

Geethanjali College of Engineering and Technology

(Approved by AICTE, New Delhi and Affiliated to JNTU, Hyderabad)
Sy.No. 33& 34, Cheeryal (V), Keesara (M), Medchal District. (A.P.)-501301.

(UGC AUTONOMOUS)

Programming For Problem Solving-II

COURSE FILE

(2020-21)



Geethanjali

**DEPARTMENT OF
FRESHMAN ENGINEERING**

Course coordinator

Program Coordinator

HOD

S.No	Contents	Page No.
1	Cover Page	3
2	Syllabus copy	4
3	Vision of the Department	6
4	Mission of the Department	7
5	PEOs, POs and PSOs	9
6	Course objectives and outcomes	15
7	Brief notes on the importance of the course and how it fits into the curriculum	15
8	Prerequisites if any	15
9	Instructional Learning Outcomes	16
10	Course mapping with POs	18
11	Class Time Table	28
12	Individual Time Table	29
13	Lecture schedule with methodology being used/adopted	30
14	Detailed notes	34
15	Additional topics	110
16	University Question papers of previous years	111
17	Question Bank	112
18	Assignment Questions	117
19	Unit wise Quiz Questions and long answer questions	120
20	Tutorial problems	130
21	Known gaps ,if any and inclusion of the same in lecture schedule	131
22	Discussion topics , if any	132
23	References, Journals, websites and E-links if any	132
24	Quality Measurement Sheets a. Course End Survey b. Teaching Evaluation	132
25	Student List	133
26	Group-Wise students list for discussion topics	134

GEETHANJALI COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF FE

Name of the Subject : Programming For Problem Solving-II

Course Code: 20CS12001

Programme : UG

Branch: All Branches

Version No : 01

Year: I

Updated on : 1/03/2021

Semester: II SEM

No. of pages :

Classification status (Unrestricted / Restricted) Unrestricted

Distribution List :Faculty, students, library

Prepared by :

1) Name : S.Radha

1) Name : M. Ravinder

2) Design.: Asst. Prof

2) Design.: Asst. Prof

3) Sign :

3) Sign :

4) Date :

4) Date :

Verified by :

1) Name : Dr Puja.S.Prasad

2) Sign :

3) Design. : Assoc. Prof

4) Date :

*** For O.C Only.**

1) Name :

2) Sign :

3) Design :

4) Date :

Approved by : (HOD)

1)Name : Dr. A. Sreelakshmi

2) Sign :

3) Date :

2. SYLLABUS

Code: 20CS12001

Programming For Problem Solving-II
1st year Semester-II

L-2, C-2

UNIT – I

Strings – Concepts, C Strings, String Input / Output functions, string manipulation functions, arrays of strings, string / data conversion, C program examples.

Enumerated Types–The Type Definition (typedef), Enumerated types

Structure and Union Types – Declaration, initialization, accessing structures, operations on structures, Complex structures, Structures and functions, passing structures through pointers, self referential structures, unions, bit fields.

UNIT – II

Linear list - Singly linked list implementation, insertion, deletion and searching operations on linear list

UNIT – III

Sorting –selection sort, Quick sort, insertion sort techniques (Using Arrays)

Searching- linear search, binary search techniques (Using Arrays)

UNIT - IV

Stacks – Introduction, Principle, Operations: Push and Pop, In-fix to Post-Fix Conversion and Post-Fix evaluation. (Array implementation.)

Queues - Introduction, Principle, Operations: Enqueue and Dequeue. (Arrayimplementation.)

UNIT – V

File Input and Output – Concept of a file, text files and binary files, Differences between text and binary files, State of a file, Opening and Closing files, file input / output functions (standard library input / output functions for files), file status functions (error handling), Positioning functions.

Command line arguments

Program Development – Multi-source files, Separate Compilation of functions

TEXT BOOKS:

1. Computer Science: A Structured Programming Approach Using C, B.A.Forouzan and R.F. Gilberg, Third Edition, Cengage Learning.
-

REFERENCE BOOKS:

1. The C Programming Language, B.W. Kernighan and Dennis M.Ritchie, PHI.
2. Programming in C. P. Dey and M Ghosh , Oxford University Press.
3. Programming with C, B.Gottfried, 3rd edition, Schaum's outlines, TMH.
4. Problem Solving and Program Design in C, J.R. Hanly and E.B. Koffman, 7th Edition, Pearson education.
5. C & Data structures – P. Padmanabham, 3rd Edition, BS. Publications.

**VISION OF THE DEPARTMENT
CSE & CSE (AI&ML,DS,CS,IOT)**

To produce globally competent and socially responsible computer science engineers contributing to the advancement of engineering and technology which involves creativity and innovation by providing excellent learning environment with world class facilities.

VISION OF THE DEPARTMENT - IT

The department of Information Technology endeavors to bring out technically competent, socially responsible technocrats through continuous improvement in teaching learning processes and innovative research practices.

VISION OF THE DEPARTMENT – ECE

To impart quality technical education in Electronics and Communication Engineering emphasizing analysis, design/synthesis and evaluation of hardware/ embedded software, using various Electronic Design Automation (EDA) tools with accent on creativity, innovation and research thereby producing competent engineers who can meet global challenges with societal commitment.

VISION OF THE DEPARTMENT – EEE

To provide excellent Electrical and Electronics education by building strong teaching and research environment

VISION OF THE DEPARTMENT – CE

To provide competent and qualitative professionals with innovative ideas in Civil Engineering.

VISION OF THE DEPARTMENT – ME

The Mechanical Engineering department strives to be recognized globally for outstanding education and research, imparting quality education, churning well-qualified engineers, who are creative, innovative, and entrepreneurial, solving problems for societal development.

3. MISSION OF THE DEPARTMENT CSE & CSE (AI&ML,DS,CS,IOT)

1. To be a centre of excellence in instruction, innovation in research and scholarship, and service to the stake holders, the profession, and the public.
2. To prepare graduates to enter a rapidly changing field as a competent computer science engineer.
3. To prepare graduate capable in all phases of software development, possess a firm understanding of hardware technologies, have the strong mathematical background necessary for scientific computing, and be sufficiently well versed in general theory to allow growth within the discipline as it advances.
4. To prepare graduates to assume leadership roles by possessing good communication skills, the ability to work effectively as team members, and an appreciation for their social and ethical responsibility in a global setting.

MISSION OF THE DEPARTMENT - IT

1. Inculcate into the students, the technical and problem solving skills to ensure their success in their chosen profession.
2. Impart the essential skills like teamwork, lifelong learning etc. to the students which make them globally acceptable technocrats.
3. Facilitate the students with strong fundamentals in basic sciences, mathematics and Information Technology areas to keep pace with the growing challenges in field.
4. Enrich the faculty with knowledge in the frontiers of the Information Technology area.

MISSION OF THE DEPARTMENT – ECE

1. To impart quality education in fundamentals of basic sciences, mathematics, electronics and communication engineering through innovative teaching-learning processes.
2. To facilitate Graduates define, design, and solve engineering problems in the field of Electronics and Communication Engineering using various Electronic Design Automation (EDA) tools.
3. To encourage research culture among faculty and students thereby facilitating them to be creative and innovative through constant interaction with R & D organizations and Industry.

4. To inculcate teamwork, imbibe leadership qualities, professional ethics and social responsibilities in students and faculty.

MISSION OF THE DEPARTMENT – EEE

1. To offer high quality graduate program in Electrical and Electronics education and to prepare students for professional career or higher studies.
2. The department promotes excellence in teaching, research, collaborative activities and positive contributions to society

MISSION OF THE DEPARTMENT – CE

To offer the best quality education in under graduation and post graduation, research guidance, professional consultancy and manpower training as well as leadership in civil engineering.

MISSION OF THE DEPARTMENT – ME

1. Imparting quality education to students to enhance their skills and make them globally competitive.
2. Prepare graduates to engage in life-long learning, possess intellectual capabilities, serving society with a strong commitment to their profession, meeting technical challenges and exhibiting ethical responsibility for societal development.
3. Conduct quality research, create opportunities for students and faculty to showcase their talent, disseminate knowledge, and promote community development, leading to peace and harmony in society.

5. PEO's, PO's & PSO's

PROGRAM EDUCATIONAL OBJECTIVES (PEO's)

CSE & CSE (AI&ML,DS,CS,IOT)

1. To provide graduates with a good foundation in mathematics, sciences and engineering fundamentals required to solve engineering problems that will facilitate them to find employment in industry and / or to pursue postgraduate studies with an appreciation for lifelong learning.
2. To provide graduates with analytical and problem-solving skills to design algorithms, other hardware / software systems, and inculcate professional ethics, inter-personal skills to work in a multi-cultural team.
3. To facilitate graduates to get familiarized with the art software / hardware tools, imbibing creativity and innovation that would enable them to develop cutting-edge technologies of multi-disciplinary nature for societal development.

PROGRAM EDUCATIONAL OBJECTIVES (PEO's) – IT

1. Exhibit sound knowledge in the fundamentals of Information Technology and apply practical experience with programming techniques to solve real world problems.
2. To inculcate professional behavior with strong ethical values, leadership qualities, innovative thinking and analytical abilities into the student.
3. To empower the student with the qualities of effective communication, teamwork, continuous learning attitude, leadership and proficiency in cutting edge technologies needed for a successful Information Technology professional.
4. Imbibe sound knowledge in mathematics, basic sciences, first principles to form a strong base for the student to keep up with the growing challenges in the field of Information Technology.

PROGRAM EDUCATIONAL OBJECTIVES - ECE

1. To prepare students with excellent comprehension of basic sciences, mathematics and engineering subjects facilitating them to gain employment or pursue postgraduate studies with an appreciation for lifelong learning.
2. To train students with problem solving capabilities such as analysis and design with adequate practical skills that are Program Specific wherein they demonstrate creativity and innovation that would enable them to develop state of the art equipment and technologies of multidisciplinary nature for societal development.
3. To inculcate positive attitude, professional ethics, effective communication and interpersonal skills which would facilitate them to succeed in the chosen profession exhibiting creativity and innovation through research and development both as team member and as well as leader.

PROGRAM EDUCATIONAL OBJECTIVES – EEE

1. To prepare students with excellent comprehension of mathematics, Basic Sciences and Engineering subjects facilitating them to find gainful employment or pursue post graduate program with an appreciation for lifelong learning.
2. To inculcate problem solving capabilities in students with analysis, design and practical skills that are Program Specific which would facilitate them to exhibit creativity and innovation that would enable them to develop modern equipment with emerging technologies of multidisciplinary nature for societal development.
3. To inculcate positive attitude, professional ethics, effective communication and interpersonal skills which would facilitate them to succeed in the chosen profession through research and development both as team member and as well as leader.

PROGRAM EDUCATIONAL OBJECTIVES – CE

1. Provide a strong foundation in Mathematics, Basic Sciences and Engineering fundamentals to the students, enabling them to excel in the various careers in Civil Engineering.
2. Impart necessary theoretical and practical background in Civil Engineering to the students, so that they can effectively compete with their contemporaries in the National / International level.
3. Motivate and prepare the Graduates to pursue higher studies, Research and Development, thus contributing to the ever-increasing academic demands of the country.
4. Enrich the students with strong communication and interpersonal skills, broad knowledge and an understanding of multicultural and global perspectives, to work effectively in multidisciplinary teams, both as leaders and team members.

PROGRAM EDUCATIONAL OBJECTIVES – ME

1. To prepare students with strong fundamental knowledge in basic sciences mathematics, engineering courses which would facilitate them find gainful employment with a sense of appreciation to pursue life-long learning for professional development.
2. To inculcate problem solving skills in students, imbibing creativity and innovation which would enable them to develop modern machinery involving cutting edge technologies of multidisciplinary nature for societal development.
3. To develop critical thinking with an aptitude to conduct research and development, instill professional ethics, develop effective communication and interpersonal skills with positive attitude to contribute significantly towards their chosen profession, thereby supporting community development.

PROGRAM OUTCOMES (Common to all branches)

Engineering Graduates would be able to:

PO 1:Engineering knowledge: Apply the knowledge of mathematics, science, engineering

fundamentals, and an engineering specialization to the solution of complex engineering problems.

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

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

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

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

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

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

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

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

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

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

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

PROGRAM SPECIFIC OUTCOMES (PSO's) - (CSE)

PSO 1: Demonstrate competency in Programming and problem-solving skills and apply these skills in solving real world problems

PSO 2: Select appropriate programming languages, Data structures and algorithms in combination with modern technologies and tools, apply them in developing creative and innovative solutions

PSO 3: Demonstrate adequate knowledge in emerging technologies

PROGRAM SPECIFIC OUTCOMES (PSO's) - (CSE-AI&ML)

PSO 1: Demonstrate competency in Programming and problem-solving skills and apply those skills in solving computing problems

PSO 2: Select appropriate programming languages, Data structures and algorithms in combination with modern technologies and apply them in developing innovative solutions

PSO 3: Demonstrate adequate knowledge in the concepts and techniques of artificial intelligence and machine learning, apply them in developing intelligent systems to solve real world problems

PROGRAM SPECIFIC OUTCOMES (PSO's) - (CSE-DS)

PSO 1: Demonstrate competency in Programming and problem-solving skills and apply those skills in solving computing problems.

