices:

On the other hand, output devices display or present the information processed by the computer to the user in a human-readable form. They convert machine-readable data into a format that users can understand. Here are some common examples of output devices:

a. Monitor/Display: The monitor, also called the screen or display, is the primary output device for visual information. It presents text, images, videos, and graphical user interfaces (GUIs) to the user.
b. Printer: A printer produces hard copies of digital documents on paper. There are various types of printers, including inkjet, laser, and dot matrix printers.
c. Speakers: Speakers output audio generated by the computer, allowing users to hear sounds, music, and voice communication.
d. Projector: A projector takes the visual output from the computer and projects it onto a larger screen or surface, making it useful for presentations and media viewing in larger settings.
e. Headphones: Headphones provide a private audio output for users, useful in situations where you don't want to disturb others.

2.1 Input Devices

Input devices are peripherals or hardware components that allow users to interact with a computer system by providing data, commands, or instructions to be processed. These devices enable users to input information and control the computer's operations.

Fig- Wired and wireless keyboards

Some common examples of input devices:

Keyboard

- A keyboard is a primary input device consisting of a set of keys arranged in a specific layout. Users press the keys to input letters, numbers, symbols, and other characters.

  Key points about keyboards:

  - Layout: Keyboards typically have a standard QWERTY layout, named after the first six letters in the top row of keys. This layout is the most common and widely used for English-language keyboards.
  - Alphanumeric Keys: Alphanumeric keys include letters (A-Z) and numbers (0-9). They form the primary input for typing text and numbers.
  - Function Keys (F1-F12): The function keys are usually located at the top of the keyboard and serve various functions depending on the operating system and software used.
  - Special Character Keys: Special character keys include punctuation marks, symbols, and other characters. These keys are accessed by pressing specific modifier keys (e.g., Shift, Alt, or Ctrl) in combination with the alphanumeric keys.
  - Modifier Keys: Modifier keys, such as Shift, Ctrl, Alt, and the Windows/Command key, modify the function of other keys when pressed in combination
  - Navigation Keys: Navigation keys include Arrow keys (Up, Down, Left, Right), Home, End, Page Up, and Page Down keys, which allow users to navigate within documents or interface elements.
  - Enter/Return Key: The Enter or Return key inputs a line break or executes a command, depending on the context.
  - Backspace and Delete Keys: The Backspace key erases the character to the left of the cursor, while the Delete key erases the character to the right.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

- Numeric Keypad: Many keyboards have a separate numeric keypad on the right side, allowing users to enter numbers quickly. It often includes arithmetic operators and a Num Lock key.
- Caps Lock: The Caps Lock key toggles between uppercase and lowercase letters. When activated, all typed letters appear in uppercase until it is turned off.
- Shift Lock: The Shift Lock key functions similarly to Caps Lock, but instead of toggling between letter cases, it locks the Shift key in the pressed position
- Ctrl (Control) and Alt (Alternate) Keys: Ctrl and Alt keys are used as modifiers to execute specific commands or keyboard shortcuts
- Windows/Command Key: Found on PC and Mac keyboards, this key is combined with other keys to perform various system-level functions.
- Ergonomics: Many keyboards are designed with ergonomic considerations to promote comfort and reduce the risk of repetitive strain injuries (RSIs) during extended use.
- Wireless and Mechanical Keyboards: Keyboards come in various types, including wired and wireless. Mechanical keyboards with individual switches under each key provide tactile feedback and are favored by some users for typing comfort.

Mouse

A mouse is a pointing device that allows users to move a cursor on the screen and select or interact with objects, icons, and menus by clicking buttons on the mouse.

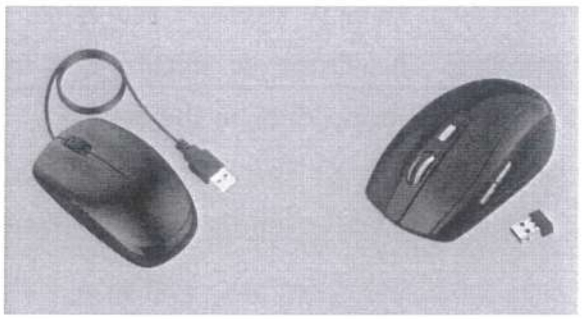


Fig - Wired and wireless mouse

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Key points about a computer mouse:

- Cursor Control: The primary function of a mouse is to control the cursor's movement on the computer screen. Users can move the mouse across a flat surface to move the cursor correspondingly.
- Buttons: A standard mouse typically has two buttons: left-click and right-click. Left-click is used for selecting and interacting with objects, while right-click provides context-specific options and menus
- Scroll Wheel: Many mice have a scroll wheel between the left and right buttons. The scroll wheel allows users to scroll vertically through documents, webpages, and other content.
- Optical or Laser Sensor: Modern mice use optical or laser sensors to track movement. These sensors provide precise cursor control on various surfaces.
- Wired and Wireless Mice: Mice can be connected to a computer using a wired USB or PS/2 cable or wireless, connecting through Bluetooth or a USB dongle.
- DPI (Dots Per Inch): DPI measures the mouse's sensitivity. Higher DPI settings result in faster cursor movement on the screen.
- Gaming Mice: Gaming mice are designed with additional features, customizable buttons, and higher DPI settings to suit the needs of gamers
- Mouse Pad: While modern mice work on various surfaces, a mouse pad can provide a smoother and more consistent tracking experience.
- Gesture Support: Some mice, especially in the context of touch-sensitive surfaces or trackpads, offer gesture support for specific actions like pinch-to-zoom or swiping.
- Ergonomics: Ergonomic designs ensure that mice fit comfortably in the hand, reducing strain during extended use.
- Bluetooth Mouse: Bluetooth mice offer wireless connectivity without the need for a separate USB dongle.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

- Gaming Mouse Software: Some gaming mice come with dedicated software that allows users to customize button functions, DPI settings, and other aspects
- Ambidextrous and Left-Handed Mice: While most mice are designed for right-handed users, some models are ambidextrous or specifically designed for left-handed users
- Multi-Button Mice: Mice with additional programmable buttons are used for productivity or gaming, where specific functions or macros can be assigned to these buttons.
- Touchpad vs. Mouse: Touchpads, commonly found on laptops, offer similar cursor control and gesture support without needing an external device.

Other Inputting Devices

- Touchpad: Commonly found on laptops, a touchpad is an alternative to a mouse. Users can move the cursor and perform actions by tapping or gestures on the touch-sensitive surface.

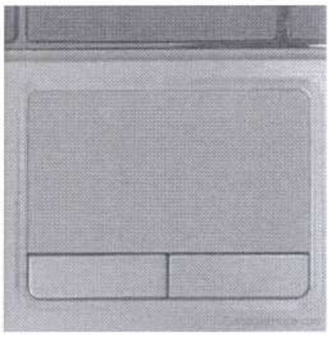


Fig – Touchpad

- Touchscreen: Touchscreens display touch-sensitive surfaces allowing users to interact directly with the screen using their fingers or a stylus. They are commonly used in smartphones, tablets, and interactive kiosks.

Fig - Touchscreen display

- Graphics Tablet or Digitizer: Graphics tablets allow artists and designers to draw or write directly on the surface with a stylus or digital pen. The input is captured and displayed on the computer screen.

Fig – Digitizer

- Scanner: Scanners are used to convert physical documents or images into digital formats by capturing their contents and saving them as digital files.

[illegible]
For Vivekananda Global University, Jaipur
Registrar

Fig - Scanner

- Webcam: A webcam is a camera that captures live video or images, allowing users to participate in video conferencing, online chats, or recording videos.

Fig – Webcam

- Microphone: A microphone is an input device used to record audio or voice input. It is commonly used for voice commands, online communication, and audio recording.

Fig – Microphone

- Barcode Reader: Barcode readers scan barcodes on products or documents to retrieve information quickly. They are often used in retail and inventory management.

For Vivekananda Global University, Jaipur

[illegible]

Registrar

Fig - Barcode and QR code Reader

- QR Code Reader: QR code readers are specialized devices or smartphone apps that scan QR codes to access embedded information or perform specific actions.
- Joystick: Joysticks are used for gaming and controlling the movement of objects in video games or flight simulators.

Fig - Joystick or game controller

- Gamepad or Controller: Gamepads are input devices designed for gaming, featuring buttons, triggers, and analog sticks to control game characters or actions.
- Biometric Sensors: Biometric input devices, such as fingerprint scanners or facial recognition cameras, capture unique physical characteristics to provide secure authentication.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Fig- Biometric Sensors

- Motion Sensors: Motion sensors, like accelerometers or gyroscopes, detect movement and orientation. They are used in gaming consoles, smartphones, and virtual reality systems.

Fig – Motion Sensors

2.2 Output Devices

Output devices for computers are peripherals or hardware components that display or present information, data, or results generated by the computer system to the user or other external devices. These devices enable users to receive and interact with the processed information. Here are some common examples of output devices for computers:

- Monitor (Display): The primary computer output device monitors display visual information, including text, images, videos, and user interfaces. They come in various sizes, resolutions, and technologies, such as LCD, LED, OLED, and CRT monitors.

[illegible]
For Vivekananda Global University, Jaipur
Registrar

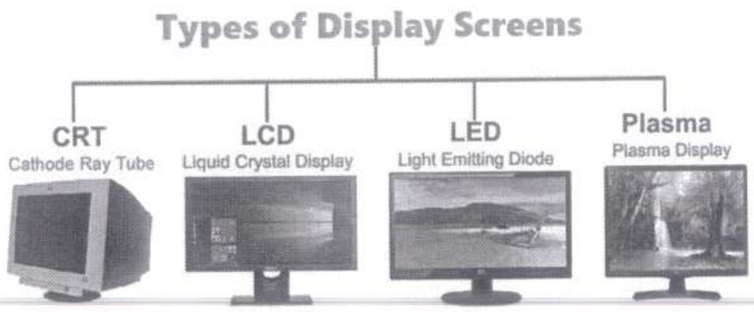


Fig - Different types of Displays

Displays come in various types, each employing different technologies and characteristics to present visual information. Here are some common types of displays:

- LCD (Liquid Crystal Display): LCDs are widely used in computer monitors, laptops, and TVs. They utilize liquid crystals to control light and create images. They offer good color reproduction and energy efficiency.
- LED (Light Emitting Diode) Display: LED displays are a type of LCD that uses LED backlighting instead of traditional CCFL (Cold Cathode Fluorescent Lamp) backlighting. LED displays are thinner, more energy-efficient, and can produce brighter images.
- OLED (Organic Light Emitting Diode): OLED displays use organic compounds that emit light when an electric current is applied. OLEDs offer better contrast, deeper blacks, and faster response times compared to LCDs. They are commonly used in smartphones, TVs, and some high-end monitors
- AMOLED (Active Matrix OLED): AMOLED is an advanced version of OLED with an active matrix that provides better pixel control, making it suitable for high-resolution displays and fast refresh rates.
- QLED (Quantum Dot LED): QLED displays use quantum dots to enhance the color and brightness of LED displays. They offer improved color accuracy and wider color gamut's.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

- Plasma Display Panel (PDP): Plasma displays use charged gas cells to emit ultraviolet light, which then illuminates phosphors to produce images. PDPs were popular for large-screen TVs but have become less common due to the rise of LCD and OLED displays.
- E-ink (Electronic Ink) Display: E-ink displays are used in e-readers and some electronic devices. They mimic the appearance of printed text and use very little power, providing a paper-like reading experience.
- CRT (Cathode Ray Tube): CRT displays use electron beams to create images on a fluorescent screen. They were widely used in older computer monitors and TVs but have been largely replaced by newer technologies.
- Micro LED: Micro LED displays use tiny LED pixels that emit their light. They offer high brightness, deep blacks, and excellent color reproduction. Micro LED displays are still in the early stages of development.
- Transparent Displays: Transparent displays allow users to see both the displayed content and objects behind the screen. They find applications in retail, advertising, and augmented reality.
- Curved Displays: Curved displays are concave or convex in shape, providing a more immersive viewing experience, especially for gaming and multimedia content.
- Projection Displays: Projection displays use projectors to display images on screens or surfaces, commonly used in home theaters, presentations, and large venues.
- Flexible Displays: Flexible displays use bendable or rollable materials to create screens that can be curved or folded, enabling new form factors for devices.
- Holographic Displays: Holographic displays create three-dimensional images using light diffraction, offering a unique visual experience.

- Printer: Printers produce hard copies of digital documents, images, or other content onto paper or print media. There are different types of printers, including inkjet, laser, and dot matrix printers.

Fig – Some common types of printers

Main types of printers:

- Inkjet Printers: Inkjet printers use liquid ink droplets to create images on paper. They are versatile, cost-effective, and suitable for printing both text and color graphics. Inkjet printers are commonly used for home and small office printing.
- Laser Printers: Laser printers use a laser beam to create static charges on a drum, which attracts toner particles to form images on paper. They offer high-speed printing and are commonly used in offices and business settings.
- Multifunction Printers (MFPs): Multifunction printers, also known as all-in-one printers, combine printing, scanning, copying, and sometimes faxing capabilities into a single device.
- Dot Matrix Printers: Dot matrix printers use a print head with pins that strike an inked ribbon to create characters and images. They are known for their impact printing and are commonly used in industrial settings and for printing forms.
- 3D Printers: 3D printers create three-dimensional objects layer by layer, using materials such as plastic, metal, or resin. They are used in various industries, including manufacturing, healthcare, and design prototyping.

For Vivekananda Global University, Jaipur
[illegible]
Registrar

- Dye Sublimation Printers: Dye-sublimation printers use a heat transfer to create high-quality photo prints on specialized paper or materials. They are commonly used for photo printing and producing ID cards.
- Thermal Printers: Thermal printers use heat to create images on heat-sensitive paper. There are two main types: direct thermal printers and thermal transfer printers. They are used in various applications, including receipts, labels, and barcode printing.
- Wireless Printers: Wireless printers connect to a computer or network via Wi-Fi, allowing users to print from multiple devices without needing physical cables.
- Large Format Printers: Large format printers are designed to print oversize documents, such as posters, banners, blueprints, and other large graphical outputs.
- Plotter Printers: Plotters are specialized printers used for precise line drawings and technical illustrations. They are commonly used in engineering, architecture, and design fields.
- Mobile Printers: Mobile printers are compact and portable, designed for on-the-go printing from smartphones and other mobile devices.
- Card Printers: Card printers are used to print plastic cards, such as ID cards, access cards, and loyalty cards, with personalized information.

- Projector: Projectors display computer-generated content on a larger screen or projection surface, making them suitable for presentations, movies, or large group displays.

Fig – Projector

For Vivekananda Global University, Jaipur

Registrar

- Speakers: Speakers produce audio output from the computer, allowing users to listen to music, sound effects, videos, and other audio content

Fig – Speakers

- Headphones: Headphones and earphones provide private audio output, enabling users to listen to audio without disturbing others.

Fig – Headphones and earphone

- Headsets: Headsets combine headphones and a microphone, allowing users to listen to audio and participate in voice communication or recordings.
- Sound Cards: While not an external device, sound cards play a crucial role in processing audio and delivering it to speakers or headphones.
- Plotter: Plotters are specialized output devices that draw detailed and precise graphical output on paper. They are commonly used in engineering and technical applications.

For Vivekananda Global University, Jaipur

[illegible]

Registrar

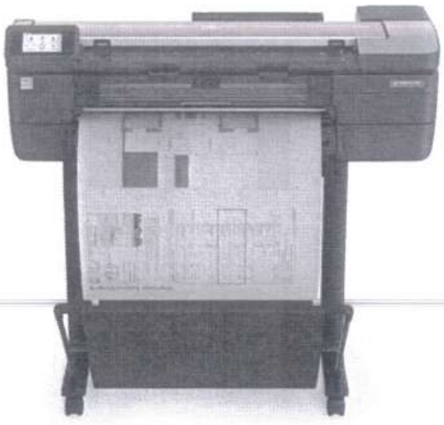


Fig – Plotter

- Braille Embosser: Braille embossers create tactile representations of digital text for visually impaired users, converting text into Braille patterns.

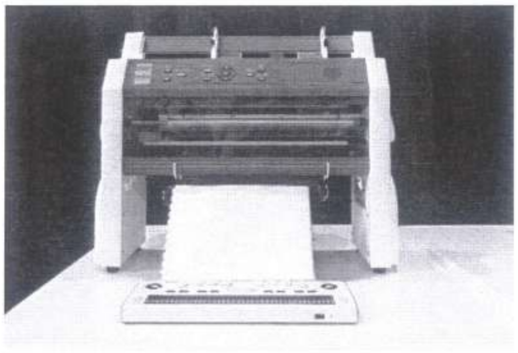


Fig – Braille Embosser

- Touchscreen Display: Touchscreen displays act as both input and output devices, allowing users to interact with the screen and receive visual feedback simultaneously.
- LED Indicators: LED indicators are small lights on the computer or external devices that provide visual cues about the system's status, such as power on/off, battery level, or activity.
- Data Projector: Data projectors are specialized projectors used to display computer data, presentations, and slideshows on screens or surfaces.
- VR Headsets: Virtual Reality (VR) headsets provide immersive visual and auditory output for virtual reality experiences.

For Vivekananda Global University, Jaipur

[illegible]

Registrar

Fig – VR headset

- Smart TVs: Smart TVs act as computer output devices with built-in displays, speakers, and internet connectivity, allowing users to access digital content and apps.
- External Displays: Besides built-in monitors, computers can be connected to external displays, like TVs or larger monitors, for extended screen real estate.

2.3 Software

Software refers to the collection of programs, data, and instructions that enable a computer system to perform specific tasks or operations. It is the non-tangible part of a computer system, as opposed to hardware, which consists of physical components. Software is essential for the functioning of modern computers and plays a crucial role in various industries and everyday life.

Types of Software:

- System Software: This software manages and controls the computer hardware and provides a platform for running other software applications.
- Application Software: These are software programs designed to perform specific tasks or provide functionality to end-users
- Programming Software: These tools are used by programmers and developers to create, debug, and maintain software applications. Examples include:
  - Integrated Development Environments (IDEs): Visual Studio, Eclipse, Xcode, etc.

- Text Editors: Examples include Visual Studio Code, Sublime Text, Atom, etc.
- Compilers and Interpreters: Software that translates programming code into machine-readable code or executes it directly.

- Middleware: This software acts as an intermediary between different software applications or between applications and the operating system. Middleware is commonly used in networking, databases, and distributed systems.
- Embedded Software: This type of software is specifically designed to run on embedded systems or specialized hardware devices. Examples include software in smartphones, routers, smart home devices, etc.
- Firmware: Firmware is software that is permanently programmed into a hardware device. It provides the necessary instructions for the device to operate and is usually not meant to be changed or updated by the end-user.
- Proprietary Software: Software that a company or individual owns, and its source code is not freely available. Users can only access the compiled version of the software and are subject to license terms and restrictions.

2.4 System Software's

System software is a category of computer software that provides the foundational infrastructure and essential services for a computer system to operate and run other software applications. Unlike application software, which is designed for specific tasks or user-oriented functions, system software manages and controls the hardware, facilitates communication between components, and provides an efficient application environment. It acts as an intermediary between the hardware and the end users, ensuring that the computer system functions smoothly and securely.

Some key components and functionalities of system software include:

- Operating System (OS): The core component of the system software, the operating system, manages computer resources, provides an interface for users to interact with the system, and coordinates the execution of application software. Examples of operating systems include Windows, macOS, Linux, iOS, and Android.

- Device Drivers: Device drivers are software programs that enable communication between the operating system and specific hardware components, such as printers, graphics cards, and network adapters. They allow the OS to control and interact with the hardware effectively.
- Firmware: Firmware is software embedded in hardware components, such as BIOS (Basic Input/Output System) in a computer's motherboard or firmware in peripheral devices. It provides essential instructions for the hardware to function during startup and initialization.
- Virtualization Software: Virtualization software allows multiple virtual machines to run on a single physical machine, enabling more efficient use of hardware resources and isolation of different computing environments.
- Utility Software: Utility software includes various tools and programs that help manage and optimize system performance, such as disk defragmenters, antivirus software, system cleanup tools, and backup utilities.
- Compiler and Interpreter: System software includes compilers and interpreters that translate high-level programming languages into machine code or execute code directly, enabling the execution of application software.
- Linker and Loader: These components are responsible for linking multiple pieces of code and loading them into memory for execution.

2.5 Application Software's

Application software, often referred to simply as "applications" or "apps," is a computer software designed to perform specific tasks or applications for end-users. Unlike system software, which manages and controls computer hardware and provides the foundation for running other software, application software is created to serve the needs of users and help them accomplish various tasks efficiently. Application software can be further categorized into different types based on their functionality and purpose. Here are some common examples of application software:

- Word Processors: Word processing software, such as Microsoft Word or Google Docs, allows users to create, edit, and format text-based documents. It is widely used for tasks like writing letters, reports, essays, and other types of written content.
- Spreadsheets: Spreadsheet software like Microsoft Excel or Google Sheets is used for organizing and analyzing data in tabular form. It is commonly employed for financial calculations, data analysis, and creating charts and graphs.
- Presentation Software: Presentation applications like Microsoft PowerPoint or Apple Keynote enable users to create visually engaging slideshows for presentations, lectures, and meetings.
- Web Browsers: Web browsers like Google Chrome, Mozilla Firefox, and Microsoft Edge allow users to access and navigate the internet, view websites, and interact with web-based applications.
- Email Clients: Email clients such as Microsoft Outlook, Gmail, and Apple Mail facilitate email management and sending.
- Graphic Design Software: Graphic design applications like Adobe Photoshop, Illustrator, and CorelDRAW are used for creating and editing images, illustrations, and other visual content.
- Video Editing Software: Video editing tools like Adobe Premiere Pro, Final Cut Pro, and Sony Vegas allow users to edit and manipulate video footage for various purposes, such as creating videos for social media, films, or presentations.
- Audio Editing Software: Audio editing applications like Audacity or Adobe Audition help users edit, record, and mix audio files.
- Multimedia Players: VLC Media Player, Windows Media Player, and iTunes enable users to play audio and video files.

[illegible]

- Antivirus Software: Antivirus applications like Norton, McAfee, and Avast protect computers from malware, viruses, and other security threats.
- Gaming Software: Video games and gaming platforms fall under application software, offering entertainment and interactive experiences for users.

2.6 Commercial Software's

Commercial software, also known as proprietary software, refers to software developed and distributed by a company or individual intending to make a profit. It is the opposite of open-source software, where the source code is freely available and can be modified by anyone. Commercial software is typically protected by copyright, and users must obtain a license or pay a fee to use the software legally.

Key characteristics of commercial software include:

- Licensing: Users are required to purchase a license to use the software, and the terms and conditions of the license dictate how the software can be used and distributed.
- Intellectual Property Protection: Commercial software is protected by copyright, trademarks, or patents to prevent unauthorized copying, distribution, or modification.
- Support and Updates: Commercial software often comes with customer support and regular updates or patches to improve performance, fix bugs, and address security issues.
- Quality Assurance: Commercial software undergoes rigorous testing and quality assurance processes to ensure it meets certain standards and functions reliably.
- Distribution: Commercial software is typically distributed through various channels, including retail stores, online marketplaces, and the software developer's website.

2.7 Open Source Software's

Open-source software (OSS) is a type of software that provides the source code openly to the public. Anyone can view, use, modify, and distribute the software's source code without restrictions. Open-source software is typically developed collaboratively by a community of developers contributing to its improvement and maintenance. This development model fosters transparency, collaboration, and innovation within the software community. Here are some popular open-source software examples:

- Linux Operating System: Linux is an open-source operating system widely used in servers, supercomputers, smartphones, and various other devices. Distributions like Ubuntu, Fedora, Debian, and CentOS are Linux-based.
- Mozilla Firefox: Firefox is an open-source web browser emphasizing speed, privacy, and customization. It is available for various platforms, including Windows, macOS, and Linux.
- Apache HTTP Server: Apache is one of the most popular web servers globally, powering many websites. It handles HTTP requests and serves web pages to clients.
- LibreOffice: LibreOffice is a free and open-source office suite offering applications like Writer (word processor), Calc (spreadsheet), Impress (presentation), and more.
- GIMP: The GNU Image Manipulation Program (GIMP) is a powerful open-source image editing software akin to Adobe Photoshop.
- VLC Media Player: VLC is a versatile open-source media player capable of playing various audio and video formats across different platforms.
- Blender: Blender is a 3D computer graphics software for modeling, animation, rendering, and more. It is popular among artists and animators.
- WordPress: WordPress is an open-source content management system (CMS) widely used for creating websites and blogs.
- MySQL: MySQL is an open-source relational database management system (RDBMS) used for data storage and retrieval.

For Vivekananda Global University, Jaipur

[illegible]

Registrar

- Git: Git is an open-source version control system that tracks changes in software development projects.
- Python: Python is a popular open-source programming language known for its simplicity and readability in various applications and fields.

2.8 Domain Software's

"Domain software" typically refers to specialized software applications designed to serve the specific needs of a particular industry or field, commonly known as a "domain." These software solutions are tailored to address their domain's unique challenges and requirements. Here are some examples of domain software in various industries:

Healthcare Domain Software:

- Electronic Health Record (EHR) Systems: Software healthcare providers use to store and manage patient health information.
- Picture Archiving and Communication Systems (PACS): Software for storing and managing medical imaging data.
- Medical Billing Software: Applications for managing billing and invoicing in healthcare settings.

Education Domain Software:

- Learning Management Systems (LMS): Software used by educational institutions for creating and delivering online courses and managing educational content.
- Student Information Systems (SIS): Software for managing student data, including enrollment, grades, attendance, etc.
- Academic Plagiarism Checkers: Tools that help educators detect plagiarism in students' academic work.

Financial Domain Software:

- Accounting Software: Applications for managing financial transactions, bookkeeping, and generating financial reports.

- Trading Platforms: Software used by traders to execute trades in financial markets.
- Personal Finance Software: Tools that help individuals manage their personal finances, budgeting, and investments.

Engineering Domain Software:

- Computer-Aided Design (CAD) Software: Tools engineers and designers use to create detailed 2D and 3D models of products and structures.
- Finite Element Analysis (FEA) Software: Applications for simulating and analyzing engineering designs and structures under various conditions.
- Electronic Design Automation (EDA) Software: Software used for designing and verifying electronic circuits and systems.

Manufacturing Domain Software:

- Enterprise Resource Planning (ERP) Systems: Software used to manage various aspects of a manufacturing business, including production, inventory, and supply chain.
- Computer-Aided Manufacturing (CAM) Software: Tools for creating CNC (Computer Numerical Control) programs for automated manufacturing processes.
- Product Lifecycle Management (PLM) Software: Applications for managing the entire lifecycle of a product, from design to disposal.

Legal Domain Software:

- Legal Practice Management Software: Tools used by law firms to manage cases, documents, and client information.
- Legal Research Software: Applications that help legal professionals access and analyze legal resources and case law.

2.9 Freeware Software's

Freeware software refers to software that is available for use at no cost. Users can download, install, and use freeware without paying any licensing fees. It is important to note that while freeware is free to use, it might still be subject to certain terms and conditions outlined by the software developer or distributor. Here are some key points to understand about freeware software:

- Free of Charge: Freeware is completely free, and users do not need to purchase a license or pay any fees to use it. It can be freely downloaded and installed on a computer or other compatible devices.
- Proprietary vs. Open Source: Freeware can be both proprietary and open source. Some freeware applications come with closed-source code and are developed by companies or individuals who choose to offer their software for free. Others are open-source, allowing users to freely access and modify the source code.
- Limited Rights: While freeware is free to use, it may have certain limitations on usage, distribution, or modification. Users should review the terms of use provided by the software developer or distributor to understand any restrictions.
- No Cost Updates: Freeware developers might release updates to their software to fix bugs or add new features, and these updates are typically available to users at no additional cost.
- Not Always Ad-Free: Some freeware applications are supported by advertising, and users may encounter ads within the software interface or during installation. However, not all freeware includes ads.
- Support and Documentation: Support and documentation for freeware can vary. Some developers offer user guides, forums, or online communities where users can find help, while others may not provide extensive support resources.
- Examples: There are many popular freeware applications available, some of which include:
  - Cleaner: A utility for cleaning and optimizing a computer's performance.
  - 7-Zip: A file compression and decompression tool.
  - Audacity: An open-source audio editing software.

    - Notepad++: A text editor with additional programming features.
    - Paint.NET: A free image and photo editing software.
    - PDFCreator: A tool for converting documents to PDF format.

2.10 Summary

Input Devices:

Input devices are hardware components that allow users to interact with a computer system by providing data or commands. They enable users to input information and instructions for processing by the computer. Some common input devices include:

- Keyboard: The primary input device for entering alphanumeric characters, numbers, and special symbols.
- Mouse: A pointing device used to control the cursor on the screen and select objects.
- Touchscreen: A display with touch-sensitive capabilities that allow users to input commands directly by touching the screen.
- Trackpad: Similar to a mouse but integrated into laptops, enabling cursor control through finger movements.
- Scanner: Device used to convert physical documents, photos, or images into digital formats.
- Microphone: Captures audio input, allowing users to record voice or provide voice commands.
- Webcam: Captures video input, enabling video conferencing and video recording.

Output Devices:

Output devices display or present information the computer processes to the user in various formats. They allow users to view, hear, or receive the results of their actions or computer operations. Some common output devices include:

- Monitor/Display: Displays text, images, and graphical user interfaces to the user.

- Printer: Produces hard copies of digital documents or images on paper.
- Speakers: Output sound and audio, allowing users to hear audio content, system alerts, and multimedia.
- Projector: Projects computer screen content onto a larger screen or surface for presentations.
- Headphones: Provides personal audio output for private listening.

System Software:

- System software provides the foundational infrastructure for a computer system to operate and manage hardware resources effectively.
- It includes the operating system (OS), device drivers, firmware, virtualization software, and utility software.

Application Software:

- Application software is designed to perform specific tasks or applications for end-users.
- It allows users to accomplish various functions, from productivity and entertainment to communication and creativity.
- Examples include word processors, spreadsheets, presentation software, web browsers, and graphic design tools.
- Application software enhances productivity, creativity, communication, and entertainment in the digital age.

Open Source Software:

- Open source software is a type of software that is freely available to the public, allowing users to view, modify, and distribute the source code.
- It is developed collaboratively by a community of volunteers, and its source code is openly accessible.
- Examples include the Linux operating system, Apache web server, and Mozilla Firefox browser.
- Open-source software promotes transparency, collaboration, and customization.

Freeware:

- Freeware refers to software that is available for free to use and distribute, but the source code is not necessarily open or accessible.
- It is provided at no cost to users, but some freeware may have limitations or come with advertisements.
- Freeware can be used for personal and non-commercial purposes without charge.
- Examples include many free applications and tools available for download online.

2.11 Review Questions

1 What are input devices, and how do they enable users to interact with a computer system?

2 Name three common input devices used for data entry and interaction with computers.

3 Describe the primary function of a mouse and how it is used to control the cursor on a computer screen.

4 How does a touchscreen differ from traditional input devices like a keyboard and mouse?

5 What is the purpose of a scanner, and how does it convert physical documents into digital formats?

6 Describe an output device and its function in presenting information processed by the computer.

7 How do input/output devices contribute to human-computer interaction, and why are they essential for user experience?

8 What is software, and how does it differ from hardware in a computer system?

9 Differentiate between system software and application software. Provide examples of each.

10 Explain the primary functions of an operating system and its significance in computer systems.

2.12 Keywords

- Keyboard - Device for entering text and commands by pressing keys.
- Mouse - Pointing device used to control the cursor on a screen.
- Touchscreen - Display with touch-sensitive capabilities for direct input.
- Scanner - Device that converts physical documents into digital formats.
- Microphone - Captures audio input for recording or voice commands.
- Webcam - Captures video input for video conferencing or recording.
- Monitor/Display - Output device that displays text, images, and graphics.
- Printer - Output device that produces hard copies of digital documents.
- Speakers - Output sound and audio for multimedia playback.
- Projector - Output device that projects computer screen content onto a larger surface.
- Operating System (OS) - Software managing computer resources and providing user interface.
- Application Software - Software designed for specific tasks or user-oriented functions.
- System Software - Software providing infrastructure for computer system operation.
- Freeware - Software available for free use, without cost or licensing fees.
- Open Source - Software with publicly accessible source code for viewing and modification.

2.13 References

1. E.Balaguruswamy, "Computing Fundamentals & C Programming", Tata McGraw Hill, 2008.
2. B. A. Forouzan & R. F. Gilberg "Computer Science – A structured programming Approach Using C", 2011
3. E. Bala Guruswamy, "Programming in ANSI" Tata McGraw Hill, 2011.

Table of Content

[illegible]

Learning Objectives

After studying this unit, the student will be able to:

- Define primary memory (RAM) and its role in a computer system.
- Explain the concept of volatile memory and its implications for primary memory.
- Describe secondary memory and its function as long-term storage.
- Compare and contrast the characteristics of primary and secondary memory in terms of speed, volatility, and capacity.
- Understand the purpose of cache memory and its role in improving CPU performance.
- Explain how cache memory acts as a buffer between the CPU and primary memory.
- Identify the advantages and limitations of cache memory in terms of size, speed, and cost.
- Identify the components of the Windows operating system.
- Describe the desktop as the primary workspace in Windows.
- Recognize the start menu as a central feature of the Windows interface.
- Understand the purpose of icons and their representation of files, folders, applications, or functions.
- Explain how to access and navigate the start menu to open programs and access various features and settings.
- Describe how to place files, folders, and shortcuts on the desktop for easy access.
- Differentiate between the Windows desktop, taskbar, and Start Menu, and their respective functionalities.

Introduction to Computer Memory

Computer memory refers to the electronic components a computer uses to store and access data and instructions. It is a crucial aspect of any computing device, enabling it to perform tasks efficiently and effectively. There are several types of computer memory, each serving different purposes.

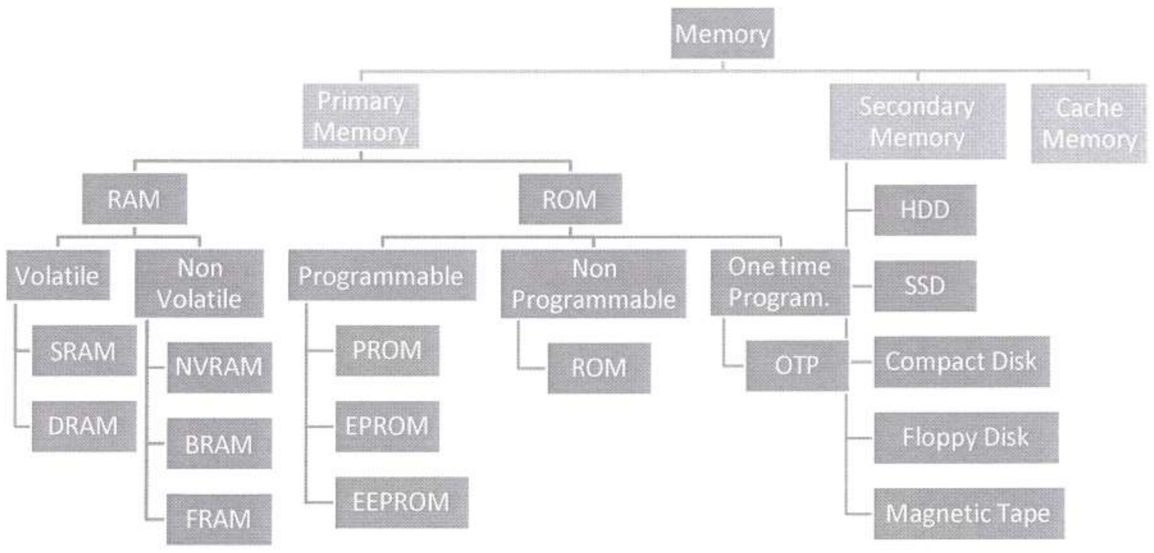


Fig – Memory types

Here are some of the most common types:

- Random Access Memory (RAM): RAM is volatile memory that provides temporary storage for data and programs currently used by the computer's operating system and applications. It allows quick access to data, which helps in the fast execution of tasks. However, when the computer is powered off or restarted, the data in RAM is lost.
- Read-Only Memory (ROM): ROM is non-volatile memory used to store essential firmware and software instructions required during the boot-up

For Vivekananda Global University, Jaipur
Registrar

process. It contains instructions that the user cannot modify, and it retains its content even when the power is turned off.

- Hard Disk Drive (HDD): HDD is a non-volatile data storage device that stores data on magnetic disks. It provides long-term storage for the operating system, software applications, and user files. Data on an HDD remains intact even when the power is turned off.
- Solid State Drive (SSD): Like an HDD, an SSD is a non-volatile storage device, but it uses NAND-based flash memory to store data instead of magnetic disks. SSDs are faster, more reliable, and more energy-efficient than traditional HDDs.
- Cache Memory: Cache memory is a small, high-speed memory located close to the CPU. It stores frequently accessed data and instructions to speed up the execution of tasks, as accessing data from the cache is much faster than fetching it from RAM or storage devices.
- Virtual Memory: Virtual memory is a technique that allows the operating system to use a portion of the hard disk as an extension of RAM. When the RAM is full, the OS moves less frequently used data from RAM to virtual memory, freeing up space in RAM for other tasks.
- Flash Memory: Flash memory is a non-volatile memory commonly used in USB drives, memory cards, and SSDs. It can retain data without a continuous power supply and is widely used for portable storage.
- The amount and type of memory in a computer significantly impact its performance. Modern computers typically combine RAM and storage devices like SSDs or HDDs to balance speed and capacity requirements. Memory capabilities continue to improve as technology advances, leading to more powerful and efficient computing devices.

3.1 Primary Memory (Main Memory)

Primary memory, also known as main memory or internal memory, refers to the immediate storage used by the CPU to store data and instructions that are actively used during the computer's operation. The two most common types of primary memory are Random Access Memory (RAM) and Read-Only Memory (ROM).

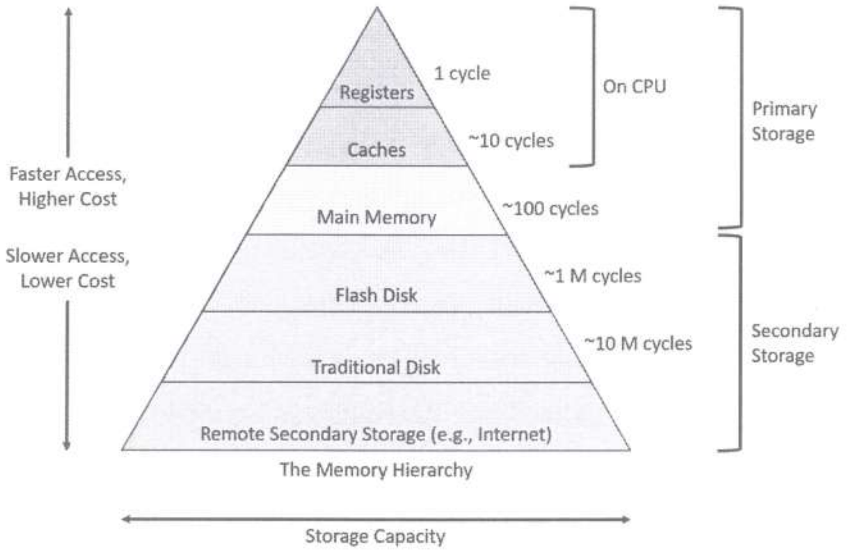


Fig – Memory Triangle

There are primarily two types of primary memory:

Random Access Memory (RAM):

> RAM is the most common type of primary memory used in modern computers. It is volatile memory, meaning its contents are lost when the power is turned off. RAM allows the CPU to access data and instructions quickly, making it ideal for running applications and performing tasks efficiently.

There are different types of RAM, including

- Dynamic RAM (DRAM): DRAM is the most common type of RAM used in personal computers. It requires periodic refreshing to maintain data, which makes it slower than other types of RAM but more cost-effective and suitable for everyday computing tasks.
- Static RAM (SRAM): SRAM is faster and more expensive than DRAM. It does not require refreshing and is often used in cache memory due to its high-speed access.

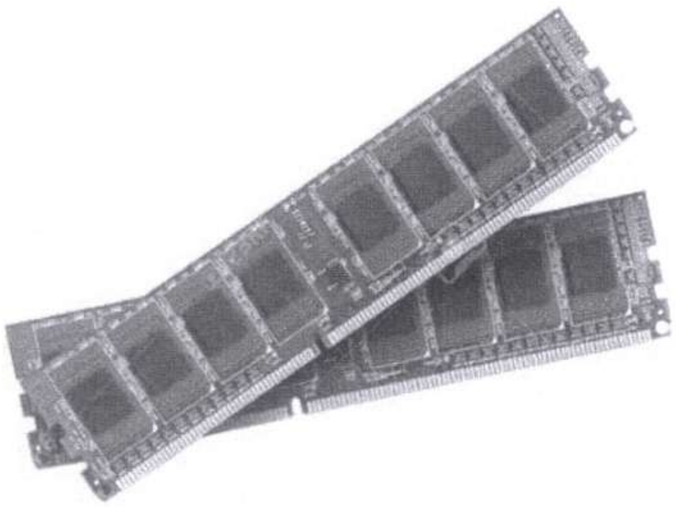


Fig - Random Access Memory

- Synchronous Dynamic RAM (SDRAM): SDRAM synchronizes its operations with the system clock, enabling faster data access compared to traditional DRAM.
- Double Data Rate (DDR) RAM: DDR RAM is an improved version of SDRAM that can transfer data on both the rising and falling edges of the clock cycle, effectively doubling the data transfer rate.

Read-Only Memory (ROM):

ROM is another type of primary memory, but unlike RAM, it is non-volatile, meaning its contents are retained even when the power is turned off. ROM is used to store essential firmware, bootstrap loaders, and other critical instructions required during the computer's boot-up process. The data

For Vivekananda Global University, Jaipur

[illegible]

Registrar

stored in ROM cannot be modified or changed by the user, making it a secure storage medium for essential system-level software.