PSO 2: Select appropriate programming languages, Data structures and algorithms in combination with modern technologies and apply them in developing innovative solutions.

PSO 3: Apply techniques of data modelling, analysis and visualization which include statistical techniques to solve real world problems delivering actionable insights for decision making.

PROGRAM SPECIFIC OUTCOMES (PSO's) - (CSE-CS)

PSO 1: Demonstrate competency in Programming and problem-solving skills and apply those skills in solving computing problems.

PSO 2: Select appropriate programming languages, Data structures and algorithms in combination with modern technologies and apply them in developing innovative solutions.

PSO 3: Apply cryptographic algorithms for ensuring cyber security as per cyber laws, demonstrate awareness of all the security related issues and employ state of the art technologies to protect the digital assets of an organization.

PROGRAM SPECIFIC OUTCOMES (PSO's) - (CSE-IoT)

PSO 1: Demonstrate competency in Programming and problem-solving skills and apply those skills in solving computing problems.

PSO 2: Select appropriate programming languages, Data structures and algorithms in combination with modern technologies and apply them in developing innovative solutions.

PSO 3: Demonstrate an ability in using IoT devices and protocols to develop IoT based solutions for real world problems.

PROGRAM SPECIFIC OUTCOMES (PSO's) –IT

PSO1: To inculcate algorithmic thinking and problem-solving skills, applying different programming paradigms.

PSO2: Develop an ability to design and implement various processes/methodologies/practices employed in design, validation, testing and maintenance of software products

PROGRAM SPECIFIC OUTCOMES (PSO's) - ECE

PSO 1: An ability to design an Electronic and Communication Engineering system, component, or process and conduct experiments, analyze, interpret data and prepare a report with conclusions to meet desired needs within the realistic constraints such as economic, environmental, social, political, ethical, health and safety, manufacturability and sustainability.

PSO 2: An ability to use modern Electronic Design Automation (EDA) tools, software and electronic equipment to analyze, synthesize and evaluate Electronics and Communication Engineering systems for multidisciplinary tasks.

PROGRAM SPECIFIC OUTCOMES (PSO's) - EEE

PSO1: An ability to simulate and determine the parameters like voltage profile and current ratings of transmission lines in Power Systems.

PSO2: An ability to understand and determine the performance of electrical machines namely, speed, torque, efficiency etc.

PSO3: An ability to apply electrical engineering and management principles to Power Projects

PROGRAM SPECIFIC OUTCOMES (PSO's) - CE

PSO 1: Apply knowledge in core areas of Civil Engineering such as Structural, Geotechnical, Water Resources, Transportation and Environmental Engineering to Civil Engineering practice

PSO 2: Utilize Civil Engineering principles that are appropriate to produce detailed drawings, design reports, quantity and cost estimates, specifications, contracts and other documents appropriate for the design, construction, operations and maintenance of Civil Engineering projects.

PSO 3: Shall interact and collaborate with stakeholders; execute quality construction works applying Civil Engineering tools namely, Total Station, Global Positioning System (GPS), ArcGIS, AutoCAD, STAAD and other necessary tools.

PROGRAM SPECIFIC OUTCOMES (PSO's) - ME

PSO 1: Apply Continuity, Energy and Momentum equations to mechanical systems, design and perform experiments in all fields of mechanical engineering

PSO 2: Able to analyze, design and develop/model mechanical and its allied systems using software tools such as AUTOCAD, ANSYS, Creo etc.

PSO 3: Able to design layouts for process and manufacturing industry taking into consideration optimization of resources for effective operation and maintenance.

6. COURSE OBJECTIVES AND OUTCOMES

Course Objectives

Develop ability to

1. Understand the concepts of string, structure union, and enumerated types
2. Understand linear lists and their implementation using arrays and linked list.
3. Understand the classical approaches to sorting arrays: selection sort, Quick sort, insertion sort; sequential and binary searching algorithms.
4. Concepts and principles of stacks and queues and their applications.
5. Understand the basic characteristics of text, binary files and C implementation of file I/O using streams and command line arguments.

Course Outcomes

After completion of the course, student would be able to

- CO1. Implement string ,functions and use enumerated types, define and use structures, unions in programs using C language.
- CO2. Ability to implement linear lists in programs using C Language.
- CO3. Write programs that sort data using selection, Quick, insertion sort techniques and perform search mechanisms either by sequential or binary search techniques using C language program.
- CO4. 4. Concepts and principles operations of stacks and queues using C program.
- CO5. Write programs that read and write text, binary files using the formatting and character I/O functions.

7. BRIEF NOTES ON THE IMPORTANCE OF THE COURSE AND HOW IT FITS INTO THE CURRICULUM

1. Learn the advanced concepts in C programming language.
2. Understand the concepts of how to develop and build user defined data types using structure concepts.
3. Use the concepts of programming to implement the same for searching, sorting and build basic data structures.
4. Use the concept of files in C programming to work with persistent data.
5. Provide programmers with the means of writing efficient, maintainable, and portable code.
6. Understand the introductory concept of Non-Linear data structures.

8. PREREQUISITES IF ANY

- i) Programming for Problem Solving.(20CS11001)

9. INSTRUCTIONAL LEARNING OUTCOMES

Upon completing this course, it is expected that a student will be able to do the following:

- a) **Coding in C:** Create and work with strings and user defined data types using structures.
- b) **Sorting and Searching:** To work with unstructured data and use various sorting techniques to structure the data. Search for elements within the given data.
- c) **Data Structures:** Understand and implement the basic data structures used in C.
- d) **Files:** Work with files using C programming concepts.
- e) **Modular approach:** Can divide a program into several modules and make them work together

UNIT 1

- 1. Use string to give a name to a data type.
- 2. Declare a user-defined data type and using enumerated type.
- 3. Different type of string function implementation.
- 4. Differentiate arrays and structures, declare, initialize and access structures.
- 5. Implement nested structures, array of structures, and arrays within a structure using C programming.
- 6. Write programs by passing structures to functions using pass by value and pass by address.
- 7. Declare a self-referential structure and its applications.
- 8. Understand the differences and similarities of a structure and union. Write C programs using unions.

UNIT 2

- 1. Differentiate arrays and linked lists.
- 2. Implement insertion, deletion and search of nodes in a linked list.

UNIT 3

1. Write a program to sort data using selection sort, quick sort, insertion sort in C language.
2. Understand different search techniques. Implement Sequential search and binary search technique using C language program.

UNIT 4

1. Understand and 4. Concepts and principles operations of stacks(push and pop).
2. Convert a given infix expression into prefix and postfix expressions and implement using C program.
3. Implement the evaluation of a postfix expression using a stack in C program.
4. Understand and 4. Concepts and principles operations of queues (enqueue,dequeue) using C program.

UNIT 5

1. Understand the basic properties and characteristics of external files, C implementation of file I/O using streams.
2. Understand differences between text files and binary files.
3. Understand different states and modes in which files can be opened.
4. Write programs that read and write text, binary files using the formatting and character I/O functions.
5. Write programs that handle file status questions. Write programs that process files randomly.
6. Pass arguments to the main function from the command line and understand how a program can be controlled from outside.
7. Understand the advantage of bit-fields and implement it using C programming.
8. Use different preprocessor commands in the C program

10. COURSE MAPPING WITH PO's & PSO's

Mapping of Course Outcomes with Program Outcomes & Program Specific Outcomes- CSE:

1: LOW 2: MEDIUM 3: HIGH

POs	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
Data Structures															
CO1.1 Understand the concepts of strings, structure union, and enumerated types.	3	2	3	2	2				2			2	2	2	-
CO2. Understand linear lists and their implementation using arrays and linked list.	3	2	3	2	2				2			2	2	2	-
CO3. Understand the classical approaches to sorting arrays: selection sort, Quick sort, insertion sort; sequential and binary searching algorithms. program	3	2	3	2	2				2			2	2	2	-
CO4.4. Concepts and principles operations of stacks and queues and their applications.	3	2	3	2	2				2			2	2	2	-
CO 5. 5. Understand the basic characteristics of text, binary files and C implementation of file I/O using streams and command line arguments.	3	2	3	2	2				2			2	2	2	-

Mapping of Course Outcomes with Program Outcomes & Program Specific Outcomes- CSE(DS):

1: LOW 2: MEDIUM 3: HIGH

POs	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
Data Structures															
CO1.1 Understand the concepts of string, structure union, and enumerated types.	3	2	3	2	2				2			2	2	2	-
CO2. Understand linear lists and their implementation using arrays and linked list.	3	2	3	2	2				2			2	2	2	-
CO3. Understand the classical approaches to sorting arrays: selection sort, Quick sort, insertion sort; sequential and binary searching algorithms. program	3	2	3	2	2				2			2	2	2	-
CO 4 Concepts and principles operations of stacks and queues and their applications.	3	2	3	2	2				2			2	2	2	-
CO 5. Understand the basic characteristics of text, binary files and C implementation of file I/O using streams and command line arguments..	3	2	3	2	2				2			2	2	2	

**Mapping of Course Outcomes with Program Outcomes & Program Specific Outcomes-
CSE(IOT):**

1: LOW 2: MEDIUM 3: HIGH

POs	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
Data Structures															
CO1. Understand the concepts of string, structure union, and enumerated types.	3	2	3	2	2				2			2	2	2	-
CO2. Understand linear lists and their implementation using arrays and linked list.	3	2	3	2	2				2			2	2	2	-
CO3. Understand the classical approaches to sorting arrays: selection sort, Quick sort, insertion sort; sequential and binary searching algorithms. program	3	2	3	2	2				2			2	2	2	-
CO4. Concepts and principles operations of stacks and queues using C program	3	2	3	2	2				2			2	2	2	-
CO5. 5. Understand the basic characteristics of text, binary files and C implementation of file I/O using streams and command line arguments.	3	2	3	2	2				2			2	2	2	-

**Mapping of Course Outcomes with Program Outcomes & Program Specific Outcomes-
CSE(AI&ML):**

1: LOW 2: MEDIUM 3: HIGH

POs	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
Data Structures															
CO1.1 Understand the concepts of string, structure union, and enumerated types.	3	2	3	2	2				2			2	2	2	-
CO2. Understand linear lists and their implementation using arrays and linked list.	3	2	3	2	2				2			2	2	2	-
CO3.3. Understand the classical approaches to sorting arrays: selection sort, Quick sort, insertion sort; sequential and binary searching algorithms. program	3	2	3	2	2				2			2	2	2	-
CO4.4. Concepts and principles operations of stacks and queues using C program	3	2	3	2	2				2			2	2	2	-
CO5. 5. Understand the basic characteristics of text, binary files and C implementation of file I/O using streams and command line arguments.	3	2	3	2	2				2			2	2	2	-

Mapping of Course Outcomes with Program Outcomes & Program Specific Outcomes- CSE(CS):

1: LOW 2: MEDIUM 3: HIGH

POs	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
Data Structures															
CO1.1 Understand the concepts of string, structure union, and enumerated types.	3	2	3	2	2				2			2	2	2	-
CO2. Understand linear lists and their implementation using arrays and linked list.	3	2	3	2	2				2			2	2	2	-
CO3.3. Understand the classical approaches to sorting arrays: selection sort, Quick sort, insertion sort; sequential and binary searching algorithms. program	3	2	3	2	2				2			2	2	2	-
CO4.4. Concepts and principles operations of stacks and queues using C program	3	2	3	2	2				2			2	2	2	-
CO5. Understand the basic characteristics of text, binary files and C implementation of file I/O using streams and command line arguments.	3	2	3	2	2				2			2	2	2	-

Mapping of Course Outcomes with Program Outcomes & Program Specific Outcomes- IT:

1: LOW 2: MEDIUM 3: HIGH

POs	1	2	3	4	5	6	7	8	9	10	11	12	1	2
Data Structures														
CO1.1 Understand the concepts of string, structure union, and enumerated types.	3	2	3	2	2				2			2	2	2
CO2. Understand linear lists and their implementation using arrays and linked list.	3	2	3	2	2				2			2	2	2
CO3.3. Understand the classical approaches to sorting arrays: selection sort, Quick sort, insertion sort; sequential and binary searching algorithms. program	3	2	3	2	2				2			2	2	2
CO4.4. Concepts and principles operations of stacks and queues using C program	3	2	3	2	2				2			2	2	2
CO5. Understand the basic characteristics of text, binary files and C implementation of file I/O using streams and command line arguments.	3	2	3	2	2				2			2	2	2

Mapping of Course Outcomes with Program Outcomes & Program Specific Outcomes- ECE:

POs	1	2	3	4	5	6	7	8	9	10	11	12	1	2
Data Structures														
CO1. Use the type definition, enumerated types, define and use structures, unions in programs using C language.	3	2	3	2	2				2			2	2	
CO2. Understand linear lists and their implementation using arrays and linked list.	3	2	3	2	2				2			2	2	
CO3.3. Understand the classical approaches to sorting arrays: selection sort, Quick sort, insertion sort; sequential and binary searching algorithms. program	3	2	3	2	2				2			2	2	
CO4. Concepts and principles operations of stacks and queues using C program	3	2	3	2	2				2			2	2	
CO5. Understand the basic characteristics of text, binary files and C implementation of file I/O using streams and command line arguments.	3	2	3	2	2				2			2	2	

Mapping of Course Outcomes with Program Outcomes & Program Specific Outcomes- EEE:

POs	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
Data Structures															
CO1. Use the type definition, enumerated types, define and use structures, unions in programs using C language.	3	2	3	2	2				2			2	2		
CO2. Understand linear lists and their implementation using arrays and linked list.	3	2	3	2	2				2			2	2		
CO3. Understand the classical approaches to sorting arrays: selection sort, Quick sort, insertion sort; sequential and binary searching algorithms. program	3	2	3	2	2				2			2	2		
CO4. Concepts and principles operations of stacks and queues using C program	3	2	3	2	2				2			2	2		
CO5. Understand the basic characteristics of text, binary files and C implementation of file I/O using streams and command line arguments.	3	2	3	2	2				2			2	2		

Mapping of Course Outcomes with Program Outcomes & Program Specific Outcomes- CE:

POs	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
Data Structures															
CO1. Use the type definition, enumerated types, define and use structures, unions in pro3.grams using C language.	3	2	3	2	2				2			2			
CO2. Understand linear lists and their implementation using arrays and linked list.	3	2	3	2	2				2			2			
CO3.3. Understand the classical approaches to sorting arrays: selection sort, Quick sort, insertion sort; sequential and binary searching algorithms. program	3	2	3	2	2				2			2			
CO4.4. Concepts and principles operations of stacks and queues using C program	3	2	3	2	2				2			2			
CO5. Understand the basic characteristics of text, binary files and C implementation of file I/O using streams and command line arguments..	3	2	3	2	2				2			2			

Mapping of Course Outcomes with Program Outcomes & Program Specific Outcomes- ME:

POs	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
Data Structures															
CO1. Use the type definition, enumerated types, define and use structures, unions in pro3.grams using C language.	3	2	3	2	2				2			2			
CO2. Understand linear lists and their implementation using arrays and linked list.	3	2	3	2	2				2			2			
CO3. Understand the classical approaches to sorting arrays: selection sort, Quick sort, insertion sort; sequential and binary searching algorithms. program	3	2	3	2	2				2			2			
CO4. Concepts and principles operations of stacks and queues using C program	3	2	3	2	2				2			2			
CO5. Understand the basic characteristics of text, binary files and C implementation of file I/O using streams and command line arguments.	3	2	3	2	2				2			2			

11. CLASS TIME TABLE

Geethanjali College of Engineering and Technology					
Department of Freshman Engineering					
Time Table (Online Classes)					
Year/Sem/Branch & Sec : I-B.Tech II-Semester, CSE- A				Acad. Year : 2020 -21	W.E.F 30-03-2021
Class Incharge: Mr. S. Rajesham				Version-1	
Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2		3
Monday	BEE	BREAK	MVC	LUNCH	EG
Tuesday	PPS-II		DM		SD lab
Wednesday	SD		PPS-II		EG
Thursday	MVC		BEE		PPS-II Lab
Friday	DM		SD		BEE lab
Saturday	EWS Lab		—		—

S.No	Subject(T/P)	Course Code	Faculty Name
1	Semiconductor Devices(SD)	20PH12001	Mr. S. Rajesham
2	Multi Variable Calculus(MVC)	20MA12001	Ms. S. Lalitha
3	Programming for Problem Solving-II(PPS-II)	20CS12001	Ms. Keerthi
4	Basic Electrical Engineering(BEE)	20EE12001	Ms. Azra Zaineb
5	Engineering Graphics(EG)	20ME12002	Mr. K. Raju, Mr. P. Satyanarayana
6	Discrete Mathematics(DM)	20CS12002	Ms. K. Durga Kalyani
7	Semiconductor Devices Lab(SD Lab)	20PH12L01	Mr. S. Rajesham, Ms. V. Manjula
8	Programming for Problem Solving-II LAB (PPS-II LAB)	20CS12L01	Ms. Keerthi
9	Basic Electrical Engineering Lab(BEE LAB)	20EE12L01	Ms. Azra Zaineb
10	Engineering Workshop(EWS LAB)	20ME12L01	Mr. R. Mahipal Reddy
11	Mentors	Mr. S. Rajesham, Ms. S. Lalitha, Ms. B. Keerthi	

TT. Coord: _____ HOD: _____
Principal: _____

Dean Academics:- _____

Geethanjali College of Engineering and Technology

Department of Freshman Engineering

Time Table (Online Classes)

Year/Sem/Branch & Sec : I-B.Tech II-Semester, CSE- B				Acad. Year : 2020 -21 W.E.F 30-03-2021	
Class Incharge: Ms. V. Manjula				Version-1	
Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1	BREAK	2	LUNCH	3
Monday	SD		BEE		EG
Tuesday	DM		MVC		PPS-II Lab
Wednesday	PPS-II		DM		SD lab
Thursday	BEE		SD		EG
Friday	MVC		PPS-II		BEE lab
Saturday	EWS Lab		—		—

S.No	Subject(T/P)	Course Code	Faculty Name
1	Semiconductor Devices(SD)	20PH12001	Dr. G. Neeraja Rani/ Ms. V. Manjula
2	Multi Variable Calculus(MVC)	20MA12001	Dr. V. S. Triveni
3	Programming for Problem Solving-II(PPS-II)	20CS12001	Dr. L. Venkateswarlu
4	Basic Electrical Engineering(BEE)	20EE12001	Mr. M. Prashanth Kumar
5	Engineering Graphics(EG)	20ME12002	Mr. N. Rajender, Mr. N. S. Raghavendra
6	Discrete Mathematics(DM)	20CS12002	Ms. K. Durga Kalyani
7	Semiconductor Devices Lab(SD Lab)	20PH12L01	Ms. V. Manjula, Ms. Ch. Kalyani
8	Programming for Problem Solving-II LAB (PPS-II LAB)	20CS12L01	Dr. L. Venkateswarlu
9	Basic Electrical Engineering Lab(BEE LAB)	20EE12L01	Mr. M. Prashanth Kumar
10	Engineering Workshop(EWS LAB)	20ME12L01	Mr. B. Subba Rao
11	Mentors	Ms. V. Manjula, Dr. L. Venkateswarlu, Ms. K. Durga Kalyani	

TT. Coord: _____ HOD: _____
Principal: _____

Dean Academics:- _____

Geethanjali College of Engineering and Technology

Department of Freshman Engineering

Time Table (Online Classes)

Year/Sem/Branch & Sec : I-B.Tech II-Semester, CSE- C				Acad. Year : 2020 -21 W.E.F 30-03-2021	
Class Incharge: Ms. M. P. Molimol				Version-1	
Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2		3
Monday	DM	BREAK	MVC	LUNCH	BEE lab
Tuesday	BEE		SD		PPS-II Lab
Wednesday	PPS-II		BEE		EG
Thursday	MVC		PPS-II		SD Lab
Friday	SD		DM		EG
Saturday	EWS Lab		—		—

S.No	Subject(T/P)	Course Code	Faculty Name
1	Semiconductor Devices(SD)	20PH12001	Ms. Ch. Kalyani
2	Multi Variable Calculus(MVC)	20MA12001	Ms. M.P. Molimol
3	Programming for Problem Solving-II(PPS-II)	20CS12001	Mr. M. Ravinder
4	Basic Electrical Engineering(BEE)	20EE12001	Mr. S. Poornachandar Rao
5	Engineering Graphics(EG)	20ME12002	Mr. R. Sudarshan, Mr. B. Bhasker
6	Discrete Mathematics(DM)	20CS12002	Mr. B. Mamatha
7	Semiconductor Devices Lab(SD Lab)	20PH12L01	Ms. Ch. Kalyani , Mr. S. Rajesham
8	Programming for Problem Solving-II LAB (PPS-II LAB)	20CS12L01	Mr. M. Ravinder
9	Basic Electrical Engineering Lab(BEE LAB)	20EE12L01	Mr. S. Poornachandar Rao, Mr. Ch. Adithya
10	Engineering Workshop(EWS LAB)	20ME12L01	Mr. M. Ravi Kumar
11	Mentors	Ms. M.P. Molimol, Mr. A. Ramesh, Ms. Ch. Kalyani	

TT. Coord: _____ HOD: _____
Principal: _____

Dean Academics:- _____

Geethanjali College of Engineering and Technology

Department of Freshman Engineering

Time Table (Online Classes)

Year/Sem/Branch & Sec : I-B.Tech II-Semester, CSE- D			Acad. Year : 2020 -21 W.E.F 30-03-2021		
Class Incharge: Mr. M. Ravinder			Version-1		
Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2		3
Monday	BEE	BREAK	SD	LUNCH	PPS-II Lab
Tuesday	BEE		PPS-II		SD lab
Wednesday	DM		MVC		BEE lab
Thursday	SD		DM		EG
Friday	MVC		PPS-II		EG
Saturday	EWS Lab		—		—

S.No	Subject(T/P)	Course Code	Faculty Name
1	Semiconductor Devices(SD)	20PH12001	Dr. J. Shankar
2	Multi Variable Calculus(MVC)	20MA12001	Ms.S. Lalitha
3	Programming for Problem Solving-II(PPS-II)	20CS12001	Mr. M. Ravinder
4	Basic Electrical Engineering(BEE)	20EE12001	Mr. V. Rakesh
5	Engineering Graphics(EG)	20ME12002	Ms. B. Anitha, Mr. V. Rajashekar
6	Discrete Mathematics(DM)	20CS12002	Ms. B. Mamatha
7	Semiconductor Devices Lab(SD Lab)	20PH12L01	Dr. J. Shankar, Mr. A. Shiva Kumar
8	Programming for Problem Solving-II LAB (PPS-II LAB)	20CS12L01	Mr. M. Ravinder
9	Basic Electrical Engineering Lab(BEE LAB)	20EE12L01	Mr. V. Rakesh
10	Engineering Workshop(EWS LAB)	20ME12L01	Mr. P. V.R. Girish Kumar
11	Mentors	Dr. J. Shankar, Mr. M. Ravinder, Ms. G. Padma	

TT. Coord:_____ HOD:_____

Principal:_____

Dean Academics:-_____

Geethanjali College of Engineering and Technology

Department of Freshman Engineering

Time Table (Online Classes)

Year/Sem/Branch & Sec : I-B.Tech II-Semester, CSE-DS				Acad. Year : 2020 -21 W.E.F 30-03-2021	
Class Incharge: Dr. G. Mahesh				Version-1	
Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2		3
Monday	DM	BREAK	MVC	LUNCH	EG
Tuesday	PPS-II		BEE		SD lab
Wednesday	SD		DM		EG
Thursday	PPS-II		SD		BEE lab
Friday	MVC		BEE		PPS-II Lab
Saturday	–		–		EWS Lab

S.No	Subject(T/P)	Course Code	Faculty Name
1	Semiconductor Devices(SD)	20PH12001	Dr. J. Anjaiah
2	Multi Variable Calculus(MVC)	20MA12001	Dr. G. Mahesh
3	Programming for Problem Solving-II(PPS-II)	20CS12001	Prof. V. V. Appaji
4	Basic Electrical Engineering(BEE)	20EE12001	Ms. B. Soujanya
5	Engineering Graphics(EG)	20ME12002	Mr. P. Sudheer Rao, Mr. R. Mahipal Reddy
6	Discrete Mathematics(DM)	20CS12002	Ms. K. Vandhana
7	Semiconductor Devices Lab(SD Lab)	20PH12L01	Dr. J. Anjaiah, Dr. Mohammad Ali
8	Programming for Problem Solving-II LAB (PPS-II LAB)	20CS12L01	Prof. V. V. Appaji
9	Basic Electrical Engineering Lab(BEE LAB)	20EE12L01	Ms. B. Soujanya
10	Engineering Workshop(EWS LAB)	20ME12L01	Mr. R. Mahipal Reddy
11	Mentors	Dr. G. Mahesh, Prof. V. V. Appaji, Dr. J. Anjaiah	

TT. Coord: _____ HOD: _____
Principal: _____

Dean Academics:- _____

Geethanjali College of Engineering and Technology

Department of Freshman Engineering

Time Table (Online Classes)

Year/Sem/Branch & Sec: I-B.Tech II-Semester, CSE- CS				Acad. Year : 2020 -21	W.E.F 30-03-2021
Class Incharge: Mr. M. Ajay Kumar				Version-1	
Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2		3
Monday	BEE	BREAK	MVC	LUNCH	SD lab
Tuesday	DM		PPS-II		EG
Wednesday	SD		DM		PPS-II Lab
Thursday	MVC		SD		BEE lab
Friday	EG		BEE		PPS-II
Saturday	–		–		EWS Lab