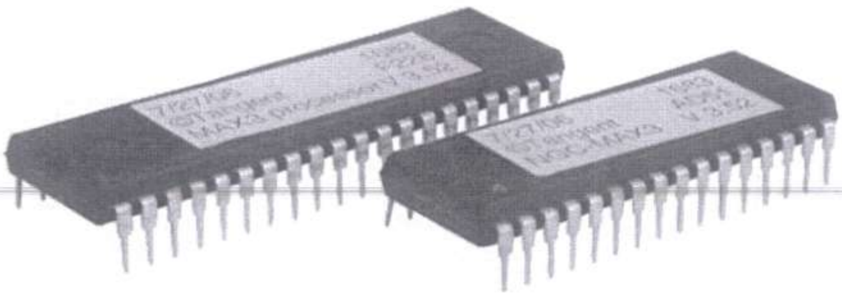


Fig – Read Only Memory

There are different types of ROM, including:

- Mask ROM: In this type of ROM, the data is permanently written during manufacturing and cannot be modified afterward.
- Programmable Read-Only Memory (PROM): PROM can be programmed by the user after purchase, allowing limited customization of its contents.
- Erasable Programmable Read-Only Memory (EPROM): EPROM can be erased and reprogrammed using ultraviolet light. However, the erasing process is time-consuming and requires special equipment.
- Electrically Erasable Programmable Read-Only Memory (EEPROM): EEPROM can be electrically erased and reprogrammed, making it more user-friendly than EPROM. It is often used for storing BIOS settings and firmware updates.

These primary memory types work together to facilitate the smooth execution of programs and provide quick access to data required by the CPU during processing. The combination of RAM and ROM ensures the proper functioning of a computer system and allows users to run applications and store data efficiently.

Advantages of Primary Memory:

- Speed: Primary memory is much faster than secondary memory (storage devices like HDDs or SSDs) in terms of data access and retrieval, which leads to faster execution of programs and overall system performance.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

- Volatile: RAM, a type of primary memory, is volatile, meaning its contents are lost when the power is turned off. This characteristic allows for quick data changes and efficient management by the operating system.
- Direct Access: The CPU can directly access data from primary memory, making it the preferred location for running applications and storing intermediate results during processing.
- Data Manipulation: Data stored in primary memory can be easily read, written, and manipulated by the CPU, making it ideal for dynamic tasks like multitasking and running programs.

Disadvantages of Primary Memory:

- Limited Capacity: Primary memory is typically more expensive and has limited capacity than secondary memory. This limitation restricts the amount of data that can be accessed and processed at any given time.
- Volatility: The volatile nature of primary memory means that data is lost when the power is turned off, necessitating secondary memory for long-term data storage.
- Cost: Primary memory components like RAM are more costly per unit of storage than secondary memory devices like HDDs or SSDs, making it expensive to add large amounts of primary memory to a computer.

3.2 Secondary Memory

Secondary memory, also known as auxiliary memory or external memory, refers to long-term storage devices used to store data and programs that persist even when the power is turned off. Unlike primary memory (RAM and ROM), secondary memory has larger capacity but slower access times. It serves as a crucial component for data storage, enabling users to retain their files, applications, and other data beyond the current session. Secondary memory also refers to long-term storage devices that retain data even when the power is turned off. Common types of secondary memory include Hard Disk Drives (HDDs) and Solid State Drives (SSDs).

[illegible]
For Vivekananda Global University, Jaipur
Registrar

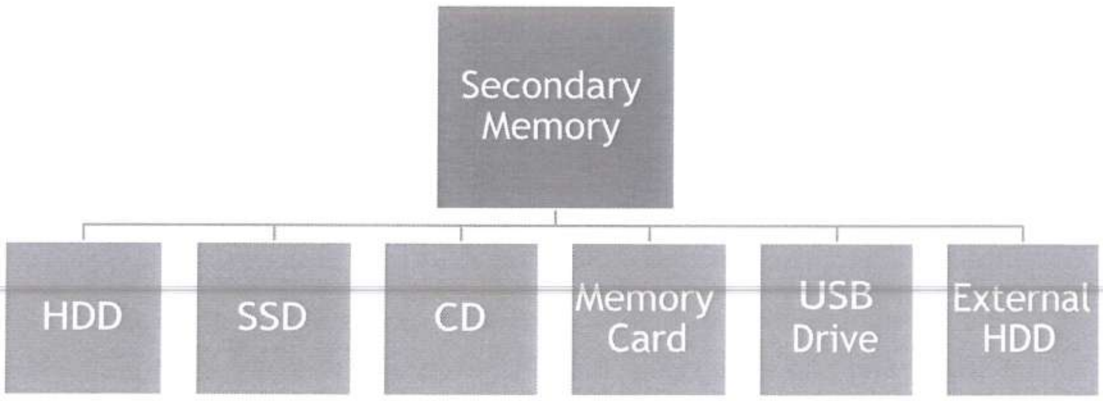


Fig - Secondary Memory

There are several types of secondary memory, each with its advantages and use cases:

- Hard Disk Drive (HDD): HDDs are one of the most commonly used types of secondary memory. They store data on magnetic disks (platters) that spin at high speeds. Data is read and written using magnetic heads, which move over the surface of the platters. HDDs provide a large amount of storage capacity at a relatively low cost and are suitable for long-term data storage, including operating systems, software applications, and user files.

Fig - Hard Disk Drive

[illegible]
For Vivekananda Global University, Jaipur
Registrar

- Solid State Drive (SSD): SSDs are a newer secondary memory type that uses NAND-based flash memory to store data. Unlike HDDs, SSDs have no moving parts, making them faster, more durable, and more energy-efficient. They offer faster data access times and read/write speeds than traditional HDDs. SSDs are commonly used in laptops, desktops, and servers to improve system performance.

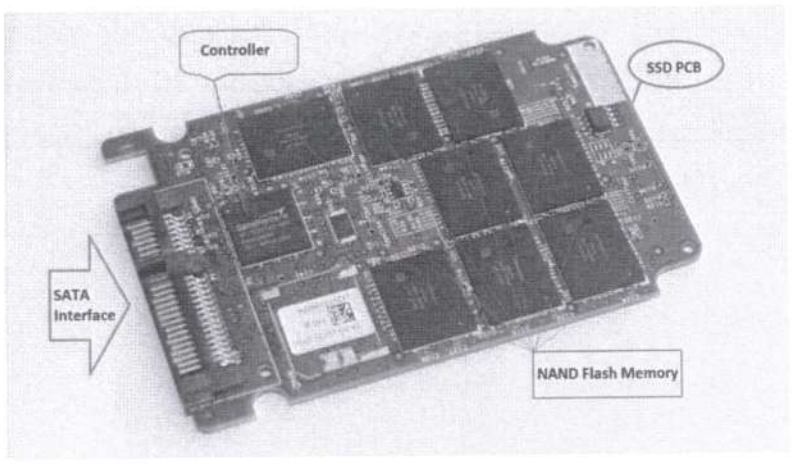


Fig – Solid State Drive circuit

- USB Flash Drives: USB flash drives, pen or thumb drives, are portable storage devices that use flash memory to store data. They connect to computers via USB ports and are small, lightweight, and convenient for storing data. USB flash drives are commonly used for data backup, file transfer, and portable application storage.

For Vivekananda Global University, Jaipur

[illegible]

Registrar

Fig - USB Flash Drive and Memory card

- Memory Cards: Memory cards are commonly used in cameras, smartphones, tablets, and other portable devices to expand their storage capacity. They come in various formats such as Secure Digital (SD), microSD, CompactFlash (CF), and more.
- Optical Discs: Optical discs, such as CDs, DVDs, and Blu-ray discs, are secondary memory options that use lasers to read and write data on their surfaces. CDs and DVDs are commonly used for storing music, videos, and software applications, while Blu-ray discs offer higher storage capacity and are used for high-definition videos and data backups.

Fig – Optical Disc

- External Hard Drives: External hard drives are HDDs or SSDs housed in an external casing with a USB or Thunderbolt interface. They provide additional storage capacity and are useful for backing up data or expanding a computer's storage.
- Cloud Storage: Cloud storage is a form of secondary memory that involves storing data on remote servers accessed through the internet. Services like Google Drive, Dropbox, and Microsoft OneDrive allow users to store and access data from any device connected to the internet.

For Vivekananda Global University, Jaipur
[illegible]
Registrar

Table - Hard Disk Drive (HDD) vs. Solid State Drive (SSD) Comparison:

Parameter	HDD	SSD
Technology:	Uses magnetic spinning platters to store data and read/write heads to access data.	Utilizes NAND-based flash memory to store data, with no moving parts, making it more reliable and durable.
Speed	Slower access times due to the mechanical movement of read/write heads, resulting in longer application boot and load times.	Significantly faster access times as data retrieval is electronic with no mechanical movement, leading to quicker boot and application launch times.
Power Consumption:	Consumes more power as it requires spinning disks and moving parts to operate.	Requires less power as there are no moving parts, resulting in improved energy efficiency and longer battery life for laptops and portable devices.
Reliability	Slightly less reliable compared to SSDs due to mechanical components and higher susceptibility to failure.	More reliable due to the absence of moving parts and the overall robust design.
Noise	Produces noise during operation due to the spinning disks and moving parts.	Silent operation since there are no moving parts, resulting in a noise-free user experience.
Durability	Susceptible to mechanical failures and data loss due to shock, impact, and wear over time.	More durable and resistant to physical shocks and vibrations, ensuring data integrity and a longer lifespan.

[illegible]

Capacity	Generally available in larger capacities at a lower cost per gigabyte than SSDs.	Offers a wide range of capacities but is relatively more expensive per gigabyte.
Cost	More cost-effective in terms of storage capacity, making it a popular choice for mass storage.	Generally more expensive than HDDs, but prices have been decreasing over time as technology advances.
Heat Generation	Generates more heat during operation due to mechanical movements.	Produces less heat, contributing to a cooler and quieter computing environment.
Weight	HDD is Heavier due to the presence of spinning disks and mechanical components.	SSD Lightweight, making it ideal for portable devices and laptops.

Each secondary memory type has advantages and disadvantages regarding speed, capacity, portability, and cost. Users often choose a combination of these storage options based on their specific needs for data retention and accessibility.

Advantages of Secondary Memory:

- Non-volatility: Unlike primary memory, secondary memory is non-volatile, meaning it retains data even when the power is turned off, making it suitable for long-term storage of files and applications.
- Larger Capacity: Secondary memory devices generally have a much larger storage capacity than primary memory, allowing users to store large amounts of data, including applications, files, and multimedia content.
- Cost-Effectiveness: Secondary memory devices are more cost-effective for storing large volumes of data, making them an economical choice for long-term storage requirements.
- Portability: Some secondary memory devices like USB drives and external SSDs are portable and allow users to carry their data with them for use on different computers.

[illegible]

For Vivekananda Global University, Jaipur
Registrar

Disadvantages of Secondary Memory:

- Slower Access: Secondary memory is typically slower than primary memory regarding data access and retrieval. This can result in longer application load times and slower overall system performance.
- Indirect Access: The CPU cannot directly access data from secondary memory; data must be first transferred to primary memory before processing, leading to additional latency.
- Wear and Tear: Mechanical storage devices like HDDs have moving parts that can wear out over time, potentially leading to data loss or drive failure.

Table – Primary vs Secondary Memory

Parameter	**Primary Memory**	**Secondary Memory**
Volatility	Primary memory is volatile, meaning it loses its data when the power is turned off. It requires continuous power to maintain the stored information.	Secondary memory is non-volatile, meaning it retains data even when the power is turned off. It provides long-term storage for files, applications, and data.
Speed	Primary memory is much faster than secondary memory, allowing the CPU to access and modify data quickly.	Secondary memory is slower than primary memory. Access times are higher, leading to relatively slower data retrieval than RAM.
Capacity	The capacity of primary memory is limited compared to secondary memory. RAM is typically smaller in size and more expensive per unit of storage.	Secondary memory has a much larger storage capacity compared to primary memory. HDDs and SSDs offer terabytes of storage space.
Role	Primary memory is directly accessible by the CPU and is used for temporarily storing	Secondary memory serves as long-term storage for data, programs, and multimedia files

[illegible]

For Vivekananda Global University, Jaipur

Registrar

	data, program instructions, and the operating system during system operation.	that need to be retained beyond the current session or system power cycle.
Cost	Primary memory (especially RAM) is generally more expensive than secondary memory on a per-gigabyte basis.	Secondary memory (especially HDDs) is generally more cost-effective in terms of storage per unit than primary memory.
Example	RAM, ROM, Cache Memory	HDD, SSD, CD, Tape, Memory Cards

In summary, primary memory (RAM) offers speed and direct access advantages but has limited capacity and is volatile, while secondary memory (HDDs, SSDs) provides non-volatility, larger storage capacity, and cost-effectiveness but may have slower access times and require indirect access through primary memory. Both types of memory play essential roles in the functioning of a computer system, complementing each other to ensure efficient data processing and storage.

3.3 Cache Memory

Cache memory is high-speed memory located between the central processing unit (CPU) and a computer system's main memory (RAM). Its purpose is to store frequently accessed data and instructions, allowing the CPU to retrieve them, significantly improving overall system performance quickly. Cache memory acts as a buffer between the slower main memory and the fast CPU, bridging the speed gap and reducing the CPU's time waiting for data to be fetched from RAM.

There are typically three levels of cache memory in modern computer architectures:

- Level 1 (L1) Cache: L1 cache is the smallest and fastest cache, located directly on the CPU chip itself. It is divided into a separate instruction cache

[illegible]
For Vivekananda Global University, Jaipur
Registrar

(L1i) and data cache (L1d). L1 cache has extremely low latency and is used to store instructions and data that the CPU is currently working with.

- Level 2 (L2) Cache: L2 cache is larger than L1 cache and is usually integrated into the CPU but slightly slower than L1 cache. It is a backup for L1 cache and provides additional storage for frequently accessed data and instructions.
- Level 3 (L3) Cache: Some CPUs may have an L3 cache, which is larger but slower than L2 cache. It is often shared among multiple CPU cores in a multi-core processor and helps reduce contention for shared data.

The cache memory works on the "principle of locality," which suggests that programs tend to access a relatively small portion of data and instructions frequently and sequentially. When the CPU needs data, it first checks the cache. If the required data is present in the cache (cache hit), the CPU can retrieve it quickly. If the data is not in the cache (cache miss), the CPU will fetch it from the main memory and load it into the cache for future use.

Advantages of Cache Memory:

- Faster Access Times: Cache memory provides much faster access times than main memory (RAM). This allows the CPU to fetch data and instructions quickly, reducing the time spent waiting for data to arrive.
- Improved System Performance: Cache memory improves overall system performance by reducing memory access latency. It allows the CPU to process data more efficiently and run applications faster.
- Optimizing CPU Utilization: Cache memory helps keep the CPU busy by providing a constant supply of frequently accessed data, reducing the processor's idle time.
- Energy Efficiency: Accessing data from cache consumes less power than accessing it from the main memory. This energy efficiency contributes to better power management and reduced heat generation in the system.

Disadvantages of Cache Memory:

[illegible]

- Limited Capacity: Cache memory has limited capacity compared to main memory. It can only hold a small portion of the data present in the main memory. Larger caches can be expensive and may not always fit on the CPU chip.
- Expensive: Cache memory is more expensive than main memory per storage unit. High-speed cache memory technologies come at a premium cost.
- Cache Coherency: In multi-core processors, maintaining cache coherency between multiple cores can be complex and may lead to increased cache miss penalties.
- Despite these limitations, cache memory plays a critical role in modern computer systems, ensuring efficient data retrieval and improving the overall performance of CPUs in various computing tasks.

Registers

Registers are small, ultra-fast storage locations within the central processing unit (CPU) of a computer. While registers are not considered traditional memory like RAM or secondary storage, they are often referred to as memory because they temporarily hold data and instructions during CPU operations.

Key points about registers as memory:

- Speed: Registers are the fastest form of memory in a computer, with access times measured in picoseconds. They are built directly into the CPU and offer lightning-fast data retrieval.
- Capacity: Registers have a limited capacity compared to other memory types. They can store only a small amount of data (usually a few bytes to a few words)
- Purpose: Registers are primarily used for rapid data manipulation and temporary storage of operands, addresses, and intermediate results during arithmetic, logical, and control operations.

[illegible]
For Vivekananda Global University, Jaipur
Registrar

- Hierarchy: Registers are part of the CPU's memory hierarchy, with higher levels (e.g., L1, L2, and L3 cache) having larger capacities but slower access times.
- Register Types: CPUs typically have different registers, including general-purpose registers, special-purpose registers (e.g., program counter, instruction register), and floating-point registers for floating-point arithmetic.
- Register Access: Access to registers does not require memory addresses like traditional memory. Instead, registers are directly accessible by the CPU's arithmetic and control units.
- Functionality: Registers play a crucial role in improving the overall performance of the CPU by reducing the need to fetch data from slower memory locations.

3.4 Graphical User Interface

A Graphical User Interface (GUI) is a visual interface that allows users to interact with a computer, software application, or electronic device through graphical elements such as icons, buttons, menus, and windows. GUIs are designed to make computing more intuitive and user-friendly by presenting information and actions visually appealing and easily understandable.

Key features of a GUI include:

- Icons: Small graphical representations of files, folders, or applications that users can click or tap to access and perform specific actions.
- Menus: Dropdown or popup lists displaying various options and commands that users can select to perform specific tasks.
- Windows: Overlapping and resizable graphical containers that display application interfaces and content, allowing users to work on multiple tasks simultaneously.
- Buttons: Graphical elements that users can click or tap to trigger specific actions or commands.

[illegible]
For Vivekananda Global University, Jaipur
Registrar

- Pointing Device Support: GUIs are typically designed to work with pointing devices like a mouse, touchpad, or touchscreen, allowing users to interact with the interface by pointing and clicking.
-

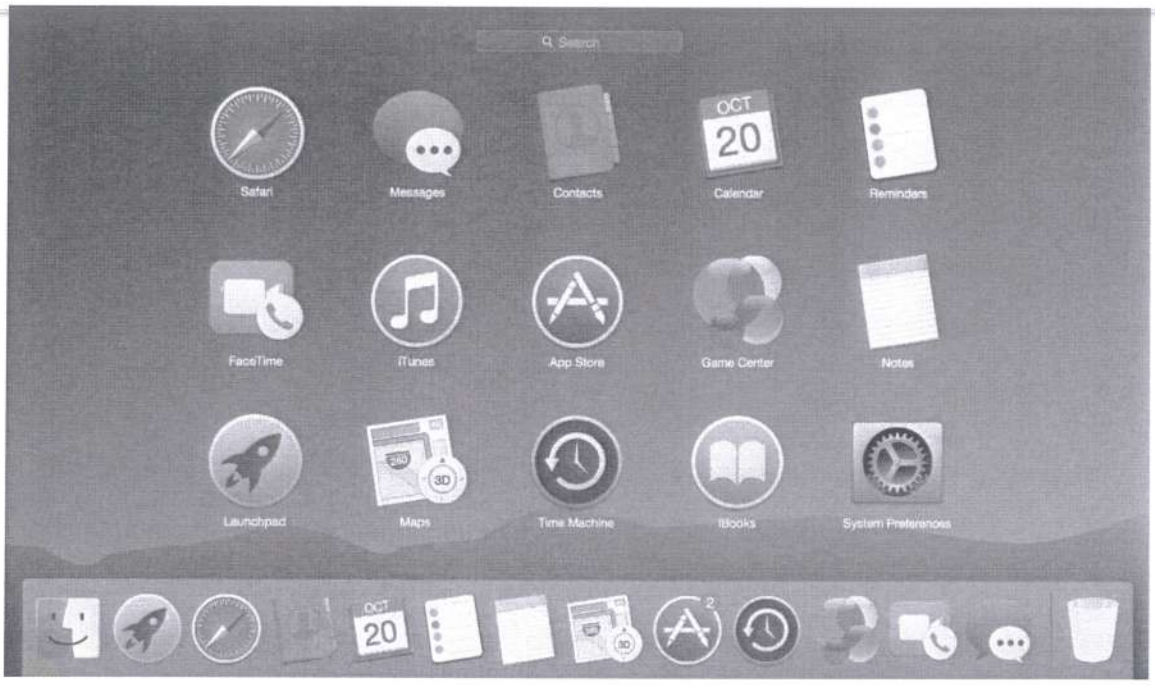


Fig – GUI Interface

- Event-driven Interaction: GUIs respond to user inputs (such as mouse clicks or keyboard input) and display real-time changes in response to actions
- Graphical Feedback: GUIs provide visual feedback to indicate the current status or progress of actions, such as progress bars or notifications
- Drag-and-Drop: Users can drag graphical elements (e.g., files or icons) and drop them into different locations or applications for easy file management and data transfer.

GUIs have revolutionized how people interact with computers and technology, enabling even non-technical users to perform complex tasks relatively quickly. They have become a standard interface paradigm in modern operating systems, software applications, and electronic devices, crucial in enhancing user

[illegible]
For Vivekananda Global University, Jaipur
Registrar

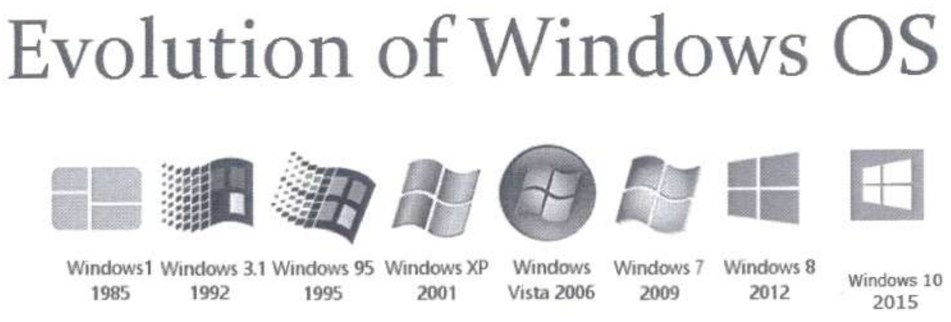
experience and productivity. GUIs have contributed to the widespread adoption and accessibility of computing devices, making technology more approachable and user-centric.

3.5 Windows Operating System

It is a widely used operating system developed by Microsoft Corporation. It provides a graphical user interface (GUI) and is designed to be user-friendly, making it accessible to a broad range of users. As of my last update in September 2021, Windows 11 is the latest version,

Here are some key Windows basics:

- Desktop: The desktop is the main screen of the Windows operating system. It is a workspace where you can place shortcuts to programs, files, and folders for easy access.
- Start Menu: The Start menu is the central hub for launching programs and accessing various system functions. It is located at the bottom-left corner of the desktop and contains a list of frequently used apps, live tiles (in Windows 10), and various system tools.
- Taskbar: The taskbar is at the bottom of the screen and provides quick access to running applications, system tray icons, and the Start menu. You can pin your favorite applications to the taskbar for easy access.



[illegible]

Fig – Windows Evolution

- File Explorer: File Explorer (formerly known as Windows Explorer) is the file management tool in Windows. It allows browsing and managing files and folders on your computer and connected storage devices.
- Control Panel: The Control Panel is a centralized location for configuring and customizing various system settings, devices, and applications. In newer versions of Windows, some settings are also accessible through the Settings app.
- Settings App: The Settings app is available in more recent versions of Windows and is designed to replace many functions of the Control Panel. It provides an easy-to-navigate interface for adjusting system settings and preferences.
- User Accounts: Windows supports multiple user accounts, allowing different individuals to have personalized settings and access to their files and applications. Administrators can manage user accounts and control access to the system.
- Installing and Uninstalling Programs: To install new software on Windows, you typically run an installer package (.exe or .mis) provided by the software's developer. To uninstall a program, you can use the "Add or Remove Programs" (Windows 7) or "Apps & Features" (Windows 10) feature in the Control Panel or Settings app, respectively.
- Updates: Windows regularly receives updates from Microsoft to improve security, fix bugs, and add new features. It's essential to keep your system up-to-date to ensure it remains secure and performs optimally.
- Shutting Down and Restarting: To shut down or restart your computer, click on the Start button, select the power icon, and then choose either "Shut down" or "Restart."

Advantages of Windows Operating System:

- User-Friendly Interface: Windows provides a user-friendly and intuitive interface, making it easy for beginners and experienced users to navigate and interact with the system.
- Broad Software Support: Windows enjoys a vast array of software and applications compatibility, offering users a wide selection of programs for various needs, including productivity, creativity, gaming, and more.
- Hardware Compatibility: Windows OS is designed to work with diverse hardware components, making it relatively easy to find compatible devices and peripherals.
- Regular Updates and Support: Microsoft regularly releases updates and security patches to improve functionality, fix bugs, and protect the system from potential vulnerabilities.
- Multimedia Capabilities: Windows is equipped with multimedia features that enable seamless video and audio playback, making it a suitable platform for entertainment purposes.
- Gaming Support: Windows is a favored platform for gaming enthusiasts, with extensive gaming libraries and compatibility with various gaming peripherals.
- Networking Capabilities: Windows provides robust networking support, facilitating easy connections to both local and internet-based networks.

Disadvantages of Windows Operating System:

- Security Vulnerabilities: Windows has been historically prone to security issues, making it a target for malware, viruses, and other cyber threats. Users must remain vigilant and keep their systems updated to stay protected.
- System Slowness: Windows systems may experience reduced performance over time and slow down due to software clutter, which requires regular maintenance and optimization.
- Cost: The Windows OS typically comes with a licensing cost, especially for premium editions, which might be a drawback for budget-conscious users.

- Bloatware: Some Windows versions come with pre-installed software (bloatware) that users may find unnecessary or unwanted, leading to reduced system performance.
- Privacy Concerns: Windows has faced criticism regarding data collection and privacy concerns. Users may need to configure settings to control the information shared with Microsoft.
- Regular Updates: While updates are essential for security and functionality improvements, some users might find frequent updates intrusive or time-consuming.
- Resource Intensive: The latest versions of Windows might require more system resources, including memory and processing power, which could be a challenge for older or low-end hardware.

3.6 Window Desktop

The desktop is the main screen that appears when you start up your computer. It is a workspace where you can place shortcuts, files, and folders for easy access. The desktop background or wallpaper is customizable, allowing you to set an image or solid colour as the backdrop.

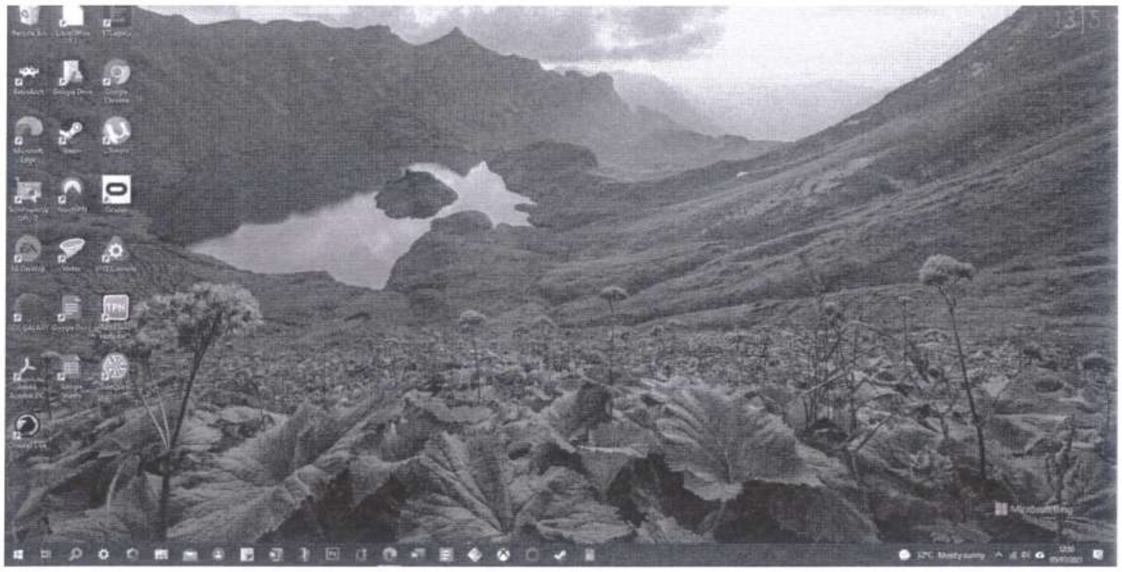


Fig – Desktop in a Windows operating system

For Vivekananda Global University, Jaipur
[illegible]
Registrar

On the desktop, you can perform various tasks such as:

- Creating new folders and files.
- Organizing files and folders by dragging and dropping them to different locations.
- Accessing frequently used applications through desktop shortcuts.
- Displaying gadgets (in earlier versions of Windows) or widgets to get quick information like time, weather, etc.
- Right-clicking to access a context menu with various options for managing files and folders.

3.7 Start Menu

The Start menu is a crucial component of the Windows operating system. In Windows 10 and previous versions, it is usually located at the screen's bottom-left corner, represented by the Windows icon. When you click on the Start button, it opens a menu with various options and features, including:

- A list of frequently used and pinned applications.
- A search bar to search for applications, files, and settings.
- Access to the Settings app for configuring system preferences.
- Power options for shutting down, restarting, or putting the computer to sleep.
- User account options, such as signing out or switching users.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

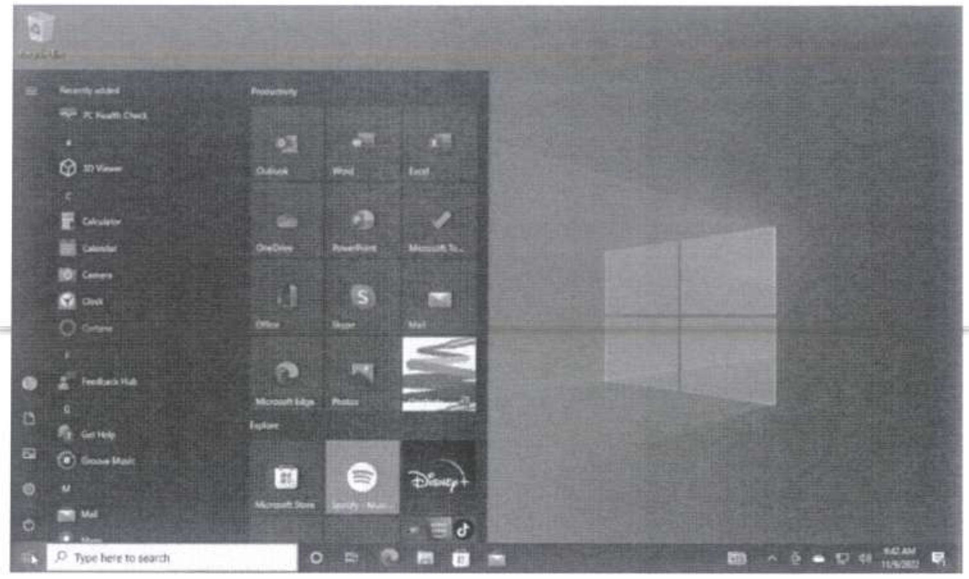


Fig – Start Menu in Windows 10 OS

In Windows 10 and newer versions, the Start menu may also include live tiles, and interactive app icons that display real-time information or updates from certain applications.

3.8 Icons

Icons are small graphical representations of applications, files, and folders. They are displayed on the desktop and within the Start menu, providing a visual way to access various operating system elements. Each icon represents a specific program or file, and clicking on an icon typically opens the associated application or file.

You can place shortcuts to your favourite applications or frequently used files and folders on the desktop by creating new shortcuts or dragging existing items from File Explorer.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

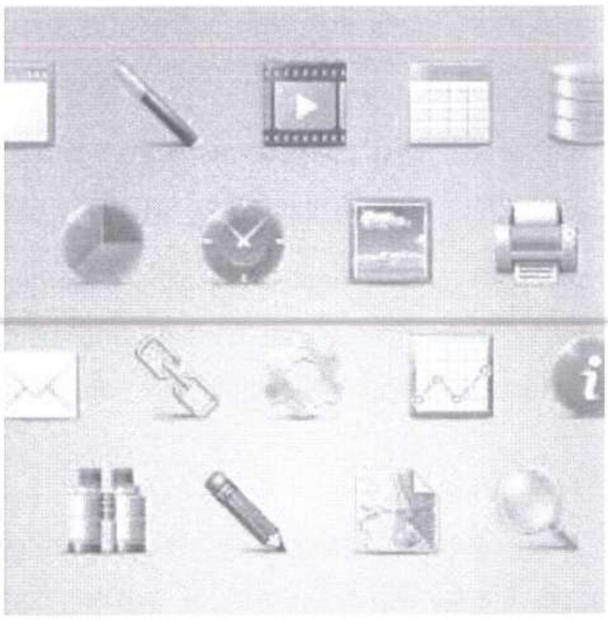


Fig – Icons in Windows OS

In the Start menu, icons are used to represent installed applications, allowing you to launch them with a single click. You can customize the Start menu by pinning or unpinning apps for quick access.

Benefits of Icons:

Visual Representation: Icons use graphical symbols and images to represent functions, applications, files, or settings. They provide a visual cue, making it easier for users to recognize and remember their purpose.

User-Friendly: Icons are generally more user-friendly, especially for novice computer users. Users can click on familiar icons to perform tasks without remembering specific commands or text.

Intuitive Navigation: Icons can serve as visual shortcuts, allowing users to quickly access frequently used programs or features without navigating through complex menus or remembering command names.

Ease of Discovery: Icons are often arranged on the desktop or within the start menu, making them easy to discover and access.

Touchscreen Compatibility: Icons are well-suited for touchscreen devices, as users can easily tap on the desired icon with their fingers.

Consistency: Icons are often used consistently across different applications and operating systems, providing a standardized user interface experience.

Overall, the desktop, Start menu, and icons are fundamental elements of the Windows user interface. They offer a user-friendly and efficient way to navigate and access various features and applications on your computer.

3.9 Summary

Computer memory is an essential component that allows computers to store and access data quickly. It is a fundamental part of modern computing, enabling the execution of programs and the storage of both temporary and permanent data. Computer memory is broadly categorized into two types: primary memory (also known as main memory) and secondary memory (also known as auxiliary memory).

Primary Memory (Main Memory):

Primary memory is directly accessible by the CPU (Central Processing Unit) and is critical in actively running applications and processing data. It is volatile, meaning its contents are lost when the computer is powered off. The two main types of primary memory are:

- RAM (Random Access Memory): RAM is the primary memory used by the computer to store data temporarily while the system is running. It allows the CPU to access and manipulate data quickly. RAM is characterized by its speed but limited capacity compared to secondary memory. It is classified into different generations based on technology, such as DDR (Double Data Rate) and DDR2, DDR3, DDR4, DDR5, etc.
- Cache Memory: Cache memory is a small, ultra-fast memory located close to the CPU. It stores frequently accessed data and instructions to reduce the time it takes for the CPU to fetch data from RAM. Cache memory comes in

multiple levels, such as L1, L2, and L3 cache, with L1 being the fastest and closest to the CPU.

Secondary Memory (Auxiliary Memory):

Secondary memory is non-volatile and serves as long-term storage for data that needs to be retained even when the computer is turned off. Unlike primary memory, secondary memory is slower but has much larger storage capacity. The main types of secondary memory include:

- Hard Disk Drive (HDD): HDDs are traditional storage devices that use magnetic disks to store data. They offer substantial storage capacity and are commonly used in computers for data storage.
- Solid State Drive (SSD): SSDs are newer storage devices that use flash memory to store data. They are faster, more reliable, and more power-efficient than HDDs, making them increasingly popular in modern computers.
- Optical Storage (e.g., CD/DVD/Blu-ray): Optical discs are read-only (CD-ROM, DVD-ROM) or writable (CD-R, DVD-R) media used for distributing software and data.
- USB Flash Drives: These portable storage devices use flash memory and are popular for transferring data between computers.

Windows OS:

Windows OS, developed by Microsoft, is one of the world's most widely used operating systems. It provides a graphical user interface (GUI) that allows users to interact with their computers easily. Windows OS supports a wide range of applications and hardware, making it suitable for both personal and business use. Over the years, Microsoft has released various versions of Windows, each with new features and improvements.

Windows Desktop:

For Vivekananda Global University, Jaipur
[illegible]
Registrar

The Windows Desktop is the primary graphical interface that appears when the operating system starts up. It serves as the central workspace for users to access and organize their files, folders, and applications. The desktop typically displays icons, shortcuts, and the taskbar, making it a convenient launching point for various activities.

Icons:

Icons are small graphical representations of files, folders, or applications. They appear on the Windows Desktop and provide an intuitive way for users to access and open their favourite programs or frequently used files. By clicking on an icon, users can quickly launch applications or open specific files, enhancing the overall usability and accessibility of the operating system.

Start Menu:

The Start Menu is a key feature of the Windows OS, accessible from the bottom-left corner of the desktop. It provides a comprehensive menu with installed applications, settings, and frequently used features. Users can easily search for applications, access system settings, shut down or restart the computer, and perform various other tasks through the Start Menu. With the introduction of Windows 10 and later versions, the Start Menu has evolved to incorporate live tiles, providing users with dynamic updates and interactive elements.

3.10 Review Questions

1 What is the primary purpose of primary memory in a computer system, and why is it essential for system operation?
2 Differentiate between RAM and Cache Memory, highlighting their respective roles in computer processing.
3 Explain the concept of volatility in memory and provide examples of volatile and non-volatile memory types.
4 How does the speed of primary memory (RAM) affect the overall performance of a computer?

[illegible]

5 Describe one practical scenario where secondary memory (e.g., SSD or HDD) is more suitable than primary memory (RAM).

6 Compare and contrast primary memory (RAM) and secondary memory (e.g., SSD or HDD) in terms of their characteristics, uses, and performance.

7 Discuss the importance of cache memory in modern computer systems, explaining how it enhances CPU performance and reduces memory latency.

8 Explain the memory hierarchy in a computer system, incorporating primary memory (RAM), cache memory, and secondary memory (e.g., SSD or HDD), and their respective roles.

9 Describe the process of virtual memory management and how it allows a computer system to utilize both primary and secondary memory effectively.

10 Evaluate the advantages and disadvantages of using solid-state drives (SSDs) compared to traditional hard disk drives (HDDs) as secondary memory devices.

3.11 Keywords

- Computer Memory: The essential computer system component that stores data, instructions, and programs for processing and retrieval.
- Primary Memory: The main memory that directly interacts with the CPU and provides temporary storage for data and instructions during program execution (e.g., RAM, Cache Memory).
- Secondary Memory: Non-volatile memory used for long-term data storage and retrieval, separate from the CPU, with larger capacity but slower access times (e.g., SSD, HDD, optical storage).
- RAM (Random Access Memory): A type of primary memory that allows the CPU to access and modify data quickly, essential for running applications and the operating system.
- Cache Memory: A small, high-speed memory located close to the CPU, used to store frequently accessed data for faster processing.

- Volatile Memory: Memory loses its contents when the power is turned off, like RAM.
- Non-Volatile Memory: Memory that retains its contents even when the power is turned off, like SSD and HDD.
- Virtual Memory: A technique used to extend the effective size of the primary memory by utilizing secondary storage as an extension of RAM.
- Memory Hierarchy: The organization of memory into different levels, with varying access speeds and capacities, to optimize overall system performance.
- Memory Management Unit (MMU): Hardware maps virtual memory addresses to physical memory locations, facilitating access and protection.
- GUI (Graphical User Interface): A visual interface that allows users to interact with computers and software through graphical elements like icons, windows, and menus.
- Windows OS: Microsoft's operating system with a GUI, used widely for personal and business computing.
- Start Menu: A central menu in Windows OS, providing access to applications, settings, and system functions.
- Icon: A small graphical representation on the desktop used for easy access and launching of files, folders, or applications.

3.12 References

1. E.Balaguruswamy, "Computing Fundamentals & C Programming", Tata McGraw Hill, 2008.
2. B. A. Forouzan & R. F. Gilberg "Computer Science – A structured programming Approach Using C", 2011
3. E. Bala Guruswamy, "Programming in ANSI" Tata McGraw Hill, 2011.

[illegible]

For Vivekananda Global University, Jaipur
Registrar

Table of Content

[illegible]

Learning Objectives

After studying this unit, the student will be able to:

- Gain a comprehensive understanding of the basic concepts and syntax of the C programming language, including variables, data types, operators, and control structures.
- Learn to write simple C programs, including input and output operations, arithmetic calculations, and conditional statements.
- Gain proficiency in identifying and resolving common errors in C programs, enhancing code reliability.
- Explore techniques for writing efficient and optimized C code, focusing on performance improvement.
- Apply C fundamentals to design and develop small-scale programs to solve real-world problems and enhance problem-solving skills.

[illegible]

For Vivekananda Glob... Registrar

Introduction

C is a general-purpose programming language that was developed in the early 1970s by Dennis Ritchie at Bell Laboratories. It has since become one of the most widely used programming languages and has influenced the development of many other languages, including C++, Java, and C#.

C is known for its efficiency, flexibility, and low-level programming capabilities. It is commonly used for system programming, embedded systems, and developing operating systems. C is also a popular choice for developing applications that require high performance, such as game engines and scientific simulations.

Here are some key features and concepts of C programming:

- Syntax: C has a relatively simple syntax compared to some other programming languages. It uses a combination of keywords, variables, data types, operators, and control structures to write programs.
- Variables and Data Types: You declare variables to store data in C. Various data types are available, including integers, floating-point numbers, characters, and more. You can also define your own data types using structures and unions.
- Functions: C allows you to define functions to perform specific tasks. Functions help in organizing code and making it more modular. They can be reusable and can accept parameters and return values.
- Control Structures: C provides control structures such as if-else statements, loops (like for and while), and switch statements to control the flow of execution in a program.
- Pointers: Pointers are a powerful feature of C. They allow you to manipulate memory directly and work with addresses of variables and data structures. Pointers enable efficient memory management and the ability to work with complex data structures.
- Arrays and Strings: C supports arrays, which are collections of elements of the same type. Arrays are useful for storing and manipulating multiple

values. C also treats strings as arrays of characters and provides functions to manipulate them.

- Input and Output: C provides functions for input and output operations. Using standard library functions, you can read input from the keyboard and write output to the screen or files.
- Preprocessor Directives: C has a preprocessor that processes the source code before compilation. Preprocessor directives, denoted by the '#' symbol, allow you to include header files, define constants, and perform other preprocessing tasks.

To start programming in C, you would typically write your code in a text editor and save it with a .c extension. You would then compile the code using a C compiler, which translates the code into machine-readable instructions. The compiled code can be executed to produce the desired output.

Many C compilers, such as GCC (GNU Compiler Collection), Clang, and Microsoft Visual C++, are available. You can choose a compiler based on your operating system and personal preference.

Learning C can provide a strong foundation for understanding programming concepts and open up various software development and system programming opportunities.