S.No	Subject(T/P)	Course Code	Faculty Name
1	Semiconductor Devices(SD)	20PH12001	Mr.A. Shiva Kumar
2	Multi Variable Calculus(MVC)	20MA12001	Mr. K. Nagaraju
3	Programming for Problem Solving-II(PPS-II)	20CS12001	Mr.M. Ajay Kumar
4	Basic Electrical Engineering(BEE)	20EE12001	Mr. K. Murali
5	Engineering Graphics(EG)	20ME12002	Mr. P. Sudheer Rao, Mr. P. Satyanarayana
6	Discrete Mathematics(DM)	20CS12002	Mr. K. Ambedkar
7	Semiconductor Devices Lab(SD Lab)	20PH12L01	Mr. A. Shiva Kumar, Mr. S. Rajesham
8	Programming for Problem Solving-II LAB (PPS-II LAB)	20CS12L01	Mr.M. Ajay Kumar
9	Basic Electrical Engineering Lab(BEE LAB)	20EE12L01	Mr. K. Murali, Mr. A. Raghu Rama Chandra
10	Engineering Workshop(EWS LAB)	20ME12L01	Mr. B. Subba Rao
11	Mentors	Mr.A. Shiva Kumar, Ms. B. Vanaja Rani, Mr. M. Ajay Kumar	

TT. Coord:_____ HOD:_____ Dean Academics:-_____

Principal:_____

Geethanjali College of Engineering and Technology

Department of Freshman Engineering

Time Table (Online Classes)

Year/Sem/Branch & Sec : I-B.Tech II-Semester, CSE(AI&ML)				Acad. Year : 2020 -21 W.E.F 30-03-2021	
Class Incharge: Dr. K. Venkateshwarlu				Version-1	
Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2		3
Monday	PPS-II	BREAK	DM	LUNCH	SD lab
Tuesday	MVC		BEE		PPS-II Lab
Wednesday	SD		PPS-II		BEE lab
Thursday	BEE		MVC		EG
Friday	DM		SD		EG
Saturday	–		–		EWS Lab

S.No	Subject(T/P)	Course Code	Faculty Name
1	Semiconductor Devices(SD)	20PH12001	Dr. SK. Mohammad Ali
2	Multi Variable Calculus	20MA12001	Dr. K.Venkateswarlu
3	Programming for Problem Solving-II(PPS-II)	20CS12001	Prof. V. V. Appaji
4	Basic Electrical Engineering(BEE)	20EE12001	Mr. K. Nagaraju
5	Engineering Graphics(EG)	20ME12002	Mr. P. Sudheer Rao, Mr. P. Sandeep Kumar
6	Discrete Mathematics	20CS12002	Ms. A. Mounika
7	Semiconductor Devices Lab(SD Lab)	20PH12L01	Dr. SK. Mohammad Ali, Dr. B. Mamatha
8	Programming for Problem Solving-II LAB (PPS-II LAB)	20CS12L01	Prof. V. V. Appaji
9	Basic Electrical Engineering Lab(BEE LAB)	20EE12L01	Mr. K. Nagaraju, Mr. K. Mahender
10	Engineering Workshop(EWS LAB)	20ME12L01	Mr. U. Srikanth
11	Mentors	Dr. SK. Mohammad Ali, Dr. K.Venkateswarlu, Mr. N. Nagi Reddy	

TT. Coord: _____ HOD: _____
Principal: _____

Dean Academics:- _____

Department of Freshman Engineering

Time Table (Online Classes)

Year/Sem/Branch & Sec : I-B.Tech II-Semester, CSE- IoT			Acad. Year : 2020 -21		W.E.F 30-03-2021
Class Incharge: Ms. P. Sailaja			Version-1		
Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2		3
Monday	MVC	BREAK	PPS-II	LUNCH	BEE lab
Tuesday	SD		BEE		EG
Wednesday	BEE		MVC		SD lab
Thursday	DM		SD		PPS-II Lab
Friday	PPS-II		DM		EG
Saturday	–		–		EWS Lab

S.No	Subject(T/P)	Course Code	Faculty Name
1	Semiconductor Devices(SD)	20PH12001	Dr. B. Mamatha
2	Multi Variable Calculus(MVC)	20MA12001	Ms. P. Sailaja
3	Programming for Problem Solving-II(PPS-II)	20CS12001	Mr. M. Ajay Kumar
4	Basic Electrical Engineering(BEE)	20EE12001	Mr. S. Hareesh Reddy
5	Engineering Graphics(EG)	20ME12002	Mr. D. Ramchander, Ms. P. Supriya
6	Discrete Mathematics(DM)	20CS12002	Mr. K. Ambedkar
7	Semiconductor Devices Lab(SD Lab)	20PH12L01	Dr. B. Mamatha, Mr. A. Shiva Kumar
8	Programming for Problem Solving-II LAB (PPS-II LAB)	20CS12L01	Mr. M. Ajay Kumar
9	Basic Electrical Engineering Lab(BEE LAB)	20EE12L01	Mr. S. Hareesh Reddy, Ms. B. Sowjanya
10	Engineering Workshop(EWS LAB)	20ME12L01	Mr. K. Praveen
11	Mentors	Ms. P. Sailaja, Dr. B. Mamatha, Dr.Sk. Nuslin Bibi	

TT. Coord: _____ HOD: _____
Principal: _____

Dean Academics:- _____

Geethanjali College of Engineering and Technology

Department of Freshman Engineering

Time Table (Online Classes)

Year/Sem/Branch & Sec: I-B.Tech II-Semester, IT				Acad. Year : 2020 -21	W.E.F 30-03-2021
Class Incharge: Ms. Fathima				Version-1	
Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2		3
Monday	PPS-II	BREAK	SD	LUNCH	EG
Tuesday	MVC		DM		EG
Wednesday	BEE		SD		PPS-II Lab
Thursday	DM		MVC		BEE Lab
Friday	BEE		PPS-II		SD Lab
Saturday	—		—		—

S.No	Subject(T/P)	Course Code	Faculty Name
1	Semiconductor Devices(SD)	20PH12001	Dr. P. Raju
2	Multi Variable Calculus(MVC)	20MA12001	Dr. Mahesh
3	Programming for Problem Solving-II(PPS-II)	20CS12001	Ms.Fathima
4	Basic Electrical Engineering(BEE)	20EE12001	Mr. K. Jayakar Babu
5	Engineering Graphics(EG)	20ME12002	Mr.G. Raju, Mr. G. Sampath
6	Discrete Mathematics(DM)	20CS12002	Ms. K. Vandhana
7	Semiconductor Devices Lab(SD Lab)	20PH12L01	Dr.B. Mamatha, Ms. Ch. Kalyani
8	Programming for Problem Solving-II LAB (PPS-II LAB)	20CS12L01	Ms.Fathima
9	Basic Electrical Engineering Lab(BEE LAB)	20EE12L01	Mr. K. Jayakar Babu, Mr. S. Hareesh Reddy
10	Mentors	Ms.Fathima, Dr. P. Raju, Mr. K. Nagaraju	

TT. Coord: _____ HOD: _____
Principal: _____

Dean Academics:- _____

Geethanjali College of Engineering and Technology

Department of Freshman Engineering

Time Table (Online Classes)

Year/Sem/Branch & Sec : I-B.Tech II-Semester, ECE-A			Acad. Year : 2020 -21 W.E.F 30-03-2021		
Class Incharge: Mr. V. Ramaiah Chary			Version-1		
Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2		3
Monday	ENG	BREAK	SDC	LUNCH	PPS-II LAB
Tuesday	EC		PPS-II		SDC LAB
Wednesday	MVC		SDC		EC LAB
Thursday	SDC		PPS-II		ELCS LAB
Friday	MVC		EC		ENG
Saturday	—		—		—

S.No	Subject		Course Code	Faculty Name
1	English(Eng)		20EN12001	Mr. V. Ramaiah Chary
2	Multi Variable Calculus(MVC)		20MA12001	Dr. N. Subhadra/Dr. Triveni
3	Programming for Problem Solving-II(PPS-II)		20CS12001	Ms. Keerthi
4	Engineering Chemistry(EC)		20CH12001	Mr. K. Satheesh
5	Semiconductor Devices and Circuits(SDC)		20EC12001	Mr. U.Appala Raju
6	English Language Communication Skills Lab(ELCS LAB)		20EN12L01	Mr. V. Ramaiah Chary
7	Programming for Problem Solving-II Lab (PPS-II LAB)		20CS12L01	Ms. Keerthi
8	Engineering Chemistry Lab(EC Lab)		20CH12L01	Mr. K. Satheesh, Ms. J. Bhargavi Lakshmi
9	Semiconductor Devices and Circuits Lab(SDC LAB)		20EC12L01	Mr. U.Appala Raju, Ms. M. Laxmi
10	Mentors		Mr. V. Ramaiah Chary, Mr. K. Satheesh, Ms. J. Bhargavi Lakshmi	

TT. Coord: _____ HOD: _____
Principal: _____

Dean Academics:- _____

Geethanjali College of Engineering and Technology

Department of Freshman Engineering

Time Table (Online Classes)

Year/Sem/Branch & Sec : I-B.Tech II-Semester, ECE-B				Acad. Year : 2020 -21 W.E.F 30-03-2021	
Class Incharge: Dr. J.V. Madhuri				Version-1	
Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2		3
Monday	MVC	BREAK	SDC	LUNCH	EC LAB
Tuesday	MVC		PPS-II		SDC LAB
Wednesday	PPS-II		EC		ELCS LAB
Thursday	SDC		ENG		EC
Friday	ENG		SDC		PPS-II LAB
Saturday	—		—		—

S.No	Subject		Course Code	Faculty Name
1	English(Eng)		20EN12001	Dr. B. Nagamani
2	Multi Variable Calculus(MVC)		20MA12001	Dr. N. Subhadra
3	Programming for Problem Solving-II(PPS-II)		20CS12001	Dr. Puja Prasad
4	Engineering Chemistry(EC)		20CH12001	Dr. J.V. Madhuri
5	Semiconductor Devices and Circuits(SDC)		20EC12001	Ms. J. Mrudula
6	English Language Communication Skills Lab(ELCS LAB)		20EN12L01	Dr. B. Nagamani
7	Programming for Problem Solving-II Lab (PPS-II LAB)		20CS12L01	Dr. Puja Prasad
8	Engineering Chemistry Lab(EC Lab)		20CH12L01	Dr. J.V. Madhuri, Mr. K. Satheesh
9	Semiconductor Devices and Circuits Lab(SDC LAB)		20EC12L01	Ms. J.Mrudula, Mr. Prasad
10	Mentors		Dr. J.V. Madhuri, Dr. B. Nagamani, Dr. Puja Prasad	

TT. Coord:_____ HOD:_____

Principal:_____

Dean Academics:-_____

Geethanjali College of Engineering and Technology

Department of Freshman Engineering

Time Table (Online Classes)

Year/Sem/Branch & Sec : I-B.Tech II-Semester, ECE-C			Acad. Year : 2020 -21		W.E.F 30-03-2021
Class Incharge: Mr.M. Upender			Version-1		
Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2		3
Monday	SDC	BREAK	EC	LUNCH	PPS-II LAB
Tuesday	MVC		SDC		EC LAB
Wednesday	EC		ENG		SDC LAB
Thursday	SDC		PPS-II		ENG
Friday	PPS-II		MVC		ELCS LAB
Saturday	—		—		—

S.No	Subject		Course Code	Faculty Name
1	English(Eng)		20EN12001	Dr. M. Upender
2	Multi Variable Calculus(MVC)		20MA12001	Ms. M. P.Molimol
3	Programming for Problem Solving-II(PPS-II)		20CS12001	Ms.S. Radha
4	Engineering Chemistry(EC)		20CH12001	Dr.Anurag Gautam
5	Semiconductor Devices and Circuits(SDC)		20EC12001	Ms.M.Laxmi
6	English Language Communication Skills Lab(ELCS LAB)		20EN12L01	Dr. M. Upender
7	Programming for Problem Solving-II Lab(PPS-II LAB)		20CS12L01	Ms. S. Radha
8	Engineering Chemistry Lab(EC Lab)		20CH12L01	Dr.Anurag Gautam, Dr. K. Shashikala
9	Semiconductor Devices and Circuits Lab(SDC LAB)		20EC12L01	Ms.M.Laxmi, Ms. M. Umarani
10	Mentors		Dr. M. Upender, Dr.Anurag Gautam, Ms.S. Radha	

TT. Coord: _____ HOD: _____
Principal: _____

Dean Academics:- _____

Geethanjali College of Engineering and Technology

Department of Freshman Engineering

Time Table (Online Classes)

Year/Sem/Branch & Sec : I-B.Tech II-Semester, ECE-D				Acad. Year : 2020 -21	W.E.F 30-03-2021
Class Incharge: Mr. Y. Anil				Version-1	
Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2		3
Monday	PPS-II	BREAK	SDC	LUNCH	ELCS LAB
Tuesday	SDC		MVC		EC
Wednesday	ENG		SDC		PPS-II LAB
Thursday	EC		ENG		SDC LAB
Friday	MVC		PPS-II		EC LAB
Saturday	—		—		—

S.No	Subject	Course Code	Faculty Name
1	English(Eng)	20EN12001	Mr. Y. Anil
2	Multi Variable Calculus(MVC)	20MA12001	Ms. P. Sailaja
3	Programming for Problem Solving-II(PPS-II)	20CS12001	Ms. S. Radha
4	Engineering Chemistry(EC)	20CH12001	Dr. R. Sanjeev
5	Semiconductor Devices and Circuits(SDC)	20EC12001	Mr. Prasad
6	English Language Communication Skills Lab(ELCS LAB)	20EN12L01	Mr. Y. Anil
7	Programming for Problem Solving-II Lab (PPS-II LAB)	20CS12L01	Ms. S. Radha
8	Engineering Chemistry Lab(EC Lab)	20CH12L01	Dr. R. Sanjeev, Dr. P. Sreedhar
9	Semiconductor Devices and Circuits Lab(SDC LAB)	20EC12L01	Mr. Prasad, Dr. S. Vallisree
10	Mentors	Mr. Y. Anil, Dr. R. Sanjeev, Ms.K. Swarupa	

TT. Coord: _____
Principal: _____

HOD: _____

Dean Academics:- _____

Geethanjali College of Engineering and Technology

Department of Freshman Engineering

Time Table (Online Classes)

Year/Sem/Branch & Sec : I-B.Tech II-Semester, EEE			Acad. Year : 2020 -21		W.E.F 30-03-2021
Class Incharge: Dr. K. Shasikala			Version-1		
Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2		3
Monday	MVC	BREAK	EC	LUNCH	SDC LAB
Tuesday	SDC		PPS-II		—
Wednesday	EC		MVC		CM LAB
Thursday	PPS-II		SDC		EC LAB
Friday	CM		SDC		PPS-II LAB
Saturday	—		—		—

S.No	Subject	Course Code	Faculty Name
1	Multi Variable Calculus(MVC)	20MA12002	Dr. Sk. Nuslin Bibi
2	Computational Mathematics(CM)	20MA12002	Dr. K.. Venkateswarlu
3	Engineering Chemistry(EC)	20CH12001	Dr. K. Shashikala
4	Programming for Problem Solving-II(PPS-II)	20CS12001	Ms.S. Sudha
5	Semiconductor Devices and Circuits(SDC)	20EC12001	Ms.M.Umarani
6	Computational Mathematics Lab(CM LAB)	20MA12L01	Dr. K. Venkateswarlu
7	Engineering Chemistry Lab(EC Lab)	20CH12L01	Dr. K. Shashikala, Dr. Anurag Gautam
8	Programming for Problem Solving-II Lab(PPS-II LAB)	20CS12L01	Ms.S. Sudha
9	Semiconductor Devices and Circuits Lab(SDC LAB)	20EC12L01	Ms.M.Umarani, Dr. S. Vallisree
10	Mentors	Dr, K, Shashikala, Ms. S. Sudha	

TT. Coord: _____ HOD: _____
Principal: _____

Dean Academics:- _____

Geethanjali College of Engineering and Technology

Department of Freshman Engineering

Time Table (Online Classes)

Year/Sem/Branch & Sec : I-B.Tech II-Semester, CE			Acad. Year : 2020 -21 W.E.F 30-03-2021		
Class Incharge: Ms.P. Mercy Kavitha			Version-1		
Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2		3
Monday	ENG	BREAK	EC	LUNCH	PPS-II LAB
Tuesday	PPS-II		MVC		DT
Wednesday	PPS-II		E.GEO		EC LAB
Thursday	EC		E.GEO		ELCS LAB
Friday	MVC		ENG		E. GEO LAB
Saturday	–		–		–

S.No	Subject		Course Code	Faculty Name
1	English (Eng)		20EN12001	Ms. P. Mercy Kavitha
2	Multi Variable Calculus(MVC)		20MA12002	Dr. Sk. Nuslin Bibi
3	Programming for Problem Solving-II(PPS-II)		20CS12001	Ms.Fathima
4	Engineering Chemistry(EC)		20CH12001	Dr. P. Sreedhar
5	Engineering Geology(E.Geo)		20CE12001	Dr. N. Mahendra
6	English Language Communication Skills Lab(ELCS LAB)		20EN12L01	Ms. P. Mercy Kavitha
7	Programming for Problem Solving-II Lab (PPS-II LAB)		20CS12L01	Ms.Fathima
8	Engineering Chemistry Lab(EC Lab)		20CH12L01	Dr. P. Sreedhar, Dr. R. Sanjeev
9	Design Thinking(DT)		20CE12001	Ms. M. Priyanka
10	Engineering Geology Lab (E.Geo Lab)		20CE12001	Dr. N. Mahendra
11	Mentors		Ms. P. Mercy Kavitha, Dr. P. Sreedhar, Ms. M. Priyanka	

TT. Coord: _____ HOD: _____
Principal: _____

Dean Academics:- _____

Geethanjali College of Engineering and Technology

Department of Freshman Engineering

Time Table (Online Classes)

Year/Sem/Branch & Sec : I-B.Tech II-Semester, ME				Acad. Year : 2020 -21 W.E.F 30-03-2021	
Class Incharge: Ms. B. Vanaja Rani				Version-1	
Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1	BREAK	2	LUNCH	3
Monday	MMD		PPS-II		ELCS Lab
Tuesday	MVC		EC		ENG
Wednesday	ENG		DT		PPS-II LAB
Thursday	EC		MMD		DT
Friday	PPS-II		MVC		EC LAB
Saturday	–		–		–

S.No	Subject		Course Code	Faculty Name
1	English (Eng)		20EN12001	Ms. B. Vanaja Rani
2	Multi Variable Calculus(MVC)		20MA12002	Mr. K. Nagaraju
3	Programming for Problem Solving-II(PPS-II)		20CS12001	Ms. S. Sudha
4	Engineering Chemistry(EC)		20CH12001	Ms. J. Bhargavi Lakshmi
5	Mechanics and Mechanical Drives(MMD)		20ME12001	Mr. N. S. Raghavendra
6	English Language Communication Skills Lab(ELCS LAB)		20EN12L01	Ms. B. Vanaja Rani
7	Programming for Problem Solving-II Lab (PPS-II LAB)		20CS12L01	Ms. S. Sudha
8	Engineering Chemistry Lab(EC Lab)		20CH12L01	Ms. J. Bhargavi Lakshmi, Dr. J. V. Madhuri
9	Design Thinking(DT)		20ME12P01	Ms. V. Sandeepa
10	Mentors		Mr. P. Sudheer Rao, Mr. R. Mahipal Reddy	

TT. Coord:_____ HOD:_____

Principal:_____

Dean Academics:-_____

12. INDIVIDUAL TIME TABLE

Dr. Puja Prasad

PPS-II

ECE-B

Period	1	BREAK	2	LUNCH	3
Mon					ECE-A Lab
Tue	CSE-A		ECE-A		
Wed			CSE-A		
Thu			ECE-A		CSE-A Lab
Fri					
Sat	–				

M. Ajay Kumar

PPS-II

CSE-CS, IoT

Time	09.00-10.30	BREAK	11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2	LUNCH	3
Mon			CSE-IoT		
Tue			CSE-CS		
Wed					CSE-CS
Thu					CSE-IoT
Fri	CSE-IoT				CSE-CS
Sat	–				

M. Ravinder

PPS-II

CSE-C, D

Time	09.00-10.30	BREAK	11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2	LUNCH	3
Mon					CSE-D Lab
Tue			CSE-D		CSE-C Lab
Wed	CSE-C				
Thu			CSE-C		
Fri			CSE-D		
Sat	–				

Fathima

PPS-II

CE, IT

Time	09.00-10.30	BREAK	11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2	LUNCH	3
Mon	IT				CE
Tue	CE				
Wed	CE				IT
Thu			IT		
Fri					
Sat	–				

Prof. Appaji
PPS II
(AIML,DS)

Time	09.00-10.30		11.00-12.30	12.30-1.30	1.30-3.00
Period	1		2		3
Mon	CSE-AIML	BREAK		LUNCH	
Tue			CSE-DS		CSE-AIML
Wed			CSE-AIML		
Thu	CSE-DS				
Fri					CSE-DS
Sat	–				

13. LECTURE SCHEDULE WITH METHODOLOGY BEING USED/ADOPTED

<i>Period No.</i>	<i>Topic to be covered</i>	<i>Instructional learning outcomes Students should be able to</i>	<i>BTL</i>	<i>Date</i>	<i>Mode of teaching</i>	<i>Remarks</i>
1	UNIT I: C String Concepts ,Input output functions	Use string definition to give a name to a data type	L1		BB	
2	String manipulation function,string conversion	Declare a user-defined data type and using enumerated type	L2		BB	
3	Introduction to Structures, Structure Initialization, Enumerated Types Declaration and Accessing	Differentiate union and structures, declare, initialize and access structures	L2		BB	
4	Operations on Structures, Complex Structures	Implement nested structures, array of structures, and arrays within a structure using C programming	L3		OHP/BB	
5	Structures and Functions	Write programs by passing structures to functions using pass by value	L3,L4		OHP/BB	
6	Structures and Functions	Write programs by passing structures to functions using pass by value	L3,L4		OHP/BB	
7	Passing Structures through pointers	Write programs by passing structures to functions using pass by address	L3,L4		OHP/BB	
8	Self Referential Structures	Declare a self-referential structure and its applications	L2		OHP/BB	
9	Unions	Understand the differences and similarities of a structure and union. Write C programs using unions.	L2,L3		OHP/BB	
10	Unions	Understand the differences and similarities of a structure and union. Write C programs using unions.	L2,L4		OHP/BB	
11	Bit Fields	Understand the advantage of bit-fields and implement it using C programming	L2,L3		LCD/BB	
	UNIT II :				LCD/BB	

12	Introduction to Linear Lists	Differentiate arrays and linked lists.	L2		BB	
13	Singly Linked List: Insertion	Implement insertion of nodes in a linked list.	L3, L4,L5		BB	
14	Singly Linked List: Deletion	Implement deletion of nodes from a linked list.	L3, L4,L5		BB	
15	Singly Linked List: Insertion, Deletion Searching	Implement all operations of a linked lists.	L3, L4,L5		LCD/BB	
16	UNIT III: Introduction to Sorting, Selection Sort	Write a program to sort data using selection sort in C language.	L3, L4,L5		LCD/BB	
17	Quick Sort Sort	Write a program to sort data using Quick Sort sort in C language.	L3, L4,L5		LCD/BB	
18	Insertion Sort	Write a program to sort data using insertion sort in C language.	L3, L4,L5		LCD/BB	
19	Introduction to searching, Linear Search	Understand different search techniques. Implement Sequential search technique using C language program	L1,L3, L4		BB	
20	Binary Search	Implement Binary search technique using C language program	L3, L4		LCD/BB	
21	Binary Search	Implement Binary search technique using C language program	L3, L5		LCD/BB	
22	UNIT IV: Introduction to Stacks, Principal of Stack	Understand the basic operations of stacks	L2		LCD/BB	
23	Stack Operations	Demonstrate the basic operations of stacks(push ,pop) using C program	L3, L4,L5		OHP/BB	
24	Stack Operations	Demonstrate the basic operations of stacks(push ,pop) using C program	L3, L4,L5		BB	
25	Stack notations : Infix, Postfix	Convert a given infix expression into prefix and postfix expressions	L3, L4,L5		BB	
26	Infix, Postfix conversions	Implement the conversion of a given infix expression into postfix expressions using C program	L3, L4,L5		BB	

27	Postfix evaluation	Implement the evaluation of a postfix expression using a stack in C program	L3, L4,L5		BB	
28	Postfix evaluation	Implement the evaluation of a postfix expression using a stack in C program	L3, L4,L5		BB	
29	Introduction to Queues	Understand the basic operations of queues	L2		BB	
30	Principal and operations: Enqueue and Dequeue	4. Concepts and principles operations of queues (enqueue,dequeue) using C program	L3, L4,L5		BB	
31	Principal and operations: Enqueue and Dequeue	4. Concepts and principles operations of queues (enqueue,dequeue) using C program	L3, L4,L5		BB	
32	UNIT V: Introduction to Files: Concept of Files, Streams	Understand the basic properties and characteristics of external files,C implementation of file I/O using streams	L2		BB	
33	Text and Binary Files	Understand differences between text files and binary files	L2		BB	
34	State of files, opening and Closing Files	Understand different states and modes in which files can be opened	L2		BB	
35	File I/O Functions(Standard library)	Write programs that read and write text, binary files using the formatting and character I/O functions.	L2,L3, L4		BB	
36	File I/O Functions(Standard library)	Write programs that read and write text, binary files using the formatting and character I/O functions.	L2,L3, L4		BB	
37	File I/O Functions(Formatting I/O)	Write programs that read and write text, binary files using the formatting and character I/O functions.	L2,L3, L4		BB	
38	File Status Functions	Write programs that handle file status questions	L2,L3, L4		BB	
39	Positioning functions	Write programs that process files randomly	L2,L3, L4		BB	
40	Program Development Multi-source files, Separate Compilation of functions	Write multi- source files and compile different files together	L2,L3, L4		BB	
41	Program Development Multi-source files, Separate Compilation of functions	Write multi- source files and compile different files together	L2,L3, L4		BB	

42	Command Line Arguments	Pass arguments to the main function from the command line and understand how a program can be controlled from outside.	L2,L3		LCD/BB	
43	Preprocessor Commands	Use different preprocessor commands in the C program	L2,L3		LCD/BB	

14. DETAILED NOTES.

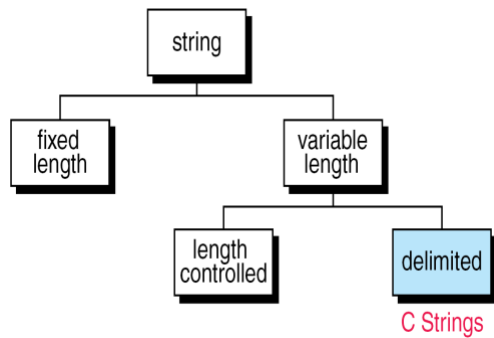
UNIT – I

Strings:

A string is a one dimensional array of characters terminated by a null character(`\0`). A string constant in C can be represented by enclosing its characters with double quotes without a null character.