4.1 History of C

The history of C programming dates back to the early 1970s when Dennis Ritchie created it at Bell Laboratories. Here's a brief timeline of the major milestones in the history of C:

- 1969: The development of C began as a successor to the B programming language, which Ken Thompson created. Dennis Ritchie wanted to improve upon B's capabilities and efficiency.
- 1972: The C programming language was developed primarily to support the development of the Unix operating system. Ken Thompson and Dennis

[illegible]
For Vivekananda Global University, Jaipur
Registrar

Ritchie rewrote Unix in C, which helped in porting the operating system to different computer architectures easily

- 1978: The first edition of "The C Programming Language," also known as the "K&R C," was published. 90Brian Kernighan and Dennis Ritchie wrote it, becoming the authoritative reference for C programming.
- 1983: The American National Standards Institute (ANSI) formed a committee to establish a standard for the C language. The committee, known as X3J11, worked to define the ANSI C standard. The resulting standard, called ANSI C or C89, was published in 1989.
- 1990: The International Organization for Standardization (ISO) adopted the ANSI C standard with some modifications and released it as ISO/IEC 9899:1990. This version is commonly referred to as C90.
- 1999: The ISO/IEC 9899:1999 standard, also known as C99, was released. C99 introduced several new features to the language, including support for variable-length arrays, inline functions, and improved support for comments
- 2011: The ISO/IEC 9899:2011 standard, known as C11, was published. C11 introduced additional features such as multi-threading support, improved Unicode support, and new library functions.

Since then, the C language has continued to evolve, with the most recent standard being C18, released in 2018. However, it's important to note that C programming is generally backward compatible, meaning that programs written in earlier versions of C should still work in newer compilers.

C's simplicity, efficiency, and low-level capabilities have made it a widely used language for system programming, embedded systems, and performance-critical applications. It has also influenced the development of numerous other programming languages, including C++, Java, and C#. The widespread adoption of C has contributed to its longevity and popularity among programmers.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

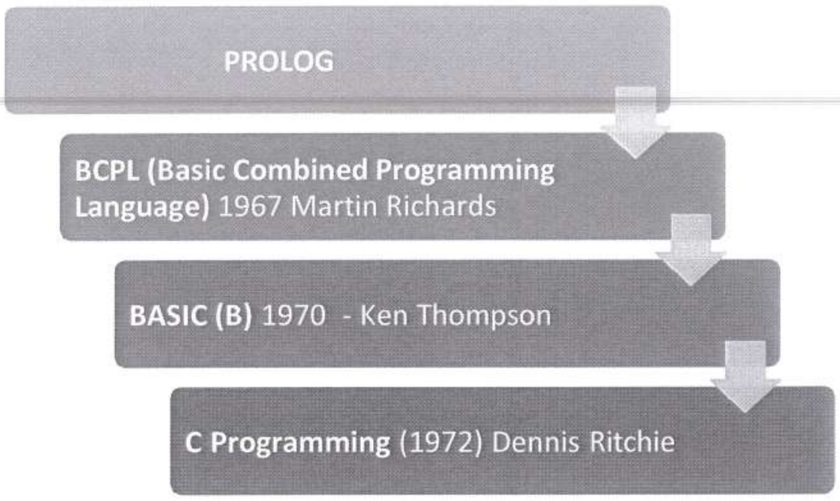

PROLOG
BCPL (Basic Combined Programming Language) 1967 Martin Richards
BASIC (B) 1970 - Ken Thompson
C Programming (1972) Dennis Ritchie

Fig - Evolution of C

Despite the emergence of newer languages, C remains an important language in the field of software development and continues to be widely used today. Many operating systems, libraries, and applications are written in C, making it an essential language for aspiring programmers to learn.

- C is a general-purpose programming language.
- It was developed at AT& T's Bell Laboratories of USA in 1972.
- It was designed and written by a man named Dennis Ritchie.
- Its simplicity, reliability, easy to use, and ease of learning are the major reasons we learn C language.
- C language is case-sensitive, i.e. it can distinguish between lower case letters (a, b, c...) and upper case letters (A, B, C).
- C has facilities for structured and procedural programming.
- C is one of the most widely used programming languages of all time.
- There are very few computer architectures for which a C compiler does not exist.

[illegible]
For Vivekananda Global University, Jaipur
Registrar

- C does not include some features found in newer, more modern high-level languages, including object orientation and garbage collection.

4.2 Structure of C Program

Syntax of C program

- Syntax: The syntax of a language describes the possible combinations of symbols that form a correct program.

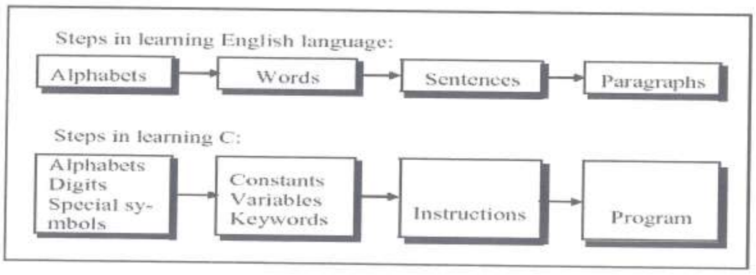

Steps in learning English language:
Alphabets
Words
Sentences
Paragraphs
Steps in learning C:
Alphabets Digits Special symbols
Constants Variables Keywords
Instructions
Program

Fig – Steps in learning C

Preprocessor Directive:

In C programming, a preprocessor directive is a command that instructs the compiler to perform specific actions before the actual compilation of the source code begins. Preprocessor directives start with the '#' symbol and are processed by the preprocessor, a built-in component of the C compiler. They are not part of the C language itself, but they are essential for performing various tasks before the actual code is compiled.

Common preprocessor directives include:

#include: Used to include header files into the source code, allowing the program to access functions and declarations defined in those header files.

#define: Used to define constants, macros, or function-like macros for use throughout the program.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

#ifdef, #ifndef, #else, #endif: Used for conditional compilation, allowing certain code blocks to be included or excluded based on predefined conditions.

#error: Used to generate a compilation error with a specified error message.

#pragma: Provides additional instructions to the compiler for specific purposes, such as compiler-specific optimizations or configurations.

Header Files:

Header files contain function prototypes, data type declarations, macro definitions, and other relevant information that allows the C program to access functions and features defined in external libraries or other source files. Header files have a '.h' extension and are typically included in C programs using the '#include' preprocessor directive

Commonly used standard C library header files include:

stdio.h: Contains standard input/output functions like 'print()' and 'scan()'.

styli: Provides functions for dynamic memory allocation, random number generation, and other utility functions.

math's: Contains mathematical functions like 'sqrt()', 'sin()', 'cos()', and others.

string.h: Contains functions for string manipulation, such as 'strcpy()', 'strcat()', 'strlen()', and more.

Header files play a crucial role in organizing and modularizing code, allowing developers to easily reuse functions and declarations from external sources. They provide a way to separate interface declarations from the implementation details of functions, promoting better code organization and maintainability in C programs.

Tokens

[illegible]

For Vivekananda Global University, Jaipur

Registrar

In C programming language, tokens are the basic building blocks of a program. Tokens can be classified into several categories, and one important category is identifiers.

Identifiers:

- Identifiers are names used to identify variables, functions, types, labels, etc.
- Rules for identifiers in C:
- Must start with a letter (a-z or A-Z) or underscore (_).
- Can be followed by letters, digits (0-9), or underscores.
- Cannot be a keyword.
- Case-sensitive (e.g., "my Variable" and "myvariable" are different identifiers).

Keywords:

- Keywords are reserved words with predefined meanings in the C language. Total 32 Keywords in C (Discussed later)
- Examples: if, else, for, while, int, char, etc.

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

Fig - List of 32 C keywords

Constants:

- Constants represent fixed values that cannot be modified during the execution of a program.

[illegible]
For Vivekananda Global University, Jaipur
Registrar

- Examples: integer constants (e.g., 123, 0xFF), floating-point constants (e.g., 3.14, 1.0e-5), character constants (e.g., 'A', '\n'), string constants (e.g., "Hello", "C programming"), etc.

Operators and Punctuators:

- Operators perform various operations on operands.
- Punctuators are symbols with special meanings in the C language.
- Examples: +, -, *, /, =, ==, ++, --, ,, ;, (,), {, }, etc.
- String literals:

String literals represent sequences of characters enclosed in double-quotes.

Example: "Hello, World!".

Comments:

- Comments are used to add explanatory notes or disable code segments.
- Single-line comments start with //, and multi-line comments start with /*and end with */.
- These are some of the important token categories in C programming language.

Main () :

The main() function has the following characteristics:

- Return Type: The main() function has a return type of int, which indicates the status of program execution. A return value of 0 indicates successful execution, while non-zero values indicate some error or abnormal termination.
- Function Name: The function name must be main. It is case-sensitive, so Main or MAIN will not work.
- Parameters: The main() function can accept two parameters: argc and argv. These parameters are used to pass command-line arguments to the program. The argc parameter represents the count of command-line arguments, and argv is an array of strings containing the actual arguments.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Note that you can omit these parameters if you don't need to process command-line arguments.

- Body: The body of the main() function contains the statements and code that make up the program. Here, you can write the logic and functionality of your program using variable declarations, control structures, function calls, and more.
- Return Statement: The main() function usually ends with a return statement. A return value of 0 is commonly used to indicate successful program execution, while non-zero values can convey specific error codes or exceptional conditions. The return statement is optional; if omitted, the main() function implicitly returns 0 by default.

An example of a C program with the main() function:

```
#include <stdio.h>
int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

In this example, the program starts executing from the main() function, which prints "Hello, world!" to the console using the printf() function. Finally, the return 0; statement indicates successful execution, and the program terminates.

Understanding the structure and purpose of the main() function is essential for writing and running C programs. It allows you to control the flow of execution and define the behaviour of your program.

4.3 Constants and Variables

Constants

- Constants are those data whose value cannot be changed.
  Eg.100, 1235, 15.24, 6.3, A, G, help etc.

[illegible]
For Vivekananda Global University, Jaipur
Registrar

$$\begin{tikzcd}
 & \text{Constant} \arrow[dl] \arrow[dr] & & \\
\text{Numeric Constant} \arrow[d] \arrow[dr] & & \text{Character Constant} \arrow[d] \arrow[dr] & \\
\textbf{Without Decimal} & \textbf{With Decimal} & \textbf{Single Character} & \textbf{Multiple Characters}
\end{tikzcd}$$

Eg.1, 2, 3, 56 ...	1.25, 356.24	A, a, G, j, k	EEE, me	
(Integers)	(Floating Point Numbers)	(Character)	(String)	
int	float	char	$\longrightarrow$	*DATA TYPES*
%d	%f	%c	$\longrightarrow$	*FORMAT SPEC*

Note :- We will talk about string later on.

In C, constants are values that cannot be changed during the execution of a program. They provide fixed values that are used in calculations, assignments, and comparisons. Here are different types of constants in C, along with examples:

- *Integer Constants:*
  - Integer constants are whole numbers without a fractional part. They can be specified in decimal, octal, or hexadecimal format.
  - Decimal: 42, 100, -10
  - Octal: 052 (decimal value 42), 0777 (decimal value 511)
  - Hexadecimal: 0x2A (decimal value 42), 0xFF (decimal value 255)
- *Floating-Point Constants:*
  - Floating-point constants represent real numbers with a fractional part. They can be written in decimal or scientific notation.
  - Decimal: 3.14, -0.5, 1.0
  - Scientific Notation: 2.5e-3 (2.5 x 10^-3), 6.022e23 (Avogadro's number)
- *Character Constants:*
  - Character constants represent individual characters enclosed in single quotes.

- Example: 'A', 'b', '5', '?'
- *String Constants:*
  - String constants are sequences of characters enclosed in double quotes. They represent a collection of characters or a string of text.
  - Example: "Hello", "C Programming", "123"
- *Symbolic Constants:*
  - Symbolic constants are identifiers that represent fixed values. They are defined using the #define preprocessor directive.
  - Example: #define PI 3.1415, #define MAX_VALUE 100
- *Enumeration Constants:*
  - Enumeration constants are identifiers that represent a set of named integer values. They are defined using the Enum keyword.
  - Enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };
- *Constant Expressions:*
  - Constant expressions are arithmetic expressions involving constants that are evaluated at compile time.
  - Example: 2 + 3, 5 * (7 - 2)
  - Constants in C provide fixed values that remain unchanged during the execution of a program. They are useful for defining values that should not be modified and for improving code readability by giving meaningful names to fixed values.

```
#include <stdio.h>
#define PI 3.1415
int main()
{
    const int maxCount = 10;
    const char message[] = "Welcome!";
    enum Month { JANUARY, FEBRUARY, MARCH };
    printf("The value of PI is: %.2f\n", PI);
    printf("The maximum count is: %d\n", maxCount);
    printf("The message is: %s\n", message);
    printf("The month is: %d\n", MARCH);
    return 0;
```

For Vivekananda Global University, Jaipur
[illegible]
Registrar

}

PI is defined as a symbolic constant using the #define directive in this example. maxCount is a constant integer variable defined using the const keyword. message is a constant character array, and MARCH is an enumeration constant. These constants are used within the main() function to print their respective values.

Variables

In C, variables are used to store and manipulate data during the execution of a program. They have a specific data type, a name, and a memory location. Here's an overview of variables in C:

- *Variable Declaration:*
    - Before using a variable must be declared with a specific data type before using it. The declaration specifies the variable's name and the type of data it can hold.
    - Syntax: datatype variable_name;
    - Example: int age;, float salary;, char grade;
- *Variable Initialization*:
    - Variables can be initialized with an initial value at the time of declaration.
    - Syntax: datatype variable_name = value;
    - Example: int age = 25;, float salary = 5000.50;, char grade = 'A';
- *Assigning Values to Variables:*
    - After declaration, variables can be assigned values using the assignment operator (=).
    - Example: age = 30;, salary = 6000.75;, grade = 'B';
- *Data Types:*
    - C provides various data types, including:
    - Integers: int, short, long, unsigned int, etc.
    - Floating-point numbers: float, double
    - Characters: char
    - Boolean: bool (in C99 and later)

[illegible]

    - User-defined types: struct, Enum, etc.

- *Scope:*
    - Variables have a scope that determines where they can be accessed within a program.
    - The scope of a variable can be at the global level (accessible throughout the program) or at the local level (accessible within a specific block of code, such as a function).
- *Rules for Naming Variables:*

  Variable names in C must follow specific rules:

    1. Must start with a letter or underscore
    2. Can contain letters, digits, and underscores
    3. Must not be a reserved keyword
    4. Case-sensitive (e.g., age and Age are different variables)
- *Constants:*
    - Constants are fixed values that do not change during program execution. They are declared using the const keyword.
    - Example: const int MAX_VALUE = 100;
- *Storage Class Specifiers:*
    - C provides storage class specifiers to define variables' storage duration and scope. Examples include auto, static, extern, and register.

An example that demonstrates the usage of variables in C:

```
#include <stdio.h>
int main()
{
    int age = 25;
    float salary = 5000.50;
    char grade = 'A';
    printf("Age: %d\n", age);
    printf("Salary: %.2f\n", salary);
    printf("Grade: %c\n", grade);
    return 0;
}
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

In this example, variables age, salary, and grade are declared and initialized with values. The printf() function is then used to display the values of these variables on the console.

Variables in C allow storing and manipulating data dynamically during program execution. They are a fundamental concept in programming and play a crucial role in implementing algorithms and solving problems.

4.4 Input Output Functions

In C, input/output (I/O) operations allow you to interact with the user, display information on the screen, or read from/write to files. Here's an overview of simple I/O operations in C:

- Output Operations:
  - print(): Used to display formatted output on the console.
  - Example: print("Hello, World!\n");
- Input Operations:
  - scan(): Used to read input from the user or from a file.
  - Example: scan("%d", &age);
- Standard I/O Files:
  - stdin, stout, and stderr: These are predefined file pointers representing standard input (keyboard), standard output (screen), and standard error (screen for error messages).
  - Example: print(stdout, "This is a message.\n");
- File Operations:
  - FILE Data Type: Used to declare file pointers to read from/write to files.
    - Example: FILE *file_ptr;
  - fopen(): Used to open a file.
    - Example: file_ptr = fopen("filename.txt", "r");
  - fclose(): Used to close a file.
    - Example: fclose(file_ptr);

[illegible]

- fscanf(): Used to read formatted data from a file.
    - Example: fscanf(file_ptr, "%d", &num);
- fprintf(): Used to write formatted data to a file.
    - Example: fprintf(file_ptr, "The result is %d\n", result);

Here's a simple example that demonstrates the usage of I/O operations in C:

```
#include <stdio.h>
int main()
{
    int age;
    float salary;
    printf("Enter your age: ");
    scanf("%d", &age);
    printf("Enter your salary: ");
    scanf("%f", &salary);
     printf("You are %d years old and your salary is
     %.2f\n", age, salary);
    return 0;
}
```

In this example, the program prompts the user to enter their age and salary using printf(). The values are then read from the user using scanf() and stored in age and salary variables. Finally, the program displays the values using printf().

Note: Make sure to include the <stdio.h> header file at the beginning of your program to access the input/output functions.

I/O operations in C are essential for interacting with users, reading input, and writing output. They provide the means to create interactive programs and handle data from various sources.

Commonly used *escape sequences* are:

- \n (newline)
- \t (tab)
- \v (vertical tab)
- \f (new page)

[illegible]

- \b (backspace)
- \r (carriage return)

4.5 Operators

Operators and expressions are fundamental for performing calculations, comparisons, and other operations in C. They enable you to manipulate variables and constants to create complex computations and decide based on conditions.

C language supports various types of operators, which can be broadly categorized as follows:

1. **Arithmetic Operators:**
    - Addition (+): Performs Addition between operands.
    - Subtraction (-): Performs subtraction between operands.
    - Multiplication (*): Performs multiplication between operands.
    - Division (/): Performs division between operands.
    - Modulus (%): Calculates the remainder of the division between operands.
2. **Relational Operators:**
    - Equal to (==): Checks if two operands are equal.
    - Not equal to (!=): Checks if two operands are not equal.
    - Greater than (>): Checks if the left operand is greater than the right operand.
    - Less than (<): Checks if the left operand is less than the right operand.
    - Greater than or equal to (>=): Checks if the left operand is greater than or equal to the right operand.
    - Less than or equal to (<=): Checks if the left operand is less than or equal to the right operand.
3. **Logical Operators:**
    - Logical AND (&&): Performs a logical AND operation between two expressions.
    - Logical OR (||): Performs a logical OR operation between two expressions.
    - Logical NOT (!): Negates the result of a logical expression.
4. **Assignment Operators:**

[illegible]

- Assignment (=): Assigns the value of the right operand to the left operand.
- Add and assign (+=): Adds the right operand to the left operand and assigns the result to the left operand.
- Subtract and assign (-=): Subtracts the right operand from the left operand and assigns the result to the left operand.
- Multiply and assign (*=): Multiplies the left operand by the right operand and assigns the result to the left operand.
- Divide and assign (/=): Divides the left operand by the right operand and assigns the result to the left operand.
- Modulus and assign (%=): Calculates the Modulus of the left operand with the right operand and assigns the result to the left operand.

5. **Increment and Decrement Operators:**
   - Increment (++): Increases the value of the operand by 1.
   - Decrement (--): Decreases the value of the operand by 1.
6. **Bitwise Operators:**
   - Bitwise AND (&): Performs bitwise AND operation between operands.
   - Bitwise OR (|): Performs bitwise OR operation between operands.
   - Bitwise NOT (~): Performs bitwise NOT operation on the operand, flipping all bits.
   - Bitwise XOR (^): Performs bitwise exclusive OR (XOR) operation between operands.
   - Left Shift (<<): Shifts the bits of the left operand to the left by the number of positions specified by the right operand.
   - Right Shift (>>): Shifts the bits of the left operand to the right by the number of positions specified by the right operand.

Here's a detailed understanding of operators and expressions in C:

- *Arithmetic Operators:*

  Addition (+), Subtraction (-), Multiplication (*), Division (/), Modulus (%), and Increment (++) or Decrement (--).

  Example: int result = a + b;, float quotient = x / y;, counter++;

Eg. +, -, /, *, % (modulus / remainder operator)

int a=4, b=10;

- c=a+b;

  c=14
- c=12%2;

  c=0
- c=13%2;

  c=1

Precedence	Associativity
*, /, %	Left to Right
+, -	Left to Right

- *Relational Operators:*

  Used for comparisons. They evaluate to a Boolean value (0 for false, 1 for true).

  Equal to (==), Not equal to (!=), Greater than (>), Less than (<), Greater than or equal to (>=), Less than or equal to (<=).

  Example: if (a == b) { /* code */ }, if (x > y) { /* code */ }

  Eg.>, <, >=, <=, == (equals to), != (not equals to)

  A relational operator always gives results in 1(for true expression) or 0(for false expression). For true expressions, the value might be 1 or a non-zero value.

  Eg. a = 4, b = 10

  c = a < b

  c = 4 < 10

  c = 1

  c = a > b = 0

  Note:- "=" is the assignment operator, and "==" is equal to operator.

  a = 4, b = 4

[illegible]

For Vivekananda Global University, Jaipur

Registrar

c = a == b

c = 4 == 4

c = 1 (i.e. true)

c = a != b (a not equals to b)

c = 4 != 4

c = 0 (i.e. false)

- *Logical Operators:*

  Used to combine and manipulate logical expressions. They evaluate a Boolean value (0 for false, 1 for true).

  AND (&&), OR (||), NOT (!).

  Example: if (condition1 && condition2) { /* code */ }, if (condition1 || condition2) { /* code */ }

  These operators are used to combine two or more conditions.

  Eg. && - AND operator

  || - OR operator

  ! – NOT operator

  (i) <u>AND operator</u> (multiply)

  a = 4, b = 5, c = 10

  d = a>b && b<c

  = 4>5 && 5<10

  = 0 && 1

  = 0 (i.e. False)

  (ii) <u>OR operator</u>(addition)

  a = 4, b = 5, c = 10

  d = a>b || b<c

  = 0 || 1

  = 1 (i.e. True)

  (iii) <u>NOT operator</u>(complement of the result)

  0 -> 1

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
1 -> 0
b = 10
c = !(b>12)
   c = !(10>12)
   c = !(0)
   c = 1
```

- *Assignment Operators:*

  Used to assign values to variables.

  Assignment (=), Addition assignment (+=), Subtraction assignment (-=), Multiplication assignment (*=), Division assignment (/=), Modulus assignment (%=).

  Example: x = 10;, y += 5;

- *Conditional Operator:*

  Also known as the ternary operator. It is a shorthand way to write if-else statements.

  Syntax: condition ? expression1 : expression2

  Example: max = (a > b) ? a : b;

  Unary operators are those operators which require only one operand.

  Eg. ++, --, !, sizeof() etc.

  Binary operators are those operators which require two operands.

  Eg. +, -, /, *, %, <=, >=, <, > etc.

  Ternary operators are those operators which require three operands.

  Syntax:

  ```
  Condition ? expression1:expression2
  ```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

If the Condition is true, expression1 will be executed and in case of false expression2 will be executed.

Eg.

(i) a=10, b=20
d= (a>b)? a: b

(10>20) is true, so a will be executed.

Therefore, d = a = 10.

(ii) a=10, b=20, c=30
d=(a>b && b<c)?23:42
(a>b && b<c) is false, therefore
d=42.

- *Bitwise Operators:*

  Used for manipulating individual bits in integral types.

  Bitwise AND (&), Bitwise OR (|), Bitwise XOR (^), Bitwise complement (~), Left Shift (<<), Right Shift (>>).

  Example: result = a & b;, value = ~mask;

- Other Operators:trfgdsszgf …/,

  Address-of (&), Dereference (*), Sizeof (sizeof), Comma (,), etc.

  It is a unary operator. It returns the memory size of a variable.

  Eg.int a, b;

  b = sizeof(a);

  Output: b = 2 (an integer have 2 bytes of memory space).

  Similarly,

  float a;

  int x;

[illegible]

x=sizeof(a);

Output: x = 4

Expressions in C are combinations of variables, constants, and operators that produce a value. They can be used in assignments, calculations, and control flow statements.

Here's an example demonstrating the usage of operators and expressions in C:

```
#include <stdio.h>
int main()
{
    int a = 5, b = 3;
    int sum = a + b;
    int product = a * b;
    int quotient = a / b;
    int remainder = a % b;
    printf("Sum: %d\n", sum);
    printf("Product: %d\n", product);
    printf("Quotient: %d\n", quotient);
    printf("Remainder: %d\n", remainder);
    return 0;
}
```

Arithmetic operators perform addition, multiplication, division, and Modulus operations in this example. The results are then displayed using printf().

Operators and expressions provide the foundation for manipulating data and making decisions in C programming. Understanding their usage is crucial for writing effective and efficient code.

4.6 Operator Precedence & Associativity

In C, operators have different precedence and associativity, which determine the order in which they are evaluated when multiple operators are present in an expression. Precedence refers to the priority of operators, while associativity determines the order of evaluation for operators with the same precedence. Here's an overview of the precedence and associativity of some common operators in C:

Operators with higher precedence are evaluated before operators with lower precedence.

Example: Multiplication (*) has higher precedence than Addition (+), so 2 + 3 * 4 is evaluated as 2 + (3 * 4).

Operator Associativity:

Associativity determines the order of evaluation when multiple operators of the same precedence appear consecutively.

- Left Associativity: Operators are evaluated from left to right.
- Example: Addition (+) has left associativity, so 2 + 3 + 4 is evaluated as (2 + 3) + 4.
- Right Associativity: Operators are evaluated from right to left.
- Example: Exponentiation (^) in C doesn't exist natively, but if it were defined with right associativity, 2 ^ 3 ^ 4 would be evaluated as 2 ^ (3 ^ 4).

Table - operator Precedence and Associativity

Precedence	Associativity
1. Unary operators Eg. ++, --, sizeof(), ! etc.	Right to left
2. Arithmetic operators Eg. +, -, /, *, %	Left to Right
3. Relational operators Eg. >, <, >=, <= etc.	Left to Right
4. Logical operators Eg. &&, ||	Left to Right

Evaluation of Expressions

In C, expressions are evaluated based on the precedence and associativity of operators. The evaluation follows specific rules to determine the order in

[illegible]

which operators and operands are processed. Here's an overview of how expressions are evaluated in C:

1. **Parentheses Evaluation:**

   Expressions within parentheses are evaluated first. The expressions inside parentheses are treated as sub-expressions and are evaluated independently.

   Example: (2 + 3) * 4 is evaluated by first evaluating 2 + 3 as 5, then multiplying the result by 4 to get 20.

2. **Unary Operators Evaluation:**

   Unary operators, such as unary plus (+), unary minus (-), increment (++), decrement (--), logical NOT (!), and bitwise NOT (~), are evaluated next. They operate on a single operand.

   Example: -5 evaluates to -5, ++counter increments the value of counter by 1.

3. **Multiplicative Operators Evaluation:**

   Multiplication (*), division (/), and Modulus (%) operators are evaluated next. They perform multiplication, division, and remainder calculations on operands.

   Example: 6 / 3 evaluates to 2, 5 % 2 evaluates to 1.

4. **Additive Operators Evaluation:**

   Addition (+) and subtraction (-) operators are evaluated next. They perform addition and subtraction operations on operands.

   Example: 2 + 3 evaluates to 5, 10 - 4 evaluates to 6.

5. **Relational and Equality Operators Evaluation:**

   Relational operators (<, >, <=, >=) and equality operators (==, !=) are evaluated next. They compare operands and produce Boolean results (0 for false, 1 for true).

[illegible]

Example: 4 < 7 evaluates to 1, 2 == 2 evaluates to 1.

6. **Logical Operators Evaluation:**

   Logical AND (&&) and logical OR (||) operators are evaluated next. They perform logical operations on Boolean operands and produce Boolean results.

   Example: 1 && 0 evaluates to 0, 1 || 0 evaluates to 1.

7. **Assignment Operators Evaluation:**

   Assignment operators (=, +=, -=, *=, /=, %=) are evaluated next. They assign values to variables based on the right-hand side of the assignment.

   Example: x = 5 assigns the value 5 to variable x, y += 3 increments the value of y by 3.

8. **Comma Operator Evaluation:**

   The comma operator (,) is evaluated last. It evaluates all expressions separated by commas and returns the value of the rightmost expression.

   Example: x = (a = 3, b = 4, a + b) assigns the value 7 to variable x by evaluating a = 3, b = 4, and a + b.

It's important to note that operators with higher precedence are evaluated before operators with lower precedence. Within the same precedence level, the associativity of the operators determines the evaluation order.

Here's an example demonstrating the evaluation of an expression in C:

```
#include <stdio.h>
int main() {
    int a = 2, b = 3, c = 4;
    int result = a + b * c / 2;
    printf("Result: %d\n", result);
    return 0;
}
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

This example evaluates the expression a + b * c / 2 based on operator precedence and associativity. Multiplication (*) and division (/) have higher precedence than Addition (+), so the expression is evaluated as a + ((b * c) / 2), resulting in a value of 8.

Understanding the order of evaluation is crucial for writing correct and predictable expressions in C.

4.7 Type Conversion in C

In C, type conversion, also known as type casting, is the process of converting a value from one data type to another. It allows you to change the interpretation of data to perform operations that may not be allowed by default due to type mismatch. Here's an overview of type conversion in C:

- *Implicit Type Conversion*
    - Implicit type conversion occurs automatically by the compiler when assigning a value of one type to a variable of another compatible type.
    - Example: Assigning an int value to a float variable: int a = 5; float b = a;
- *Explicit Type Conversion (Type Casting):*
    - The programmer performs Explicit type conversion using type cast operators to explicitly convert a value from one type to another.
    - Syntax: type name(expression)
    - Example: Converting a float value to an int: float x = 3.14; int y = (int)x;
- *Widening Conversion:*
    - Widening conversion, also known as upcasting, involves converting a value from a smaller data type to a larger data type.
    - Example: Converting an int to a double: int a = 5; double b = (double)a;
- *Narrowing Conversion:*
    - Narrowing conversion, also known as down casting, involves converting a value from a larger data type to a smaller data type.
    - Example: Converting a double to an int: double x = 3.14; int y = (int)x;
- *Loss of Precision:*
    - When performing narrowing conversions, there may be a loss of precision

[illegible]
For Vivekananda Global University, Jaipur
Registrar

or truncation of data if the value cannot be fully represented in the target type.

- Example: Converting a float to an int can result in loss of decimal places: float a = 3.99; int b = (int)a; (b will be 3)

- *Explicit Type Conversion Functions:*
  - C provides explicit type conversion functions, such as atoi(), atof(), and itoa(), to convert between strings and numeric data types.
  - Example: Converting a string to an integer: char str[] = "123"; int num = atoi(str);

It's important to note that improper or inappropriate use of type conversion can lead to unexpected results or data loss. Care should be taken to ensure compatibility and maintain data integrity during type conversions

Here's an example demonstrating type conversion in C:

```
#include <stdio.h>
int main() {
    int a = 5;
    float b = 3.14;
    int result;
    // Implicit type conversion
    result = a + b;
    printf("Implicit Conversion: %d\n", result);
    // Explicit type conversion
    result = (int)b;
    printf("Explicit Conversion: %d\n", result)
    return 0;
}
```

In this example, the program demonstrates both implicit and explicit type conversion. The a + b expression performs implicit conversion, where the int value is implicitly converted to float for the addition operation. The explicit conversion (int)b converts the float value b to an int before the assignment to result.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

4.8 Summary

Tokens: In C programming, a token is the smallest program unit recognized by the compiler. It can be a keyword, identifier, constant, string literal, operator, or special symbol.

Operators: C language supports various operators, including arithmetic, relational, logical, bitwise, and assignment operators, which perform specific operations on operands.

Operator Precedence: Operator precedence determines the order in which operators are evaluated when an expression contains multiple operators. Operators with higher precedence are evaluated first.

Associativity: Associativity determines the grouping of operands when an expression contains multiple operators with the same precedence. Operators can be left-associative (evaluated left to right) or right-associative (evaluated right to left).

Input/Output Functions: C provides standard input/output functions like 'printf()' for displaying output and 'scanf()' for reading input from the user. 'printf()' formats and prints data to the standard output stream, while 'scanf()' reads data from the standard input stream based on specified format specifiers.

4.9 Review Questions

1 What is the purpose of a C compiler?
2 Explain the difference between a variable and a constant in C.
3 What are data types in C? Provide examples of different data types.
4 What is the significance of the 'main()' function in a C program?
5 How do you declare and initialize a variable in C?
6 What is the difference between 'print()' and 'scan()' functions in C?
7 How do you use conditional statements (if-else) in C? Provide an example.
8 What are loops in C? How do you use 'for' and 'while' loops?
9 Explain the concept of arrays in C, and how to access elements in an array.

10 How do you define and call a user-defined function in C? Provide an example.

4.10 Keywords

- Compiler: A program that translates C source code into machine code or intermediate code, making it executable on a computer.
- Variable: A named location in memory used to store data that can be modified during program execution.
- Constant: A fixed value that remains unchanged throughout the program's execution.
- Data Types: The classification of data in C based on their storage requirements and the operations that can be performed on them (e.g., int, float, char).
- main(): The main function in C, which serves as the entry point of a program and from where the program's execution begins.
- printf(): A function used to display output to the standard output stream (usually the console).
- scanf(): A function used to read input from the standard input stream (usually the keyboard).
- Conditional Statements: Statements used to make decisions in a program based on specified conditions (e.g., if-else).

4.11 References

1. E.Balaguruswamy, "Computing Fundamentals & C Programming", Tata McGraw Hill, 2008.
2. B. A. Forouzan & R. F. Gilberg "Computer Science - A structured programming Approach Using C", 2011
3. E. Bala Guruswamy, "Programming in ANSI" Tata McGraw Hill, 2011.

[illegible]

Table of Content

[illegible]

Learning Objectives

After studying this unit, the student will be able to:

- Learn to use conditional statements like if, else, and else if to control the flow of your program based on specific conditions.
- Understand how Boolean expressions evaluate to either true or false and are used in decision-making statements.
- Practice using nested if-else statements to create complex decision-making structures with multiple conditions.
- Learn to use the switch statement to handle multiple cases based on the value of an expression.
- Logical Operators: Explore logical operators (&&, ||, !) to combine multiple conditions in decision-making statements.
- Understand how the for loop works with its initialization, condition, and increment/decrement to perform a fixed number of iterations.
- Learn how the while loop continues executing as long as a specified condition remains true.
- Understand the do-while loop, which guarantees that the code block is executed at least once before checking the loop condition.
- Use of break and continue statements to modify the flow of a loop and control when to terminate or skip iterations.

[illegible]

Introduction to Control Statements

In C programming, control statements are used to control the flow of execution in a program. They allow the program to make decisions, repeat blocks of Code, and perform different actions based on specific conditions. The main types of control statements in C are:

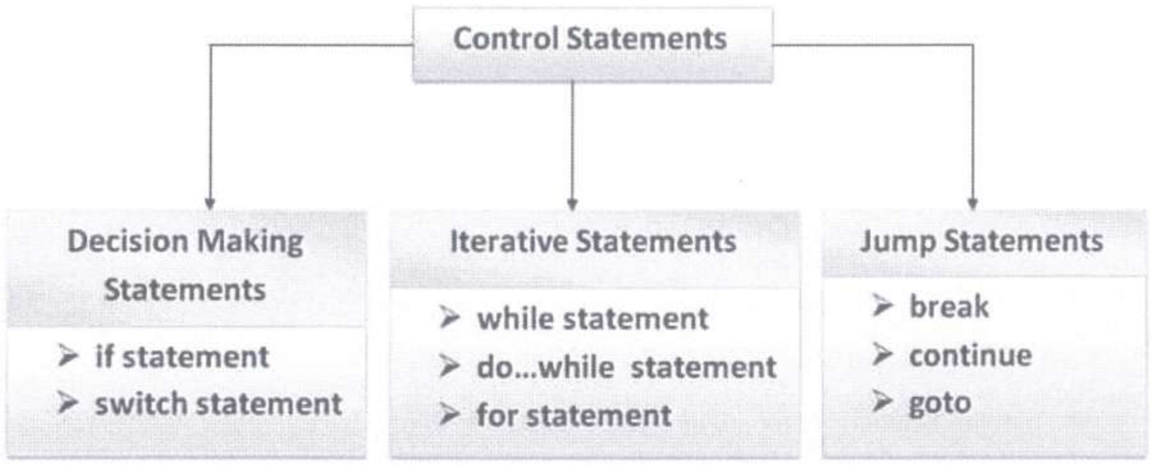


Fig – Types of Control Statements

There are three main types of control statements in C:

Decision Making Statements:

- if statement: The if statement is used to execute a block of code if a given condition is true.
- if-else statement: The if-else statement allows you to execute one block of code if the condition is true and another block of code if the condition is false.
- nested if-else: You can have if-else statements inside another if or else block to create nested conditions.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

- Switch Statement: The switch statement allows you to select one of many code blocks to be executed based on the value of an expression.

Iterative (Loop) Statements:

- for loop: The for loop is used to repeatedly execute a block of code for a specified number of times.
- while loop: The while loop executes a block of code as long as a specified condition is true.
- do-while loop: Similar to the while loop, but it executes the block of code first and then checks the condition.

Jump Statements:

- break: It is Used inside loops (like for, while, do-while) and switch statements. It terminates the current loop or switch block.
- continue: It is Used inside loops (like for, while, do-while). When encountered, it immediately jumps to the next iteration of the loop, skipping the rest of the code below the continue statement.
- goto: Allows you to transfer control to a labeled statement anywhere in the code.

5.1 Control Statements

- ***if-else Statements:***

Allows binary decision-making. Executes a block of Code if the condition is true and another block if the condition is false.

Syntax:

```
if (condition) {
    // Code block executed when the condition is true
} else {
```

[illegible]

```
        // Code block executed when the condition is false
    }
```

- ***Nested if-else Statements:***

Enables multiple levels of decision-making by placing one if-else Statement inside another.

Syntax:

```
if (condition1) {
    // Code block executed when condition1 is true
    if (condition2) {
        // Code block executed when both condition1 and
condition2 are true
    } else {
        // Code block executed when condition1 is true but
condition2 is false
    }
} else {
    // Code block executed when condition1 is false
}
```

- ***else if Ladder:***

Used when there are multiple conditions to check, and the program needs to make a decision based on the first condition that evaluates to true.

Syntax:

```
if (condition1) {
    // Code block executed when condition1 is true
} else if (condition2) {
    // Code block executed when condition2 is true, and
condition1 is false
} else if (condition3) {
    // Code block executed when condition3 is true, and
both condition1 and condition2 are false
}
// more else if statements...
else {
```

[illegible]
For Vivekananda Global University, Jaipur
Registrar

```
        // Code block executed when all conditions are false
    }
```

- ***switch Statement:***

  Allows multi-way decision-making based on the value of an expression.

  Syntax:

```
switch (expression) {
    case value1:
        // Code block executed when expression equals value1
        break.
    case value2:
        // Code block executed when expression equals value2
        break.
    // more cases...
    default:
        // Code block executed when none of the cases match
}
```

- ***Loops (for, while, do-while):***

  Used to repeat a block of Code multiple times until a specific condition is met.

  Syntax:

```
// for loop
for (initialization; condition; update) {
    // Code block executed as long as the condition is true
}
// while loop
while (condition) {
    // Code block executed as long as the condition is true
}
// do-while loop
do {
    // Code block executed at least once, and then repeatedly as long as the condition is true
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
} while (condition).
```

- ***Go to (Jump statement):***

The goto statement in C is a jump statement that allows you to transfer the control flow of a program to a labelled statement within the same function. While the goto statement can be used to alter the normal flow of execution, it is generally considered bad practice due to its potential to make code harder to read and maintain. It can lead to spaghetti code and make it difficult to understand the program's logic. In most cases, other control statements like if, else, for, while, and do-while loops are preferred over goto.

Syntax of the goto statement:

```
goto label;
//...
//...
label: // The labelled statement
// Code to be executed when the goto statement is
encountered
```

Example of using goto statement in C:

```
#include <stdio.h>
int main() {
    int num;
    printf("Enter a positive number: ");
    scanf("%d", &num);
    if (num <= 0) {
        printf("Invalid input! Please enter a positive
number.\n");
        goto end; // Jump to the 'end' label and skip the
rest of the code
    }
    printf("The number you entered is: %d\n", num);
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
    end: // The 'end' label
    printf("Program ends.\n");
    return 0;
}
```

In this example, we use the goto statement to jump to the end label if the user enters a non-positive number. The program will then skip the rest of the code after the goto statement and directly execute the code following the end label.

Control statements provide the necessary tools to create complex and dynamic programs by controlling the sequence of code execution based on conditions and repetitions. They are vital for creating efficient and logical flow in C programs, allowing developers to handle various scenarios and build versatile applications.

5.2 Statements and Blocks

In C programming, statements and blocks are essential components that determine the flow of execution and code organization within a program.

- **Statements:**

A statement in C is a single instruction or action that performs a specific task. Each Statement in C typically ends with a semicolon (;). Statements can be simple or compound.

- Simple Statement: A simple statement is a single line of Code that performs a specific operation.
  For example:

  ```
  int x = 5; // Declaration and initialization statement

  printf("Hello, World!"); // Function call statement
  ```

- Compound Statement (Block): A compound statement, also known as a block, is a group of multiple statements enclosed within curly braces { }.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Compound statements allow grouping multiple statements together to form a single unit. Blocks are often used with control structures like if-else, loops, and functions.

- **Blocks:**

In C, a block is a set of statements enclosed within curly braces { }. A block is used to define the scope of variables and to group statements together. Blocks are often associated with control structures and function definitions.

Example of a block in an if-else statement:

```
if (condition) {
    // Code block for the if branch
    printf("Condition is true.\n");
} else {
    // Code block for the else branch
    printf("Condition is false.\n");
}
```

Example of a block in a function definition:

```
void my Function() {
    // Start of the function block
    int x = 10;
    printf("Value of x: %d\n", x);
    // End of the function block
}
```

Blocks are essential for defining the scope of local variables and for controlling the flow of execution in the program. They allow for more organized and maintainable Code, especially when handling complex logic or multiple statements in a single code unit. The use of blocks enhances code readability and helps in preventing unintended variable name conflicts by restricting the visibility of variables to their specific block scope.

5.3 Switch Case Statement

Organizations decide whether to adopt a centralized or decentralized approach to managing their IT infrastructure and operations in the modern IT environment.

Here's an overview of the centralization versus decentralization debate in the context of the modern IT environment:

In C programming, the switch case statement is a control statement used for multi-way decision-making based on the value of an expression. It provides an efficient alternative to long chains of if-else if-else statements when the program needs to select one code block from multiple options based on the value of a single expression.