Strings are classified as

1. Fixed length strings:
2. Variable length strings:



Fixed length

- Reserves the fixed amount of memory for the string variable
- Disadvantages

- 1) Complete string can not be stored if the size is small
- 2) Memory is wasted if the size is too big

Ex:

```
Char a[10]="hello"; // waste memory
```

Variable length

- Size of storage can be expanded and compressed to accommodate the data
- There are two common techniques to use length-controlled strings and delimited strings
 - a. Length controlled string add a count that specifies the number of characters in the string. This count is used by the string manipulation functions to determine the actual length of data.

7	T	E	S	T	I	N	G
---	---	---	---	---	---	---	---

- b. Delimited Strings: C strings are delimited using the ASCII null character(`\0`).

S	T	R	I	N	G	\0
---	---	---	---	---	---	----

C strings: C uses variable-length ,delimited strings.

Storing Strings:

In C, a string is stored in an array of characters. It is terminated the null character(‘\0’). The string “TESTING” is stored in the memory as

T	E	S	T	I	N	G	\0
---	---	---	---	---	---	---	----

Difference between one character in memory and one-character string . The character requires only one memory location, but a one character string requires two memory locations :one for data and one for the delimiter

Char ‘H’

String “H”

Empty string” “

H

H	\0
---	----

\0

Difference between string and character array

String

S	T	R	I	N	G	\0
---	---	---	---	---	---	----

Character array

S	T	R	I	N	G
---	---	---	---	---	---

Because strings are variable in length , the null character is used as an end-of-string marker. Enough room must be provided to store the maximum length string , plus one for delimiter.

String literals

A string literal also known as string constant is a sequence of characters enclosed in double quotes.

Ex: “Hello”

“ C is a high level language”

C automatically creates an array of characters ,initializes it to a null-delimited string, and stores it, remembering its address.

Referencing String Literals

A string literal is stored in memory. We can refer to it using pointers.

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

```
int main()
```

```
{
```

```
printf(“%c”,”HELLO”[1]);
```

```
return 0;
```

```
}
```

Output

E

“Hello”[0]

“Hello”[1]

“Hello”[2]

“Hello”[3]

“Hello”[4]

“Hello”[5]

H	E	L	L	O	\0
---	---	---	---	---	----

String Declaration

Syntax: char string_name[size];

Example: char student_name[20];

Whenever we declare a String, it will contain garbage values inside it. We have to initialize String or Character array before using it. Process of assigning some legal default data to String is Called **Initialization of String**.

There are different ways of initializing String in C Programming –

1. Initializing Unsized Array of Character
2. Initializing String Directly
3. Initializing String Using Character Pointer

1.Initializing Unsized Array of Character

1. **Unsized Array** : Array Length is not specified while initializing character array using this approach
2. Array length is Automatically calculated by Compiler
3. Individual Characters are written inside Single Quotes , Separated by comma to form a list of characters. Complete list is wrapped inside **Pair of Curly braces**
4. **Please Note** : NULL Character should be written in the list because it is ending or terminating character in the String/Character Array

```
char name [] = {'P','R','I','T','E','S','H','\0'};
```

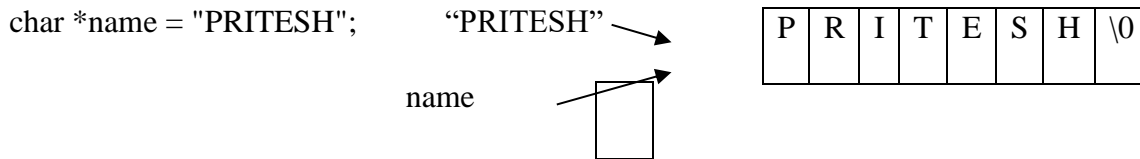
2. Initializing String Directly

1. In this method we are directly assigning String to variable by writing text in double quotes.
2. In this type of initialization, we don't need to put NULL or **Ending / Terminating character** at the end of string. It is appended automatically by the compiler.

```
char name [ ] = "PRITESH";
```

3.Initializing String Using Character Pointer

1. Declare Character variable of pointer type so that it can hold the **base address of “String”**
2. Base address means address of first array element i.e (**address of name[0]**)
3. NULL Character is appended **Automatically**



Topic: String Input / Output functions

- Formatted input/output functions(`printf/scanf`)
- String –only functions(line to string(`gets/fgets`), string to line(`puts/fputs`))

String Input functions:

In C, there are 3 input functions to read a string. They are

1. `scanf()`
2. `getchar()`
3. `gets()`

1. `scanf()`: In C Programming we can use `scanf` function from `stdio.h` header file to read string. `scanf` function is commonly used for accepting string.

1. Format Specifier **%s** is used for Accepting String
2. `scanf()` function accepts only String before Space

Syntax:

```
scanf("%s",Name_Of_String_Variable);
```

Example

```
#include<stdio.h>

int main()
{
    char name[20];
    printf("\nEnter the Name : ");
    scanf("%s",name);
    printf("Name of the Company : %s",name);
    return(0);
}
```

Scan Set conversion code([.....])

The `scanf()` function can have a scanset, which defines a set of characters to be read by the `scanf()`.

You define a scanset by putting the characters inside square brackets.

For example, the following scanset tells `scanf()` to read only the characters X, Y, and Z:

```
%[XYZ]
```

When using a scanset, `scanf()` continues to read characters until it encounters a character outside the scanset.

To specify an inverted set, append `^` in front of the set. `^` tells `scanf()` to read characters that are not in the scanset.

To specify a range in the scan set, use a hyphen. For example, this tells `scanf()` to accept the characters A through Z:

```
%[A-Z]
```

The scanset is case sensitive. To scan for both upper- and lowercase letters, you must specify them individually.

```
#include <stdio.h>

int main(void)
{
    int i; // www.java2s.com
    char str[80], str2[80];

    printf("testing the scanf scan set. %[abcdefg] \n");

    scanf("%d%[abcdefg]%s", &i, str, str2);
    printf("%d %s %s", i, str, str2);

    return 0;
}
```

Result

```
testing the scanf scan set. [abcdefg]
this is a test
2009347448  ?;?w??
```

www.java2s.com

To read a line, we code the `scanf` as

`scanf("%81[^\n]",line);` //scanf reads until it finds the newline and then stops.

2.getchar()

getchar() function is also one of the function which is used to accept the single character from the user.

Syntax

char ch = getchar();

Example

```
#include<stdio.h>
main()
{
char ch[10],c,i=0;
while((c = getchar())!='\n')
{
ch[i]=c;
i++;
}
ch[i]='\0';
printf("Accepted string : %s",ch);
}
```

3.gets()/fgets() line to string)

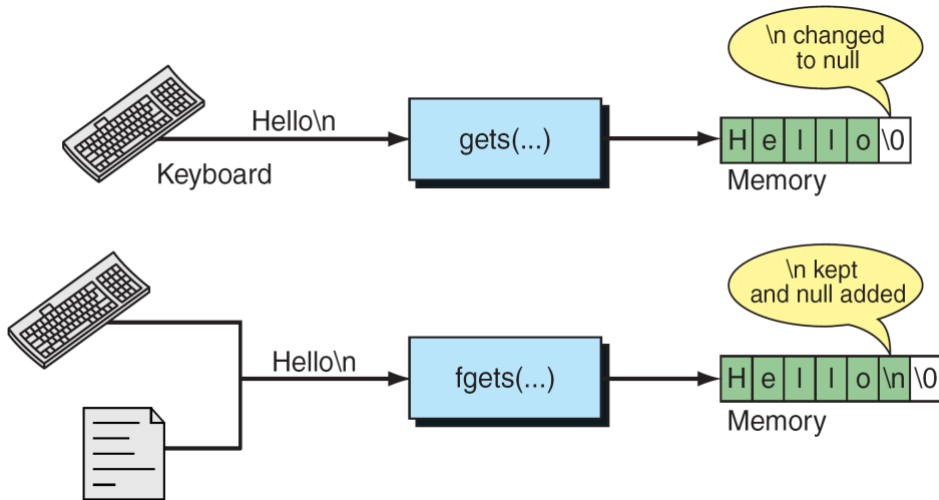
Reads characters from the standard input (stdin) and stores them as a C string into str until a newline character or the end-of-file is reached.

Syntax

Function Declaration

char* gets(**char*** strptr):

The gets function takes a line(terminated by a newline) from the input stream and makes a null-terminated string out of it. The newline is converted to end-of-string character(`\0`).



Example

```
#include<stdio.h>

void main()
{
char name[20];
printf("\nEnter the Name : ");
gets(name);
printf("%s",name);
}
```

String output functions:

C Language supports 3 output functions are there to read a string. They are

1. printf()
2. putchar()
3. puts()

1. printf():

- 1.printf is included in header file “**stdio.h**”
- 2.As name suggest it used for **Printing or Displaying Messages or Instructions**

Syntax

```
printf ( " Type your Message / Instruction " ) ;
```

and

```
printf ("control string ", variable name) ;
```

Example

Justification flag(-) is used to left justify the output. It is used only when a width is also specified, and the length of the string is less than the specified width(minimum width).

```
printf("|%-30s\n","This is a string");
```

Output:

```
|This is a string      |
```

Precision is the maximum number of characters to be printed .

```
Printf("|%-15.14s","12345678901234567890");
```

Output:

```
|12345678901234 |
```

Program to read and display a string:

```
#include<stdio.h>

void main()
{
char str[10];
printf("Enter the String : ");
scanf("%s",str); // Accept String
printf("String is : %s ",str);
}
```

2.putchar():

putchar() is used to print one character at a time. To print more than one character, write the putchar() in loop.

Syntax

```
putchar (char variable name);
```

Example

```
#include< stdio.h>

void main()
{
char string[]="This is an example string\n";
int i=0;
while(string[i]!='\0')
```

```

{
putchar(string[i]);
i++;
}
}

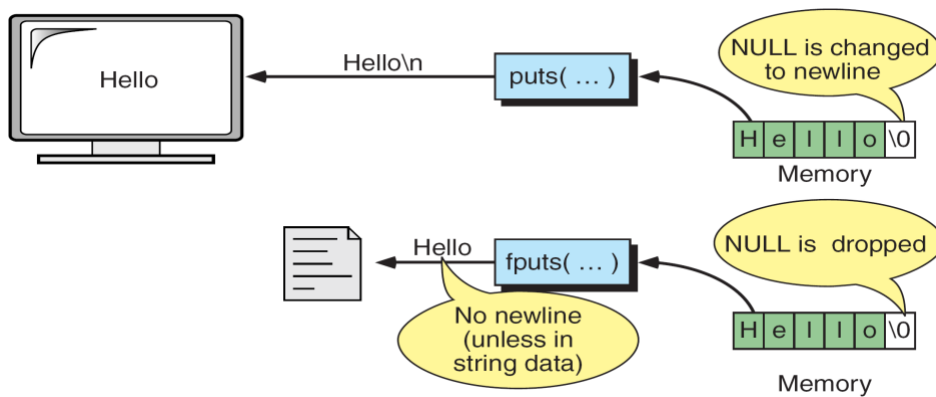
```

3.puts()/fputs()

puts() is used to print multiple characters including white spaces at a time. It takes a null-terminated string from the memory and writes it to a file or the monitor. The null character is replaced by a newline.

Declaration Syntax

```
int puts(const char* strptr);
```



Example

```

#include< stdio.h>

void main()
{
    char string[] = "This is an example string\n";
    puts(string);  // String is variable Here
    puts("String"); // String is in Double Quotes
}

```

Output:

This is an example string
String

Topic: String manipulation functions

Strings handling functions are defined under "string.h" header file.

```
#include <string.h>
```

C – String functions

strlen - Finds out the length of a string
strlwr - It converts a string to lowercase
strupr - It converts a string to uppercase
strcat - It appends one string at the end of another
strncat - It appends first n characters of a string at the end of another.
strcpy - Use it for Copying a string into another
strncpy - It copies first n characters of one string into another
strcmp - It compares two strings
strncmp - It compares first n characters of two strings
strcmpi - It compares two strings without regard to case ("i" denotes that this function ignores case)
stricmp - It compares two strings without regard to case (identical to strcmpi)
strnicmp - It compares first n characters of two strings, Its not case sensitive
strdup - Used for Duplicating a string
strchr - Finds out first occurrence of a given character in a string
strrchr - Finds out last occurrence of a given character in a string
strstr - Finds first occurrence of a given string in another string
strset - It sets all characters of string to a given character
strnset - It sets first n characters of a string to a given character
strrev - It Reverses a string

C String function – strlen

Syntax:

```
size_t strlen(const char *str)
```

size_t represents unsigned short

It returns the length of the string without including end character (terminating char '\0').

Example of strlen:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str1[20] = "Geethanjali";
    printf("Length of string str1: %d", strlen(str1));
    return 0;
}
```

Output:

```
Length of string str1: 11
```

strlen vs sizeof

strlen returns you the length of the string stored in array, however sizeof returns the total allocated size assigned to the array. So if I consider the above example again then the following statements would return the below values.

strlen(str1) returned value 11.

sizeof(str1) would return value 20 as the array size is 20 (see the first statement in main function).

C String function – strlen

Syntax:

```
size_t strlen(const char *str, size_t maxlen)
```

size_t represents unsigned short

It returns length of the string if it is less than the value specified for maxlen (maximum length) otherwise it returns maxlen value.

Example of strlen:

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

```

{
    char str1[20] = " Geethanjali";
    printf("Length of string str1 when maxlen is 30: %d", strnlen(str1,
30));
    printf("Length of string str1 when maxlen is 10: %d", strnlen(str1,
10));
    return 0;
}

```

Output:

Length of string str1 when maxlen is 30: 11

Length of string str1 when maxlen is 10: 10

Have you noticed the output of second printf statement, even though the string length was 11 it returned only 10 because the maxlen was 10.

C String function – strcmp

```

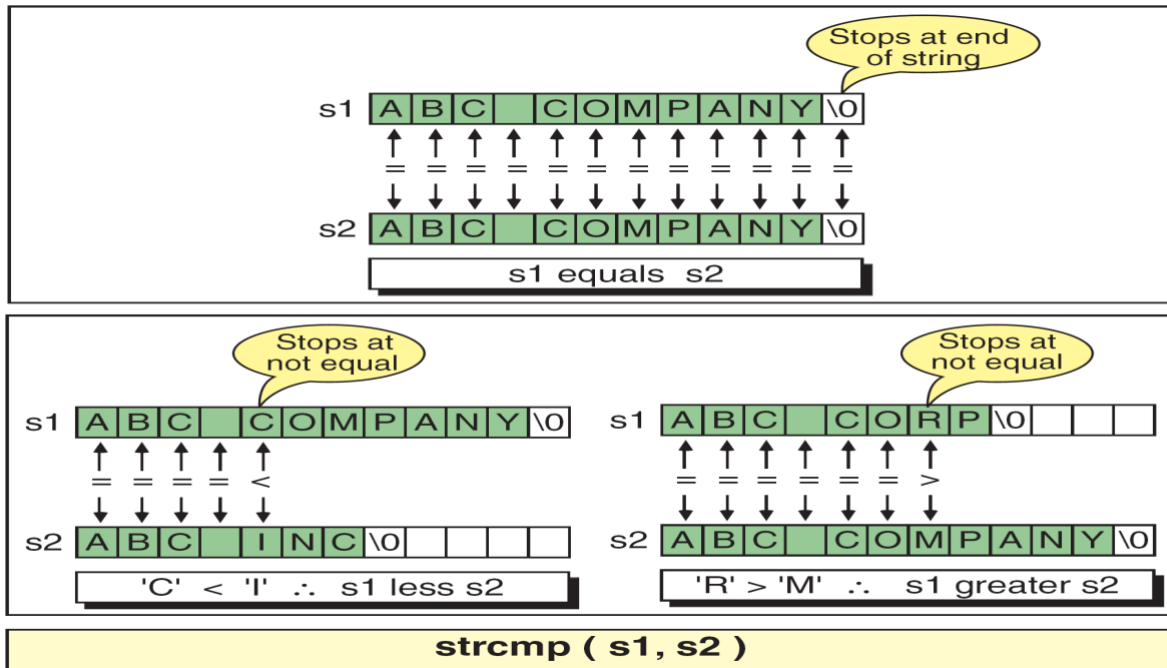
int strcmp(const char *str1, const char *str2)

```

It compares the two strings and returns an integer value. If both the strings are same (equal) then this function would return 0 otherwise it may return a negative or positive value based on the comparison.

If string1 < string2 OR string1 is a substring of string2 then it would result in a negative value. If string1 > string2 then it would return positive value.

If string1 == string2 then you would get 0(zero) when you use this function for compare strings.



Example of strcmp:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[20] = " Geethanjali";
    char s2[20] = " Geethanjali College ";
    if (strcmp(s1, s2) ==0)
    {
        printf("string 1 and string 2 are equal");
    }else
    {
        printf("string 1 and 2 are different");
    }
    return 0;
}
```

Output:

```
string 1 and 2 are different
```

C String function – strcmp

```
int strcmp(const char *str1, const char *str2, size_t n)
```

size_t is for unsigned short

It compares both the string till n characters or in other words it compares first n characters of both the strings.

string1	string2	Size	Results	Returns
"ABC123"	"ABC123"	8	equal	0
"ABC123"	"ABC456"	3	equal	0
"ABC123"	"ABC456"	4	string1 < string2	< 0
"ABC123"	"ABC"	3	equal	0
"ABC123"	"ABC"	4	string1 > string2	> 0
"ABC"	"ABC123"	3	equal	0
"ABC123"	"123ABC"	-1	equal	0

Example of

strcmp:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[20] = " Geethanjali";
    char s2[20] = " Geethanjali College";
    /* below it is comparing first 8 characters of s1 and s2*/
    if (strcmp(s1, s2, 8) ==0)
    {
        printf("string 1 and string 2 are equal");
    }else
    {
        printf("string 1 and 2 are different");
    }
    return 0;
}
```

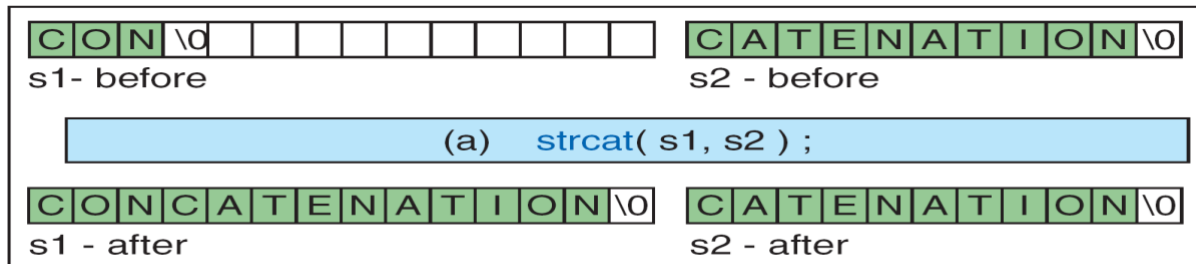
Output:

string1 and string 2 are equal

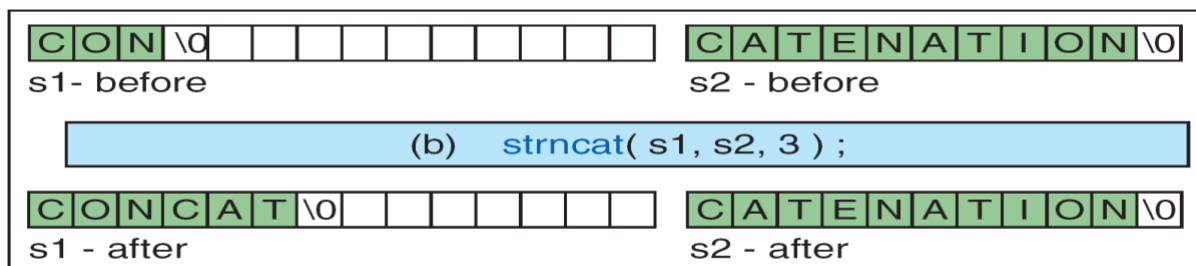
C String function – strcat

```
char *strcat(char *str1, char *str2)
```

It concatenates two strings and returns the concatenated string.



String Concatenate



String N Concatenate

Example of

strcat:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[10] = "Hello";
    char s2[10] = "World";
    strcat(s1,s2);
    printf("Output string after concatenation: %s", s1);
    return 0;
}
```

Output:

```
Output string after concatenation: HelloWorld
```

C String function – strncat

```
char *strncat(char *str1, char *str2, int n)
```

It concatenates n characters of str2 to string str1. A terminator char ('\0') will always be appended at the end of the concatenated string.

Example of strncat:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[10] = "Hello";
    char s2[10] = "World";
    strncat(s1,s2, 3);
    printf("Concatenation using strncat: %s", s1);
    return 0;
}
```

Output:

```
Concatenation using strncat: HelloWor
```

C String function – strcpy

```
char *strcpy( char *str1, char *str2)
```

It copies the string str2 into string str1, including the end character (terminator char '\0').

Example of strcpy:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[30] = "string 1";
    char s2[30] = "string 2 : I'm gonna be copied into s1";
```

```

/* this function has copied s2 into s1*/
strcpy(s1,s2);
printf("String s1 is: %s", s1);
return 0;
}

```

Output:

```
String s1 is: string 2: I'm gonna be copied into s1
```

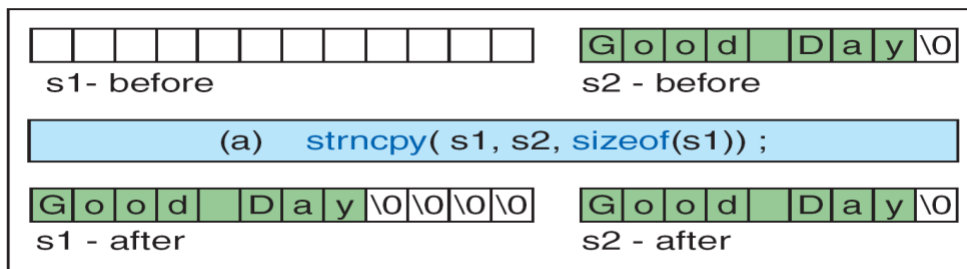
C String function – strncpy

`char *strncpy(char *str1, char *str2, size_t n)`

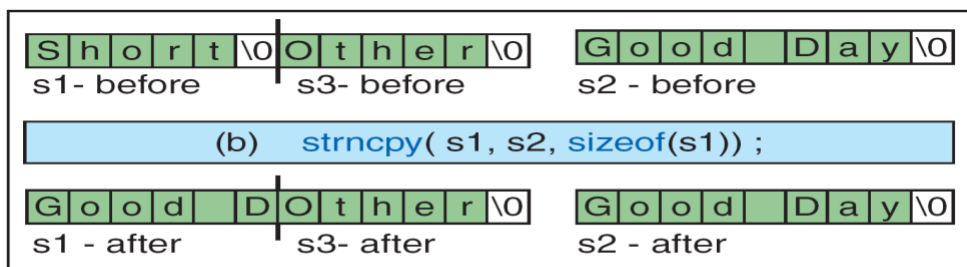
`size_t` is unassigned short and `n` is a number.

Case1: If length of `str2 > n` then it just copies first `n` characters of `str2` into `str1`.

Case2: If length of `str2 < n` then it copies all the characters of `str2` into `str1` and appends several terminator chars(`'\0'`) to accumulate the length of `str1` to make it `n`.



Copying Strings



Copying Long Strings

Example of strncpy:

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

```
#include <string.h>
int main()
{
    char first[30] = "string 1";
    char second[30] = "string 2: I'm using strncpy now";
    /* this function has copied first 10 chars of s2 into s1*/
    strncpy(s1,s2, 12);
    printf("String s1 is: %s", s1);
    return 0;
}
```

Output:

```
String s1 is: string 2: I'm
```

C String function – String reverse

String reverse function reverses the contents of the string.