Syntax:

```
switch (expression) {
    case constant1:
        // Code block executed when expression equals
constant1
        break;
    case constant2:
        // Code block executed when expression equals
constant2
        break;
    // more cases...
    default:
        // Code block executed when none of the cases match
the expression
}
```

The switch keyword initiates the switch case statement.

Syntax

Expression is the value to be evaluated.

Each case label specifies a constant value that the expression can match.

The code block following each case label is executed when the expression matches the corresponding constant.

The break statement is used to exit the switch case block after executing the Code associated with a matched case. Without break, the program will continue executing the Code for subsequent cases regardless of whether their expressions match.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Example 1:

```
int day = 3;

switch (day) {

    case 1:

        printf("Sunday\n");

        break;

    case 2:

        printf("Monday\n");

        break;

    case 3:

        printf("Tuesday\n");

        break;

    case 4:

        printf("Wednesday\n");

        break;

    case 5:

        printf("Thursday\n");

        break;

    case 6:

        printf("Friday\n");

        break;

    case 7:

        printf("Saturday\n");

        break;

    default:

        printf("Invalid day\n");

}
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

In this example, the day variable is evaluated against each case, and the corresponding day name is printed to the console. If the value of day does not match any of the cases, the default block is executed, providing a default response for invalid inputs.

Example 2

```
#include<math.h>
int main() {
char day;
Printf("enter day");
Scanf("%s", &day);
switch(day) {
case m : printf("Monday \n");
break;
case t : printf("Tuesday \n");
break;
case w : printf("Wednesday \n");
break;
case T : printf("Thursday \n");
break;
case f : printf("Friday \n");
break;
case s: printf("Saturday \n");
break;
case S : printf("Sunday \n");
     break;
     default :printf("Not a valid day");
}
return 0;
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
}
```

This example is using the character as case sequences.

Switch case statements are particularly useful when dealing with multiple cases based on discrete values, such as days of the week, menu choices, or enumerated types. They enhance code readability and maintainability by providing a more concise and structured approach to multi-way decision-making.

5.4 Decision Control Instructions

Decision control instructions are a fundamental part of programming and refer to the statements that allow a program to make decisions based on certain conditions. These instructions help direct the flow of the program, enabling it to perform different actions based on the outcome of specific tests or comparisons.

In C, there are four types of if-else statements based on their structure and usage:

- **Simple if-else Statement:**

  The simple if-else Statement is the most basic type for making binary decisions. It executes one block of Code if the condition is true and another block of Code if the condition is false.

Syntax:

```
if (condition) {
    // Code block executed when the condition is true
} else {
    // Code block executed when the condition is false
}
```

- **Nested if-else Statement:**

  Nested if-else statements are used for multiple levels of decision-making. It involves placing one if-else Statement inside another, allowing more complex choices based on various conditions.

Syntax:

```
if (condition1) {
    // Code block executed when condition1 is true
    if (condition2) {
        // Code block executed when both condition1 and
condition2 are true
    } else {
        // Code block executed when condition1 is true but
condition2 is false
    }
} else {
    // Code block executed when condition1 is false
}
```

- **else if Ladder:**

An else if ladder is used when there are multiple conditions to check, and the program needs to make a decision based on the first condition that evaluates to true.

Syntax:

```
if (condition1) {
    // Code block executed when condition1 is true
} else if (condition2) {
    // Code block executed when condition2 is true, and
condition1 is false
} else if (condition3) {
    // Code block executed when condition3 is true, and
both condition1 and condition2 are false
}
// more else if statements...
else {
    // Code block executed when all conditions are false
}
```

- **Ternary Operator (Conditional Operator):**

The ternary operator is a shorthand form of the if-else Statement

```
variable = (condition) ? value_if_true : value_if_false;
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Example:

```
int num = 10;

int result = (num > 0) ? 1 : 0; // If num is positive, result
will be 1; otherwise, it will be 0.
```

These types of if-else statements provide powerful tools for making decisions in C programs based on different conditions. They allow for a wide range of decision-making scenarios, from simple binary choices to complex multi-level branching logic. Choosing the appropriate type of if-else Statement depends on the specific requirements and complexity of the decision-making process in a given program.

5.5 Looping Constructs in C

Decentralization in the modern IT environment refers to delegating IT responsibilities, decision-making authority, and resources to different departments or business units. Here are some benefits and drawbacks associated with decentralization:

In C, looping constructs are used to repeat a block of Code multiple times. They are essential for creating repetitive tasks and iterating over data structures. The main looping constructs in C are:

- ***for loop:***

  The for loop allows you to execute a block of Code repeatedly for a specified number of times. It consists of an initialization, a condition, and an increment/decrement. The syntax is as follows:

```
for (initialization; condition; increment/decrement) {
    // code block to execute repeatedly as long as the
condition is true
}
```

Example:

```
for (int i = 1; i <= 5; i++) {
    printf("Iteration %d\n", i);
}
```

[illegible]

For Vivekananda Global University, Jaipur
Registrar

- ***while loop:***

  The while loop repeatedly executes a block of Code as long as the given condition remains true. The loop will continue until the condition evaluates to false. The syntax is as follows:

```
while (condition) {
    // code block to execute repeatedly as long as the
condition is true
}
```

Example:

```
        int count = 0;
        while (count < 5) {
            printf("Count: %d\n", count);
            count++;
        }
```

- ***do-while loop:***

  The do-while loop is similar to the while loop, but it ensures that the code block is executed at least once before checking the condition. The loop will continue as long as the condition remains true. The syntax is as follows:

```
do {
    // code block to execute repeatedly as long as the
condition is true
} while (condition);
```

Example:

```
int num = 1;
do {
    printf("Number: %d\n", num);
    num++;
} while (num <= 5);
```

- ***break statement:***

  The break statement is used to exit a loop prematurely, regardless of the loop's condition. It is often used to terminate a loop early based on certain conditions. For example:

[illegible]
For Vivekananda Global University, Jaipur
Registrar

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break; // Exit the loop when i becomes 5
    }
    printf("Iteration %d\n", i);
}
```

- ***continue Statement:***

  The continue statement is used to skip the current iteration and proceed to the next iteration of the loop. It allows you to avoid executing certain parts of the loop's Code under specific conditions. For example:

  ```
  for (int i = 1; i <= 5; i++) {
      if (i == 3) {
          continue; // Skip iteration when i is 3
      }
      printf("Iteration %d\n", i);
  }
  ```

These looping constructs allow C programmers to repeat code blocks, iterate over data structures, and handle various scenarios more efficiently.

5.6 If Else Statements

.IT security in enterprises refers to the practices and measures implemented to protect the organization's information technology systems, networks, and data from unauthorized access, use, disclosure, disruption, modification, or destruction. Here are some key aspects and considerations related to IT security in enterprises:

- ***if Statement:***

  The if Statement allows you to execute a block of Code if a given condition is true.

[illegible]
For Vivekananda Global University, Jaipur
Registrar

Example:

```
#include <stdio.h>
int main() {
    int age = 25;

    if (age >= 18) {
        printf("You are an adult.\n");
    }
    return 0;
}
```

Output:

```
You are an adult.
```

- ***if-else Statement:***

  The if-else Statement provides an alternate block of Code to execute when the condition is false.

Example:

```
#include <stdio.h>
int main() {
    int num = 12;
    if (num % 2 == 0) {
        printf("The number is even.\n");
    } else {
        printf("The number is odd.\n");
    }
    return 0;
}
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Output:

```
The number is even.
```

- ***if-else-if Statement:***

  The else if Statement allows you to test multiple conditions in sequence and execute different code blocks based on the outcome of those conditions.

Example:

```
#include <stdio.h>
int main() {
    int score = 80;
    if (score >= 90) {
        printf("Excellent!\n");
    } else if (score >= 70) {
        printf("Good job!\n");
    } else if (score >= 50) {
        printf("You passed.\n");
    } else {
        printf("You failed.\n");
    }
    return 0;
}
```

Output:

```
Good job!
```

- ***Nested if-else Statement:***

  A nested if-else statement is an if-else statement inside another if or else block. This allows for more complex decision-making.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Example:

```
#include <stdio.h>

int main() {

    int num = 15;

    if (num >= 10) {

        if (num < 20) {

            printf("The number is between 10 and 19.\n");

        }

     else {

            printf("The number is greater than or equal to
20.\n");

        }

    } else {

        printf("The number is less than 10.\n");

    }

    return 0;

}
```

Output:

```
The number is between 10 and 19.
```

In this example, the nested if-else first checks if num is greater than or equal to 10. If it is, it further checks if num is less than 20. If both conditions are true, it prints "The number is between 10 and 19." If the first condition is false, it prints "The number is less than 10."

5.7 For Loop

In C, the for loop is used to execute a block of Code repeatedly for a specific number of iterations. It consists of three components: initialization, condition, and

[illegible]

increment/decrement. The loop continues as long as the condition remains true. Here's the basic syntax of a for loop in C:

```
for (initialization; condition; increment/decrement) {
    // code block to execute repeatedly as long as the
condition is true
}
```

Let's look at an example of using a for loop to print the numbers from 1 to 5:

```
#include <stdio.h>
int main() {
    for (int i = 1; i <= 5; i++) {
        printf("%d ", i);
    }
    return 0;
}
```

Output:

```
1 2 3 4 5
```

In this example, the for loop initializes the variable i to 1 (int i = 1), sets the condition to execute the loop as long as i is less than or equal to 5 (i <= 5), and increments i by one after each iteration (i++).

The loop starts with i = 1, and it prints the i value (1) using printf(). Then, it increments i to 2 and repeats the process until i becomes 6, at which point the condition i <= 5 becomes false, and the loop terminates.

The for loop is commonly used when you know the number of iterations you want to perform, making it easier to control the loop flow and execute a block of Code a fixed number of times.

- ***Nested for loop***

[illegible]
For Vivekananda Global University, Jaipur
Registrar

In the nested loop structure, an outer loop contains an inner loop. The outer loop will run based on its own conditions, and the inner loop will run completely for each iteration of the outer loop.

Syntax

```
for (initialization; condition; update) {
    // Outer loop code
    for (initialization; condition; update) {
        // Inner loop code
    }
    // More outer loop code (optional)
}
```

Example 1

```
#include <stdio.h>

int main() {

    int rows = 10;

    int columns = 10;

    printf("Multiplication Table:\n");

    // Outer loop controls the rows

    for (int i = 1; i <= rows; i++) {

        // Inner loop controls the columns

        for (int j = 1; j <= columns; j++) {

            printf("%4d", i * j);

        }

        printf("\n");

    }

    return 0;

}
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Output:

```
Multiplication Table:

 1  2  3  4  5  6  7  8  9 10

 2  4  6  8 10 12 14 16 18 20

 3  6  9 12 15 18 21 24 27 30

 4  8 12 16 20 24 28 32 36 40

 5 10 15 20 25 30 35 40 45 50

 6 12 18 24 30 36 42 48 54 60

 7 14 21 28 35 42 49 56 63 70

 8 16 24 32 40 48 56 64 72 80

 9 18 27 36 45 54 63 72 81 90

10 20 30 40 50 60 70 80 90 100
```

5.8 While loops

Vendor selection is choosing the most suitable vendor or supplier to meet an organization's needs. It involves steps and considerations to ensure the selected vendor can deliver the required goods or services effectively. Here are key aspects related to vendor selection:

In C, the while loop is used to repeatedly execute a block of Code as long as a given condition remains true. The loop will continue until the condition evaluates to false. Here's the basic syntax of a while loop in C:

```
while (condition) {
    // code block to execute repeatedly as long as the
condition is true
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
}
```

Let's look at an example of using a while loop to print the numbers from 1 to 5:

```
#include <stdio.h>
int main() {
    int i = 1;
    while (i <= 5) {
        printf("%d ", i);
        i++;
    }
    return 0;
}
```

Output:

```
1 2 3 4 5
```

In this example, the while loop checks the condition i <= 5 before each iteration. The loop starts with i = 1, and it prints the value of i (which is 1) using printf(). Then, it increments i by one (i++) and repeats the process until i becomes 6. When i becomes 6, the condition i <= 5 becomes false, and the loop terminates.

It's essential to ensure that the condition inside the while loop eventually becomes false; otherwise, the loop will continue indefinitely, resulting in an infinite loop.

The while loop is commonly used when the number of iterations is not known beforehand or when you want to repeat a task until a specific condition is no longer true.

5.9 Do While Loops

In C, the do-while loop is similar to the while loop, but it guarantees that the block of Code inside the loop is executed at least once before checking the loop condition. After the code block is executed, the loop will continue to execute as long as the specified condition remains true. Here's the basic syntax of a do-while loop in C:

```
do {
    // code block to execute repeatedly as long as the
condition is true
} while (condition);
```

Let's look at an example of using a do-while loop to print the numbers from 1 to 5:

```
#include <stdio.h>

int main() {

    int i = 1;

    do {

        printf("%d ", i);

        i++;

    } while (i <= 5);

    return 0;

}
```

Output:

```
1 2 3 4 5
```

In this example, the do-while loop first executes the block of Code inside the loop (printing the value of i) and then increments i by one (i++). After each iteration, it checks the condition i <= 5. The loop continues until i becomes 6. When i becomes 6, the condition i <= 5 becomes false, and the loop terminates.

Unlike the while loop, the do-while loop guarantees that the Code inside the loop will run at least once, regardless of the initial condition. It's useful to perform an action before evaluating the condition. Just be cautious not to create an infinite loop by ensuring the condition eventually becomes false.

Table – Difference between while and do while loop

Sr.	While Loop	Do While Loop
1	In a while loop, the condition is checked at the beginning of the loop, before executing the loop's body.	In a do-while loop, the condition is checked at the end of the loop, after executing the loop's body.
2	If the condition is initially false, the loop's body is never executed.	This guarantees that the loop's body will be executed at least once, even if the condition is false from the beginning.
3	The loop will continue to execute as long as the condition remains true.	The loop will continue to execute as long as the condition remains true.
4	Syntax : `while (condition) {` `// Code to be executed as long as the condition is true.` `}`	Syntax : `do {` `// Code to be executed at least once, and then as long as the condition is true.` `} while (condition);`
5	Example `int count = 1;` `while (count <= 5) {` `printf("Count:  %d\n", count);` `count++;` `}` Output Count: 1 Count: 2 Count: 3 Count: 4 Count: 5	`int count = 1;` `do {` `printf("Count:%d\n", count);` `count++;` `} while (count <= 5);` Output Count: 1 Count: 2 Count: 3 Count: 4 Count: 5

[illegible]
For Vivekananda Global University, Jaipur

Registrar

5.10 Break and Continue Statements

Ongoing relationship management refers to nurturing and maintaining a positive and productive relationship with vendors beyond the initial contract phase. It involves regular communication, monitoring, and collaboration to ensure that the vendor meets the organization's needs and expectations. Here are key aspects related to ongoing relationship management with vendors

In C, break and continue are two control flow statements used within loops to alter the normal flow of execution.

- ***Break statement:***

  The break statement is used to exit a loop prematurely, regardless of the loop's condition. When the break statement is encountered within a loop, the loop is immediately terminated, and the program execution continues with the Statement immediately after the loop.

  Here's an example of using break in a for loop to stop the loop when i becomes 5:

```
#include <stdio.h>
int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            break; // Exit the loop when i becomes 5
        }
        printf("%d ", i);
    }
    return 0;
}
```

Output:

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
1 2 3 4
```

In this example, the loop starts with i = 1 and continues printing the values of i (1, 2, 3, 4). When i becomes 5, the if condition is true, and the break statement is executed, terminating the loop immediately.

- ***Continue Statement:***

The continue statement is used to skip the current iteration of a loop and proceed to the next iteration. When the continue statement is encountered within a loop, the rest of the loop code is skipped for the current iteration, and the program execution jumps to the next iteration.

Here's an example of using continue in a for loop to skip printing the value of i when it is equal to 5:

```
#include <stdio.h>
int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            continue;// Skip the iteration when i is 3
        }
        printf("%d ", i);
    }
    return 0;
}
```

Output:

```
1 2 4 5
```

In this example, the loop starts with i = 1, and it prints the value of i (1). Then, it increments i to 2 and prints it (2). When i becomes 3, the if condition is true, and the continue statement is executed, skipping the rest of the loop code for this

[illegible]

iteration. The loop proceeds to the next iteration, which is i = 4, and continues printing the remaining values (4, 5).

5.11 Summary

Decision Making:

Decision-making statements in programming allow you to control the flow of your Code based on specific conditions. The main decision-making statements in C are if, else, and else if. The if Statement executes a block of Code if a given condition is true. The else Statement provides an alternate block of Code to execute when the initial condition is false. The else if Statement allows you to test multiple conditions in sequence and execute different code blocks based on the outcomes of these conditions. These statements enable your program to make informed choices and handle different scenarios based on user input or variable values.

Loops:

Loops are used to execute a block of Code repeatedly until a specified condition is no longer true. The primary loops in C are for, while, and do-while. The for loop performs a fixed number of iterations based on an initialization, condition, and increment/decrement. The while loop continues executing as long as a given condition remains true. The do-while loop ensures that the code block is executed at least once before checking the loop condition. Loops are useful for iterating over data, performing repetitive tasks, and processing large amounts of information.

Conditional Statements:

Conditional statements, including decision-making statements and loops, enable you to create more flexible and interactive programs. By using conditions, you can control which parts of the Code get executed, allowing your program to adapt to different situations and user input.

[illegible]

Break and Continue:

The break statement is used to exit a loop prematurely, terminating the loop regardless of the loop's condition. It helps to stop the loop's execution based on specific conditions or user interactions. The continue statement is used to skip the current iteration of a loop and proceed to the next iteration.

5.12 Review Questions

1 What is the purpose of the if Statement in C, and how does it work?

2 What is the difference between an if-else statement and an if-else-if statement?

3 When would you use a switch statement instead of multiple if-else statements?

4 Explain the usage of the break and continue statements within loops and switch cases.

5 Write a C program that uses the if-else Statement to check if a number is positive, negative, or zero.

6 Write a C program that uses a switch statement to display the name of the month based on the user's input (1 for January, 2 for February, etc.).

7 What is the purpose of the for loop in C, and what are its three components?

8 Describe the difference between a while loop and a do-while loop.

9 What does an infinite loop mean, and how can you avoid creating one?

10 Write a C program using a for loop to print the first 10 even numbers.

5.13 Keywords

- for: A loop that executes a block of Code repeatedly for a specified number of times.
- while: A loop that repeatedly executes a block of Code as long as a given condition remains true.
- do-while: A loop that guarantees the code block is executed at least once before checking the loop condition.

- if: A conditional statement that executes a block of Code if a given condition is true.
- else: Used with if to execute an alternate block of Code when the initial condition is false.
- else if: Used to test multiple conditions in sequence and execute different code blocks based on the outcome.
- break: Used to exit a loop prematurely, terminating the loop regardless of the loop's condition.
- continue: Used to skip the current loop iteration and proceed to the next iteration.
- Iteration: Iteration is the process of repeating a set of statements or a block of Code multiple times in a computer program.
- Nested Loops: Nested loops refer to the concept of having one loop inside another loop in a computer program. It allows you to perform repeated operations on two-dimensional data structures, such as arrays or matrices.

5.14 References

1. E.Balaguruswamy, "Computing Fundamentals & C Programming", Tata McGraw Hill, 2008.
2. B. A. Forouzan & R. F. Gilberg "Computer Science – A structured programming Approach Using C", 2011
3. E. Bala Guruswamy, "Programming in ANSI" Tata McGraw Hill, 2011.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Table of Content

[illegible]

Learning Objectives

After studying this unit, the student will be able to:

- Learn the fundamental concept of arrays as a data structure to store multiple elements of the same data type.
- Learn how to declare arrays, specify their size, and initialize them with values.
- Understand how to access individual elements in an array using their index.
- Learn how to use loops to traverse through the elements of an array.
- Explore the concept of multidimensional arrays, such as 2D arrays or matrices, and their application in solving problems.
- Practice common array operations, including finding the minimum and maximum elements, calculating the sum and average, and searching for specific elements.
- Understand various sorting algorithms, working principles, and time and space complexity.
- Learn about comparison-based sorting algorithms like Bubble Sort, Selection Sort, and Insertion Sort.
- Explore non-comparison-based sorting algorithms like Counting Sort, Radix Sort, and Bucket Sort, and understand their advantages.
- Learn how to select an appropriate sorting algorithm based on the size and characteristics of the dataset.
- Understand the concept of stability in sorting and its significance in specific applications.
- Explore the applications of sorting in various fields, such as database management, search algorithms, and data analysis.

[illegible]

Introduction to Array

An array represents a fundamental data structure utilized in computer programming, enabling the storage of multiple elements of the same data type under a single variable name. This feature facilitates the efficient organization and manipulation of multiple values as a cohesive entity. Arrays hold significant importance in data manipulation and algorithm implementation and find extensive application across various programming languages.

In C language, an array constitutes a fixed-size collection of similar data items, which are stored in contiguous memory locations. This attribute allows the array to accommodate various primitive data types, including integers, characters, floating-point numbers, and more, as well as derived and user-defined data types like pointers and structures.

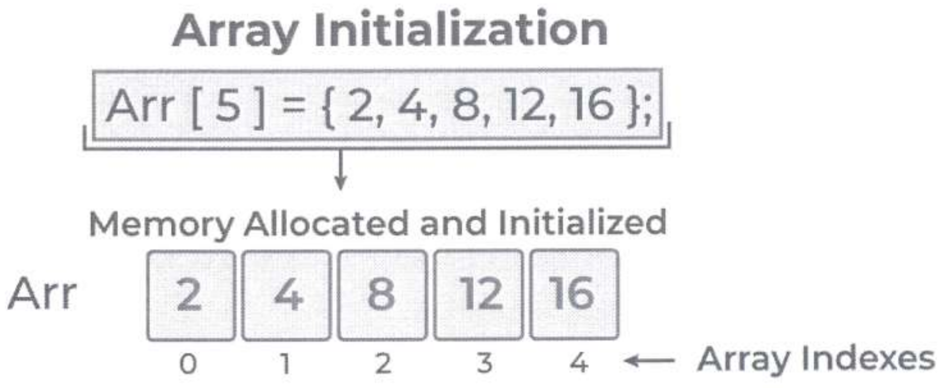

Array Initialization
Arr [5] = { 2, 4, 8, 12, 16 };
Memory Allocated and Initialized
Arr
2
4
8
12
16
0
1
2
3
4
Array Indexes

Fig – Array Initialization

Key Characteristics of Arrays:

- Homogeneous Elements: Arrays exclusively store elements of a single data type, such as integers, floating-point numbers, characters, or any other specific data type.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

- Indexed Elements: Each element in an array is associated with an index, starting from 0 for the first element, 1 for the second, and so on. The index enables direct access to any specific element within the array.
- Fixed Size: Arrays possess a predetermined size defined during their declaration, and this size remains unchanged throughout their lifetime. To accommodate more elements, a new array with a larger size must be created, necessitating the copying of elements from the old array.
- Contiguous Memory Allocation: Array elements are stored in adjacent memory locations, allowing rapid access to individual elements using their respective indices.

Advantages of Arrays:

- Efficient Access: Arrays provide direct and efficient access to individual elements using their indices, facilitating swift data retrieval and manipulation.
- Contiguous Memory Allocation: The contiguous memory layout of arrays results in efficient memory access and cache utilization, leading to improved performance.
- Fixed Size: The fixed size of arrays ensures that the necessary memory is allocated in advance, enhancing memory management and avoiding unexpected runtime allocation.
- Element Ordering: Arrays maintain the order of elements based on their indices, making it advantageous in scenarios where element order preservation is crucial.
- Random Access: Arrays support random access to elements, allowing direct access to any element based on its index, regardless of its position in the array.

Disadvantages of Arrays:

- Fixed Size Limitation: The fixed size of arrays can be problematic when the exact number of elements to be stored is unknown or may change during runtime. Dynamic resizing can be inefficient, necessitating the creation of a new array and element copying.

- Contiguous Memory Requirement: Large arrays may encounter challenges in finding a sufficiently large block of contiguous memory, impacting performance.
- Wasted Memory Space: Declaring an array with a size larger than the actual number of elements results in wasted memory space and inefficient memory usage.
- No Built-in Bound Checking: Most programming languages lack built-in boundary checking for array indices, potentially leading to unpredictable behavior or memory-related errors when accessing elements beyond the valid index range.
- Homogeneous Elements: Arrays can only store elements of the same data type, necessitating more complex data structures, such as lists or maps, to handle heterogeneous data.
- Insertion and Deletion Complexity: It is not possible to add or delete array elements from the middle of array, as it involves shifting elements to accommodate new elements or close gaps left by deleted elements.

6.1 Array Declaration in C

Array declaration syntax in C -

```
        data_type array_name[size];
```

- *data_type:* The data type of elements stored in the array. It can be any valid C data type, such as int, float, char, etc.
- *array_name:* The name of the array you use to access its elements.
- *Size:* The number of elements that the array can hold. It must be a non-negative integer constant or an integer variable with a positive value.

Example to declare an integer array named numbers that can hold five elements, you would write:

```
                int numbers[5];
```

[illegible]

This statement declares an array of 5 integers, allocating contiguous memory locations for each element. Access to individual elements is achieved through indices ranging from 0 to 4. To modify or retrieve elements, the array name is followed by the desired index enclosed within square brackets:

```
numbers[0] = 10; // Assigning 10 to the first element of the array
numbers[2] = 30; // Assigning 30 to the third element of the array
int value = numbers[1]; // Retrieving the value of the second element of the array
```

In C, arrays are zero-indexed, signifying that the first element is accessed using index 0, the second with index 1, and so on. It is crucial to ensure that the index remains within the valid range of the array (0 to size-1) to prevent accessing out-of-bounds memory locations, as this can lead to undefined behavior and bugs in the program.

Alternatively, we can initialize the array at the time of declaration

```
int numbers[5] = {50, 60, 70, 80, 90};
```

With this declaration, the numbers array is both declared and initialized with the specified values. If the number of elements provided in the initialization list is less than the size of the array, the remaining elements will be automatically initialized to zero for numeric types (e.g., int, float) or the null character ('\0') for character arrays

6.2 Array Initialization in C

In C programming, we have the option to initialize an array during its declaration using an initializer list. This list comprises values enclosed in curly braces { }, separated by commas, corresponding to the elements of the array in the order of declaration.

[illegible]

The syntax for array initialization in C is as follows:

data_type array_name[size] = {value1, value2, ..., valueN};

Here is an explanation of the components involved:

- data_type: Represents the data type of elements stored in the array, which can be any valid C language data type, such as int, float, char, and so on.
- array_name: Specifies the name of the array to be initialized.
- Size of array: It denotes the number of elements that the array can accommodate. It must be a non-negative integer constant or an integer variable with a positive value.
- value1, value2, ..., valueN: Refers to the values intended to be assigned to the individual elements of the array. The number of values in the initializer list should not exceed the size of the array.

Here are some examples of array initialization in C:

// Integer array initialization

int numbers [5] = {50, 60, 70, 80, 90};

// Character array (string) initialization

char message [6] = {'A', 'R', 'R', 'A', 'Y', '\0'};

// Float array initialization

float temperatures [3] = {88.6, 145.2, 103.9};

In the first example, we initialize an integer array called numbers with five elements, each assigned a specific value. The second example demonstrates the initialization of a character array named message, functioning as a string in C, with the characters representing the word "Hello," followed by the null terminator '\0'. The null terminator is crucial as it indicates the end of the string. Lastly, the third

[illegible]

For Vivekananda Global University, Jaipur

Registrar

example showcases the initialization of a float array named temperatures with three floating-point values representing temperature readings.

If the number of elements in the initializer list is less than the size of the array, the remaining elements will be automatically initialized to zero for numeric types (e.g., int, float) or the null character ('\0') for character arrays. For instance:

int data[6] = {10, 20, 30}; // data[3], data[4], data[5] will be initialized to 0

char name[10] = "John"; // name[4] to name[9] will be initialized to '\0'

6.3 Accessing Array Elements

To access elements in an array, you use the array name followed by the index of the element you want to access, enclosed in square brackets []. The index represents the position of the element within the array, base indexstarts from 0 for the very first element and it increments by 1 for the remaining element.

For access an element in an array, the Syntax is :

array_name [index]

Keep in mind the following points when accessing array elements:

- Indexing: Array indices start from 0 (for the first element) and go up to size - 1 (for the last element), where size is the number of elements in the array.
- Bounds Checking: It's essential to ensure that the index you provide is within the valid range of the array (0 to size - 1). Accessing elements outside this range can lead to undefined behavior and may result in a runtime error or garbage values.
- Looping Through an Array: You can use loops (e.g., for loop) to iterate through all the elements of an array efficiently.

Remember that arrays are zero-indexed, so the position of first element in the array is at index 0, and the second element in the array is at index 1, and so on.

[illegible]

6.4 Two Dimensional Array

A two-dimensional array, commonly known as a 2D array, it is a data structure specially designed to store elements in a grid or matrix-like arrangement with two dimensions. It essentially consists of an array of arrays, where each element in the 2D array is uniquely identified by two indices: a row index and a column index.

In most programming languages, a 2D array can be visualized as a rectangular table or grid, where each cell within the table holds a specific value. The first index corresponds to the row number, while the second index represents the column number.

The syntax for declaring a two-dimensional array in C is as follows:

```
data_type array_name[rows][columns];
```

An explanation of the components involved:

data_type: Refers to the data type of the elements that the 2D array will store. This can be any valid C data type, such as int, char, float, double, and so on.

array_name: Represents the identifier or name of the 2D array.

rows: Denotes the number of rows in the 2D array.

columns: Signifies the number of columns in the 2D array.

Two-dimensional arrays are particularly useful when there is a need to represent data in a tabular form, such as matrices, game boards, images, or any other data with two dimensions.

To access an element in a 2-dimensional array, you use the subscripts, i.e., the row index and column index of the array. For example:

```
int val = a[1][2];
```

The above statement will retrieve the value from the cell located in the 2nd row and 3rd column of the array.

[illegible]

A program where we have used nested loop to handle a two dimensional array:

```
#include <stdio.h>

int main() {
    // Define a 2D array with 3 rows and 4 columns
    int twoDArray[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };

    // Display the original 2D array
    printf("Original 2D Array:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            printf("%d ", twoDArray[i][j]);
        }
        printf("\n");
    }

    // Modify the elements in the 2D array using nested loops
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
             twoDArray[i][j] *= 2; // Multiply each element by
2
        }
    }

    // Display the modified 2D array
    printf("\nModified 2D Array:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
```

[illegible]
For Vivekananda Global University, Jaipur
Registrar

```
    // Input array elements
    printf("Enter array elements:\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
              printf("Element at row %d, column %d: ", i,
j);
            scanf("%d", &arr[i][j]);
        }    }
    // Print the array in Row-major form
    printf("Array in Row-major form:\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
    // Print the array in Column-major form
    printf("Array in Column-major form:\n");
    for (int j = 0; j < n; j++) {
        for (int i = 0; i < m; i++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

OutPut

```
Enter the number of rows (m): 2
Enter the number of columns (n): 3
Enter the elements of the array:
Element at row 0, column 0: 1
Element at row 0, column 1: 2
Element at row 0, column 2: 3
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
            printf("%d ", twoDArray[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

After the compilation and executed of above code, it produces the result like:

```
    a[0][0]: 1
    a[0][1]: 2
    a[1][0]: 5
    a[1][1]: 6
    a[2][0]: 3
    a[2][1]: 7
    a[3][0]: 4
    a[3][1]: 8
```

6.5 Array Representation

To demonstrate the representation of a 2D array in both row-major and column-major forms, we can write a C program that takes input for an array of order m x n and then prints the array contents in both forms.

```
#include <stdio.h>
int main() {
    int m, n;
    // Input the dimensions of the array
    printf("Enter the number of rows (m): ");
    scanf("%d", &m);
    printf("Enter the number of columns (n): ");
    scanf("%d", &n);
    int arr[m][n];
```

```
Element at row 1, column 0: 4
Element at row 1, column 1: 5
Element at row 1, column 2: 6
Array in Row-major form:
1 2 3
4 5 6
Array in Column-major form:
1 4
2 5
3 6
```

6.6 Sorting

Sorting is the process of arranging a collection of elements in a specific order, and sorting algorithms are used to achieve this arrangement. The order can be ascending or descending for numerical data, and lexicographical order for textual data (i.e., based on the alphabetical order of characters).

Some sorting examples are

- Employee Payroll: Sorting employee records based on salary or seniority level can help in making payroll processing more efficient and organized.
- E-commerce Websites: Sorting products based on various attributes such as price, popularity, or customer ratings allows users to find the desired products quickly.

6.7 Types of Sortings

I. **Bubble Sort**
II. **Insertion Sort**
III. **Selection Sort**
IV. **Quick Sort**
V. **Merge Sort**
VI. **Radix Sort**

[illegible]

6.8 Bubble Sort

Bubble Sort is a basic and direct sorting algorithm utilized to organize elements in a list or array. It achieves sorting by repeatedly comparing adjacent elements and performing swaps if they are in the incorrect order. Its name stems from the fact that smaller elements "bubble" to the beginning of the list as the sorting process advances.

The fundamental steps of the Bubble Sort algorithm are as follows:

1. Begin with the first element (index 0) and compare it with the next element (index 1).
2. If the first element is greater than the next element, perform a swap.
3. Move on to the next pair of elements (index 1 and index 2) and compare them.
4. Continue this process, comparing and swapping adjacent elements until reaching the end of the list.
5. Repeat the above steps for a number of passes equal to the number of elements in the list.

However, it is important to note that Bubble Sort is not efficient for large data sets due to its time complexity, which is O(n^2) in the worst, average, and best cases. Consequently, sorting large lists using Bubble Sort can result in significant time consumption.

Example:

Let's say we have the following unsorted list: [64, 34, 25, 12, 22, 11, 90]

Pass 1:

Compare 64 and 34. Swap them because 64 > 34. Result: [34, 64, 25, 12, 22, 11, 90]

Compare 64 and 25. Swap them because 64 > 25. Result: [34, 25, 64, 12, 22, 11, 90]

Compare 64 and 12. Swap them because 64 > 12. Result: [34, 25, 12, 64, 22, 11, 90]

Compare 64 and 22. Swap them because 64 > 22. Result: [34, 25, 12, 22, 64, 11, 90]

[illegible]

Compare 64 and 11. Swap them because 64 > 11. Result: [34, 25, 12, 22, 11, 64, 90]

Compare 64 and 90. No need to swap. Result: [34, 25, 12, 22, 11, 64, 90]

Pass 2:

Compare 34 and 25. Swap them because 34 > 25. Result: [25, 34, 12, 22, 11, 64, 90]

Compare 34 and 12. Swap them because 34 > 12. Result: [25, 12, 34, 22, 11, 64, 90]

Compare 34 and 22. Swap them because 34 > 22. Result: [25, 12, 22, 34, 11, 64, 90]

Compare 34 and 11. Swap them because 34 > 11. Result: [25, 12, 22, 11, 34, 64, 90]

Compare 34 and 64. No need to swap. Result: [25, 12, 22, 11, 34, 64, 90]

Pass 3:

Compare 25 and 12. Swap them because 25 > 12. Result: [12, 25, 22, 11, 34, 64, 90]

Compare 25 and 22. Swap them because 25 > 22. Result: [12, 22, 25, 11, 34, 64, 90]

Compare 25 and 11. Swap them because 25 > 11. Result: [12, 22, 11, 25, 34, 64, 90]

Compare 25 and 34. No need to swap. Result: [12, 22, 11, 25, 34, 64, 90]

Pass 4:

Compare 12 and 22. No need to swap. Result: [12, 22, 11, 25, 34, 64, 90]

Compare 22 and 11. Swap them because 22 > 11. Result: [12, 11, 22, 25, 34, 64, 90]

Compare 22 and 25. No need to swap. Result: [12, 11, 22, 25, 34, 64, 90]

Pass 5:

Compare 11 and 12. No need to swap. Result: [11, 12, 22, 25, 34, 64, 90]

Compare 12 and 22. No need to swap. Result: [11, 12, 22, 25, 34, 64, 90]

Pass 6:

Compare 11 and 12. No need to swap. Result: [11, 12, 22, 25, 34, 64, 90]

Since no swaps are performed during the last pass, the list is now sorted in ascending order: [11, 12, 22, 25, 34, 64, 90].

Bubble Sort in C

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

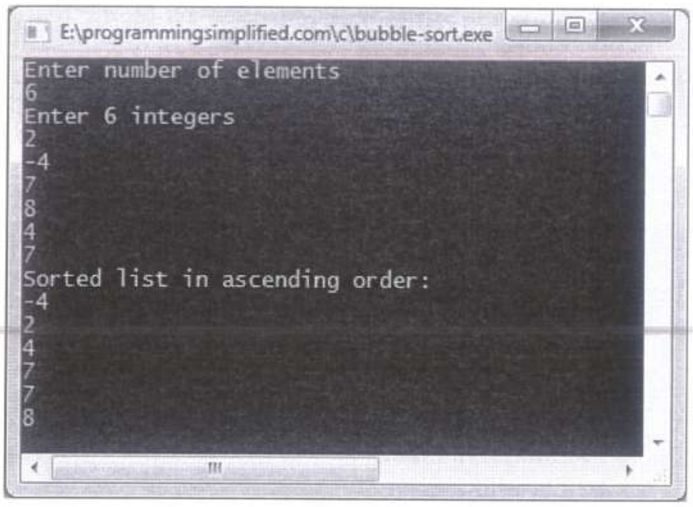
[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
  int array[100], n, c, d, swap;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  for (c = 0 ; c < n - 1; c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
       if (array[d] > array[d+1]) /* For decreasing order use
'<' instead of '>' */
      {
        swap       = array[d];
        array[d]   = array[d+1];
        array[d+1] = swap;
      }
    }
  }
  printf("Sorted list in ascending order:\n");
  for (c = 0; c < n; c++)
     printf("%d\n", array[c]);
 return 0;
}
```

[illegible]
For Vivekananda Global University, Jaipur



```
E:\programmingsimplified.com\c\bubble-sort.exe
Enter number of elements
6
Enter 6 integers
2
-4
7
8
4
7
Sorted list in ascending order:
-4
2
4
7
7
8
```

6.9 Insertion Sort

Insertion Sort is a sorting algorithm that constructs a sorted list by gradually inserting elements from an unsorted portion of the list into their appropriate positions within the sorted section. It operates as an in-place sorting algorithm, negating the need for additional data structures during the sorting process. Insertion Sort is particularly suitable for small lists or partially sorted data.

The underlying concept of Insertion Sort can be summarized as follows:

1. Begin with the first element of the list. As a single element can be viewed as a sorted list in itself, it is already in its correct position.
2. Move on to the next element and compare it with all the elements to its left in the sorted portion.
3. Insert the current element at the suitable position in the sorted segment, potentially shifting other elements if necessary to create space.
4. Repeat this process for all elements in the unsorted segment until the entire list is sorted.

Example:

Let's consider the following unsorted list: [12, 11, 13, 5, 6]

Initial State: [12, 11, 13, 5, 6]

[illegible]

For Vivekananda Global University, Jaipur

Registrar

- **Pass 1:** The first element 11 is compared with the element to its left, 12. Since 11 is smaller, they are swapped. Result: [11, 12, 13, 5, 6]
- **Pass 2:** The third element 13 is already in its correct position in the sorted part. No swaps are needed. Result: [11, 12, 13, 5, 6]
- **Pass 3:** The fourth element 5 is compared with the elements in the sorted part [11, 12, 13]. It is smaller than all of them and needs to be inserted at the beginning. The list becomes: [5, 11, 12, 13, 6]
- **Pass 4:** The last element 6 is compared with the elements in the sorted part [5, 11, 12, 13]. It is smaller than all of them and needs to be inserted at the beginning. The list becomes: [5, 6, 11, 12, 13]

The final sorted list in ascending order is: [5, 6, 11, 12, 13]

The time complexity of Insertion Sort is $O(n^2)$ in the worst, average, and best cases. This occurs because, in the worst case, each element needs to be compared and inserted for each position in the sorted part.

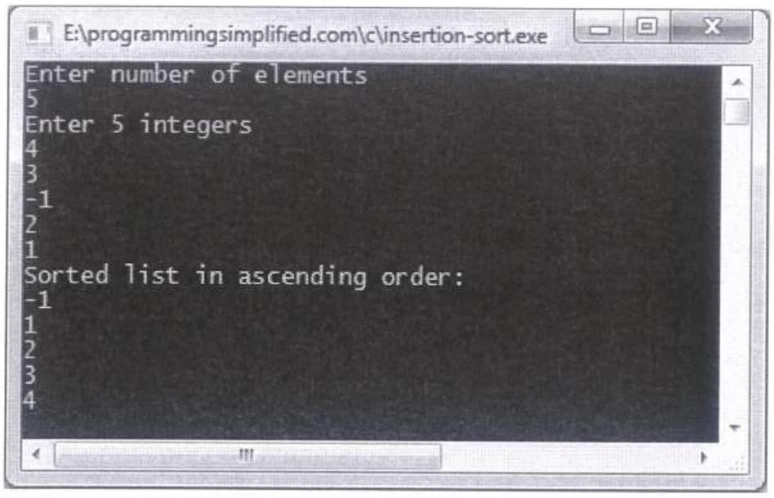
/* Insertion sort ascending order */

```
#include <stdio.h>
int main()
{
  int n, array[1000], c, d, t, flag = 0;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  for (c = 1 ; c <= n - 1; c++) {
    t = array[c];
    for (d = c - 1 ; d >= 0; d--) {
      if (array[d] > t) {
        array[d+1] = array[d];
        flag = 1;
```

[illegible]

```
        }
        else
          break;
      }
      if (flag)
        array[d+1] = t;
    }
    printf("Sorted list in ascending order:\n");
    for (c = 0; c <= n - 1; c++) {
      printf("%d\n", array[c]);
    }
    return 0;
}
```

Output:



E:\programmingsimplified.com\c\insertion-sort.exe

```
Enter number of elements
5
Enter 5 integers
4
3
-1
2
1
Sorted list in ascending order:
-1
1
2
3
4
```

1.11 Selection Sort

Selection Sort is an additional sorting algorithm that functions by repeatedly choosing the smallest (or largest, depending on the sorting order) element from the unsorted portion of an array and exchanging it with the leftmost element of the unsorted part. This procedure persists until the entire array is sorted.

A step-by-step explanation of how Selection Sort works:

[illegible]

For Vivekananda Global University, Jaipur

Registrar

1. The algorithm starts with the entire array as the unsorted part and an empty sorted part.
2. In each iteration, the algorithm searches for the smallest element in the unsorted part.
3. Once the smallest element is found, it is swapped with the leftmost element of the unsorted part, and this element becomes a part of the sorted section.
4. The boundary between the sorted and unsorted parts is moved one element to the right.
5. Steps 2 to 4 are repeated until the entire array becomes sorted.

While Selection Sort is easy to understand and implement. The time complexity of Selection sort is $O(n^2)$ in the average and worst cases.

Example:

Suppose we have the following unsorted list: [64, 34, 25, 12, 22, 11, 90]

Step 1:

The entire list is unsorted, and the sorted part is empty.

The smallest element is 11, and it is swapped with the leftmost element 64. Result: [11, 34, 25, 12, 22, 64, 90]

Step 2:

The first element 11 is now sorted.

The smallest element in the remaining unsorted part is 12, and it is swapped with the leftmost element 34. Result: [11, 12, 25, 34, 22, 64, 90]

Step 3:

The first two elements [11, 12] are now sorted.
The smallest element in the remaining unsorted part is 22, and it is swapped with the leftmost element 25. Result: [11, 12, 22, 34, 25, 64, 90]

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Step 4:

The first three elements [11, 12, 22] are now sorted.

The smallest element in the remaining unsorted part is 25, and it is swapped with the leftmost element 34. Result: [11, 12, 22, 25, 34, 64, 90]

Step 5:

The first four elements [11, 12, 22, 25] are now sorted.

The smallest element in the remaining unsorted part is 34, and it is swapped with the leftmost element 64. Result: [11, 12, 22, 25, 34, 64, 90]

Step 6:

The first five elements [11, 12, 22, 25, 34] are now sorted.

The smallest element in the remaining unsorted part is 64, and it is swapped with the leftmost element 90. Result: [11, 12, 22, 25, 34, 64, 90]

Step 7:

The entire list [11, 12, 22, 25, 34, 64, 90] is now sorted.

The final sorted list in ascending order is: [11, 12, 22, 25, 34, 64, 90]

Program:/* Selection Sort Program in C*/

```
#include <stdio.h>
int main()
{
  int array[100], n, c, d, position, t;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
   for (c = 0; c < (n - 1); c++) // finding minimum element
(n-1) times
  {
```

[illegible]

For Vivekananda Global University, Jaipur

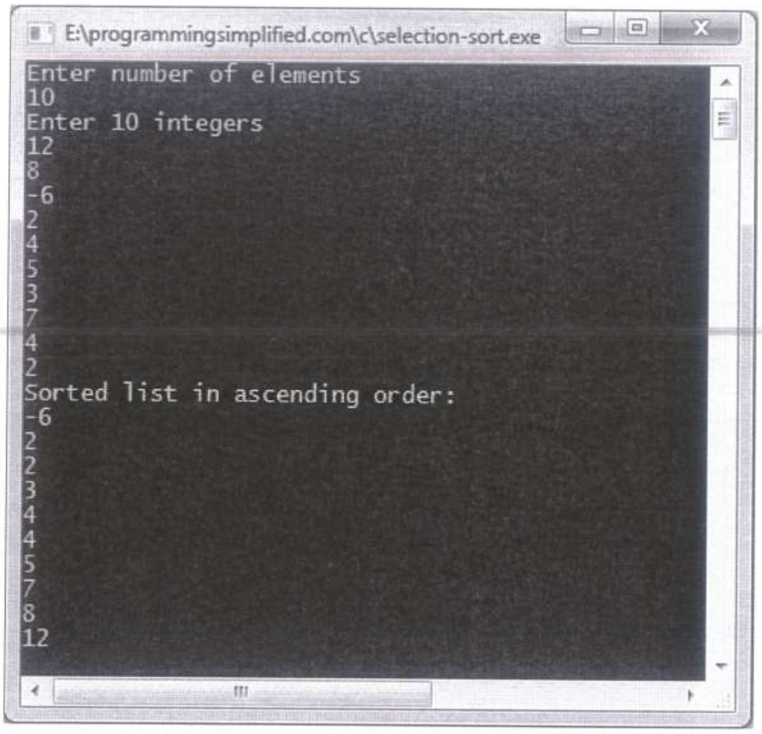
Registrar

```
    position = c;
    for (d = c + 1; d < n; d++)
    {
      if (array[position] > array[d])
        position = d;
    }
    if (position != c)
    {
      t = array[c];
      array[c] = array[position];
      array[position] = t;
    }
  }
  printf("Sorted list in ascending order:\n");
  for (c = 0; c < n; c++)
    printf("%d\n", array[c]);
  return 0;
}
```

Output

For Vivekananda Global University, Jaipur

[illegible]



```
Enter number of elements
10
Enter 10 integers
12
8
-6
2
4
5
3
7
4
2
Sorted list in ascending order:
-6
2
2
3
4
4
5
7
8
12
```

1.12 Quick Sort

Quick Sort is a widely used and efficient sorting algorithm based on the divide-and-conquer approach. It sorts an array by selecting a "pivot" element and partitioning the array into two arrays (sub-arrays): The elements having value less than the pivot and elements greater than the pivot. The pivot is placed in its correct sorted position, and the same process is recursively applied to the two sub-arrays until the entire array is sorted.

Here's a high-level overview of the Quick Sort algorithm:

- **Choose a Pivot:** Select a pivot element from the array. The choice of the pivot can significantly affect the algorithm's performance. Common approaches include selecting the first, last, middle, or a randomly chosen element as the pivot.
- **Partitioning:** Reorder the array in such a way that all elements less than the pivot come before it, and all elements greater than the pivot come

[illegible]

For Vivekananda Global University, Jaipur

Registrar

after it. The pivot will be in its final sorted position. The elements on the left and right of the pivot may not be sorted yet.

- **Recursion:** Recursively apply the Quick Sort algorithm to the sub-arrays on the left and right of the pivot until each sub-array contains zero or one element. In the end, all the sub-arrays will be sorted, leading to the complete sorted array.

The Quick Sort time complexity is O(n log n) in the average and best cases. However, in the worst case (when the pivot choice results in an imbalanced partition), the time complexity can degrade to $O(n^2)$.

Example:

Suppose we have the following unsorted list: [64, 34, 25, 12, 22, 11, 90]

Step 1:

- Choose the pivot. Let's select the middle element, which is 22.
- Partition the list into two sub-arrays: elements less than 22 and elements greater than 22.
- Sub-array with elements less than 22: [11, 12]
- Sub-array with elements equal to 22: [22]
- Sub-array with elements greater than 22: [64, 34, 25, 90]

Step 2:

- Apply the Quick Sort algorithm recursively to the sub-arrays.
- For the sub-array with elements less than 22, the pivot is 11.
- Partition the sub-array: [11, 12] -> [11], [12]. Since these sub-arrays have only one element, they are considered sorted.

Step 3:

- For the sub-array with elements greater than 22, the pivot is 34.
- Partition the sub-array: [34, 25, 90] -> [25], [34, 90]. Again, these sub-arrays have only one element, so they are considered sorted.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Step 4:

- Combine the sorted sub-arrays: [11], [22], [25], [34, 90].
- Combine [34, 90] as [34, 90] is already sorted.

Step 5:

- Combine the sorted sub-arrays: [11], [22], [25], [34, 90], [64].

Step 6:

- Combine the sorted sub-arrays: [11, 22, 25, 34, 64, 90].
- The final sorted list in ascending order is: [11, 22, 25, 34, 64, 90]

/*Quick Sort in C*/

```
#include<stdio.h>
void quicksort(int number[25],int first,int last){
   int i, j, pivot, temp;

   if(first<last){
      pivot=first;
      i=first;
      j=last;
      while(i<j){
         while(number[i]<=number[pivot]&&i<last)
            i++;
         while(number[j]>number[pivot])
            j--;
         if(i<j){
            temp=number[i];
            number[i]=number[j];
            number[j]=temp;
         }
      }
      temp=number[pivot];
```

[illegible]

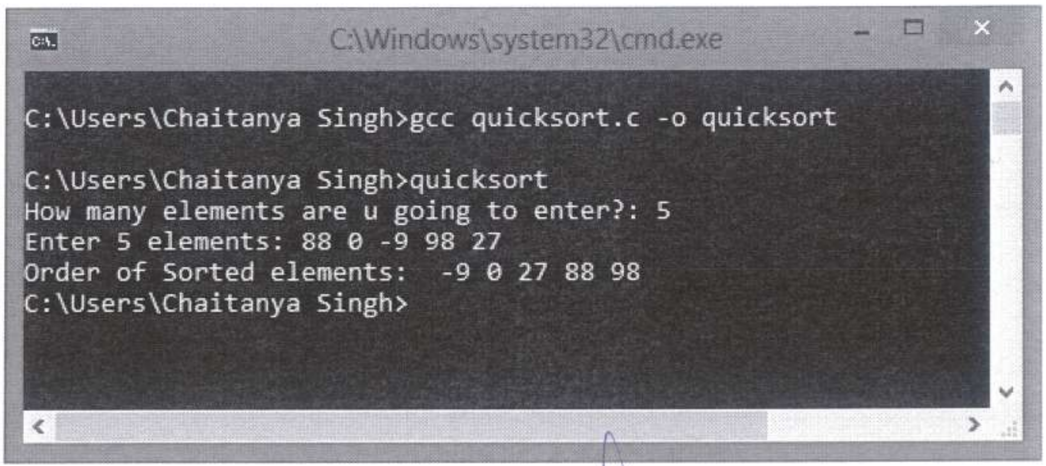
For Vivekananda Global University, Jaipur

```
         number[pivot]=number[j];
         number[j]=temp;
         quicksort(number,first,j-1);
         quicksort(number,j+1,last);
      }
   }
int main(){
   int i, count, number[25];

   printf("How many elements are u going to enter?: ");
   scanf("%d",&count);

   printf("Enter %d elements: ", count);
   for(i=0;i<count;i++)
      scanf("%d",&number[i]);
   quicksort(number,0,count-1);
   printf("Order of Sorted elements: ");
   for(i=0;i<count;i++)
      printf(" %d",number[i]);
   return 0;
}
```

Output:



C:\Windows\system32\cmd.exe

```
C:\Users\Chaitanya Singh>gcc quicksort.c -o quicksort

C:\Users\Chaitanya Singh>quicksort
How many elements are u going to enter?: 5
Enter 5 elements: 88 0 -9 98 27
Order of Sorted elements:  -9 0 27 88 98
C:\Users\Chaitanya Singh>
```

For Vivekananda Global University, Jaipur

Registrar

1.13 Merge Sort

Merge Sort is another popular sorting algorithm based on the divide-and-conquer approach. It efficiently sorts an array by dividing it into two halves, recursively sorting the two halves, and then merging them back together into a single sorted array.

Here's a high-level overview of the Merge Sort algorithm:

- Divide: Divide the unsorted array into two halves at the middle.
- Conquer: Recursively sort each of the two halves using Merge Sort.
- Merge: Merge the two sorted halves to create a single sorted array.

The merging step is the key operation in Merge Sort. It involves comparing elements from the two sorted halves and combining them into a single sorted array. During the merging process, elements are compared and arranged in their correct sorted order.

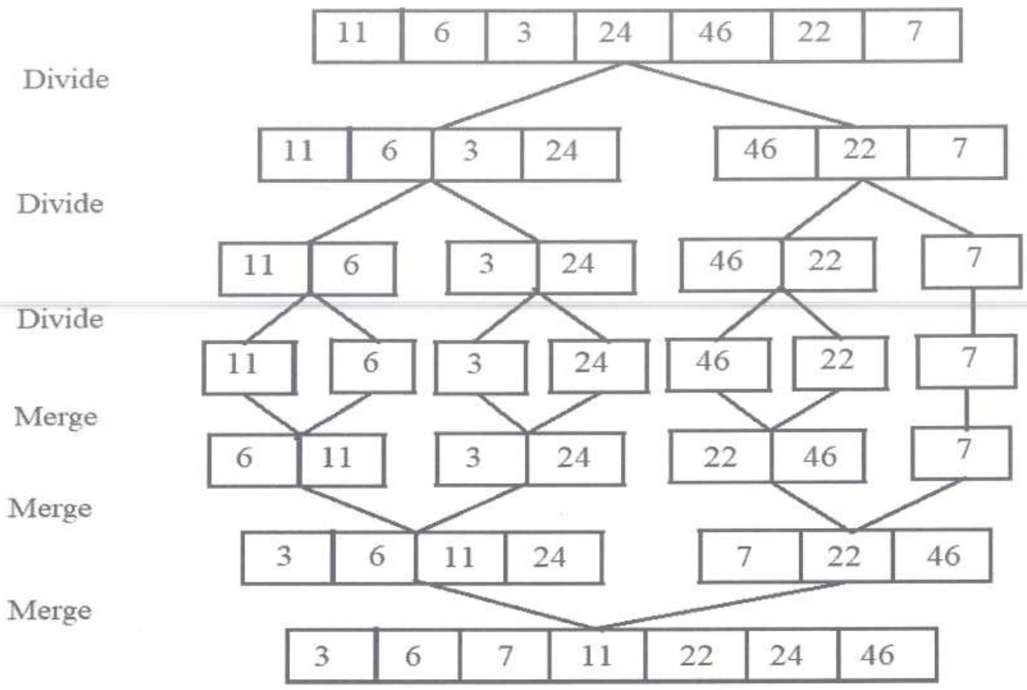
The Merge Sort time complexity is O(n log n) in all cases, including the worst, average, and best cases.

Example

[illegible]

For Vivekananda Global University, Jaipur

Registrar



1.14 Radix Sort

Radix Sort is a non-comparative sorting algorithm that efficiently organizes elements by utilizing a method of distributing them into distinct buckets based on their digits. The distribution process starts from the least significant digit and proceeds towards the most significant digit. Subsequently, the elements are combined from the buckets, resulting in the formation of the final sorted array.

This sorting method finds particular utility in sorting numbers with a fixed number of digits, such as integers or strings of equal length.

A high-level overview of the Radix Sort algorithm:

1. Determine the maximum number of digits: Identify the number with the maximum number of digits (maximum length) within the input array. This step is crucial for determining the number of passes required to achieve the sorting.
2. Bucket Distribution: Commencing from the least significant digit (rightmost digit), distribute the elements into separate buckets based on the value of

[illegible]

the current digit. Elements sharing the same digit value are placed into the same bucket.

3. Bucket Combination: Merge the elements from all the buckets to reconstruct the array. This step partially sorts the elements based on the current digit.
4. Repeat: Iterate through steps 2 and 3 for the next digit, progressing towards the most significant digit. This iterative process continues for all digits in the numbers.
5. By performing the bucket distribution and combination for each digit, Radix Sort correctly sorts the elements, taking into account the value of all digits in each pass.

Suppose we have the following unsorted list of integers: [170, 45, 75, 90, 802, 24, 2, 66]

Step 1: Find the maximum number of digits in the array

- The maximum number in the array is 802, which has three digits. So, we will perform three passes for sorting based on each digit's place value.

Step 2: Pass 1 (sorting based on the least significant digit, units place)

Bucket Distribution:

Bucket 0: [170, 90]
Bucket 1: [801]
Bucket 2: [2]
Bucket 3: [803]
Bucket 4: [24]
Bucket 5: [75]
Bucket 6: [66]
Bucket 7: [ ]
Bucket 8: [802]
Bucket 9: [45]

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Bucket Combination: Combine the elements from all the buckets: [170, 90, 801, 2, 803, 24, 75, 66, 802, 45]

Step 3: Pass 2 (sorting based on the tens place)

Bucket Distribution:

Bucket 0: [802, 2]

Bucket 1: [801]

Bucket 2: [24]

Bucket 3: [803, 45]

Bucket 4: [75]

Bucket 5: []

Bucket 6: [66]

Bucket 7: [170]

Bucket 8: [90]

Bucket 9: []

Bucket Combination: Combine the elements from all the buckets: [802, 2, 801, 24, 803, 45, 75, 66, 170, 90]

Step 4: Pass 3 (sorting based on the hundreds place)

Bucket Distribution:

Bucket 0: [2]

Bucket 1: [24]

Bucket 2: []

Bucket 3: [45]

Bucket 4: [66]

Bucket 5: [75]

Bucket 6: [90]

Bucket 7: [170]

Bucket 8: [801, 802]

Bucket 9: [803]

Bucket Combination: Combine the elements from all the buckets: [2, 24, 45, 66, 75, 90, 170, 801, 802, 803]

Step 5: The entire list [2, 24, 45, 66, 75, 90, 170, 801, 802, 803] is now sorted.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

The final sorted list in ascending order is: [2, 24, 45, 66, 75, 90, 170, 801, 802, 803]

1.15 Summary

Arrays :

- Array is a basic data structure that allows storing multiple elements of the same data type in contiguous memory locations.
- They are declared using the syntax datatype array_name[size]; and accessed using an index, starting from [0 - size -1].
- One-dimensional arrays are simple collections of elements, while multidimensional arrays can have multiple dimensions represented by multiple indices.
- C supports dynamic arrays, which can change size during runtime using functions like malloc and realloc.
- Arrays can be initialized during declaration with specific values, and elements can be traversed using loops.

Sorting Techniques in C:

- Sorting is the process of arranging elements in a particular order either ascending or descending.
- In-place sorting algorithms rearrange elements within the original array without using additional memory, while out-of-place sorting creates a new sorted array.
- Bubble sort repeatedly compares and swaps adjacent elements until the array is sorted, with a time complexity of O(n^2).
- Selection sort divides the array into sorted and unsorted parts, selecting the smallest element and placing it in the appropriate position for each iteration, also with a time complexity of O(n^2).
- Insertion sort used for the final sorted array one element at a time by inserting elements into their correct positions, with a time complexity of O(n^2) for average and worst cases, but O(n) for best case with already sorted data.

[illegible]

- Merge sort is based on divide-and-conquer algorithm that recursively divides the array, sorts the halves, and then merges them, with a time complexity of O(n log n).
- Quick sort, another divide-and-conquer algorithm, selects a pivot element, partitions the array, and recursively sorts sub-arrays, with an average-case time complexity of O(n log n) and worst-case of O(n^2).
- Radix sort is a different, non-comparative based sorting algorithm that processes elements digit by digit or character by character, making it suitable for specific scenarios.

1.16 Review Questions

1 How do you declare a one-dimensional array in C?

2 Explain the concept of the array index and how it relates to accessing elements.

3 How do you initialize an array during declaration?

4 Can you have an array of characters to represent a string in C? How do you handle strings in C?

5 What is a multidimensional array in C? Provide an example of its declaration and usage.

6 How an array passed to a function in C? Are there any special considerations?

7 What is the sorting algorithm's primary goal, and why is it essential in programming?

8 What is the time complexity of the bubble sort algorithm, and why is it generally inefficient for large datasets?

9 Describe the working principle of the selection sort algorithm. What is its time complexity?

10 Compare the efficiency of bubble sort and selection sort, and mention scenarios where each might be more suitable.

1.17 Keywords

- Index: An integer value used to access individual elements within an array.
- One-dimensional array: An array with a single index used to access elements.
- Multidimensional array: An array with multiple indices used to access elements arranged in multiple dimensions.
- Dynamic array: An array whose size can be dynamically changed during runtime using functions like malloc and realloc.
- Array initialization: Assigning initial values to elements during array declaration.
- Array traversal: Iterating through an array's elements to process or display them.
- Sorting: Arranging elements in a specific order (e.g., ascending or descending).
- In-place sorting: Sorting algorithms that rearrange elements within the original array without using additional memory.
- Bubble sort: A simple sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order.
- Selection sort: A sorting algorithm that divides the array into sorted and unsorted parts and selects the smallest element for each iteration.
- Insertion sort: A sorting algorithm that builds the final sorted array one item at a time by inserting elements into their correct position.
- Merge sort: A divide-and-conquer sorting algorithm that recursively divides the array into halves, sorts them, and then merges the sorted halves.
- Quick sort: A divide-and-conquer sorting algorithm that selects a pivot element, partitions the array, and recursively sorts sub-arrays.
- Stability in sorting: A property of sorting algorithms that preserves the relative order of equal elements in the sorted output.

[illegible]

- Time complexity: A measure of the time a sorting algorithm takes to complete based on the input size.
- adix sort: A non-comparative sorting algorithm that sorts elements by processing individual digits or characters.

1.18 References

1. E.Balaguruswamy, "Computing Fundamentals & C Programming", TataMcGraw Hill, 2008.
2. B. A. Forouzan & R. F. Gilberg "Computer Science – A structured programming Approach Using C", 2011
3. E. Balaguruswamy, "Programming in ANSI" Tata McGraw Hill, 2011.

[illegible]
For Vivekananda Global University, Jaipur
Registrar

Table of Content

[illegible]

For Vivekananda Global University, Jaipur

Learning Objectives

After studying this unit,the student will be able to:

- String Operations: Understand the fundamental operations that can be performed on strings, such as concatenation, substring extraction, string length, and string comparison. Learn how to manipulate strings to achieve specific tasks efficiently.
- Built-in String Handling Functions: Familiarize yourself with the various built-in functions programming languages provide for handling strings. Learn how to use these functions to perform common tasks like searching for substrings, converting case, replacing characters, and more.
- String Manipulation Programs: Develop the ability to design and implement programs that involve string manipulation to solve real-world problems. Practice writing code to perform tasks like reversing a string, counting occurrences of a substring, checking for palindromes, and more.
- Text Processing: Explore text processing techniques using strings. Learn about pattern-matching algorithms, regular expressions, and how to extract meaningful information from textual data.
- String Formatting: Learn about string formatting and how to construct strings with dynamic content using placeholders.

Introduction

In the C programming language, a string is a sequence of characters stored in an array, with termination marked by the null character '\0'. Strings are essential when dealing with textual data. Unlike certain high-level languages, C does not have a built-in string data type; instead, strings are represented as arrays of characters.

Features of Strings in C:

• Null-Terminated: A null-terminated string in C is one that ends with the null character '\0', indicating the string's conclusion. Upon declaring and initializing strings in C, the null character is automatically added.

• Character Array: Strings are represented as arrays of characters, where each character in the array corresponds to a letter in the string.

• Fixed Size: C arrays have a fixed size, thus the length of a string is constrained by the array's size. Care must be taken not to exceed the array size to prevent buffer overflows.

Advantages of Strings in C:

• Text Handling: Strings simplify the handling of textual data, enabling reading, modification, and display of textual material.

• Concise Representation: Expressing a sequence of characters using a single variable makes code concise and easily readable.

• Standard Library Support: C offers a standard library of string handling functions, facilitating tasks like concatenation, comparison, and copying.

Applications of Strings in C:

[illegible]

For Vivekananda Global University, Jaipur

Registrar

• Input/Output Activities: Strings are commonly used for handling textual data input and output, such as receiving user input or writing messages to the terminal.

• Data Manipulation: Strings find extensive use in data manipulation activities like parsing data, tokenization, and searching for specific patterns within text.

• Text Processing: Text processing plays a critical role in various applications, including text editors, word processors, and compilers. Strings are indispensable in these applications.

• String Operations: Numerous string manipulation operations such as concatenation, comparison, and substring extraction are essential for various string-related applications.

• File Operations: Strings are employed in file operations to read, write, and process textual data stored in files.

A strings in C are represented as character arrays and are utilized for managing textual data across diverse applications. Their null-terminated nature enables easy use with standard library functions. Strings are indispensable in C programming for text processing, data manipulation, and various other operations involving textual data..

7.1 String

A string is a sequential arrangement of characters utilized to represent textual data in various fields, including computer programming. It is a collection of letters, numbers, symbols, or spaces that form a coherent piece of text. Strings find extensive use in managing and processing textual data across diverse programming languages and applications.

Representation of Strings:

Strings are typically stored in contiguous memory regions as character sequences. Characters within strings are encoded using specific character sets,

For Vivekananda Global University, Jaipur
[illegible]
Registrar

such as ASCII, UTF-8, or UTF-16, to represent letters, numbers, symbols, and other characters.

Character arrays serve as the predominant method for representing strings in programming languages. In this representation, a string is an array of characters, terminated by a null character ('\0') that marks the end of the string. The null character acts as a sentinel value, signifying the conclusion of the text and preventing unexpected behavior during string operations.

For instance, the string "Hello" is represented as an array of characters in the following manner:

'S' | 't' | 'r' | 'i' | 'n' | '\0'

Each character in the string is stored in its designated memory location, resulting in a continuous sequence. When reading or modifying a string, the software employs the array index to access individual characters or traverse the entire string. This enables efficient handling and manipulation of textual data.

7.2 String Declaration and Initialization

Statements and blocks are critical components in C programming because they determine the flow of execution and code organization inside a program.

A string in language C is declared in the following manner:

char temp[5];

[illegible]

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

In this way, we can initialize a string of length 5.

String Initialization

String initialization is possible in many different ways and some of them are

```
 char s[]= ″temp string″;

char s[5]= ″array″;

char s[]={'t','e','m', 'p','\0'};

char s[5]={'t','e','m', 'p','\0'};
```

t	e	m	p	\0
s[0]	s[1]	s[2]	s[3]	s[4]

Assigning Values to Strings

Assigning values to string arrays is not supported directly through assignment operators. Once strings are declared, you cannot assign values to string-type variables in the same way as you do for other data types. For instance, in the C language, you cannot write and assign values to a string array like this:

char s[50];

s = "string value";

For Vivekananda Global University, Jaipur

[illegible]

Registrar

7.3 String Operations on String

Function	Description
strlen()	can determine the string's length.
Strcpy()	may transfer a string's information to another
Strcat()	enables the joining or concatenation of two strings.
Strcmp()	ability to compare two strings
Strlwr()	Can lowercase the string after conversion
Strupr()	is utilized to change the string's letters to uppercase.
Strrev()	is used to make the string reverse

7.4 Built-in String Handling Function

String handling functions are built-in and can be used to perform typical operations on strings. These routines are frequently used for string manipulation and text processing. Here are some of the most often used string handling functions built into C:

1. **strlen():**

The length of a string (means the total number of characters in the string other than the null terminator) is returned.

strlen() is another string header file function that can be used directly on strings. When you need to know the length of a string, you can use the string function in

C, strlen(). The strlen() string functions in C determine the length of a given string. However, one can construct a program manually to find the length of any string, however using this direct method can save you time, as seen below:

Example:

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[] = "Hello, World!";
    int length = strlen(str);
    printf("String: %s\n", str);
    printf("Length of the string: %d\n", length);
    return 0;
}
```

```
Output:
String: Hello, World!
Length of the string: 13
```

In this example, the strlen() function is used to calculate the length of the string "Hello, World!", which is 13 characters (including the null-terminator \0). The calculated length is then printed to the console. Remember that the length includes all characters in the string, including spaces and punctuation, up to the null-terminator.

2. **strcpy():**

The strcpy() function in C can be used to copy one string to another. The source string and the destination string are its two required parameters. Up until the null terminator ('0') of the source string, the function copies each character from the source string into the destination string.When you need to copy the contents of one string into another, you use this method. Even null characters are transferred throughout the procedure.

Syntax of the function is strcpy()

Example:

```
#include <stdio.h>
#include <string.h>
int main() {
    char source[] = "Hello, World!";
        char  destination[20];  //  Make  sure  the
destination array has enough space
    strcpy(destination, source);
    printf("Source: %s\n", source);
    printf("Copied String: %s\n", destination);
    return 0;
}
Output:
Source: Hello, World!
Copied String: Hello, World!
```

3. **strcat():**

The function "strcat" is employed to append one string to the end of another string. It allows the concatenation of two character strings, essentially joining them end-to-end. During the "strcat" operation, the null character of the destination string will be overwritten by the first character of the source string.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Subsequently, the previous null character will be added at the end of the newly created destination string, resulting from the "strcat" operation.

To use the "strcat" function, the user must provide two arguments:

i) "src" - This argument represents the source string that needs to be appended.

ii) "dest" - This argument denotes the destination string to which the source string will be appended.

The "src" string is specified in place of the "src" argument, while the destination string, to which the source string will be appended, is specified in place of the "dest" argument.

Example

```
#include <stdio.h>

#include <string.h>

int main() {

    char str1[20] = "Hello, ";

    char str2[] = "World!";

    strcat(str1, str2);

    printf("Concatenated String: %s\n", str1);

    return 0;

}

Output:

Concatenated String: Hello, World!
```

In this example, the strcat() function is used to concatenate the content of the str2 string ("World!") to the end of the str1 string ("Hello, "). The resulting concatenated string is then printed to the console. It's important to ensure that the destination string (in this case, str1) has enough space to accommodate the concatenated result, including the null-terminator. Otherwise, you might encounter buffer overflow issues.

4. **strcmp():**

lexicographically compares two strings. Returns a value of 0 if the strings are equal, a value of a negative sign if the first string is lexicographically inferior to the second, and a value of a positive sign if the first string is lexicographically superior to the second.The strcmp() function can be used to compare two strings to determine whether or not they are identical.This string compares two strings using C functions.

When comparing the strings, the following two factors are taken into consideration:

1. str1
2. str2

By comparing two string the return value be determined on the basis of the strings setup as shown below.

The function outputs a discrete value that can be one of the following: 0, >0, or 0. 'A' and 'a' are treated as different letters in this method since the two values given are case sensitive.

Example:

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

int main() {
   char str1[] = "apple";
   char str2[] = "banana";
   int result = strcmp(str1, str2);
   if (result < 0) {
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
    printf("'%s' is less than '%s'\n", str1, str2);
  } else if (result > 0) {
    printf("'%s' is greater than '%s'\n", str1, str2);
  } else {
    printf("'%s' is equal to '%s'\n", str1, str2);
  }
  return 0;
}
```

Output:

```
'apple' is less than 'banana'
```

In this example, the strcmp() function is used to compare the strings "apple" and "banana". The return value of strcmp() indicates whether the first string is less than, greater than, or equal to the second string. The comparison result is then used to determine the relationship between the two strings, and the appropriate message is printed to the console.

5. **strchr():**

Identifies the first time a character appears in a string and then either returns a pointer to that character or NULL if the character cannot be located.

Example:

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

int main() {
    char str[] = "Hello, World!";
    char target = 'W';
    char *result = strchr(str, target);
    if (result != NULL) {
```

For Vivekananda Global University, Jaipur

[illegible]

Registrar

```
            printf("Character '%c' found at position:
%ld\n", target, result - str);
    } else {
            printf("Character '%c' not found in the
string.\n", target);
    }
    return 0;
}
Output:
Character 'W' found at position: 7
```

In this example, the strchr() function is used to search for the character 'W' within the string "Hello, World!". If the character is found, the function returns a pointer to its first occurrence in the string.

6. **strstr():**

It looks for a substring's first appearance in a string and then either provides a reference to the substring's beginning or NULL if the substring cannot be found.

Example:

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[] = "Hello, World!";
    char substring[] = "World";
    char *result = strstr(str, substring);
    if (result != NULL) {
        printf("Substring '%s' found at position: %ld\n",
substring, result - str);
```

[illegible]
For Vivekananda Global University, Jaipur
Registrar

```
    } else {
                printf("Substring '%s' not found in the
string.\n", substring);
    }
    return 0;
}
Output:
Substring 'World' found at position: 7
```

In this example, the strstr() function is used to search for the substring "World" within the string "Hello, World!". If the substring is found, the function returns a pointer to its first occurrence in the string. The position of the substring is then calculated by subtracting the base address of the string (str) from the result pointer. If the substring is not found, the function returns NULL, and an appropriate message is printed to the console.

7. **puts() and gets():**

gets and puts, two popular string header file methods, are used to display and receive user input, respectively.In order to briefly describe how the string handling functions of puts and gets in C operate, it is helpful to note that the gets() method allows the user to verify that characters have been typed before pressing the enter key. The user may also add strings that are separated from one another. The puts() function, which is also one of the string types in C, is used to publish a line for the output screen. It functions similarly to how printf() does.

A program for these functions

```
#include <stdio.h>

          int main() {
              char name[50];
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
    printf("Enter your name: ");

    gets(name);

    printf("Hello, ");

    puts(name);

    printf("Nice to meet you!\n");

    return 0;

}

Enter your name: Alice

Hello, Alice

Nice to meet you!
```

In this example, the gets() is used as function to read a line of text (including spaces) from the user input into the name array. Then, the puts() function is used to print the content of the name array along with additional output. It's important to note that the gets() function is generally not recommended for use due to potential buffer overflow vulnerabilities. Instead, using fgets() is safer.

Here's an alternative version of the program using fgets():

```
#include <stdio.h>

int main() {

    char name[50];

    printf("Enter your name: ");

    fgets(name, sizeof(name), stdin);

    printf("Hello, ");

    puts(name);

    printf("Nice to meet you!\n");
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
    return 0;
}
```

7.5 Programs on String

Some example programs that involve string operations in C:

1. Reversing a String:

Explanation of the Logic:

- The program defines a function reverseString to reverse the input string.
- The function takes the string as an argument and calculates its length using strlen.
- It uses two pointers start and end, initialized to the first and last index of the string, respectively.
- It uses a while loop to swap characters from the start and end positions, and then moves the start pointer forward and the end pointer backward until they meet at the middle of the string.

```
#include <stdio.h>

#include <string.h>

// Function to reverse the string

void reverseString(char str[]) {

    int length = strlen(str);

    int start = 0;

    int end = length - 1;


    while (start < end) {
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
            char temp = str[start];

            str[start] = str[end];

            str[end] = temp;

            start++;

            end--;

        }

    }


    int main() {

        char inputString[100];

        printf("Enter a string: ");

        scanf("%[^\n]", inputString);

        // Reverse the string

        reverseString(inputString);

        printf("Reversed string: %s\n", inputString);

        return 0;

    }
```

```
Enter a string: Hello, World!

Reversed string: !dlroW ,olleH
```

2. **Counting Occurrences of a Character:**

```
#include <stdio.h>

#include <string.h>

int countOccurrences(char str[], char ch) {
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
    int count = 0;

    int length = strlen(str);

    for (int i = 0; i < length; i++) {

        if (str[i] == ch) {

            count++;

        }

    }

    return count;

}

int main() {

    char message[] = "Hello, World!";

    char searchChar = 'l';

       int occurrences = countOccurrences(message,
searchChar);

     printf("Number of occurrences of '%c': %d\n",
searchChar, occurrences);

    return 0;

}

Output: Number of occurrences of 'l': 3
```

3. **Checking for Palindrome:**

```
#include <stdio.h>

#include <string.h>

#include <stdbool.h>
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
bool isPalindrome(char str[]) {
    int left = 0;
    int right = strlen(str) - 1;

    while (left < right) {
        if (str[left] != str[right]) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
int main() {
    char word[] = "level";

    if (isPalindrome(word)) {
        printf("%s is a palindrome.\n", word);
    } else {
        printf("%s is not a palindrome.\n", word);
    }
        return 0;
}
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
Output: "level is a palindrome."
```

4. **String Palindrome Using strrev() (non-standard, Windows-specific function):**

```
#include <stdio.h>
#include <string.h>
int main() {
    char word[] = "level";
    char reversed[50];

    strcpy(reversed, word);
    strrev(reversed);

    if (strcmp(word, reversed) == 0) {
        printf("%s is a palindrome.\n", word);
    } else {
        printf("%s is not a palindrome.\n", word);
    }
        return 0;
}
Output: "level is a palindrome."
```

5. **Counting Words in a Sentence:**

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

[illegible]
For Vivekananda Global University, Jaipur
Registrar

```
#include <string.h>


int countWords(char sentence[]) {

    int count = 0;

    int length = strlen(sentence);

    bool inWord = false;

    for (int i = 0; i < length; i++) {

            if (sentence[i] == ' ' || sentence[i] ==
'\t' || sentence[i] == '\n') {

            inWord = false;

        } else if (!inWord) {

            inWord = true;

            count++;

        }

    }

    return count;

}

int main() {

    char sentence[] = "Hello, how are you today?";

    int wordCount = countWords(sentence);

    printf("Number of words in the sentence: %d\n",
wordCount);

    return 0;

}
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
Output: 6
```

6. **Checking for Anagrams:**

Explanation of the Logic:

- The program defines a function areAnagrams to check if two input strings are anagrams.
- The function takes two strings as arguments and calculates their lengths using strlen.
- It initializes a count array count of size 256 to store the frequency of each character (ASCII characters).
- It counts the frequency of characters in the first string str1 and stores it in the count array.
- It decrements the frequency of characters in the second string str2 from the count array.
- If all the frequency values in the count array are zero, it means both strings have the same characters and their frequency, making them anagrams. Otherwise, they are not anagrams.

```
#include <stdio.h>

#include <string.h>

// We are writing a program, A function to check if two
strings are anagrams

int areAnagrams(char str1[], char str2[]) {

    int len1 = strlen(str1);

    int len2 = strlen(str2);


     // If the lengths of the strings are different, they
cannot be anagrams

    if (len1 != len2)
```

For Vivekananda Global University, Jaipur
[illegible]
Registrar

```
        return 0;

      int count[256] = {0}; // Initialize a count array to store
the frequency of each character


    // Count the frequency of characters in str1

    for (int i = 0; i < len1; i++) {

        count[str1[i]]++;

    }

    // Decrement the frequency of characters in str2

    for (int i = 0; i < len2; i++) {

        count[str2[i]]--;

    }


     // Check if all the frequency values in the count array
are zero

    for (int i = 0; i < 256; i++) {

        if (count[i] != 0)

            return 0;

    }

    return 1;

}

int main() {

   char str1[100], str2[100];

    printf("Enter the first string: ");
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
    scanf("%s", str1);
    printf("Enter the second string: ");
    scanf("%s", str2);
    // Check if the strings are anagrams
    if (areAnagrams(str1, str2)) {
        printf("The two strings are anagrams.\n");
    } else {
        printf("The two strings are not anagrams.\n");
    }
    return 0;
}
```

```
Enter the first string: listen
Enter the second string: silent
The two strings are anagrams.
```

7. **Removing Duplicates from a String:**

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

void removeDuplicates(char str[]) {
    int length = strlen(str);
    int index = 0;
```

For Vivekananda Global University, Jaipur

[illegible]

Registrar

```
    for (int i = 0; i < length; i++) {
        int j;
        for (j = 0; j < index; j++) {
            if (str[i] == str[j]) {
                break;
            }
        }
        if (j == index) {
            str[index++] = str[i];
        }
    }
    str[index] = '\0';
}
int main() {
    char str[] = "programming";
    printf("Original String: %s\n", str);
    removeDuplicates(str);
    printf("String after removing duplicates: %s\n",
str);
    return 0;
}
Output: "progamin"
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

7.6 Summary

In programming, a string is represented as an array of characters, where each character is terminated by a null character ('\0'). Strings is a sequences of characters that can be manipulated and processed to accomplish various tasks. In the C programming language, there is no specific data type for strings, and they are typically handled as character arrays.

Points about strings in C:

• Strings are character arrays.

• Strings are null-terminated, with a null character ('\0') marking the end of the string.

• String operations include concatenation, comparison, searching, tokenization, and more.

• Care must be taken to avoid buffer overflows and ensure proper null-termination.

Summary of Built-in String Functions:

A set of built-in string functions through the <string.h> header. These functions simplify common string operations and help avoid manual handling of character arrays. Some essential built-in string functions are:

• strlen(): Calculates the length of a string.

• strcpy(): Copies one string to another.

• strcat(): Concatenates two strings.

• strcmp(): Compares two strings lexicographically.

• strstr(): this function finds the first occurrence of a sub-string in a string.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

• strchr(): this function finds the first occurrence of a character in a string.

• strtok(): Breaks a string into smaller tokens based on a delimiter.

String operations involve various manipulations and actions performed on strings in programming. In different programming languages, including C, strings are represented as arrays of characters, with each character terminated by a null character ('\0'). Here is a summary of common string operations:

• String Manipulation: Modifying or processing strings to achieve specific tasks, such as extracting substrings, changing case, or removing duplicates.

• Concatenation: Combining two or more strings into a single string.

• Substring: Extracting a portion of a string based on a specified range or position.

• Search: Finding the occurrence of a substring or character in a given string.

• Replace: Replacing occurrences of a substring or character with another in a string.

• Reverse: Reversing the order of characters in a string.

• Count Occurrences: Determining the number of times a substring or character appears in a string.

• Palindrome: Identifying if a string reads the same backward as forward.

• Anagrams: Checking if two strings contain the same characters but may have a different order.

[illegible]

• Duplicate Removal: Eliminating duplicate characters or substrings from a string.

• Tokenization: Breaking a string into smaller components called tokens based on a delimiter.

• Vowels and Consonants: Identifying and working with the vowels (e.g., 'a', 'e', 'i', 'o', 'u') and consonants in a string.

• Compression: Reducing the size of a string by eliminating redundancies.

• Case Conversion: Changing the letter case (uppercase or lowercase) of characters in a string.

• Capitalization: Making the first character of each word uppercase in a string.

7.7 Review Questions

1. What are string operations on strings?
2. What is the role of the <string.h> library in C programming?
3. List some common built-in string handling functions provided by <string.h>.
4. Write a C program to find the length of a given string using strlen().
5. Explain the strcpy() function and provide an example of its usage.
6. WAP to concatenate two strings using strcat().
7. What does the strcmp() function do? Provide an example of comparing two strings.
8. WAP to count the occurrences of a specific character in a string.
9. How to check a string is palindrome? Write a C program to demonstrate this.
10. Explain the process of tokenizing a sentence using strtok(). Provide a C program to tokenize a sentence into words.

For Vivekananda Global University, Jaipur
[illegible]
Registrar

7.8 Keywords

- String operations: Manipulations and actions performed on strings in programming.
- String manipulation: Modifying or processing strings to achieve specific tasks.
- Concatenation: Combining two or more strings into a single string.
- Substring: A portion of a string extracted based on a specified range.
- Search: Finding the occurrence of a substring or character in a given string.
- Replace: Replacing occurrences of a substring or character with another in a string.
- Reverse: Reversing the order of characters in a string.
- Count occurrences: Determining the number of times a substring or character appears in a string.
- Palindrome: A string that reads the same backward as forward.
- Anagrams: Two strings that contain the same characters but may have a different order.
- Duplicate removal: Eliminating duplicate characters or substrings from a string.
- Tokenization: Breaking a string into smaller components called tokens.
- Vowels: The set of characters 'a', 'e', 'i', 'o', 'u' (and sometimes 'y') in a string.
- Consonants: Characters other than vowels in a string.
- Compression: Reducing the size of a string by eliminating redundancies.
- Case conversion: Changing the letter case (uppercase or lowercase) of characters in a string.
- Capitalization: Making the first character of each word uppercase in a string.
- Built-in functions: Functions provided by the programming language or standard libraries.
- <string.h>: A header file in C containing functions for string manipulation.
- strlen(): A function to determine the length of a string.

[illegible]

- strcpy(): A function to copy one string to another.
- strcat(): A function to concatenate two strings.
- strcmp(): A function to compare two strings.
- strstr(): This function is designed to locate the initial occurrence of a substring within a given string.
- strchr(): A function to find the first occurrence of a character in a string.
- strtok(): A function to tokenize a string into smaller parts based on a delimiter.

7.9 References

1. E.Balaguruswamy, "Computing Fundamentals & C Programming", TataMcGraw Hill, 2008.

2. B. A. Forouzan & R. F. Gilberg "Computer Science – A structured programming Approach Using C", 2011

3. E. Balaguruswamy, "Programming in ANSI" Tata McGraw Hill, 2011.

[illegible]

For Vivekananda Global University, Jaipur
Registrar

Table of Content

[illegible]

Learning Objectives

After studying this unit, the student will be able to:

- Explain the concept of modular programming and how it promotes code organization, reusability, and maintainability.
- Differentiate between function declaration and definition. Write function declarations with proper return types, function names, and parameter lists.
- Demonstrate how to call functions using function names and passing arguments. Understand the flow of control when a function is called and executed.
- Recognize and differentiate between standard functions provided by the programming language and user-defined functions created by the programmer.
- Use pointers or structures to return multiple values from a function when the language allows returning only a single value.
- Implement functions that use various operators (arithmetic, logical, relational) to perform calculations, comparisons, or other tasks.
- Develop a deeper understanding of functions' importance in writing organized efficient, and maintainable code.

[illegible]

Introduction

In programming, a function is a self-contained block of code that performs a specific task. Functions allow you to break down a program into smaller, modular pieces, making the code more organized, readable, and maintainable. They provide a way to reuse code and improve the program's overall efficiency.

- return type: The data type of the value that the function returns. It can be void if the function does not return any value.
- function name: The identifier for the function, which is used to call the function from other parts of the program.
- parameter list: The list of input parameters (if any) that the function accepts. These parameters are variables used to pass data into the function.
- value: The value returned by the function (if the return type is not void).

Advantages of Functions:

- Modularity: Functions promote code modularity by dividing a complex problem into smaller, manageable tasks. Each function can be developed and tested independently, making the overall code more organized.
- Reusability: Functions can be reused in multiple parts or even in different programs, reducing code duplication and enhancing code maintainability.
- Abstraction: Functions allow you to hide the implementation details of a task, providing a higher-level interface for other parts of the program to interact with.
- Code Readability: By breaking down a program into smaller functions, the code becomes more readable and easier to understand.

Applications of Functions:

- Code Organization: Functions are extensively used to organize large programs into smaller, more manageable parts. Each function can handle a specific functionality, making the code easier to develop and maintain.
- Modular Programming: Modular programming is an essential programming paradigm that encourages the use of functions. It involves dividing the program into functional modules, making it easier to understand, test, and modify.

[illegible]

- Reusability: Functions allow you to write code once and use it multiple times, reducing redundancy and promoting efficiency.
- Mathematical Computations: Functions are used to encapsulate mathematical operations or complex calculations, making the code more concise and easier to read.
- I/O Operations: Functions are often used to handle input/output operations, such as reading data from files or displaying output to the user.
- Data Manipulation: Functions can be employed for data manipulation tasks, like sorting arrays, searching for specific elements, or modifying data structures.

8.1 Function

Functions play a fundamental role in programming, offering a way to create modular and reusable code. This preface aims to provide a concise overview of key function-related concepts.

Functions provide reusability and modularity to a program by allowing developers to break down complex tasks into smaller, more manageable code units.

Key points about functions in C:

- **Encapsulation:** Functions encapsulate specific tasks or functionality, treating them as self-contained units. This encapsulation helps in organizing and structuring the code.
- **Code Reusability:** Once a function is defined, it can be called multiple times from different program parts, promoting code reuse and reducing code duplication.
- **Modularity:** Functions enable modular programming by dividing the program into smaller, independent parts. Each function handles a specific task, making understanding and maintaining the code easier.
- **Procedure or Subroutine:** In some other programming languages, functions are known as procedures or subroutines. The concept remains the same, regardless of the name used in different languages.

[illegible]

8.2 Modular programming

Modular programming, or modular design or modularization, is a software development approach that emphasizes dividing a large program into smaller, independent, and self-contained modules or units. Each module represents a specific functionality or task and can be developed, tested, and maintained separately. The modules interact with each other through well-defined interfaces, promoting code reusability, ease of maintenance, and better organization.

The key principles and benefits of modular programming include:

- **Code Reusability:** By breaking down a program into smaller modules, developers can reuse these modules in multiple application parts or even in different projects. This reduces redundant code and saves development time.
- **Encapsulation:** Each module is designed to hide its internal implementation details from other modules, exposing only necessary interfaces. This encapsulation protects the integrity of the module's functionality and prevents unintended dependencies.
- **Ease of Maintenance:** With modularization, modifying or debugging a specific part of the program becomes easier since changes are localized to individual modules. It also helps in isolating and fixing bugs more efficiently.
- **Readability and Understandability:** Smaller modules with well-defined responsibilities are easier to understand and maintain. Developers can focus on the specific functionality of each module without being overwhelmed by the entire program's complexity.
- **Parallel Development:** When a project is divided into modules, multiple developers can work simultaneously on different modules, speeding up the development process.
- **Testing:** Modules can be tested independently, simplifying the testing process and reducing the scope of testing for each module. This improves overall code quality and reliability.

- **Scalability:** Modular programs are more scalable as new features or functionality can be added by creating new modules or modifying existing ones without affecting the entire program.

To achieve modular programming, developers follow certain guidelines:

- **High Cohesion:** Modules should have high cohesion, meaning that the elements within a module are related and contribute to a single, well-defined purpose.
- **Low Coupling:** Modules should have low coupling, indicating that they are loosely connected to each other. This reduces interdependencies, making it easier to modify or replace one module without affecting others.
- **Clear Interfaces:** Each module should have clear and well-defined interfaces, specifying how other modules can interact with it. This ensures that changes to the module's implementation do not affect its external usage.
- **Abstraction:** Abstract data types and interfaces allow modules to hide their internal details, providing a higher-level view of the module's functionality.

Modular programming is a widely adopted practice in software development and is supported by various programming languages and development methodologies. It helps create more maintainable, flexible, and robust software systems, making it an essential approach for managing the complexity of modern software projects. In C programming, statements and blocks are essential components that determine the flow of execution and code organization within a program.

8.3 Function Declaration

A function declaration is the process of specifying the signature of a function without providing its actual implementation. It informs the compiler about the existence of a function, its return type, name, and the types of its parameters (if any). Function declarations are typically placed in header files (also known as

[illegible]

function prototypes) so that other program parts can access and use the function without knowing the implementation details.

The syntax for function declaration is as follows:

```
return type function_name(parameter1_type parameter1_name,
parameter2_type parameter2_name, ...);
```

- **return type:** The data type of the value that the function will return to the caller. It can be any valid C data type, or void if the function does not return any value.
- **function name:** The name of the function, which serves as an identifier to call the function from other parts of the program.
- **parameter1_type, parameter2_type, etc.:** The data types of the input parameters that the function accepts (if any).
- **parameter1_name, parameter2_name, etc.:** The names of the input parameters. These names are used within the function to refer to the passed arguments.

Here's an example of a function declaration for a simple addition function:

```
int add numbers(int a, int b);
```

In this example, add numbers is the function name, and it takes two int parameters a and b. The function returns an int value, representing the sum of a and b.

Function declarations are essential in C and other programming languages because they allow you to use functions before their actual implementation appears in the code. By declaring functions in header files and including those headers in different source files, you can achieve modularity and code reusability, making your code more organized and maintainable.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

8.4 Function Definition

A function definition is the process of providing the actual implementation of a function. It defines the set of programming statements that will be executed when the function is called. The function definition contains the code that performs the specific task or tasks intended for that function.

In C and many other programming languages, a function is defined by enclosing the set of programming statements within curly braces {}. The function definition follows the function declaration, which provides the function's signature (return type, name, and parameter list).

The syntax for function definition is as follows:

```
return type function_name(parameter1_type parameter1_name,
parameter2_type parameter2_name, ...) {

  // Function body (code to be executed)

  // ...

  // Optional return statement

  return value;

}
```

- **return type:** The data type of the value that the function will return to the caller. It should match the return type specified in the function declaration.
- **function name:** The function's name should be identical to the function name in the declaration.
- **parameter1_type, parameter2_type, etc.:** The data types of the input parameters that the function accepts should match the types specified in the function declaration.

- **parameter1_name, parameter2_name, etc.:** The names of the input parameters. These names are used within the function to refer to the passed arguments.
- **Function body:** The code enclosed within the curly braces {} represents the function's implementation. It contains the programming statements that perform the desired task when the function is called.
- **Optional return statement:** If the function has a return type other than void, the function should include a return statement that specifies the value to be returned as the function's output.

Here's an example of a function definition for a simple addition function: Example:

```
int add numbers(int a, int b)

 {

    return a + b;

}
```

In this example, the add numbers function takes two int parameters a and b and returns their sum.

Function definitions are essential because they provide the actual behavior of the function, allowing the program to execute the desired tasks when the function is called. Function definitions should match their corresponding declarations to ensure consistency in the program and avoid errors during compilation. Modular programming is facilitated by separating the declaration and definition of functions, enabling code reusability and maintainability.

8.5 Function Call

Function call, also known as function invocation, executes a specific function by using its name and providing the required arguments (if any). When a function is called, the program transfers control to the function's definition, and the code

[illegible]

inside the function is executed. After the function completes its task or reaches a return statement, the control returns to the program's point immediately after the function call.

The syntax for calling a function is as follows:

return type result = function_name(argument1, argument2, ...);

- **return type:** If the function has a return type other than void, a variable of the appropriate data type should be used to store the returned value. If the function returns nothing (void), the return type should be omitted.
- **function name:** The name of the function to be called should match the function's definition and declaration.
- **argument1, argument2, etc.:** The values or variables passed to the function as input parameters. The number and types of arguments should match the function's parameter list specified in the declaration and definition.

Here's an example of calling the previously defined add numbers function:

```
#include <stdio.h>

int add numbers(int a, int b) {

    return a + b;

}

int main() {

    int result = add numbers(5, 3); // Calling the
add_numbers function with arguments 5 and 3

    printf("Result: %d\n", result);

    return 0;

}
```

For Vivekananda Global University, Jaipur

[illegible]

Registrar

Output: Result: 8

In this example, the add_numbers function is called with arguments 5 and 3. The result of the function (which is 8, the sum of the two arguments) is stored in the result variable and then printed to the console.

Function calls are essential in programming because they allow code reusability. By calling the same function multiple times with different arguments, you can perform the same task on different sets of data or inputs, making your code more efficient and concise. Functions also enable modular programming by dividing the program into smaller, manageable units, making the code easier to read, understand, and maintain.

8.6 Types of Functions

There are two types of functions in C programming:

1. **Library Functions:**

Library functions, also known as standard library functions, are pre-defined functions provided by C libraries or header files. They are an integral part of the C programming language and offer a wide range of functionalities that can be used in various C programs. These functions are written in C and are made available for developers to use, which helps save time and effort by not having to reinvent commonly used functionalities.

C standard library functions are specified in the C Standard Library and grouped into several header files. You need to include the relevant header file in your C program to use these functions. Some of the commonly used C standard library functions are:

1. Input/output functions (<stdio.h>):
   - printf: Used to print formatted output to the screen.
   - scanf: Used to read formatted input from the user.
   - getchar: Used to read a single character from the user.

   - puts: Used to print a string to the screen followed by a newline.

2. Math functions (<math.h>):
   - sqrt: Calculates the square root of a number.
   - pow: Raises a number to a given power.
   - sin, cos, tan: Trigonometric functions.
   - ceil, floor, round: Functions for rounding numbers
3. String manipulation functions (<string.h>):
   - strcpy: Copies a string from source to destination.
   - strlen: Returns the length of a string.
   - strcat: Concatenates two strings.
   - strcmp: Compares two strings.
4. Memory management functions (<stdlib.h>):
   - malloc: Allocates memory dynamically.
   - calloc: Allocates and initializes memory for an array of elements.
   - free: Deallocates memory previously allocated with malloc or calloc.
5. Character manipulation functions (<ctype.h>):
   - isalpha, isdigit, isalnum: Check if a character is alphabetic, numeric, or alphanumeric.
   - toupper, tolower: Convert characters to uppercase or lowercase.

These are just a few examples of standard library functions in C. The C Standard Library provides many more functions that cover various aspects of programming, making it easier for developers to perform common tasks without having to implement them from scratch.

2. **User-Defined Functions:**

The programmer creates user-defined functions to perform specific tasks per a particular program's requirements. They are essential building blocks that allow you to break down complex tasks into smaller, more manageable parts, promoting code organization, reusability, and maintainability.

To create a user-defined function in C, you need to follow these steps:

[illegible]

For Vivekananda Global Uni aipur

Registrar

1. **Function Declaration (Prototype):**

Declare the function's prototype, including the function's name, return type, and the parameters (if any). The prototype serves as a blueprint for the function, informing the compiler about its existence and how it should be used.

Example:

// Function declaration (prototype)

int add(int a, int b);

2. **Function Definition:**

Define the body of the function, which contains the actual code to perform the desired task. The function body is enclosed within curly braces {}.

Example:

```
// Function definition

int add(int a, int b) {

      return a + b;

}
```

3. **Function Call:**

Call the user-defined function from the main program or other functions when you need to execute the code inside it. To call the function, use its name followed by parentheses containing the arguments (if any).

Example:

```
#include <stdio.h>

// Function declaration
```

[illegible]

For Vivekananda Global University, Jaipur

```
int add(int a, int b);

int main() {

    // Function call

    int result = add(5, 3);

    printf("The result is: %d\n", result);

    return 0;

}
```

In this example, add is a user-defined function that takes two integer arguments a and b and returns their sum. When add(5, 3) is called in the main function, it computes the sum of 5 and 3 and returns the result, which is then printed as output.

User-defined functions allow programmers to modularize their code, making it easier to understand and maintain. They are a fundamental concept in structured programming and are crucial in writing efficient and organized C programs.

8.7 Function Returning more Values

A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

Let's see a simple example of C function that doesn't return any value from the function.

```
Example without return value:

void hello(){

printf("hello c");

}
```

[illegible]

If you want to return any value from the function, you need to use any data type such as int, long, char, etc. The return type depends on the value to be returned from the function.

Let's see a simple example of C function that returns int value from the function.

```
Example with return value:

int get(){

return 10;

}
```

In the above example, we have to return 10 as a value, so the return type is int. If you want to return floating-point value (e.g., 10.2, 3.1, 54.5, etc), you need to use float as the return type of the method.

Different aspects of function calling

A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different aspects of function calls.

- function without arguments and without return value
- function without arguments and with return value
- function with arguments and without return value
- function with arguments and with return value

1. **function without arguments and without return value:**

Function that takes no arguments and does not return any value. Such functions are commonly used when you need to perform specific actions or tasks without requiring any input or generating an output result. The return type for such functions is void.

[illegible]

Here's an example of a function without arguments and return value:

```
#include <stdio.h>

//function without arguments and return value (void return
type)

void printHello() {

    printf("Hello, World!\n");

}

int main() {

    // Function call, no need to pass any arguments or
capture the return value

    printHello();

    return 0;

}

Output: Hello, World!
```

In this example, we have a function named printHello(), which does not take any arguments (void argument list) and does not return any value (void return type). Inside the function, it simply prints "Hello, World!" to the console using printf.

In the main() function, we call printHello() with printHello();. Since the function does not require any arguments and does not return any value, we can call it directly without providing any arguments or assigning the result to any variable.

Functions without arguments and return value are useful when you want to perform actions that don't depend on any input and don't need to provide any

[illegible]
For Vivekananda Global University, Jaipur
Registrar

output to the calling code. They are often used for tasks like displaying messages, updating global variables, or executing certain operations with fixed behavior.

2. **function without arguments and with return value**

Function that takes no arguments and returns a value. Such functions are useful when calculating or generating a result based on some internal logic without relying on any external input.

Here's an example of a function without arguments and with a return value:

#include <stdio.h>

```
//function without arguments and with return value
(int return type)

int getRandomNumber() {

    // Logic to generate a random number (example)

    return rand() % 100 + 1; // Returns a random number
between 1 and 100

}

int main() {

    int randomNumber;

    // Function call, no need to pass any arguments

    randomNumber = getRandomNumber();

    printf("Random Number: %d\n", randomNumber);

    return 0;

}
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

In this example, we have a function named getRandomNumber(), which does not take any arguments (void argument list) but returns an integer value. Inside the function, we use some logic to generate a random number between 1 and 100 using the rand() function and modulo operation.

In the main() function, we declare an integer variable randomNumber, and then we call getRandomNumber () to retrieve a random number. The value returned by getRandomNumber () is assigned to randomNumber, and we print it using printf.

Functions without arguments and with return value are useful when you want to encapsulate complex calculations or logic that doesn't require any external input. They help in modularizing your code and improving its readability and maintainability.

3. **function with arguments and without return value.**

Function that takes arguments (input parameters) but does not return any value. Such functions are commonly used when you need to perform specific actions or modify values based on the provided input, without needing to generate an output result.

Here's an example of a function with arguments and without a return value:

#include <stdio.h>

```
//function with arguments and without return value (void
return type)

void printSum (int a, int b) {

    int sum = a + b.

    printf("Sum of %d and %d is: %d\n", a, b, sum);

}
```

[illegible]
For Vivekananda Global University, Jaipur
Registrar

```
int main() {

    int num1 = 5, num2 = 7;

    //function call with arguments, no need to capture the
return value

    printSum(num1, num2);

    return 0;

}

Output: Sum of 5 and 7 is: 12
```

In this example, we have a function named printSum () that takes two integer arguments a and b and does not return any value (void return type). Inside the function, it calculates the sum of the two arguments and prints the result using printf.

In the main () function, we declare two integer variables num1 and num2, and then we call the printSum() function with these variables as arguments. The function calculates and prints the sum of num1 and num2, but it does not return any value.

Functions with arguments and without a return value are commonly used for tasks that require specific inputs to perform an action or update global variables but don't need to produce a result that needs to be captured by the calling code. They help in organizing code and making it more readable and modular.

4. **function with arguments and with return value**

Function that takes arguments (input parameters) and returns a value. Functions with arguments and return value are widely used to perform calculations or generate results based on the provided input.

```
Here's an example of a function with arguments and return
value:
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
#include <stdio.h>

//function with arguments and return value (int return
type)

int add (int a, int b) {

    return a + b.

}

int main () {

    int num1 = 5, num2 = 7;

    int sum;

    //function call with arguments, capturing the return
value

    sum = add(num1, num2);

    printf("Sum of %d and %d is: %d\n", num1, num2, sum);

    return 0;

}

Output: Sum of 5 and 7 is: 12
```

In this example, we have a function named add() that takes two integer arguments a and b and returns their sum (int return type). Inside the function, it calculates the sum of the two arguments and uses the return statement to provide the result.

In the main () function, we declare two integer variables num1 and num2, and then we call the add() function with these variables as arguments. The function calculates the sum of num1 and num2 and returns the result, which is then captured and stored in the variable sum. Finally, we print the sum using printf.

[illegible]

Functions with arguments and return value are commonly used for calculations, data processing, and other tasks that require input and produce a result. They are essential to C programming as they enable code reusability and allow you to create modular and maintainable programs.

8.8 Function with Operators

Functions that use operators to perform various operations. Functions with operators can be used to carry out calculations, comparisons, and other operations, making your code more modular and organized. Here are some examples of functions using different operators:

1. **Function with arithmetic operators:**

This function takes two integers as arguments and returns their sum.

```
int add(int a, int b) {

    return a + b;

}
```

2. **Function with relational operators:**

This function takes two integers as arguments and returns 1 if the first argument is greater than the second, otherwise returns 0.

```
int isGreater(int a, int b) {

    return a > b ? 1 : 0;

}
```

3. **Function with logical operators:**

[illegible]

This function takes a boolean value as an argument and returns the negation of that value.

```
int negate(int value) {
    return !value;
}
```

4. **Function with bitwise operators:**

This function takes two integers as arguments and performs bitwise AND operation on them.

```
int bitwiseAND(int a, int b) {
    return a & b;
}
```

5. **Function with assignment operators:**

This function takes two pointers to integers and adds the second value to the first value using the assignment operator '+='.

```
void addToValue(int* num1, int num2) {
    *num1 += num2;
}
```

6. **Function with ternary operator:**

This function takes an integer as an argument and returns "Even" if the number is even, otherwise returns "Odd".

```
const char* evenOrOdd(int num) {
    return (num % 2 == 0) ? "Even" : "Odd";
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
}
```

These are just a few examples of functions using different operators. You can create functions to perform any kind of operation using C's wide range of operators. Functions with operators help encapsulate specific functionalities, making the code more readable and maintainable.

8.9 Summary

Modular Programming:

- Functions in C enable modular programming by breaking down complex tasks into smaller, manageable parts. Each function performs a specific task, making the code more organized and easier to understand, maintain, and reuse.
- Function Declaration, Definition, and Function Call:
- Function Declaration: Functions must be declared before they are used. The declaration includes the function's name, return type, and the types of its parameters (if any).
- Function Definition: The function's definition contains the actual code that executes the task defined in the function. It is enclosed within curly braces {}.
- Function Call: Functions are called to execute their code. The calling code can pass arguments to the function (if required) and may capture the return value (if any)

Types of Functions:

- Value-Returning Functions: These functions calculate a result and return a value of a specified data type to the calling code.
- Void Functions: Void functions do not return any value. They are used to perform specific actions without producing a result.
- Recursive Functions: Recursive functions call themselves to solve problems that can be broken down into smaller subproblems.

[illegible]

- Inline Functions: Inline functions are optimized for performance by directly replacing the function call with the actual function code.

Function Returning More Values:

- Functions in C can return only one value directly. However, multiple values can be returned using techniques such as pointers, structures, or global variables.

Function with Operators:

- Functions in C can use various operators to perform calculations, comparisons, bitwise operations, etc. Functions with operators help encapsulate specific functionalities and improve code readability.

8.10 Review Questions

1. What is the main purpose of using functions in C programming?
2. Explain the process of creating a function, including function declaration and definition.
3. What are the two main types of functions based on their return behavior? Provide examples of each.
4. How can you return multiple values from a function in C? Provide at least two different methods.
5. Describe the use of "void" as the return type in a function. When is it commonly used?
6. What is a recursive function, and how does it work? Provide an example of a recursive function.
7. How can you pass arguments to a function in C? Give an example of a function that takes two integers as arguments and returns their sum.
8. Explain the concept of "modular programming" and how functions facilitate it.
9. Give an example of a function that uses arithmetic operators to perform a calculation.

[illegible]

10. What are inline functions in C programming, and what is their purpose?

8.11 Keywords

- Function: A modular programming unit that performs a specific task.
- Return Type: The data type of the value returned by a function.
- Arguments/Parameters: Inputs passed to a function when it is called.
- Void Function: A function that does not return any value.
- Value-Returning Function: A function that calculates and returns a result.
- Function Declaration: Providing the function's name, return type, and parameters to inform the compiler.
- Function Definition: The actual implementation of the function's code.
- Recursive Function: A function that calls itself execution.
- Inline Function: A small function optimized for performance by replacing its call with the actual code.
- Function Pointer: A pointer that holds the address of a function.

8.12 References

1. E. Balaguruswamy, "Computing Fundamentals & C Programming", TataMcGraw Hill, 2008.

2. B. A. Forouzan & R. F. Gilberg "Computer Science – A structured programming Approach Using C", 2011

3. E. Balaguruswamy, "Programming in ANSI" Tata McGraw Hill, 2011.

[illegible]

Table of Content

[illegible]
For Vivekananda Global University, Jaipur
Registrar

Learning Objectives

After studying this unit, the student will be able to:

- Understand the concept of functions and their role in code organization and modularity.
- Learn how to define and call functions in C programming.
- Explore the use of decision statements (if, else, else-if) to make choices based on conditions.
- Comprehend the integration of functions within decision statements for code reuse and efficiency.
- Grasp the concept of loop operators (for, while, do-while) and their role in controlling loop execution.
- Learn how to use functions with loop operators to handle repetitive tasks.
- Explore the benefits of encapsulating loop logic within functions for improved code organization and readability.
- Understand the concept of arrays and their representation of collections of elements in contiguous memory.
- Learn how to pass arrays to functions as arguments for efficient element manipulation.
- Explore the use of functions to return arrays or modify array elements, promoting code reusability and maintainability.
- Recognize the importance of code modularity in breaking down complex tasks into manageable functions.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Introduction Function

Functions with decision statements, loops, and arrays are essential building blocks in programming. They enable you to implement complex logic and handle data efficiently. Let's explore each component and how they can be combined in functions:

1. Functions:

   Functions are self-contained blocks of code that perform specific tasks. They are defined with a name, return type (if any), and optional parameters. Functions help in modularizing the code and promoting code reusability.

1. Decision Statements:

   Decision statements, like if and else, allow you to make choices based on certain conditions. They help in controlling the flow of execution based on whether a condition is true or false.

2. Loops:
3. Loops, like for, while, and do-while, enable you to execute a block of code repeatedly based on a particular condition. They are useful when working with arrays and performing repetitive tasks.

Functions with arrays:

- Modularization: Functions with arrays promote modularization, allowing you to break down complex tasks into smaller, manageable functions. This enhances code readability, maintainability, and reusability.
- Function Definition: To create a function that works with arrays, define the function with appropriate parameters to accept the array and its size. The function can then perform specific operations on the array's elements.
- Array Access: Inside the function, you can use loops (like for, while, or do-while) to access and process elements of the array. Loops enable you to perform operations on each element efficiently.
- Function Return Type: Functions can have a return type that specifies the data type of the value the function returns after processing the array. This allows you to obtain results from the function.

- Function Call: To use the function, call it from the main function or any other part of the program, passing the array and its size as arguments. The function can then perform the desired operation and optionally return a result.
- Data Manipulation: Functions with arrays are commonly used for data manipulation tasks, such as finding the average, sum, maximum, or minimum of array elements, sorting arrays, or performing statistical calculations.

9.1 Function

A function is a fundamental programming building block representing a self-contained block of code designed to perform a specific task. In most programming languages, including C, functions serve as reusable and modular units of code that help organize and structure programs. They allow you to break down complex tasks into smaller, manageable pieces, making code more readable, maintainable, and efficient.

Key Characteristics of Functions:

1. **Modularity:** Functions provide modularity by encapsulating specific actions or operations within a single block of code. This lets you focus on individual tasks independently, making it easier to understand and debug your program.
2. **Reusability:** Once you define a function, you can call it multiple times from different parts of your program. This reusability saves time and effort and promotes efficient code development.
3. **Abstraction:** Functions abstract away the internal details of their implementation, providing only the necessary interface to interact with them. This abstraction makes it easier to use complex operations without worrying about their implementation details.
4. **Parameter Passing:** Functions can accept input data through parameters, allowing you to provide different data each time the function is called.

Parameters enable dynamic behavior and customization within the same function.

5. **Return Value:** Functions can return a value to the caller, which allows them to pass back results or computed values. This feature is crucial for passing information from the function's execution back to the calling code.
6. **Function Signature:** The function signature comprises the function's name, return type, and parameter list (if any). It defines the function's interface and how it can be used.

Function Syntax in C:

In C, the syntax to declare and define a function is as follows:

```
return      Type      functionName(parameterType      parameter1,
parameterType parameter2, ...) {

    // Function body

    // Statements to perform the desired task

    // Optionally, a return statement to return a value

}
```

- **return Type:** The data type of the value the function returns to the caller. The return type is void if the function does not return a value.
- **function Name**: The unique identifier used to call the function.
- **parameter Type:** The data type of each parameter (if any) that the function accepts. If the function takes no parameters, the parameter list is left empty.

Example of a Simple Function in C:

```
#include <stdio.h>

//function to calculate the square of a number

int square(int Num) {
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
          return Num * num;

      }

      int main() {

          int number = 5;

          int result = square(number);

          printf("The square of %d is %d.\n", number, result);

          return 0;

      }
```

In this example, we define a function named square that takes an integer Num as a parameter and returns its square. The function is called from the main function, demonstrating the essential features of functions in C.

9.2 Decision Statements

Decision statements, also known as conditional statements, are programming constructs that allow a program to make decisions based on certain conditions. They enable the program to take different paths or execute different blocks of code depending on whether a specified condition evaluates to true or false. The primary decision statements in most programming languages, including C, are:

1. **if statement:**

The if statement executes a block of code if a given condition is true.

```
Syntax:

if (condition) {

    // Code to execute if the condition is true

}
```

2. **if-else Statement:**

[illegible]

For Vivekananda Global University, Jaipur

Registrar

The if-else statement allows the program to execute one block of code if the condition is true and another block of code if the condition is false.

```
Syntax:

     if (condition) {

         // Code to execute if the condition is true

     } else {

         // Code to execute if the condition is false

     }
```

3. **else-if Ladder:**

The else-if ladder is an extension of the if-else statement, allowing the program to test multiple conditions one by one and execute the corresponding block of code for the first true condition.

```
Syntax:

     if (condition1) {

         // Code to execute if condition1 is true

     } else if (condition2) {

         // Code to execute if condition2 is true

     } else if (condition3) {

         // Code to execute if condition3 is true

     } else {

         // Code to execute if none of the conditions are true

     }
```

4. **switch Statement:**

[illegible]

The switch statement is used to select one of many code blocks to execute based on the value of an expression.

Syntax:

```
    switch (expression) {
        case value1:
            // Code to execute if expression equals value1
            break;
        case value2:
            // Code to execute if expression equals value2
            break;
        // more cases...
        default:
            // Code to execute if none of the cases match the
    expression
    }
```

Decision statements are essential for controlling the flow of a program based on different scenarios and making programs more flexible and dynamic. They help to implement logic and conditions, enabling the program to respond appropriately to various inputs and situations.

9.3 Function Uses in Decision Statements

Functions play a significant role in decision statements by allowing you to encapsulate complex logic, computations, or operations into reusable blocks of code. Functions can be used in decision statements to perform specific tasks, compute values, or validate conditions. Here are some common uses of functions in decision statements:

1. **Function Calls in Condition Expressions:**

[illegible]

```
int is Even(int num) {

    return num % 2 == 0;

}


int main() {

    int number = 5;

    if (is Even(number)) {

        printf("%d is even.\n", number);

    } else {

        printf("%d is odd.\n", number);

    }

    return 0;

}
```

2. **Complex Decision Logic:**

Functions can be used to encapsulate complex decision-making logic, making the main code more readable and maintainable.

```
int isValidInput(int num) {

    return (num >= 0 && num <= 100);

}

int main() {

    int score;

    printf("Enter your score: ");

    scanf("%d", &score);

    if (isValidInput(score)) {
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
            printf("Your score is valid.\n");
        } else {
            printf("Invalid score. Please enter a number between 0 and 100.\n");
        }
        return 0;
    }
```

3. **Switch Case Functionality:**

Functions can be used within case statements in a switch statement, providing more complex actions based on the value of the expression.

```
    int add(int a, int b) {
        return a + b;
    }
    int subtract(int a, int b) {
        return a - b;
    }
    int main() {
        int choice, num1, num2, result;

        printf("Enter 1 for addition or 2 for subtraction: ");
        scanf("%d", &choice);

        printf("Enter two numbers: ");
        scanf("%d %d", &num1, &num2);
```

[illegible]

For Vivekananda Global University, Jaipur

```
    switch (choice) {
        case 1:
            result = add(num1, num2);
            break;
        case 2:
            result = subtract(num1, num2);
            break;
        default:
            printf("Invalid choice.\n");
            return 1;
    }
    printf("Result: %d\n", result);
    return 0;
}
```

4. **User-Defined Decision Functions:**

You can create custom functions to handle specific decision-making scenarios and call them within decision statements.

```
int isPrime(int num) {
    if (num <= 1) return 0;
    for (int i = 2; i <= num / 2; i++) {
        if (num % i == 0) return 0;
    }
    return 1;
```

[illegible]

For Vivekananda Global University, Jaipur

```
}

int main() {

     int number;

     printf("Enter a number: ");

     scanf("%d", &number);

     if (isPrime(number)) {

         printf("%d is a prime number.\n", number);

     } else {

         printf("%d is not a prime number.\n", number);

     }

     return 0;

}
```

Functions in decision statements make your code more organized, easier to understand, and enable code reuse. Breaking down complex logic into smaller functions improves code maintainability and readability, leading to better software development practices.

9.4 Loop Operators

It seems there might be a misunderstanding in the question. "Loop operators" is not a standard term in programming. However, I believe you might be referring to loop control statements or operators, which are essential in controlling the flow of loops in programming.

In programming, loop control statements are used to control the execution of loops based on certain conditions. They allow you to determine when to start, continue, or terminate a loop. The most common loop control statements in C are:

1. **for loop:**

[illegible]
For Vivekananda Global University, Jaipur
Registrar

The for loop is used to repeatedly execute a block of code as long as a specified condition is true.

```
for (initialization; condition; increment/decrement) {
    // Code to be executed in each iteration
}
```

2. **while loop:**

The while loop is used to execute a block of code repeatedly as long as a specified condition is true.

```
while (condition) {
    // Code to be executed in each iteration
}
```

3. **do-while loop:**

The do-while loop is similar to the while loop, but it executes the block of code at least once before checking the condition.

```
do {
    // Code to be executed in each iteration
} while (condition);
```

4. **break statement:**

The break statement is used to exit a loop prematurely, breaking out of the loop.

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break; // Exit the loop when i equals 5
    }
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
        printf("%d ", i);
    }
```

5. **continue statement:**

The continue statement is used to skip the remaining code in a loop iteration and jump to the next iteration.

```
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            continue; // Skip the iteration when i equals 5
        }
        printf("%d ", i);
    }
```

These loop control statements allow you to create flexible loops and control the flow of execution, enabling you to solve different types of problems and handle various scenarios efficiently.

9.5 Functions with Loop

an example of a function that uses a loop to calculate the factorial of a given positive integer:

```
    #include <stdio.h>
    //function to calculate the factorial of a positive integer
    int factorial(int num) {
        int result = 1;
        for (int i = 1; i <= num; i++) {
            result *= i;
        }
```

```
     return result;

}

int main() {

     int number;

     printf("Enter a positive integer: ");

     scanf("%d", &number);

     if (number < 0) {

          printf("Factorial  is  not  defined  for  negative
numbers.\n");

     } else {

          int fact = factorial(number);

          printf("Factorial of %d is %d.\n", number, fact);

     }

     return 0;

}
```

In this example, we define a function factorial that takes a positive integer num as an argument and returns its factorial. The function uses a for loop to calculate the factorial by multiplying all the integers from 1 to num.

In the main function, we take user input for a positive integer and call the factorial function to calculate its factorial. If the user enters a negative number, we inform them that the factorial is not defined for negative numbers.

The use of a loop in the factorial function allows us to iteratively calculate the factorial of the given integer, making the code more efficient and concise.

9.6 Functions with Loop

Functions and loop operators are fundamental components of programming that allow for code organization, reusability, and control flow. They work together seamlessly to handle repetitive tasks and implement complex logic. Here's an overview of how functions and loop operators can be used together:

Functions:

Functions are self-contained blocks of code that perform specific tasks or computations.

They provide modularity and enable code reuse, making programs more organized and maintainable.

Functions are called with arguments (input values) and can return a result (output value) after executing their code.

Loop Operators:

Loop operators control the flow of loops, allowing code to execute repeatedly based on certain conditions.

Common loop operators include for, while, and do-while loops.

Using Functions with Loop Operators:

1. **Function Calls Inside Loops:**

Functions can be called inside loop bodies to perform specific operations repeatedly.

```
//function to calculate the square of a number

int square(int num) {

    return num * num;

}
```

[illegible]

```
int main() {
     for (int i = 1; i <= 5; i++) {
          int result = square(i);
          printf("Square of %d is %d.\n", i, result);
     }
     return 0;
}
```

2. **Looping Within Functions:**

Functions can contain loop operators to handle repetitive tasks as part of their functionality.

```
//function to calculate the sum of numbers from 1 to
n
int sumToN(int n) {
     int sum = 0;
     for (int i = 1; i <= n; i++) {
          sum += i;
     }
     return sum;
}
int main() {
     int number;
     printf("Enter a positive integer: ");
     scanf("%d", &number);

     if (number <= 0) {
          printf("Invalid input.\n");
     } else {
```

For Vivekananda Global University, Jaipur
Registrar

```
            int result = sumToN(number);
            printf("Sum of numbers from 1 to %d is %d.\n",
        number, result);
        }
        return 0;
    }
```

3. **Looping and Function Composition:**

Functions can be composed to handle more complex tasks, combining loop operators as needed.

```
//function to calculate the factorial of a positive
integer
int factorial(int num) {
    int result = 1;
    for (int i = 1; i <= num; i++) {
        result *= i;
    }
    return result;
}
//function to calculate the sum of factorials from 1
to n
int sumOfFactorials(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += factorial(i);
    }
    return sum;
}
int main() {
    int number;
    printf("Enter a positive integer: ");
    scanf("%d", &number);
```

[illegible]

```
    if (number <= 0) {
        printf("Invalid input.\n");
    } else {
        int result = sumOfFactorials(number);
        printf("Sum of factorials from 1 to %d is
%d.\n", number, result);
    }
    return 0;
}
```

By using functions with loop operators, you can create more modular, efficient, and organized code to handle repetitive tasks and implement complex algorithms. Functions allow you to encapsulate logic, and loop operators facilitate repetitive execution, making them powerful tools for writing effective and maintainable programs.

9.7 Functions with Array

Functions can work seamlessly with arrays, allowing you to pass arrays as arguments to functions and perform various operations on the elements. This enables you to encapsulate array-related tasks and promote code reusability. Here re some common ways to use functions with arrays in C:

Passing Arrays to Functions:

You can pass arrays to functions by specifying the array name as a parameter in the function declaration. The size of the array is usually passed as a separate parameter, or you can use a fixed-size array in the function signature.

```
#include <stdio.h>
//function to find the sum of all elements in an array
int arraySum(int arr[], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
```

For Vivekananda Glo [illegible] Registrar

```
        sum += arr[i];
    }
    return sum;
}
int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int size = sizeof(numbers) / sizeof(numbers[0]);
    int result = arraySum(numbers, size);
    printf("Sum of elements in the array: %d\n", result);
    return 0;
}
```

Returning Arrays from Functions:

C does not allow functions to return entire arrays, but you can return a pointer to the first element of the array. This allows you to modify the original array or use the returned array in other parts of the program.

```
#include <stdio.h>

//function to create and return an array containing the first n
Fibonacci numbers

int* fibonacci(int n) {
    static int fibArray[100]; // Static to avoid memory issues
    fibArray[0] = 0;
    fibArray[1] = 1;
    for (int i = 2; i < n; i++) {
        fibArray[i] = fibArray[i - 1] + fibArray[i - 2];
    }
    return fibArray;
}
int main() {
    int n = 10;
    int* fibNumbers = fibonacci(n);
    printf("First %d Fibonacci numbers: ", n);
    for (int i = 0; i < n; i++) {
        printf("%d ", fibNumbers[i]);
    }
    printf("\n");
    return 0;
}
```

Modifying Array Elements in Functions:

Arrays can be passed to functions as pointers, allowing you to modify the array elements within the function.

```
#include <stdio.h>
//function to double all elements in an array
void doubleArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] *= 2;
    }
}
int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int size = sizeof(numbers) / sizeof(numbers[0]);
    printf("Original array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", numbers[i]);
    }
    printf("\n");
    doubleArray(numbers, size);
    printf("Array after doubling: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", numbers[i]);
    }
```

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

You can handle array-related tasks more efficiently and promote code modularity by using functions with arrays. Functions allow you to encapsulate array operations, making your code more organized and easier to maintain.

Some programming examples demonstrate using functions with decision statements, loop operators, and arrays in C.

1. Function and Decision Statements:

```
#include <stdio.h>

//function to check if a number is positive, negative, or
zero

void checkNumber(int num) {

    if (num > 0) {

        printf("%d is positive.\n", num);

    } else if (num < 0) {

        printf("%d is negative.\n", num);

    } else {

        printf("The number is zero.\n");

    }

}

nt main() {

    int number;

    printf("Enter a number: ");

    scanf("%d", &number);
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
    checkNumber(number);

    return 0;

}
```

2. Function and Loop Operators:

```
#include <stdio.h>

//function to calculate the sum of numbers from 1 to n
using a loop

int sumToN(int n) {

    int sum = 0;

    for (int i = 1; i <= n; i++) {

        sum += i;

    }

    return sum;

}

int main() {

    int number;

    printf("Enter a positive integer: ");

    scanf("%d", &number);

    if (number <= 0) {

        printf("Invalid input.\n");

    } else {

        int result = sumToN(number);

        printf("Sum of numbers from 1 to %d is %d.\n",
number, result);
```

[illegible]

For Vivekananda [illegible] Registrar

```
    }
    return 0;
}
```

3. Function with Arrays:

```
#include <stdio.h>
//function to find the maximum element in an array
int findMax(int arr[], int size) {
    int max = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
int main() {
    int numbers[] = {10, 25, 8, 12, 45};
    int size = sizeof(numbers) / sizeof(numbers[0]);
    int maxNum = findMax(numbers, size);
    printf("Maximum element in the array is: %d\n",
maxNum);
    return 0;
}
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

In these examples, we have used functions to encapsulate specific logic and computations. The first example uses a function with decision statements to check if a number is positive, negative, or zero. The second example utilizes a function with a loop operator to calculate the sum of numbers from 1 to n. The third example employs a function with an array to find the maximum element in an array. These examples showcase the versatility and usefulness of functions when combined with decision statements, loop operators, and arrays in C programming.

few more programming examples that demonstrate the use of functions with arrays in C:

1. Function to Calculate Average of Elements in an Array:

```
#include <stdio.h>

//function to calculate the average of elements in an array

double calculateAverage(int arr[], int size) {

    int sum = 0;

    for (int i = 0; i < size; i++) {

        sum += arr[i];

    }

    return (double)sum / size;

}

int main() {

    int numbers[] = {10, 20, 30, 40, 50};

    int size = sizeof(numbers) / sizeof(numbers[0]);

    double average = calculateAverage(numbers, size);

    printf("Average of elements in the array: %.2lf\n",
average);
```

For Vivekananda Global University, Jaipur

[illegible]

Registrar

```
    return 0;
}
```

2. Function to Search for an Element in an Array:

```
#include <stdio.h>

//function to search for an element in an array

int searchElement(int arr[], int size, int key) {

    for (int i = 0; i < size; i++) {

        if (arr[i] == key) {

            return i; // Return the index if element is
found

        }

    }

    return -1; // Return -1 if element is not found

}

int main() {

    int numbers[] = {10, 20, 30, 40, 50};

    int size = sizeof(numbers) / sizeof(numbers[0]);

    int searchKey = 30;


    int index = searchElement(numbers, size, searchKey);

    if (index != -1) {

        printf("Element   %d   found   at   index   %d.\n",
searchKey, index);

    } else {
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
            printf("Element %d not found in the array.\n",
searchKey);
        }
        return 0;
    }
```

3. Function to Reverse the Elements in an Array:

```
#include <stdio.h>
//function to reverse the elements in an array
void reverseArray(int arr[], int size) {
    int start = 0;
    int end = size - 1;
    while (start < end) {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}
int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int size = sizeof(numbers) / sizeof(numbers[0]);
    printf("Original array: ");
    for (int i = 0; i < size; i++) {
```

[illegible]
For Vivekananda Global University, Jaipur
Registrar

```
        printf("%d ", numbers[i]);
    }
    printf("\n");
    reverseArray(numbers, size);
    printf("Reversed array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", numbers[i]);
    }
    printf("\n");
    return 0;
}
```

9.8 Summary

Explored the integration of functions with decision statements, loop operators, and arrays in C programming. Functions are modular blocks of code that perform specific tasks and promote code organization and reusability. They can be effectively used with decision statements, loop operators, and arrays to handle various programming scenarios.

Functions and Decision Statements:

- Functions provide modularity and code reuse by encapsulating specific logic.
- Functions can be used within decision statements (if, else, else-if) to execute different code blocks based on conditions.
- The combination of functions and decision statements improves code organization and comprehensibility.

Functions and Loop Operators:

- Functions can contain loop operators (for, while, do-while) to handle repetitive tasks and computations.
- Loop operators control the flow of loops, allowing code to execute repeatedly based on specified conditions.
- Functions with loop operators promote code modularity and enhance code readability.

Functions with Arrays:

- Arrays are collections of elements stored in contiguous memory locations.
- Functions can accept arrays as parameters, allowing efficient element manipulation.
- Functions can return pointers to arrays or use static arrays to return modified arrays from functions.
- Functions with arrays enhance code reusability and improve the organization of array-related operations.

9.9 Review Questions

1. What is the purpose of the if Statement in C, and how does it work?
2. What is a function in C programming, and how does it promote code organization and reusability?
3. How can functions be used in decision statements (if, else, else-if) to execute different code blocks based on conditions? Provide an example.
4. Explain the role of loop operators (for, while, do-while) and how functions can work with loop operators to handle repetitive tasks. Provide an example.
5. How can functions be utilized to perform operations on arrays efficiently? Give an example of a function that calculates the average of elements in an array.
6. Can functions return entire arrays in C? If not, how can functions return arrays or modify array elements? Give an example of a function that modifies an array by reversing its elements.
7. How does the use of functions with arrays enhance code reusability and improve the organization of array-related operations?

8. In what situations would you use functions to encapsulate decision-making logic, loop operations, or array manipulations? Explain with specific examples.
9. Discuss the benefits of using functions with decision statements, loop operators, and arrays in C programming in terms of code modularity, readability, and maintainability.
10. Write a C function that takes an array of integers as input and returns the sum of its positive elements.

9.10 Keywords

Functions and Decision Statements:

- Function: A self-contained block of code that performs a specific task or computation.
- Decision Statements: Statements like if, else, and else if used to make choices in the code based on certain conditions.
- if: A decision statement that executes a block of code if a given condition is true.
- else: A decision statement that executes a block of code if the preceding if condition is false.
- else if: A decision statement that provides an additional condition to check if the previous if condition is false.

Functions and Loop Operators:

- Loop Operators: Statements like for, while, and do-while used to control the flow of loops in code.
- for: A loop operator that repeatedly executes a block of code for a specific number of times.
- while: A loop operator that repeatedly executes a block of code as long as a given condition is true.
- do-while: A loop operator that executes a block of code at least once and then repeatedly as long as a given condition is true.

Functions with Arrays:

- Arrays: Collections of elements of the same data type stored in contiguous memory locations.
- Array Parameter: Passing an array as an argument to a function to perform operations on its elements.
- Array Index: The position of an element in an array, starting from 0 for the first element.
- Array Size: The number of elements present in an array.
- Array Modification: Changing the values of elements in an array within a function.
- Array Return: Using a pointer to return an array or its elements from a function.

Understanding these keywords and their relationships will enable you to effectively use functions, decision statements, loop operators, and arrays in C programming to create efficient and organized code.

References

1. E.Balaguruswamy, "Computing Fundamentals & C Programming", Tata McGraw Hill, 2008.
2. B. A. Forouzan & R. F. Gilberg "Computer Science – A structured programming Approach Using C", 2011
3. E. Bala Guruswamy, "Programming in ANSI" Tata McGraw Hill, 2011.

[illegible]

Table of Content

[illegible]

Learning Objectives

After studying this unit, the student will be able to:

- Describe what structures are and how they allow you to group multiple variables of different data types under a single name.
- Write the syntax to declare and define a structure in C, including naming the structure and defining its member variables.
- Create variables of a specific structure type and understand how each variable holds the defined member variables.
- Use the dot (.) operator to access and modify individual members of a structure variable.
- Initialize structure variables at the time of declaration or later in the program using appropriate values.
- Understand how to pass structures to functions by value or reference (using pointers).
- Identify the various use cases and scenarios where structures are helpful, such as representing real-world entities, organizing complex data, and managing data within arrays or linked data structures.
- Compare structures with other data types in C, like arrays and unions, and understand the unique characteristics and advantages of using structures.
- Realize how structures are used as building blocks for implementing more complex data structures, like linked lists, stacks, and queues.
- Recognize the importance of using structures to create modular and readable code by grouping related data together and making code maintenance easier.
- By mastering the concept of structures and their applications, you will gain the ability to manage and manipulate complex data more effectively, resulting in more efficient and organized programs.

[illegible]

Introduction

Structure:

In C, a structure is a user-defined data type that allows you to group different variables of various data types under a single name. It enables you to create a composite data structure that can hold related data together. Each structure type variable contains all the members declared inside the structure.

Union:

- A union is another user-defined data type that allows you to combine different data types into a single data type. Unlike structures, unions allocate memory only for the largest member in the union. All members share the same memory location, and changing the value of one member may affect the values of other members.

Features of Structure:

- Grouping Data: Structures are useful for grouping related data together, making the code more organized and easier to manage.
- Member Access: You can access individual members of a structure using the dot (.) operator.
- Data Flexibility: Each structure member can have a different data type, giving you flexibility in creating complex data structures.
- Memory Allocation: Memory for each member in a structure is allocated separately, resulting in a structure size equal to the sum of the sizes of its members.

Features of Union:

- Memory Optimization: Unions are useful for optimizing memory usage when storing different data types in the same memory location.
- Shared Memory: All members of a union share the same memory location, and changing one member's value can overwrite the other members' values.
- Single Memory Allocation: Memory for a union is allocated only for the largest member, resulting in a union size equal to the size of the largest member.

[illegible]

- Data Overwriting: Since all members share the same memory, only one member can be used at a time, and overwriting one member with another may lead to data loss.

10..1 Introduction to Structure

A structure is a user-defined data type that allows you to group together multiple variables of different data types under a single name. Each variable within a structure is called a member or a field. Structures provide a way to represent a collection of related data as a single unit, making it easier to organize and manipulate complex data in a program.

Syntax of Declaring a Structure:

```
struct Structure Name {

     Datatype member1;

     Datatype member2;

     // more members...

};
```

Here, Structure Name is the name of the structure, and member1, member2, etc., are the individual members of the structure, each with its data type.

Creating Structure Variables:

Once you have defined a structure, you can create variables of that structure type. Each variable will contain the member variables defined in the structure.

```
struct structure Name variable Name; // Creates a structure
variable
```

Accessing Structure Members:

You can access the individual members of a structure variable using the dot (.) operator. The dot operator allows you to assign values to the members and retrieve their values.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
variableName.member1 = value1; // Assigning value to member1

variableName.member2 = value2; // Assigning value to member2

// Accessing members to retrieve values

printf("Value of member1: %d\n", variableName.member1);

printf("Value of member2: %f\n", variableName.member2);
```

Initializing Structure Variables:

You can initialize a structure variable at the time of declaration or later in the program.

```
struct structure Name variable Name = {value1, value2}; //
Initializing at declaration

            // Initializing later in the program

            variableName.member1 = value1;

            variableName.member2 = value2;
```

Use Cases of Structures:

Structures are widely used in programming for various purposes, such as:

Representing real-world entities (e.g., a person, a car, a book) by grouping related data together.

Organizing and managing complex data, like database records, file data, or data fetched from external sources.

Passing multiple related variables as arguments to functions in a convenient and organized manner.

Storing data in an organized way within arrays or linked data structures.

10.2 Declaring Structures:

Declaring structures involves defining a user-defined data type that allows you to group together multiple variables of different data types under a single name. The

[illegible]

For Vivekananda Global University, Jaipur

structure acts as a container, holding related data items as members. Each member within the structure can have its own data type, creating a composite data type.

Syntax of Declaring a Structure:

The syntax to declare a structure in C is as follows:

```
struct structure Name {

     Datatype member1;

     Datatype member2;

     // more members...

};
```

Here, structure Name is the name of the structure, and member1, member2, etc., are the individual members of the structure, each with its data type. You can have as many members as needed, and each member is separated by a semicolon.

Example:

Let's declare a structure called Person to represent a person's information with name, age, and height:

```
struct Person {

     char name[50];

     int age;

     float height;

};
```

Creating Variables of the Structure Type:

After defining the structure, you can create variables of that structure type. Each variable will contain the member variables defined in the structure.

[illegible]

```
struct structure Name variableName1; // Creates a structure
variable of type structure Name

struct structure Name variableName2; // Creates another
structure variable of type structure Name
```

Example:

Using the Person structure definition from the previous step, we can create variables of the Person type as follows:

```
struct Person person1; // Creates a structure variable of
type Person

struct Person person2; // Creates another structure
variable of type Person
```

Initializing Structure Variables:

You can initialize structure variables at the time of declaration using curly braces {} and providing values for each member variable in the order they appear in the structure definition.

```
struct Person person1 = {"John Doe", 30, 6.2}; //
Initializes person1 with the given values
```

Alternatively, you can assign values to the structure members after declaration:

```
    struct Person person2;

strcpy(person2.name, "Jane Smith");

person2.age = 25;

person2.height = 5.8;
```

Simple example of declaring structures in C to represent information about students and their grades:

```
#include <stdio.h>

// Structure declaration for Student
```

```
struct Student {

    char name[50];

    int age;

    float grade;

};

int main() {

    // Declaring structure variables

    struct Student student1;

    struct Student student2;

    // Initializing structure members for student1

    strcpy(student1.name, "Alice");

    student1.age = 20;

    student1.grade = 87.5;

    // Initializing structure members for student2

    strcpy(student2.name, "Bob");

    student2.age = 22;

    student2.grade = 78.2;

    // Displaying information about student1

    printf("Student 1\n");

    printf("Name: %s\n", student1.name);

    printf("Age: %d\n", student1.age);

    printf("Grade: %.2f\n\n", student1.grade);

    // Displaying information about student2
```

For Vivekananda Global University, Jaipur
[illegible]
Registrar

```
    printf("Student 2\n");
    printf("Name: %s\n", student2.name);
    printf("Age: %d\n", student2.age);
    printf("Grade: %.2f\n", student2.grade);
    return 0;
}
```

Output:

```
Student 1
Name: Alice
Age: 20
Grade: 87.50
Student 2
Name: Bob
Age: 22
Grade: 78.20
```

10.3 Accessing Structures:

Accessing structures in C involves accessing and modifying the member variables of a structure variable. You can use the dot (.) operator to access the individual members of a structure. The dot operator allows you to assign values to the members and retrieve their values.

Syntax for Accessing Structure Members:

To access a member of a structure, you use the dot operator as follows

```
structureVariable.memberName
```

[illegible]

Here, structure Variable is the variable of the structure type, and member Name is the name of the member variable you want to access.

Example:

Let's consider the previously defined Person structure representing a person's information:

```
struct Person {
    char name[50];
    int age;
    float height;
};
```

Accessing Structure Members:

You can access the individual members of a structure variable as shown below:

```
struct Person person1; // Create a structure variable of
type Person
// Accessing and modifying members of the structure
strcpy(person1.name, "John Doe");
person1.age = 30;
person1.height = 6.2;
// Accessing and printing members of the structure
printf("Name: %s\n", person1.name);
printf("Age: %d\n", person1.age);
printf("Height: %.2f\n", person1.height);
```

In this example, we first create a structure variable person1 of type Person. We then access and modify the members of the structure using the dot operator. For example, we use person1.name to access and set the name member, person1.age to access and set the age member, and person1.height to access and set the height member.

The printf() function is used to print the values of the members of person1 to the console.

Simple example of declaring structures in C to represent information about students and their grades:

```
#include <stdio.h>
// Structure declaration for Student
struct Student {
    char name[50];
    int age;
    float grade;
};
int main() {
    // Declaring structure variables
    struct Student student1;
    struct Student student2;
    // Initializing structure members for student1
    strcpy(student1.name, "Alice");
    student1.age = 20;
    student1.grade = 87.5;
    // Initializing structure members for student2
    strcpy(student2.name, "Bob");
    student2.age = 22;
    student2.grade = 78.2;
    // Displaying information about student1
    printf("Student 1\n");
    printf("Name: %s\n", student1.name);
    printf("Age: %d\n", student1.age);
    printf("Grade: %.2f\n\n", student1.grade);
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
    // Displaying information about student2
    printf("Student 2\n");
    printf("Name: %s\n", student2.name);
    printf("Age: %d\n", student2.age);
    printf("Grade: %.2f\n", student2.grade);
    return 0;
}
```

```
Output:
Student 1
Name: Alice
Age: 20
Grade: 87.50
Student 2
Name: Bob
Age: 22
Grade: 78.20
```

10.4 Variable user of Structures:

Structures provide a way to group together multiple variables of different data types under a single name. The variables (also called members) within a structure have various uses and advantages:

- **Grouping Related Data:** The primary purpose of using variables in structures is to group related data together. For example, if you need to store information about a person, you can create a structure with variables representing their name, age, address, etc.
- **Representing Real-World Entities:** Structures allow you to create user-defined data types that can represent real-world entities. For instance, you can define a structure to represent a car with variables for make, model, year, and price.

- **Data Organization and Readability:** Structures improve code organization and readability by encapsulating related data in a single entity. This makes the code more modular and easier to understand.
- **Passing Multiple Arguments to Functions:** You can pass a structure as an argument to a function, allowing you to work with multiple related values within the function. This simplifies the function call and enhances code clarity.
- **Returning Multiple Values from Functions:** Functions can return only one value, but by using a structure, you can effectively return multiple related values as a single unit.
- **Data Storage and File I/O:** Structures are useful for data storage, especially when dealing with records or data structures in files. You can read/write entire structures to/from files in a well-organized manner.
- **Data Management:** Structures help manage complex data effectively. For example, in a program handling employee records, each employee's information can be stored in a structure, making it easier to manage the data.
- **Implementing Complex Data Structures:** Structures serve as building blocks for implementing more complex data structures like linked lists, trees, graphs, and stacks. Each node in these data structures can be a structure itself.
- **Memory Efficiency:** Using structures can save memory by grouping related data together, avoiding the need to create multiple individual variables.
- **Data Integrity and Type Safety:** Structures enforce type safety, ensuring that data of the correct type is stored in each member variable. This helps prevent programming errors and data corruption.
- **Code Reusability:** Once you have defined a structure, you can create multiple variables of that structure type, allowing you to reuse the structure to store data for multiple instances.
- **Ease of Data Manipulation:** The dot (.) operator makes it easy to access and modify the member variables of a structure, allowing you to work with the data stored within the structure effectively.

[illegible]

Some programming examples that showcase the various uses of variables within structures in C:

Example 1: Representing Students

```
#include <stdio.h>

#include <string.h>

// Structure declaration for Student

struct Student {

    char name[50];

    int age;

    float grade;

};

int main() {

    // Declaring and initializing structure variables

    struct Student student1 = {"Alice", 20, 87.5};

    struct Student student2 = {"Bob", 22, 78.2};

    // Displaying information about student1

    printf("Student 1\n");

    printf("Name: %s\n", student1.name);

    printf("Age: %d\n", student1.age);

    printf("Grade: %.2f\n\n", student1.grade);

    // Displaying information about student2

    printf("Student 2\n");

    printf("Name: %s\n", student2.name);

    printf("Age: %d\n", student2.age);
```

For Vivekananda Global University, Jaipur

Registrar

```
    printf("Grade: %.2f\n", student2.grade);

    return 0;

}
```

Example 2: Function with Structure Argument

```
#include <stdio.h>

#include <string.h>

// Structure declaration for Student

struct Student {

    char name[50];

    int age;

    float grade;

};

// Function to display student information

void display Student(struct Student s) {

    printf("Name: %s\n", s.name);

    printf("Age: %d\n", s.age);

    printf("Grade: %.2f\n", s.grade);

}

int main() {

    // Declaring and initializing a structure variable

    struct Student student1 = {"Alice", 20, 87.5};

    // Passing the structure variable as an argument to the
function

    display Student(student1);
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
    return 0;

}
```

Example 3:- Returning Structure from a Function

```
#include <stdio.h>

#include <string.h>

// Structure declaration for Point

struct Point {

    int x;

    int y;

};

// Function to create and return a Point structure

struct Point create Point(int x, int y) {

    struct Point p;

    p.x = x;

    p.y = y;

    return p;

}

int main() {

    // Calling the function to create a Point structure

    struct Point point1 = create Point(10, 5);

    // Displaying the values of the Point structure

    printf("Point  coordinates:  (%d,  %d)\n",  point1.x,
point1.y);

    return 0;
```

For Vivekananda Global University, Jaipur
[illegible]
Registrar

```
}
```

Example 4: Structure Array

```
#include <stdio.h>

// Structure declaration for Rectangle

struct Rectangle {

    int length;

    int width;

};

int main() {

    // Declaring an array of Rectangle structures

    struct Rectangle rectangles[3];

    // Initializing structure array elements

    rectangles[0].length = 5;

    rectangles[0].width = 3;

    rectangles[1].length = 10;

    rectangles[1].width = 7;

    rectangles[2].length = 8;

    rectangles[2].width = 6;

    // Displaying the areas of the rectangles

    for (int I = 0; i < 3; i++) {

        int      area      =      rectangles[i].length      *
rectangles[i].width;

        printf("Rectangle %d area: %d\n", i+1, area);

  }
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
        return 0;

    }
```

These examples demonstrate the various uses of variables within structures, such as representing entities, passing structures to functions, returning structures from functions, and using structure arrays for data organization.

10.5 Unions

A union is a user-defined data type that allows you to store different data types in the same memory location. Unlike structures, where each member has its own memory location, all members of a union share the same memory space. The size of a union is determined by the size of its largest member.

Syntax of Declaring a Union:

The syntax to declare a union in C is similar to that of a structure:

```
union UnionName {

    datatype member1;

    datatype member2;

    // more members...

};

Example:

#include <stdio.h>

// Union declaration for storing different data types

union Data {

    int in Value;

    float float Value;

    char charValue;

};
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
int main() {

    union Data myData;

    myData.intValue = 42;

    // Accessing the integer value

    printf("Integer value: %d\n", myData.intValue);

   // Accessing the same memory location as a float

    myData.floatValue = 3.14;

    printf("Float value: %.2f\n", myData.floatValue);

    // Accessing the same memory location as a character

    myData.charValue = 'A';

    printf("Character value: %c\n", myData.charValue);

    // The value of intValue is overwritten by charValue
due to shared memory.

    printf("Integer value after setting character value:
%d\n", myData.intValue);

    return 0;

}
```

Output

```
Integer value: 42

Float value: 3.14

Character value: A

Integer value after setting character value: 65
```

n this example, we have a union named Data that can store an integer, a float, or a character at the same memory location. We create a variable myData of type Data and store an integer value (42) in it. We then access and print the integer value using the dot operator.

Next, we store a float value (3.14) in the same memory location and access and print it as a float. The union shares the memory, so when we store a character value ('A') in the union, it overwrites the integer value previously stored.

Unions are useful when you need to store different types of data in the same memory location, and you know that only one member will be used at a time. However, be cautious about data interpretation and ensure that you access the correct member based on the context.

10.6 Create Union Variables

To create union variables in C, you follow a similar syntax to declaring other variables. Here's how you create union variables:

Step 1: Define the Union:

First, you need to define the union using the union keyword and specify the member variables within curly braces.

Step 2: Create Union Variables:

After defining the union, you can create variables of that union type. Each variable will have the data of the largest member defined in the union.

The syntax for Defining Union and Creating Union Variables:

```
// Step 1: Define the Union

union UnionName {

    dataType member1;

    dataType member2;

    // more members...

};
```

For Vivekananda Global University, Jaipur
[illegible]
Registrar

```
int main() {
    // Step 2: Create Union Variables
    union UnionName variableName1;
    union UnionName variableName2;
  // more union variables...

    // Use the union variables to store and access data as needed.
    return 0;
}
```

Example:

Let's create a union to store different data types and demonstrate the creation of union variables:

```
#include <stdio.h>
// Step 1: Define the Union
union MyUnion {
    int intValue;
    float floatValue;
    char charValue;
};
int main() {
    // Step 2: Create Union Variables
    union MyUnion data1; // Creates a union variable of type MyUnion
```

[illegible]

For Vivekananda Global University, Jaipur
Registrar

```
        union MyUnion data2; // Creates another union variable
    of type MyUnion


        // Storing data in the union variables

        data1.intValue = 42;

        data2.floatValue = 3.14;


        // Accessing and printing the values of the union
    variables

        printf("data1 intValue: %d\n", data1.intValue);

        printf("data2 floatValue: %.2f\n", data2.floatValue);

    // The union variables share the same memory, so the value
of data1.intValue will be overwritten.

    data1.charValue = 'A';

    printf("data1 intValue after setting charValue: %d\n",
data1.intValue);


    return 0;

}
```

```
Output:

data1 intValue: 42

data2 floatValue: 3.14

data1 intValue after setting charValue: 65
```

In this example, we define a union MyUnion that can store an integer, a float, or a character at the same memory location. We then create two union variables data1 and data2 and store different data types in them. Since the union variables share the same memory, setting a value for one member can affect the values of other members, as shown when we set data1.charValue to 'A', which changes the integer value stored in data1.intValue.

Unions are useful when storing different types of data in the same memory location, but you should be cautious about data interpretation and ensure you access the correct member based on the context.

10.7 Accessing Union Members

You use the dot (.) operator to access union members in C, just like accessing structure members. However, remember that all union members share the same memory space. Therefore, you can access only one member at a time, and you should know which member is currently being used to interpret the data correctly.

Here's how you can access union members:

Syntax for Accessing Union Members:

```
union UnionName {

    dataType member1;

    dataType member2;

    // more members...

};


int main() {

    union UnionName variableName;

    variableName.member1 = value; // Accessing and setting
member1

    dataType data = variableName.member2; // Accessing
member2
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
    // Use the union members as needed.

    return 0;
```

Example:

Let's use the same union from the previous example and demonstrate how to access its members:

```
#include <stdio.h>

// Union declaration for MyUnion

union MyUnion {

    int intValue;

    float floatValue;

    char charValue;

};

int main() {

    union MyUnion data;


    data.intValue = 42;

    printf("Integer value: %d\n", data.intValue);

    data.floatValue = 3.14;

    printf("Float value: %.2f\n", data.floatValue);

    data.charValue = 'A';

    printf("Character value: %c\n", data.charValue);

    // Since all members share the same memory location,
the value of intValue is overwritten by charValue.

    printf("Integer value after setting character value:
%d\n", data.intValue);

    return 0;

}
```

```
Output

Integer value: 42

Float value: 3.14
```

For Vivekananda Global University, Jaipur
Registrar

```
Character value: A
Integer value after setting character value: 65
```

In this example, we create a union variable data of type MyUnion. We then access and set each member using the dot operator: data.intValue, data.floatValue, and data.charValue. Remember that modifying one member of the union can affect the values of other members since they share the same memory space.

When accessing union members, make sure to access the member that corresponds to the current data stored in the union. Unions are particularly useful when you need to switch between different data types while using the same memory location. However, be cautious about data interpretation and ensure you access the correct member based on the context.

Uses of Unions:

Unions are useful when you need to store different types of data in the same memory location, and you know that only one member will be used at a time. Some common uses of unions include:

- **Memory Optimization:** Unions can be used to save memory when dealing with multiple data types that occupy the same memory space.
- **Type Conversion:** Unions can easily convert data from one type to another. For example, you can store an integer in a union and then read it as a floating-point number.
- **Data Interchange:** Unions are often used when dealing with data interchange formats like binary data or network packets.
- **Variant Records:** Unions are useful when you have a data structure where different fields are valid under different conditions.

few more programming examples to showcase the different use cases of unions in C:

Example 1: Variant Records

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

[illegible]
For Vivekananda Global University, Jaipur
Registrar

```
// Union declaration for a variant record
union VariantRecord {
    int intValue;
    float floatValue;
    char stringValue[50];
};
int main() {
    union VariantRecord record1;
    record1.intValue = 42;
    printf("Integer value: %d\n", record1.intValue);
    record1.floatValue = 3.14;
    printf("Float value: %.2f\n", record1.floatValue);
    strcpy(record1.stringValue, "Hello");
    printf("String value: %s\n", record1.stringValue);
    return 0;
}
```

Example 2: Type Conversion

```
#include <stdio.h>
// Union declaration for type conversion
union TypeConversion {
    int intValue;
    float floatValue;
};
int main() {
    union TypeConversion data;
    data.floatValue = 3.14;
    printf("Float value: %.2f\n", data.floatValue);
    // Convert the float value to an integer using type
casting
    data.intValue = (int)data.floatValue;
    printf("Integer value: %d\n", data.intValue);
    return 0;
```

[illegible]

```
}
```

Example 3: Data Interchange

```
#include <stdio.h>
// Union declaration for data interchange
union DataInterchange {
    int intValue;
    char byteArray[4];
};
int main() {
    union DataInterchange data;
    data.intValue = 16909060; // Binary representation:
00000001 00000010 00000011 00000100
    // Access individual bytes using byteArray
    printf("Byte 0: %d\n", data.byteArray[0]); // Output:
4
    printf("Byte 1: %d\n", data.byteArray[1]); // Output:
3
    printf("Byte 2: %d\n", data.byteArray[2]); // Output:
2
    printf("Byte 3: %d\n", data.byteArray[3]); // Output:
1
    return 0;
}
```

Example 4: Union in Structure

```
#include <stdio.h>
// Union declaration for a variant record
union VariantRecord {
    int intValue;
    float floatValue;
    char stringValue[50];
};
// Structure declaration using the union
```

[illegible]

```
struct Data {
    int dataType; // 0 for int, 1 for float, 2 for string
    union VariantRecord value;
};
int main() {
    struct Data data1;
    data1.dataType = 0;
    data1.value.intValue = 42;
    struct Data data2;
    data2.dataType = 1;
    data2.value.floatValue = 3.14;
    struct Data data3;
    data3.dataType = 2;
    strcpy(data3.value.stringValue, "Hello");
    printf("Data 1: %d\n", data1.value.intValue);
    printf("Data 2: %.2f\n", data2.value.floatValue);
    printf("Data 3: %s\n", data3.value.stringValue);
    return 0;
}
```

10.8 Summary

Structures:

- Structures are user-defined data types that allow you to group together multiple variables of different data types under a single name.
- Each variable within a structure is called a member or a field.
- You declare a structure using the struct keyword, followed by the structure name and the member variables enclosed in curly braces.
- You can create structure variables and initialize their members using the dot (.) operator.
- Structures are useful for organizing related data, representing real-world entities, passing multiple arguments to functions, and returning multiple values from functions.

[illegible]

- They help in managing complex data and improving code readability and maintainability.
- You can also use structures to implement complex data structures like linked lists, trees, and graphs.

Unions :

- Unions are user-defined data types that allow you to store different data types in the same memory location.
- All members of a union share the same memory space, and the size of its largest member determines the size of the union.
- You declare a union using the union keyword, followed by the union name and the member variables enclosed in curly braces.
- You can create union variables; each variable will have the data of the largest member defined in the union.
- You can access union members using the dot (.) operator, but you should know which member is currently being used to interpret the data correctly.
- Unions are useful when you need to switch between different data types while using the same memory location.
- They are often used for variant records, type conversion, data interchange, and saving memory when dealing with multiple data types.

Differences:

- Each member has its own memory location in structures, and you can access multiple members simultaneously.
- In unions, all members share the same memory location, and you can access only one member at a time.
- Structures are useful for organizing related data, while unions are used when you need to store different types of data in the same memory location.

Similarities:

- Both structures and unions are user-defined data types.

[illegible]

- They allow the grouping multiple variables of different data types under a single name.
- They provide a way to manage complex data and improve code organization.

Overall, structures and unions are valuable features in C programming, providing flexibility and efficiency when dealing with related data and different data types. Proper usage of structures and unions can greatly enhance the organization and readability of your C programs.

10.9 Review Questions

1. What is a structure in C, and what is its primary purpose?
2. How do you declare a structure in C?
3. How do you create variables of a structure type?
4. How do you access members of a structure variable?
5. What is a union in C, and how is it different from a structure?
6. How do you declare a union in C?
7. How do you create variables of a union type?
8. What is the primary use of unions in C?
9. How do you access members of a union variable?
10. What are the similarities between structures and unions in C?

10.10 Keywords

struct: A keyword used to define a user-defined data type that allows grouping multiple variables of different data types under a single name.

typedef: A keyword used to create a new name (alias) for an existing data type, making using structures with a shorter name easier.

member: Refers to an individual variable inside a structure that holds data related to a specific attribute of the entity the structure represents.

dot (.): An operator used to access the members of a structure variable.

union: A keyword used to define a user-defined data type that allows storing different data types in the same memory location.

member: Refers to an individual variable inside a union, representing one of the data types that can be stored in the union.

dot (.): An operator used to access the members of a union variable, similar to accessing structure members.

These keywords play essential roles in defining and working with structures and unions in C, enabling efficient data organization and manipulation.

10.11 References

1. E.Balaguruswamy, "Computing Fundamentals & C Programming", Tata McGraw Hill, 2008.
2. B. A. Forouzan & R. F. Gilberg "Computer Science – A structured programming Approach Using C", 2011
3. E. Bala Guruswamy, "Programming in ANSI" Tata McGraw Hill, 2011.

[illegible]

Table of Content

[illegible]
For Vivekananda Global University, Jaipur
Registrar

Learning Objectives

After studying this unit, the student will be able to:

- Define what storage classes are and their significance in memory management.
- Differentiate between automatic, register, static, and external storage classes.
- Identify scenarios where each storage class is most appropriate.
- Define variable scoping and its role in determining variable visibility.
- Recognize the three main scopes: block scope, function scope, and global scope.
- Understand how variables with different scopes are accessed and manipulated.
- Explain the concept of automatic storage class and its usage within functions and blocks.
- Demonstrate how automatic variables are allocated and deallocated on the stack.
- Identify scenarios where automatic storage is appropriate for managing local variables.
- Describe the purpose of the register storage class in suggesting register allocation for variables.
- Analyze situations where using the register storage class can improve program performance.
- Understand the limitations and considerations when using the register storage class.
- Control Variable Lifetime with Static Storage Class:
- Explore the behavior of static variables, including their extended lifetime beyond function calls.
- Demonstrate how static variables retain their values between different function invocations.

[illegible]

Introduction

A storage class is a fundamental concept that governs how variables are allocated, accessed, and managed in memory during program execution. Storage classes determine variables' lifetime, visibility, and scope, allowing programmers to control memory usage and data manipulation effectively. Different programming languages provide various storage classes, each serving distinct purposes and offering unique behaviours.

Storage classes are crucial for understanding how variables are stored in memory and how they behave throughout the program's execution. They play a vital role in optimizing memory usage, improving program performance, and ensuring data integrity. The main storage classes in many programming languages, including C and C++, are Automatic, Register, Static, and External.

- **Automatic Storage Class:**
  The automatic storage class is the default storage class for variables declared within functions or blocks of code. These variables are typically allocated on the stack, and the compiler automatically manages their memory. Automatic variables exist when the block they are declared in is entered and destroyed when the block is exited. They are well-suited for managing local variables that need to be created and discarded frequently during function calls.
- **Register Storage Class:**
  The register storage class suggests to the compiler that a variable should be stored in a CPU register instead of memory. Registers are faster to access, and using the register storage class may improve the performance of frequently accessed variables. However, the compiler may or may not allocate the variable in a register, as it depends on hardware architecture and optimization decisions.
- **Static Storage Class:**
  The static storage class is used to control the lifetime and visibility of a variable. Variables with the static storage class are allocated memory in the

[illegible]

data segment rather than the stack. They retain their values between different function calls, making them suitable for maintaining state or sharing data across function invocations.

- **External Storage Class:**

  The external storage class is used to declare global variables that can be accessed across multiple source files. External variables have a global scope, meaning they can be accessed from any part of the program. The actual definition of an external variable must be provided in a separate source file using the "extern" keyword.

- **Scoping:**

  scoping refers to the rules that determine the visibility, accessibility, and lifetime of variables within different parts of a program. The scope of a variable defines the region of the program where the variable can be referenced and used. Understanding scoping is crucial for writing clean, organized, and bug-free code, as it helps prevent naming conflicts and ensures proper data encapsulation.

11.2 Automatic Storage Class

the automatic storage class is one of the four primary storage classes, which also include static, register, and external storage classes. The automatic storage class is the default storage class for variables declared within functions or blocks of code in languages like C and C++. It defines how variables are allocated, accessed, and managed in memory during the execution of a function or block.

Key Characteristics of Automatic Storage Class:

- **Scope:**

Variables with automatic storage class have block scope, meaning they are only accessible within the block (a pair of curly braces) where they are defined.

Once the block is exited, the variables are automatically deallocated, and their memory is released.

[illegible]

- **Lifetime:**

Automatic variables are created when the block containing their declaration is entered during the program's execution.

They are destroyed when the block is exited or when their lifetime ends due to the end of the function call.

The lifetime of automatic variables is limited to the duration of their enclosing block.

- **Memory Allocation:**

Memory for automatic variables is typically allocated on the stack.

The stack is a region of memory that is efficiently managed by the compiler, allowing for fast allocation and deallocation of memory during function calls.

- **Default Storage Class:**

The "auto" keyword, which explicitly specifies the automatic storage class, is rarely used in modern programming languages like C and C++.

The automatic storage class is implied when no other storage class (e.g., static, register, or extern) is specified.

Example:

```
void example Function() {

    int x = 10; // 'x' has automatic storage class

    // Code here

    {

        int y = 5; // 'y' also has automatic storage class but
is a separate variable

        // Code here

    }
```

[illegible]

```
    // 'y' is no longer accessible here (out of its block scope)
}
```

Advantages and Use Cases:

Automatic variables are suitable for managing local data within functions or blocks. They are created when needed and automatically deallocated, which helps manage memory efficiently.

The use of the automatic storage class helps prevent naming conflicts since variables are limited to the block where they are declared.

It is the default storage class, so the programmer does not need to explicitly specify it in most cases.

It is important to be mindful of the scope and lifetime of automatic variables, especially when using pointers or passing variables by reference to other functions. Variables with automatic storage class should not be accessed outside their block scope to avoid undefined behavior and memory-related issues.

An example demonstrating the use of the automatic storage class in C:

```
#include <stdio.h>

void exampleFunction() {

    auto int x = 10; // 'auto' is optional, as it is the
default storage class for local variables

    x++; // Increment 'x'

    printf("Value of x inside exampleFunction: %d\n", x);

}

int main() {

    exampleFunction();
    // 'x' is not accessible here, as it is within the
scope of exampleFunction

    // Let's use another block to demonstrate block scope
    {
        auto int y = 5; // 'auto' is optional here too
        printf("Value of y inside the nested block: %d\n",
y);
    }
    // 'y' is not accessible here, as it is within the
scope of the nested block

    return 0;
}
```

Output:

```
Value of x inside exampleFunction: 11
Value of y inside the nested block: 5
```

In this example, we define the exampleFunction with an automatic variable x. The variable x is automatically allocated on the stack when the function is called and automatically deallocated when the function exits. Each time exampleFunction is called, a new instance of x is created with an initial value of 10. The value of x is then incremented within the function, and the updated value is printed.

Note that the auto keyword is optional here, as it is the default storage class for local variables within functions. In modern programming, the explicit use of the auto keyword is uncommon, and most programmers omit it.

For Vivekan[illegible] [illegible] Registrar

Additionally, we demonstrate the block scope of automatic variables by declaring another variable y inside a nested block. The variable y is accessible only within the nested block and is not accessible outside of it.

Overall, automatic variables are suitable for managing local data within functions or blocks, as they are automatically created and deallocated within their respective scopes.

11.3 Register Storage Class:

The register storage class is one of the four primary storage classes, alongside automatic, static, and external storage classes. The register storage class is used to suggest to the compiler that a variable should be stored in a CPU register rather than in memory. Registers are fast-access storage locations located directly on the CPU, allowing for quicker access to the variable's value compared to accessing data from memory.

It is important to note that the register storage class is only a suggestion to the compiler. The compiler may or may not honor this suggestion based on hardware architecture, optimization decisions, and the availability of registers. In modern programming languages like C and C++, the usage of the register storage class is limited, as compilers are highly optimized and capable of making register allocation decisions on their own.

Key Characteristics of Register Storage Class:

- **Suggesting Register Allocation:**

The "register" keyword is used to suggest to the compiler that a variable should be stored in a CPU register.

The keyword is placed before the variable's declaration, like other storage class keywords.

Example :

```
void exampleFunction() {
```

[illegible]
For Vivekananda
Registra

```
    register int x; // Suggesting 'x' to be stored in a register
(Compiler may or may not honor this)

    // Code here

}
```

- **Compiler Discretion:**

The compiler may choose to store the variable in a register or in memory based on various factors.

Factors influencing the decision include the hardware architecture, the number of available registers, and the variable's usage pattern in the code.

Limitations:

The register storage class cannot be applied to variables that have a memory address, such as arrays and pointers, as registers do not have addresses that can be used for indirect access.

The use of the register storage class might be ignored by the compiler, especially if there are more variables suggested to be stored in registers than available registers in the CPU.

Use Cases:

- Historically, the register storage class was used to optimize critical sections of code, where variables needed frequent access, such as in time-critical algorithms or low-level programming.
- In modern programming, relying on the register storage class explicitly is not common, as optimizing compilers are generally capable of making efficient register allocation decisions on their own.

Overall, while the register storage class may have been more prevalent in earlier programming, its use in modern programming is limited due to the advancements in compiler optimizations. Programmers are encouraged to focus on writing clean

[illegible]

For Vivekananda Global University, Jaipur

Registrar

and maintainable code, allowing the compiler to handle register allocation and other performance optimizations effectively.

In modern programming languages like C and C++, the use of the register storage class is less common due to the advancements in compiler optimizations. Compilers are now highly optimized and can make efficient register allocation decisions on their own. As a result, the register keyword is considered a hint to the compiler rather than a strict directive.

In some cases, the register keyword may be ignored by the compiler if it determines that storing the variable in a register does not provide significant performance benefits. Moreover, some compilers might even issue a warning or ignore the keyword altogether.

However, to provide an example of the register storage class, you can do the following:

```
#include <stdio.h>

void exampleFunction() {

    register int x = 10; // Suggesting 'x' to be stored in
a register

    x++; // Increment 'x'

    printf("Value of x inside exampleFunction: %d\n", x);

}

int main() {

    exampleFunction();

    // 'x' is not accessible here, as it is within the
scope of exampleFunction

    return 0;

}
```

[illegible]
For Vivekananda Global University, Jaipur
Registrar

In this example, we have a function exampleFunction() with a variable x declared with the register storage class. We suggest to the compiler that x should be stored in a CPU register for faster access. However, as mentioned earlier, whether the compiler will allocate x to a register depends on various factors such as hardware architecture, optimization settings, and the specific usage of the variable in the code.

Again, keep in mind that the use of register is optional and might not have a significant impact on performance in modern programming. Instead, it is recommended to rely on the compiler's optimization capabilities and focus on writing clean and maintainable code.

11.4 External Storage Class

The external storage class is one of the four primary storage classes, along with automatic, static, and register storage classes. The external storage class is used to declare global variables that can be accessed across multiple source files in a program. These variables have a global scope, meaning they are visible and accessible from any part of the program after their declaration.

The external storage class allows for the sharing of data between different parts of a program, making it a useful feature when multiple source files need to access the same global variable. However, it is important to note that the actual definition of the external variable must be provided in one of the source files, and its declaration must be marked with the "extern" keyword in other files where it is used.

Key Characteristics of External Storage Class:

Declaration and Definition:

The external storage class is declared using the "extern" keyword.

[illegible]
For Vivekananda Global University, Jaipur
Registrar

The declaration of an external variable in a source file informs the compiler about the existence and type of the variable, allowing it to be accessed from other files.

The actual definition (memory allocation) of the external variable is provided in one of the source files.

Example (file1.c):

```
// file1.c
extern    int    globalVariable;    //    Declaration    of
'globalVariable' in this file


void exampleFunction() {
    // Access 'globalVariable' here
}
```

Example (file2.c):

```
// file2.c
int globalVariable = 100; // Definition of 'globalVariable'
in this file


void anotherFunction() {
    // Access 'globalVariable' here
}
```

Sharing Data Across Source Files:

The external storage class is particularly useful when multiple source files need to share the same global variable.

[illegible]
For Vivekananda Global University, Jaipur
Registrar

It enables modular programming by allowing data to be shared between different parts of a program without the need for duplication.

Use Cases:

External variables are often used for sharing configuration data, constants, or other data that needs to be accessed globally throughout the program.

They can also be used for sharing data between the main program and various libraries or modules.

It is essential to use the external storage class with caution, as global variables can lead to potential issues such as name clashes, data integrity problems, and difficulties in debugging and maintenance. Whenever possible, consider using other techniques like passing parameters and returning values from functions or using encapsulation to limit the scope of data. Global variables should be used judiciously to maintain code clarity and organization.

To demonstrate the use of the external storage class, you need to split the example into multiple source files. Here's an example of how you can use the external storage class to share a global variable between two source files:

Source file 1 (main.c):

```
#include <stdio.h

//declaration of the external variable (provided by another
source file)

     extern int globalVar;

     int main() {

          // Access and modify the external variable

          globalVar = 42;

          // Use the external variable in this source file
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
    printf("Value of globalVar in main.c: %d\n",
globalVar);

    return 0;

}
```

Source file 2 (other_file.c):

```
#include <stdio.h>

// Definition of the external variable (to be shared with
other source files)

int globalVar; // Note: 'extern' is not used here

void otherFunction() {

    // Use the external variable in this source file

    printf("Value of globalVar in other_file.c: %d\n",
globalVar);

}

Output:

Value of globalVar in main.c: 42

Value of globalVar in other_file.c: 42
```

Explanation:

In this example, we have two source files, "main.c" and "other_file.c".

In "main.c", we declare the external variable globalVar using the extern keyword, which tells the compiler that the variable is defined in another source file. We then access and modify the value of globalVar in "main.c".

In "other_file.c", we define the external variable globalVar without the extern keyword. This is where the memory for the variable is allocated.

[illegible]

For Vivekananda Global University, Jaipur

Regi

The otherFunction() in "other_file.c" also accesses the value of globalVar and prints it.

The external storage class allows us to share the same global variable (globalVar) between two separate source files, enabling modularity and reusability in large programs. The external variable can be accessed and modified in any source file that includes its declaration with the extern keyword. This way, we can avoid duplicating the variable and ensure consistent values throughout the program.

11.5 Static Storage Class

The static storage class is one of the four primary storage classes, along with automatic, register, and external storage classes. The static storage class is used to control the lifetime and visibility of variables in a program. Variables declared with the static storage class have a longer lifetime than automatic variables and retain their values across different function calls.

The static storage class is especially useful when you want a variable to maintain its value between function calls, and you want to limit its visibility to the block or function where it is defined. It is important to note that the static storage class can be applied to both global variables and local variables, each having its own implications.

Key Characteristics of Static Storage Class:

Lifetime and Value Persistence:

- Static variables have a lifetime that spans the entire execution of the program.
- When a static variable is defined, memory is allocated for it in the data segment of the program, rather than on the stack (like automatic variables).
- The value of a static variable persists between different function calls, retaining its value from the previous call.

Block and Function Scope:

- When a static variable is declared inside a function or block, it retains its value between subsequent calls to that function or block.
- If a static variable is declared outside any function or block (at the global level), it is accessible from the point of its declaration to the end of the source file where it is defined.

Default Initialization:

If a static variable is not explicitly initialized, it is automatically initialized to zero (for variables at the global level) or to its initial value (for variables declared within a function).

Example (Static Variable within a Function):

```
#include <stdio.h>

void exampleFunction() {

   static int counter = 0; // 'counter' is a static variable

   counter++; // Increment the counter

   printf("Counter value: %d\n", counter);

}

int main() {

   exampleFunction(); // Output: Counter value: 1

   exampleFunction(); // Output: Counter value: 2

   exampleFunction(); // Output: Counter value: 3

   return 0;

}
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Example (Static Variable at the Global Level):

```
#include <stdio.h>

static int globalCounter = 0; // 'globalCounter' is a static variable at the
global level

void exampleFunction() {

  globalCounter++; // Increment the global counter

  printf("Global Counter value: %d\n", globalCounter);

}

int main() {

  exampleFunction(); // Output: Global Counter value: 1

  exampleFunction(); // Output: Global Counter value: 2

  exampleFunction(); // Output: Global Counter value: 3

  return 0;

}
```

Use Cases:

Static variables are commonly used when you need to maintain state across multiple function calls, such as in counters, unique identifiers, or caching mechanisms.

They are useful for implementing singleton patterns and other design patterns that require maintaining a single instance of an object across function calls.

It is crucial to use static variables with care, as their persistence across function calls can lead to unexpected side effects and make code harder to understand and maintain. Properly managing the scope and lifetime of static variables is essential to ensure correct behavior in the program.

[illegible]

11.6 Scope

Scoping refers to the rules that determine the visibility, accessibility, and lifetime of variables within different parts of a program. The scope of a variable defines the region of the program where the variable can be referenced and used. Understanding scoping is crucial for writing clean, organized, and bug-free code, as it helps prevent naming conflicts and ensures proper data encapsulation.

Scoping rules vary based on the programming language, but the following three common scopes are prevalent in many languages:

Block Scope:

Block scope refers to the region of code enclosed within a set of curly braces ({ }).

Variables declared inside a block are accessible only within that block and its nested blocks.

Once the block is exited, the variables declared within it are typically deallocated and no longer accessible.

Block scope is often associated with local variables declared inside functions or conditional statements.

Block scope refers to the region of code enclosed within a pair of curly braces { }. Variables declared inside a block are visible and accessible only within that block and its nested blocks. Block scope is a fundamental concept in many programming languages, including C and C++, and it plays a vital role in managing variable visibility and avoiding naming conflicts.

Key Points about Block Scope:

- Visibility: Variables declared within a block are visible and accessible only within that block. They cannot be accessed from outside the block or any nested blocks.
- Nesting: Blocks can be nested within each other. Variables declared in an outer block are accessible within inner blocks, but the reverse is not true.

[illegible]
For Vivekananda Global University, Jaipur

Registrar

- Lifetime: The lifetime of variables with block scope is limited to the duration of the block they are declared in. When the block is exited, the variables are typically deallocated and no longer accessible.
- Shadowing: If a variable with the same name is declared within an inner block, it "shadows" the variable with the same name in the outer block. This means the inner variable takes precedence within the inner block, and the outer variable remains hidden until the inner block is exited.

Example:

```
#include <stdio.h>

int main() {

    int x = 10; // Outer block variable 'x'

        {

        int y = 5; // Inner block variable 'y'

        printf("Inner block: x = %d, y = %d\n", x, y);

                // Shadowing 'x' within the inner block

        int x = 20; // Inner block variable 'x' shadows the
outer 'x'

        printf("Inner block (after shadowing): x = %d, y =
%d\n", x, y);

    }

        // 'y' is not accessible here (out of its block
scope)

    // The outer 'x' is still accessible

    printf("Outer block: x = %d\n", x);

    return 0;

}
```

[illegible]

For Vivekananda Global University, Jaipur
Registrar

```
Output:

Inner block: x = 10, y = 5

Inner block (after shadowing): x = 20, y = 5

Outer block: x = 10
```

In this example, we have an outer block and an inner block. Inside the inner block, we declare an x variable, which "shadows" the x variable from the outer block. The inner x takes precedence within the inner block, and the outer x remains hidden until the inner block is exited. Once we leave the inner block, the outer x becomes accessible again.

Block scope allows us to limit the visibility of variables to specific regions of code, which helps prevent naming conflicts and makes it easier to manage variable lifetimes. It is an essential concept in programming and aids in writing clean and maintainable code.

Example :

```
void exampleFunction() {

    int x = 10; // 'x' is within the block scope of
'exampleFunction'

    // Code here

    {

        int y = 5; // 'y' is within the block scope of this
nested block

        // Code here

    }

    // 'y' is no longer accessible here

}
```

Function Scope:

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Function scope refers to the visibility and accessibility of variables declared within a function in a programming language. Variables with function scope are visible only within the function where they are declared and cannot be accessed from outside the function or any other functions. Function scope is one of the types of local scope in programming.

Key Points about Function Scope:

- **Visibility:** Variables declared within a function are visible and accessible only within that function. They are not accessible from any other functions, including the main function.
- **Lifetime:** The lifetime of variables with function scope is limited to the duration of the function. When the function returns or is exited, the variables are typically deallocated, and their memory is released.
- **No Shadowing:** Unlike block scope, there is no variable shadowing within functions. If a variable with the same name is declared within a nested block (inside the function), it does not hide or affect the variables with the same name in the outer scope (the function itself).
- **Reusability:** Function scope allows you to declare variables within functions, which enhances modularity and reusability. These variables are temporary and serve specific purposes within the function.

Function scope defines the visibility of function parameters and local static variables.

Function parameters are accessible only within the function they belong to and act as local variables.

Local static variables are variables declared with the "static" storage class within a function. They persist between function calls but remain accessible only within the function.

Function scope prevents naming conflicts between different functions.

Example1:

```
int globalVar = 100; // Global variable with global scope

void exampleFunction(int param) { // 'param' has function scope
within exampleFunction

    static int staticVar = 5; // 'staticVar' has function scope
and persists between calls

    // Code here

}
```

Example2:

```
#include <stdio.h>

void exampleFunction() {

    int x = 10; // Variable 'x' with function scope within
exampleFunction

    printf("Inside exampleFunction: x = %d\n", x);

}

int main() {

    exampleFunction(); // Calling the function

    // 'x' is not accessible here, as it is within the scope of
exampleFunction

    // Uncommenting the next line will result in a compilation
error

    // printf("Outside exampleFunction: x = %d\n", x);

    return 0;

}
```

Output:

```
Inside exampleFunction: x = 10
```

[illegible]

In this example, the variable x is declared within the exampleFunction() with function scope. It is accessible and usable only within the exampleFunction(). If we try to access x outside the function (in the main function), it will result in a compilation error.

Function scope ensures that variables within functions are local to that function, and any modifications to those variables stay confined within the function. This helps in writing more maintainable and organized code by limiting the visibility and usage of variables to where they are genuinely needed.

Global Scope:

Global scope refers to the visibility and accessibility of variables declared outside any function or block in a programming language. Variables with global scope can be accessed from any part of the program, including all functions. They have a lifetime that spans the entire execution of the program, and their memory is allocated at the beginning of the program and deallocated only when the program terminates.

Key Points about Global Scope:

- **Visibility**: Variables declared at the global level are visible and accessible from any function in the program. They have a global reach and can be used across different functions.
- **Lifetime:** The lifetime of variables with global scope is the entire duration of the program's execution. They are created when the program starts and exist until the program terminates.
- **Accessibility**: Since global variables can be accessed from any function, they allow for the sharing of data between different parts of the program. However, this also means that any function can modify the value of a global variable, which can lead to potential side effects and difficulties in debugging and maintenance.

- **No Reusability:** Unlike variables with function or block scope, global variables are not tied to specific functions or blocks. As such, they lack the modularity and reusability benefits provided by local variables.

Example:

```
#include <stdio.h>

// Global variable with global scope

int globalVar = 100;

void exampleFunction() {

    // Access the global variable within exampleFunction

    printf("Inside exampleFunction: globalVar = %d\n",
globalVar);

}

int main() {

    // Access the global variable within the main function

    printf("Inside main: globalVar = %d\n", globalVar);

    // Modify the global variable from the main function

    globalVar = 200;

    // Call the exampleFunction, and it will see the
updated value of globalVar

    exampleFunction();

    return 0;

}
```

Output:

```
Inside main: globalVar = 100
```

[illegible]

For Vivekananda Global University, Jaipur

```
Inside exampleFunction: globalVar = 200
```

In this example, we declare a global variable globalVar outside any function, giving it global scope. The variable is accessible from both the main function and the exampleFunction. When we modify the value of globalVar inside the main function, the updated value is seen within the exampleFunction when it is called later.

Global variables are powerful tools that can be used to share data across different parts of a program. However, excessive use of global variables can lead to code that is difficult to maintain, understand, and debug. It is generally recommended to use global variables judiciously and consider other alternatives such as passing parameters and returning values from functions for better code organization and encapsulation.

Variables declared outside any function or block have global scope.

Global variables can be accessed from any part of the program, including all functions.

Global scope allows sharing data between different parts of the program but requires careful consideration to avoid unintended side effects.

```
Example :

int globalVar = 100; // Global variable with global scope

void exampleFunction() {

    // 'globalVar' is accessible here

}
```

Understanding scoping is essential for organizing code and ensuring that variables are used in the intended manner. By properly defining the scope of variables, programmers can avoid naming conflicts, improve code readability, and manage data effectively throughout their programs. Scoping plays a significant role in maintaining data encapsulation and

[illegible]

preventing accidental modification of variables in unrelated parts of the code, leading to more robust and maintainable software.

11.7 Summary

Storage Classes:

Storage classes in C and C++ define how variables are allocated, accessed, and managed in memory. There are four primary storage classes:

a. **Automatic:** The default storage class for variables declared within functions or blocks. They have block scope and are allocated on the stack. They are created when the block is entered and destroyed when the block is exited.

b. **Register:** Suggests to the compiler that a variable should be stored in a CPU register for faster access. However, compilers can choose to ignore this suggestion based on optimization decisions and hardware architecture limitations.

c. **External:** Used to declare global variables that can be accessed across multiple source files in a program. They have global scope and are defined in one source file and declared as extern in other source files where they are used.

d. **Static:** Allows for variables to have a longer lifetime than automatic variables. Variables with static storage class retain their values across different function calls and can have function scope or global scope.

Scope of a Variable:

Scope refers to the region of a program where a variable is visible and accessible.

Function Scope: Variables with function scope are visible and accessible only within the function where they are declared. They have a limited lifetime, existing only during the function's execution.

Block Scope: Variables with block scope are visible and accessible only within the block (enclosed by curly braces) where they are declared. They have a limited lifetime, existing only during the block's execution.

[illegible]

For Vivekananda Global University, Jaipur

Global Scope: Variables with global scope are visible and accessible from any part of the program, including all functions. They have a lifetime that spans the entire program's execution, existing from the program's start until termination.

Understanding storage classes and scoping is crucial for writing efficient, maintainable, and modular code. Careful consideration of when and where to use each storage class and understanding the visibility and lifetime of variables can lead to better software design and programming practices.

11.8 Review Questions

1. What is the default storage class for variables declared within functions or blocks in C and C++?
2. Which storage class suggests to the compiler that a variable should be stored in a CPU register?
3. What is the primary purpose of the "extern" storage class specifier?
4. What is the lifetime of variables with static storage class?
5. In which scope are variables visible and accessible only within the block they are declared?
6. Which storage class has limited use in modern programming due to advanced compiler optimizations?
7. How do you declare an external variable in C or C++?
8. What does variable shadowing mean?
9. Which storage class is used for variables with longer lifetimes and retain their values across function calls?
10. What is the scope of a variable declared at the global level?

11.9 Keywords

Automatic: The default storage class for local variables declared within functions or blocks, with a limited lifetime and accessible only within the block of declaration.

Register: Suggests to the compiler that a variable should be stored in a CPU register for faster access, but its use is limited due to modern compiler optimizations.

[illegible]

External: Used to declare global variables with global scope, allowing data sharing across different source files.

Static: Storage class for variables with a longer lifetime, retaining their values across different function calls, can have function scope or global scope.

Scope: The region of a program where a variable is visible and accessible, which can be function scope, block scope, or global scope.

Remember that understanding of these keywords and concepts is crucial for writing efficient and well-organized code in C and C++.

11.10 References

1. E.Balaguruswamy, "Computing Fundamentals & C Programming", TataMcGraw Hill, 2008.
2. B. A. Forouzan & R. F. Gilberg "Computer Science – A structured programming Approach Using C", 2011
3. E. Balaguruswamy, "Programming in ANSI" Tata McGraw Hill, 2011.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Table of Content

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Learning Objectives

After studying this unit, the student will be able to:

- Learn the fundamental concepts of File Input/Output and its importance in computer programming for data storage and retrieval.
- Explore the standard C library functions for File I/O, such as "open()", "fclose()", "fread()", "fwrite()", "fgets()", and "fprintf()", and understand how to use them effectively.
- Learn how to read data from text and binary files using "fread()" and "fgets()", and grasp the techniques to process and manipulate the data.
- Discover how to write data to files, both in binary and text formats, using "fwrite()" and "fprintf()", and learn to format and organize the data.
- Understand the importance of proper error handling in File I/O operations to ensure robustness and prevent potential data loss or program crashes.
- Gain proficiency in using file pointers to access and navigate through files during input and output operations.
- Understand the concept of command-line arguments and how they allow users to customize program behavior during runtime.
- Learn how to access and process command-line arguments in the "main()" function using parameters "argc" and "argv".
- Learn to handle scenarios where users may not provide the correct number of arguments or provide invalid arguments.
- Grasp the ability to create interactive programs that take user inputs from the command line for versatile and flexible execution.

Introduction

File Input/Output (I/O) and command-line arguments are essential concepts in C programming that enable interaction with files and provide flexibility in executing programs with user-provided inputs.

File Input/Output (I/O):

File I/O refers to the process of reading data from files or writing data to files in a C program. Files serve as a means of storing data persistently on a computer's storage medium. C provides built-in functions, such as "fopen()", "fread()", "fwrite()", "fgets()", and "fclose()", to perform file I/O operations.

To read data from a file, the "fopen()" function is used to open the file in various modes, such as read mode ("r") or binary read mode ("rb"). After reading data from the file using functions like "fread()" or "fgets()", the file should be closed using the "fclose()" function to release resources and ensure data integrity.

Similarly, to write data to a file, the "fopen()" function is used in write mode ("w") or append mode ("a"). Data can be written to the file using functions like "fwrite()" or "fprintf()".

Command-line Arguments:

Command-line arguments allow users to pass input to a C program when it is executed in the command-line environment. These arguments are provided after the program name when running the executable.

In C, the "main()" function can accept two parameters: "argc" (argument count) and "argv" (argument vector). "argc" represents the number of command-line arguments, and "argv" is an array of character pointers (strings) containing the actual arguments.

[illegible]

For example, if a program named "my_program" is executed with the following command:

```
my_program arg1 arg2 arg3
```

Then, "argc" will be 4 (including the program name itself), and "argv" will be an array containing {"my_program", "arg1", "arg2", "arg3"}.

Command-line arguments offer a way to customize program behavior and input data without modifying the source code. They are useful for scripting, batch processing, and automation tasks.

File I/O and command-line arguments provide powerful capabilities for C programs to interact with external data and adapt to different scenarios based on user inputs. Understanding these concepts is crucial for developing versatile and user-friendly C applications.

12.1 File input/output:

File Input/Output (I/O) is a fundamental concept in programming that deals with reading data from and writing data to files on disk or other storage devices. It allows programs to interact with external files, enabling data persistence and sharing between different sessions of the same program or between different programs.

File I/O is commonly used for tasks such as:

- **Reading Input Data:** Programs often need to read input data from files. For example, a text processing program might read data from a text file, or an image processing application might read pixel data from an image file.
- **Writing Output Data:** Programs generate results that need to be saved for future reference or use. File I/O allows these results to be written to files for permanent storage or sharing with other users or applications.
- **Configuration:** Programs can read configuration settings from files during startup to determine how the program should behave.

[illegible]

For Vivekananda Global University, Jaipur

- **Data Persistence:** Files provide a way to store data permanently, allowing it to be accessed and utilized across different sessions of the program or by other programs.
- In most programming languages, File I/O involves the following basic operations:
- **Opening a File:** Before reading from or writing to a file, it needs to be opened by the program. Opening a file establishes a connection between the program and the file, enabling data transfer.
- **Reading from a File:** Once a file is opened for reading, the program can read data from it. Data can be read in different formats, depending on the type of data stored in the file (e.g., text, binary).
- **Writing to a File:** When a file is opened for writing, the program can write data to it. Similar to reading, data can be written in various formats.
- **Closing a File:** After the program finishes reading from or writing to the file, it should be closed to release system resources and ensure data is properly saved.

It's important to handle File I/O operations carefully, as errors can occur (e.g., file not found, permission issues). Proper error handling and resource management are crucial to ensure the program behaves correctly and does not cause unexpected issues.

The specifics of File I/O can vary depending on the programming language being used. Different languages provide different sets of functions or libraries to facilitate File I/O operations. Commonly used programming languages like Python, C, C++, Java, and others have built-in or third-party libraries to handle File I/O efficiently. The lifetime of automatic variables is limited to the duration of their enclosing block.

Function Name	Description
fopen()	create a new file or open a existing file
fclose()	closes a file
getc()	reads a character from a file

For Vivekananda Global University, Jaipur

[illegible]

Registrar

putc()	writes a character to a file
fscanf()	reads a set of data from a file
fprintf()	writes a set of data to a file
getw()	reads a integer from a file
putw()	writes a integer to a file
fseek()	set the position to desire point
ftell()	gives current position in the file
rewind()	set the position to the beginning point

12.2 Open a File or Creating a file:

To open a file in C, you use the fopen function. If the file exists, it will be opened, and if it doesn't exist, a new file with the specified name will be created. The mode you specify in the fopen function determines whether the file will be opened for reading, writing, or both.

Here's the syntax for fopen:

```
FILE *fopen(const char *filename, const char *mode);
```

Parameters:

filename: The name of the file you want to open or create, represented as a string.

mode: A string representing the mode in which the file should be opened. The mode determines whether the file will be opened for reading, writing, or both, as well as whether the file will be treated as a binary or text file.

Commonly used modes:

"r": Read mode. The file is opened for reading, and the file pointer is positioned at the beginning of the file.

"w": Write mode. The file is opened for writing; if it already exists, its contents will be truncated (cleared). If the file does not exist, a new empty file will be created.

"a": Append mode. The file is opened for writing, and data is written at the end of the file. If the file doesn't exist, a new empty file is created.

[illegible]

"rb", "wb", "ab": Binary modes. These modes are used for reading or writing binary files (e.g., image files, audio files).

Here's an example of opening a file in write mode to create a new file and write some content to it:

```
    #include <stdio.h>

    int main() {

        FILE *file;

        // Open the file in write mode

        file = fopen("example.txt", "w");


        if (file != NULL) {

            // Write data to the file

            fprintf(file, "This is a new file created using
    C.\n");

            fputs("Another line in the file.\n", file);

            // Close the file

            fclose(file);

            printf("File    created    and    data    written
    successfully.\n");

        } else {

            printf("Error opening the file.\n");

        }

        return 0;

}
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

The program opens the file "example.txt" in write mode in this example. If the file exists, its contents will be cleared, and if it doesn't exist, a new file will be created. The program then writes two lines of text to the file using fprintf and fputs functions.

Always check the return value of fopen to handle any potential errors that may occur during the file opening process.

12.3 Closing a File

After you have finished reading from or writing to a file, it's essential to close the file properly using the fclose function. Closing a file releases system resources associated with it and ensures that any buffered data is written to the file before it closes. Failing to close a file can lead to resource leaks and other issues in your program.

Here's the syntax for the fclose function:

```
int fclose(FILE *stream);
```

Parameters:

stream: The file pointer that represents the file you want to close. It should be the same pointer that was returned by the fopen function when the file was opened.

The return value of fclose is an integer, which is zero on success and EOF (a macro representing an end-of-file indicator) on failure.

Here's an example of how to close a file properly in C:

```
    #include <stdio.h>

    int main() {

        FILE *file;


        // Open the file in write mode
```

[illegible]

For Vivekananda Global University, Jaipur

```
    file = fopen("example.txt", "w");

    if (file != NULL) {

        // Write data to the file

        fprintf(file, "This is a new file created using
C.\n");

        fputs("Another line in the file.\n", file);

        // Close the file

        if (fclose(file) == 0) {

            printf("File closed successfully.\n");

        } else {

            printf("Error closing the file.\n");

        }

    } else {

        printf("Error opening the file.\n");

    }

    return 0;

}
```

In this example, the fclose function is called to close the file after writing data to it. The return value of fclose is checked to determine if the file was closed successfully.

Always make sure to close files when you are done with them to free up resources and ensure the integrity of the data written to the file. It's good practice to close files explicitly, even though most modern operating systems will automatically close files when the program terminates. However, relying on the system to close files can lead to resource leaks if your program handles a large number of files during its execution.

12.4 Read a character from a file

To read a character from a file in C, you can use the fgetc function. This function reads a single character from the specified file and advances the file pointer to the next character position.

Here's the syntax for fgetc:

int fgetc(FILE *stream);

Parameters:

stream: The file pointer that represents the file from which you want to read the character. It should be the same pointer that was returned by the fopen function when the file was opened for reading.

The fgetc function returns the character read as an integer value. If successful, it returns the ASCII value of the character. If it reaches the end of the file or encounters an error, it returns EOF (a macro representing an end-of-file indicator).

Here's an example of how to read characters from a file in C:

```
#include <stdio.h>

int main() {

    FILE *file;

    int ch;

    // Open the file in read mode

    file = fopen("example.txt", "r");

    if (file != NULL) {

        // Read characters until the end of the file is
reached

        while ((ch = fgetc(file)) != EOF) {

            // Print the character or process it as needed
```

[illegible]

For Vivekananda Global University, Jaipur

```
                putchar(ch);
            }
            // Close the file
            fclose(file);
        } else {
            printf("Error opening the file.\n");
        }
        return 0;
    }
```

In this example, the program opens the file "example.txt" in read mode. It then uses a while loop to read characters from the file using fgetc until the end of the file is reached (indicated by EOF). The characters are printed to the console using putchar, but you can process or store them in any way you need for your specific application.

Always remember to check for errors and handle them appropriately while opening and reading files. Additionally, don't forget to close the file using fclose once you have finished reading from it to free up resources.

12.5 Write a character to a file

To write a character to a file in C, you can use the fputc function. This function writes a single character to the specified file at the current file position. If the file doesn't exist, it will be created. If the file exists and is opened in write mode, its contents will be truncated (cleared).

Here's the syntax for fputc:

```
int fputc(int character, FILE *stream);
```

Parameters:

[illegible]

character: The character you want to write to the file. It is of type int, but it should represent the ASCII value of the character you want to write.

stream: The file pointer that represents the file to which you want to write the character. It should be the same pointer that was returned by the fopen function when the file was opened for writing.

The fputc function returns the character written as an integer value (the character itself) on success. If there's an error during writing, it returns EOF.

Here's an example of how to write a character to a file in C:

```
#include <stdio.h>

int main() {

    FILE *file;

    int ch;

    // Open the file in write mode

    file = fopen("output.txt", "w");

    if (file != NULL) {

        // Write a character to the file

        ch = 'A';  // Character to write (ASCII value of
'A' is 65)

        if (fputc(ch, file) != EOF) {

            printf("Character   written   to   the   file
successfully.\n");

        } else {

            printf("Error writing to the file.\n");

        }
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
        // Close the file
        fclose(file);
    } else {
        printf("Error opening the file.\n");
    }
    return 0;
}
```

In this example, the program opens the file "output.txt" in write mode and writes the character 'A' to the file using fputc. The ASCII value of 'A' is 65, so you can directly use the character literal in the function call.

12.6 Reads a set of data from a file

To read a set of data from a file in C, you need to determine the format and structure of the data in the file. Depending on the type of data and how it is organized in the file, you can use different techniques for reading the data.

Here's a general approach to read a set of data from a file:

- **Open the File:** Open the file in read mode using fopen. Make sure to check if the file was opened successfully.
- **Read the Data:** Depending on the structure of the data, you can use various functions like fscanf, fgets, or fread to read the data from the file.
- **fscanf:** If the data is formatted and consists of values separated by spaces or other delimiters, fscanf can be used to read the data in a formatted manner.
- **fgets:** If the data is organized as lines of text, you can use fgets to read each line of data.
- **fread:** If the data is binary and not in human-readable format, you can use fread to read binary data directly.

[illegible]
For Vivekananda Global University, Jaipur
Registrar

- **Process the Data:** After reading the data, you can process it as needed. For example, you might store it in variables, perform calculations, or use it for further operations.
- **Close the File:** Once you are done reading and processing the data, close the file using fclose to release resources associated with the file.

Here's an example demonstrating how to read a set of integers from a file:

Suppose the file "data.txt" contains the following data:

10

20

30

40

50

```
#include <stdio.h>

int main() {

    FILE *file;

    int data;

       // Open the file in read mode

    file = fopen("data.txt", "r");

    if (file != NULL) {

        // Read and process the data

        while (fscanf(file, "%d", &data) != EOF) {

            // Process the data (e.g., print it)

            printf("%d ", data);

        }
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
        // Close the file
        fclose(file);
    } else {
        printf("Error opening the file.\n");
    }
    return 0;
}
```

In this example, the program opens the file "data.txt" in read mode and reads integers from the file using fscanf. The integers are then processed (printed to the console in this case) inside the loop. The loop continues until fscanf returns EOF, which indicates the end of the file.

Remember to handle errors and close the file properly to ensure the correct behavior of your program. Additionally, adapt the code based on the specific format and organization of the data in your file.

12.7 writes a set of data to a file

To write a set of data to a file in C, you can use different functions depending on the type of data you want to write. Here's a general approach to write a set of data to a file:

- **Open the File:** Open the file in write mode using fopen. If the file does not exist, it will be created. If it already exists, its contents will be truncated (cleared) unless you use append mode.
- **Write the Data:** Depending on the type of data you want to write, you can use various functions like fprintf, fputs, or fwrite.

[illegible]

- **fprintf:** If you want to write formatted data to the file (e.g., integers, floating-point numbers, strings), use fprintf.
- **fputs:** If you want to write a string or a line of text to the file, use fputs.
- **fwrite:** If you want to write binary data to the file, use fwrite.
- **Close the File:** Once you are done writing the data, close the file using fclose to save the changes and release resources associated with the file.

Here's an example of how to write a set of integers to a file:

```
#include <stdio.h>

int main() {

    FILE *file;

    int data[] = {10, 20, 30, 40, 50};

    int i;

    // Open the file in write mode

    file = fopen("data.txt", "w");


    if (file != NULL) {

        // Write the data to the file

        for (i = 0; i < 5; i++) {

            fprintf(file, "%d\n", data[i]);

        }

        // Close the file

        fclose(file);

        printf("Data written to the file successfully.\n");

    } else {

        printf("Error opening the file.\n");
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

```
    }
    return 0;
}
```

In this example, the program opens the file "data.txt" in write mode and writes the integers from the array data to the file using fprintf. Each integer is written on a new line using the newline character '\n'.

Remember to handle errors and close the file properly after you have finished writing the data. The fclose function will save the changes and make them permanent in the file. Adapt the code based on the specific format and organization of the data you want to write.

12.8 Reads a integer from a file

To read an integer from a file in C, you can use the fscanf function. fscanf reads formatted data from a file and allows you to extract specific types of data, such as integers, floating-point numbers, or characters, from the file.

Here's the syntax for fscanf to read an integer from a file:

```
int fscanf(FILE *stream, const char *format, ...);
```

Parameters:

stream: The file pointer that represents the file from which you want to read the integer. It should be the same pointer that was returned by the fopen function when the file was opened for reading.

format: A format string that specifies the type of data to read. For reading an integer, you use %d in the format string.

The fscanf function returns the number of items successfully read from the file. If it fails to read the expected number of items or encounters an error, it returns a value less than the number of expected items or EOF.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Here's an example of how to read an integer from a file:

```
#include <stdio.h>

int main() {

    FILE *file;

    int number;

    // Open the file in read mode

    file = fopen("data.txt", "r");


    if (file != NULL) {

        // Read the integer from the file

        if (fscanf(file, "%d", &number) == 1) {

            printf("The integer read from the file is:
%d\n", number);

        } else {

            printf("Error reading the integer from the
file.\n");

                // Close the file

        fclose(file);

    } else {

        printf("Error opening the file.\n");

    }

    return 0;

}
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

In this example, the program opens the file "data.txt" in read mode and reads the integer from the file using fscanf with the format %d. The integer is then stored in the variable number, and it is printed to the console.

Remember to check for errors during file opening and reading, and handle them appropriately. Also, ensure that you close the file using fclose after you have finished reading from it to free up resources.

12.9 Writes a integer to a file

To write an integer to a file in C, you can use the fprintf function. fprintf allows you to write formatted data to a file, including integers, floating-point numbers, strings, and more.

Here's the syntax for fprintf to write an integer to a file:

```
int fprintf(FILE *stream, const char *format, ...);
```

Parameters:

stream: The file pointer that represents the file to which you want to write the integer. It should be the same pointer that was returned by the fopen function when the file was opened for writing.

format: A format string that specifies how the data should be formatted in the file. For writing an integer, you use %d in the format string.

The fprintf function returns the number of characters written to the file. If it encounters an error, it returns a negative value.

Here's an example of how to write an integer to a file:

```
#include <stdio.h>

int main() {

    FILE *file;

    int number = 42;
```

[illegible]

```
    // Open the file in write mode
    file = fopen("output.txt", "w");
    if (file != NULL) {
        // Write the integer to the file
        if (fprintf(file, "%d", number) > 0) {
            printf("Integer written to the file 
successfully.\n");
        } else {
            printf("Error writing the integer to the 
file.\n");
        }
        // Close the file
        fclose(file);
    } else {
        printf("Error opening the file.\n");
    }
    return 0;
}
```

In this example, the program opens the file "output.txt" in write mode and writes the integer 42 to the file using fprintf with the format %d.

Remember to check for errors during file opening and writing, and handle them appropriately. Also, ensure that you close the file using fclose after you have finished writing to it to save the changes and free up resources.

[illegible]

12.10 Set the position to desire point

To set the position in a file to a desired point, you can use the fseek function in C. The fseek function allows you to move the file pointer to a specific position within the file, allowing you to read or write data from that position.

Here's the syntax for fseek:

```
int fseek(FILE *stream, long offset, int origin);
```

Parameters:

- **stream:** The file pointer that represents the file where you want to set the position.
- **offset:** The number of bytes to offset the file pointer from the specified origin.
- origin: The position from where the offset is calculated. It can take one of the following values:
- **SEEK_SET (0):** The offset is relative to the beginning of the file.
- **SEEK_CUR (1):** The offset is relative to the current position of the file pointer.
- **SEEK_END (2):** The offset is relative to the end of the file.
- The fseek function returns 0 on success and a non-zero value on failure.

Here's an example of how to use fseek to set the position in a file:

Suppose the file "data.txt" contains the following data:

```
Hello, this is a text file.

    #include <stdio.h>

    int main() {

        FILE *file;

        char buffer[100];

        // Open the file in read mode

        file = fopen("data.txt", "r");
```

[illegible]

For Vivekananda Global University, Jaipur

```
    if (file != NULL) {

        // Set the position to the 7th byte from the
beginning of the file

        fseek(file, 6, SEEK_SET);

        // Read data from the new position

        fgets(buffer, sizeof(buffer), file);

        // Close the file

        fclose(file);

        // Print the data read from the file

        printf("Data read from the file: %s\n", buffer);

    } else {

        printf("Error opening the file.\n");

    }

    return 0;

}
```

In this example, the program opens the file "data.txt" in read mode and sets the file position to the 7th byte from the beginning of the file using fseek. It then reads the data from the new position using fgets and stores it in the buffer. The program then prints the data read from the file. Remember to check for errors during file opening and seeking, and handle them appropriately. Additionally, ensure that you close the file using fclose after you have finished reading from it to free up resources.

12.11 Gives current position in the file

To obtain the current position in the file, you can use the ftell function in C. The ftell function returns the current file position indicator's offset from the beginning of the file.

[illegible]

Here's the syntax for ftell:

```
long ftell(FILE *stream);
```

Parameters:

stream: The file pointer that represents the file whose current position you want to determine.

The ftell function returns the current position as a long integer value representing the offset in bytes from the beginning of the file. If an error occurs, it returns -1L.

Here's an example of how to use ftell to get the current position in a file:

```
#include <stdio.h>

int main() {

    FILE *file;

    long position;

    // Open the file in read mode

    file = fopen("data.txt", "r");

    if (file != NULL) {

        // Get the current position in the file

        position = ftell(file);

       if (position != -1L) {

            printf("Current position in the file: %ld
bytes\n", position);

        } else {

            printf("Error getting the current
position.\n");
```

[illegible]

```
            }

        // Close the file
        fclose(file);
    } else {
        printf("Error opening the file.\n");
    }
    return 0;
}
```

In this example, the program opens the file "data.txt" in read mode and uses ftell to get the current position in the file. The position is then printed to the console.

Remember to check for errors during file opening and when calling ftell, and handle them appropriately. Also, ensure that you close the file using fclose after you have finished working with it to free up resources.

12.12 Set the position to the begining point

The rewind function is another way to set the file position to the beginning of the file. It is specifically designed to move the file position indicator to the start of the file.

Here's the syntax for rewind:

```
void rewind(FILE *stream);
```

Parameters:

stream: The file pointer that represents the file whose position you want to set to the beginning.

The rewind function does not return any value (void).

[illegible]
For Vivekananda Global University, Jaipur

Here's an example of how to use rewind to set the file position to the beginning of the file:

```
#include <stdio.h>

int main() {

    FILE *file;

    char buffer[100];

    // Open the file in read mode

    file = fopen("data.txt", "r");

    if (file != NULL) {

        // Set the position to the beginning of the
file

        rewind(file);

        // Read data from the beginning of the file

        fgets(buffer, sizeof(buffer), file);

        // Close the file

        fclose(file);

        // Print the data read from the file

        printf("Data read from the file: %s\n",
buffer);

    } else {

        printf("Error opening the file.\n");

    }

    return 0;

}
```

[illegible]

In this example, the program opens the file "data.txt" in read mode and uses rewind to set the file position to the beginning of the file. It then reads the data from the beginning of the file using fgets and stores it in the buffer. The program then prints the data read from the file.

Both fseek and rewind can be used to set the file position to the beginning of the file. The difference is that fseek allows you to set the file position to any desired offset, while rewind specifically sets it to the beginning. Use the one that best suits your needs in your program.

12.13 Command-line arguments

Command-line arguments are parameters that are passed to a program when it is executed from the command line or terminal. These arguments provide input or configuration options to the program at runtime, allowing it to behave differently based on the provided values.

In C, you can access command-line arguments through the main function. The main function can take two arguments:

argc (argument count): An integer that represents the number of command-line arguments passed to the program. It includes the name of the program itself as the first argument.

argv (argument vector): An array of strings that contains the actual command-line arguments passed to the program. Each element of the array represents a command-line argument, and the first element (argv[0]) is the name of the program itself.

Here's the standard declaration of the main function in C:

```
int main(int argc, char *argv[])
```

Let's see an example of a simple C program that takes two integers as command-line arguments and prints their sum:

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

[illegible]

For Vivekananda Global University, Jaipur

```
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <num1> <num2>\n", argv[0]);
        return 1;
    }
    int num1 = atoi(argv[1]); // Convert the first argument
to an integer
    int num2 = atoi(argv[2]); // Convert the second
argument to an integer
    int sum = num1 + num2;
    printf("Sum: %d\n", sum);
    return 0;
}
```

In this example, the program checks if exactly two command-line arguments (in addition to the program name) are provided. If not, it prints a usage message and exits with a non-zero return code. If two arguments are provided, it converts them to integers using atoi (ASCII to Integer) function, calculates their sum, and prints the result.

When you run the program from the command line, you would provide the two integers as arguments, like this:

```
./program_name 10 20
```

This would output:

```
Sum: 30
```

[illegible]

For Vivekananda Global University, Jaipur

Registrar

Command-line arguments are a powerful way to provide inputs and configure programs without requiring user interaction or modifying the source code. They are commonly used to pass data to scripts, configure settings, or control the behavior of command-line utilities.

12.14 Summary

File Input/Output (I/O) and command-line arguments are essential concepts in C programming for interacting with files and passing inputs to programs.

File I/O:

- File I/O allows reading data from files (input) or writing data to files (output).
- The FILE data type represents a file stream, and fopen is used to open a file and get a file pointer.
- For reading from a file, fscanf, fgets, or fgetc functions are used to read formatted data, lines, or characters respectively.
- For writing to a file, fprintf, fputs, or fputc functions are used to write formatted data, strings, or characters respectively.
- Always close files after use with fclose.

Command-Line Arguments:

- Command-line arguments are inputs provided to a program when it is executed from the terminal.
- The main function in C can take argc (argument count) and argv (argument vector) as parameters.
- argc holds the number of command-line arguments (including the program name), and argv is an array of strings containing the arguments.
- Command-line arguments are passed as strings, so conversions like atoi are needed to use them as numbers.
- Command-line arguments are useful for providing configuration, data, or options to a program without modifying the source code.

[illegible]

Combining File I/O and Command-Line Arguments:

- You can read input data from files specified as command-line arguments, process the data, and write the results to another file.
- Command-line arguments allow for more flexibility and automation, enabling users to customize program behavior easily.

Overall, mastering file I/O and understanding command-line arguments can greatly enhance the functionality and versatility of your C programs.

12.15 Review Questions

1. What is File Input/Output (I/O) in C, and what are its key components?
2. How do you open a file in C for reading or writing, and what is the purpose of the FILE data type?
3. Explain the role of the fscanf, fgets, and fgetc functions in reading data from a file. Provide an example for each.
4. Describe the purpose of the fprintf, fputs, and fputc functions in writing data to a file. Provide an example for each.
5. Why is it essential to close a file using fclose after reading from or writing to it?
6. What are command-line arguments, and how are they passed to a C program?
7. In the main function, what do argc and argv represent, and how can you access the command-line arguments in C?
8. How can you use command-line arguments to customize the behavior of a C program?
9. Provide an example of a C program that reads integers from a file specified as a command-line argument and calculates their sum.
10. How would you modify the previous program to write the sum to a different file, also specified as a command-line argument?

12.16 Keywords

- File: A sequence of bytes on the disk used for permanent storage of data.
- fopen: A function used to open a file, returning a file pointer.
- FILE: A data type representing a file stream.
- fscanf: A function used to read formatted data from a file.
- fgets: A function used to read a line of text from a file.
- fgetc: A function used to read a single character from a file.
- fprintf: A function used to write formatted data to a file.
- fputs: A function used to write a string to a file.
- fputc: A function used to write a single character to a file.
- fclose: A function used to close a file after reading from or writing to it.
- argc: An integer representing the number of command-line arguments passed to a C program.
- argv: An array of strings containing the command-line arguments.
- main: The entry point function in C, accepting argc and argv as parameters.
- atoi: A function used to convert a string to an integer.
- atof: A function used to convert a string to a floating-point number.
- Usage: A message displayed to users when command-line arguments are incorrect or insufficient.
- SEEK_SET: A constant used with fseek to set the file position relative to the beginning of the file.
- SEEK_CUR: A constant used with fseek to set the file position relative to the current position.
- SEEK_END: A constant used with fseek to set the file position relative to the end of the file.
- rewind: A function used to set the file position to the beginning of the file.

12.17 References

1. E.Balaguruswamy, "Computing Fundamentals & C Programming", TataMcGraw Hill, 2008.

[illegible]

For Vivekananda Global University, Jaipur

Registrar

2. B. A. Forouzan & R. F. Gilberg "Computer Science – A structured programming Approach Using C", 2011
3. E. Balaguruswamy, "Programming in ANSI" Tata McGraw Hill, 2011.

[illegible]

For Vivekananda Global University, Jaipur

Registrar