The general form of the function:

```
strrev(string);
```

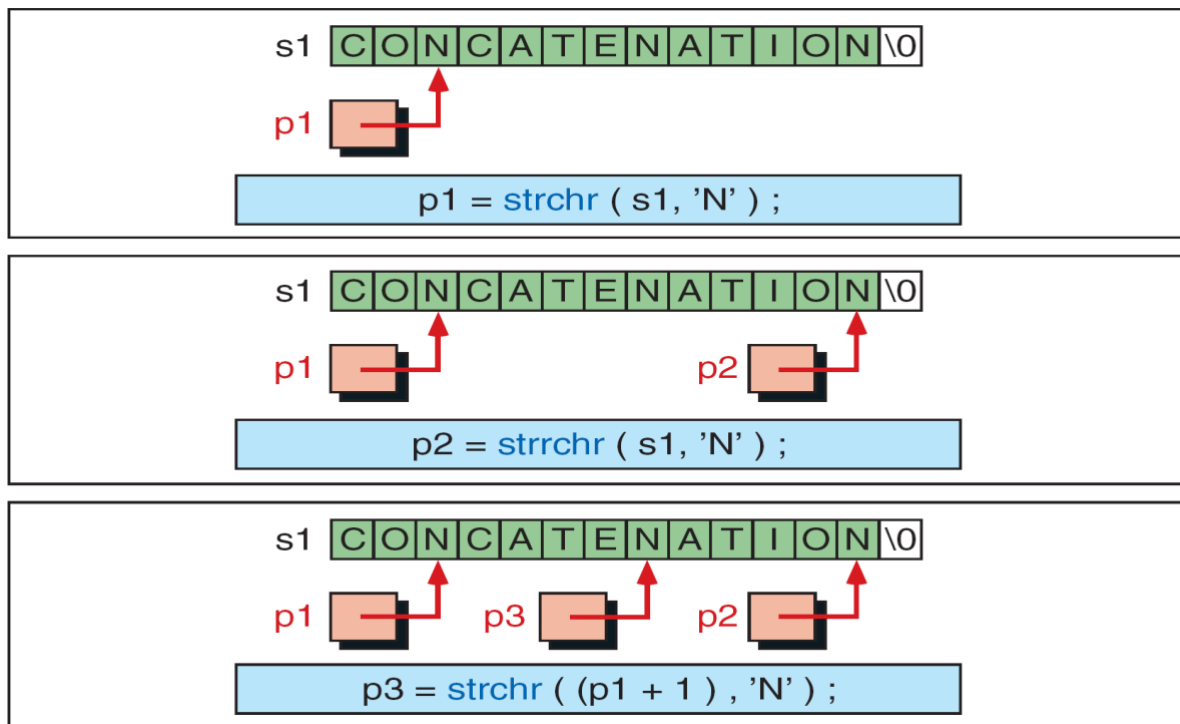
Example:

```
#include<stdio.h>
#include<string.h>
int main()
{
    char s[]="hello";
    strrev(s);
    puts(s);
    return 0;
}
```

C String function – strchr

```
char *strchr(char *str, int ch)
```

It searches string str for character ch (you may be wondering that in above definition I have given data type of ch as int, don't worry I didn't make any mistake it should be int only. The thing is when we give any character while using strchr then it internally gets converted into integer for better searching.



Example of

strchr:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char mystr[30] = "I'm an example of function strchr";
    printf ("%s", strchr(mystr, 'f'));
    return 0;
}
```

Output:

```
f function strchr
```

C String function – Strchr

```
char *strrchr(char *str, int ch)
```

It is similar to the function strchr, the only difference is that it searches the string in reverse order, now you would have understood why we have extra r in strrchr, yes you guessed it correct, it is for reverse only.

Now let's take the same above example:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char mystr[30] = "I'm an example of function strrchr";
    printf ("%s", strrchr(mystr, 'f'));
    return 0;
}
```

Output:

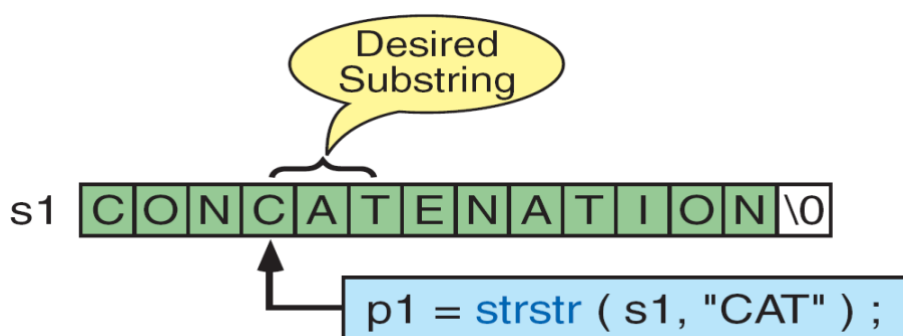
```
function strrchr
```

Why output is different than strchr? It is because it started searching from the end of the string and found the first 'f' in function instead of 'of'.

C String function – strstr

```
char *strstr(char *str, char *srch_term)
```

It is similar to strchr, except that it searches for string srch_term instead of a single char.



Example of strstr:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char inputstr[70] = "String Function in C at Geethanjali";
    printf ("Output string is: %s", strstr(inputstr, 'Geetha'));
    return 0;
}
```

Output:

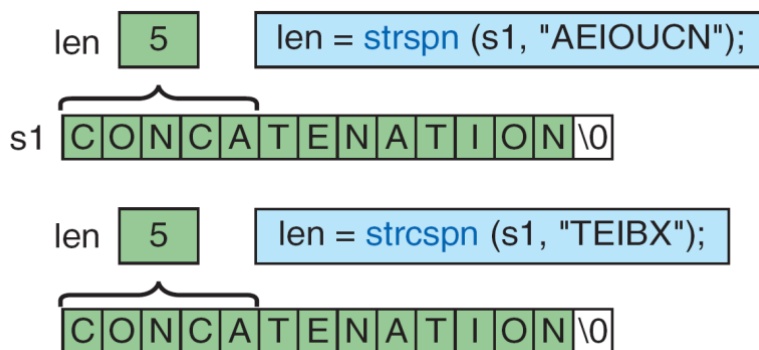
```
Output string is: Geethanjali
```

You can also use this function in place of strchr as you are allowed to give single char also in place of search_term string.

Search for character in Set

- Basic string span:** This function searches the string, spanning character that are in the set and stopping at the first character that is not in the set. They return the number of characters that matched those in the set. If no characters match those in the set, they return zero.

int strspn(const char* str, const char* set);



```
#include<stdio.h>
#include<string.h>
int main()
{
    char str[]="CONCATENATION";
    int len;
```

```

len=strspn(s1,"AEIOUCN");
printf("%d",len);
return 0;
}

```

Output:

5

- b. **Complemented String Span:** This is a string complement span; its functions stops at the first character that matches one of the characters in the set. If none of the characters in the string match the set, they return the length of the string.

```

int strcspn(const char* str, const char* set);
#include<stdio.h>
#include<string.h>
int main()
{
char str[]="CONCATENATION";
int len;
len=strcspn(s1,"TEIBX");
printf("%d",len);
return 0;
}

```

Output:

5

c. String Span-pointer

The function operates just like the string complement function except that it returns a pointer to the first character that matches the set.

```

char* strpbrk(const char* str, const char* set);

```

p stands for pointer

brk stands for break

String Token:

This function is used to locate substrings , called tokens, in a string.

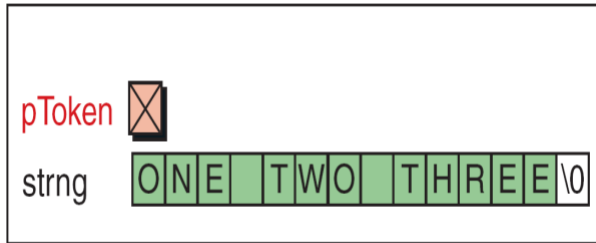
strtok() function in C tokenizes/parses the given string using delimiter.

Syntax for strtok() function is given below.

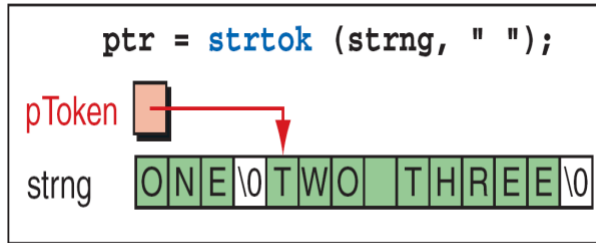
```

char * strtok ( char * str, const char * delimiters );

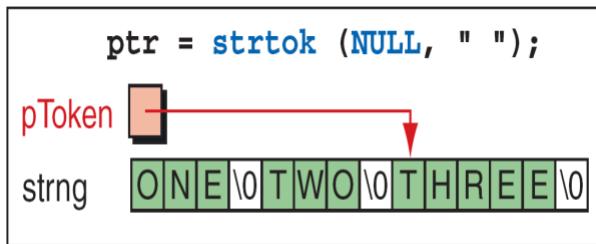
```

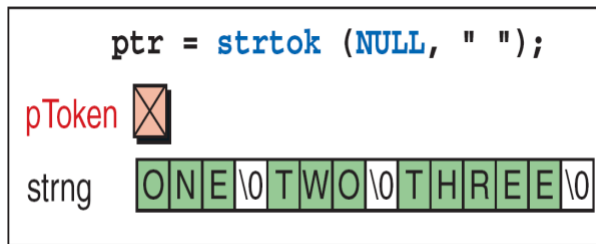
(a) Before Parsing



(b) After First Parsing



(c) After Second Parsing



(d) After Last Parsing

EXAMPLE PROGRAM FOR STRTOK() FUNCTION IN C:

In this program, input string "Test,string1,Test,string2:Test:string3" is parsed using strtok() function. Delimiter comma (,) is used to separate each sub strings from input string.

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char string[50] = "Test,string1,Test,string2:Test:string3";
    char *p;
    printf ("String \"%s\" is split into tokens:\n",string);
    p = strtok (string, ",:");
    while (p!= NULL)
    {
        printf ("%s\n",p);
        p = strtok (NULL, ",:");
    }
    return 0;
}
```

OUTPUT:

```
String "Test,string1,Test,string2:Test:string3" is split into tokens:
Test
string1
Test
string2
Test
string3
```

String to Number:

These functions are available in stdlib.h library.

The C library function long int strtol(const char *str, char **endptr, int base) converts the initial part of the

string in str to a long int value according to the given base, which must be between 2 and 35 inclusive, or be the special value 0. It stops with the first non-numeric character, which is considered to be the start of a trailing string. The address of the trailing string is stored in the second parameter, unless it is a null pointer. A third parameter determines the base of the string number.

Declaration

Following is the declaration for strtol() function.

long int strtol(const char *str, char **endptr, int base);

Parameters

str – This is the string containing the representation of an integral number.

endptr – This is the reference to an object of type char*, whose value is set by the function to the next character in str after the numerical value.

base – This is the base, which must be between 2 and 35 inclusive, or be the special value 0.

The letters a.....z or A.....Z represent the values 10.....35. Only the numeric and alphabetic characters less than the base are permitted in any string.

If the base is 0, the format is determined by the string as follows:

- If the number begins with 0x or oX, the number is a hexadecimal constant.
- If the first digit is 0 and the second digit is not x or X, the number is an octal constant.
- If the first digit is a nonzero, the number is a decimal constant.

Return Value

This function returns the converted integral number as a long int value, else zero value is returned if the string does not begin with a valid number and the trailing string pointer is set to the beginning of the string.

Example

The following example shows the usage of strtol() function.

Ex1:

```
#include <stdio.h>
#include <stdlib.h>
int main () {
    char str[30] = "2030300 This is test";
    char *ptr;
    long ret;
    ret = strtol(str, &ptr, 10);
    printf("The number(unsigned long integer) is %ld\n", ret);
    printf("String part is |%s|", ptr);
    return(0);
}
```

The number(unsigned long integer) is 2030300

String part is | This is test|

Ex2:

```
#include <stdio.h>
#include <stdlib.h>
int main () {
    long num;
    char *ptr;
    num = strtol("12345 Decimal constant:", &ptr, 0);
    printf("%s %ld\n", ptr, num);
    num = strtol("11001 Binary constant:", &ptr, 2);
    printf("%s %ld\n", ptr, num);
    num = strtol("13572 Octal constant:", &ptr, 8);
    printf("%s %ld\n", ptr, num);
}
```

```

num = strtol("7AbC Hex constant:", &ptr, 16);
printf("%s %ld\n", ptr, num);
num = strtol("11001Base 0-Decimal constant:", &ptr, 0);
printf("%s %ld\n", ptr, num);
num = strtol("01101 Base 0-Octal constant:", &ptr, 0);
printf("%s %ld\n", ptr, num);
num = strtol("0x7AbC Base 0-Hex constant:", &ptr, 0);
printf("%s %ld\n", ptr, num);
num = strtol("Invalid Input      :", &ptr, 0);
printf("%s %ld\n", ptr, num);
return(0);
}

```

Decimal Constant:12345

Binary Constant : 25

Octal Constant: 6010

Hex constant: 31420

Base 0- Decimal:11001

Base 0- Octal : 577

Base 0- Hex :31420

Invalid input: 0

Topic : String to Number Conversion

1. atof():Convert a string to an equivalent floating point value.

Syntax: float atof(const char *s);

2.atoi():Convert a string to an equivalent integer value.

Syntax: int atoi(const char *s);

3.atol():Convert a string to an equivalent long integer value.

Syntax: long atoi(const char *s);

4.ecvt():Convert floating point numbers to an equivalent null terminated strings. It specifies the total number of characters in the equivalent string.

Syntax: char *ecvt(double value,int ndig,int *dec,int *sign);

5.fcvt(): Convert floating point numbers to an equivalent null terminated strings. It specifies the equivalent number of characters after the decimal point.

Syntax:char *fcvt(double value,int ndig,int *dec,int *sign);

/*program that converts string to its equivalent integer and floating point value*/

#include<stdio.h>

#include<stdlib.h>

```

void main()
{
char *str="834.41";
printf("string=%s\n\n integer data value=%d\n\n float data value=%.2f",str,atoi(str),atof(str));
}

```

Output:

String=834.41

Integer data value=834

Float data value=834.41

String/Data Conversion

Description

sscanf() function is a file handling function in C programming language which is used to read formatted input from a string/buffer.

The C library function int sscanf(const char *str, const char *format, ...) reads formatted input from a string.

Declaration

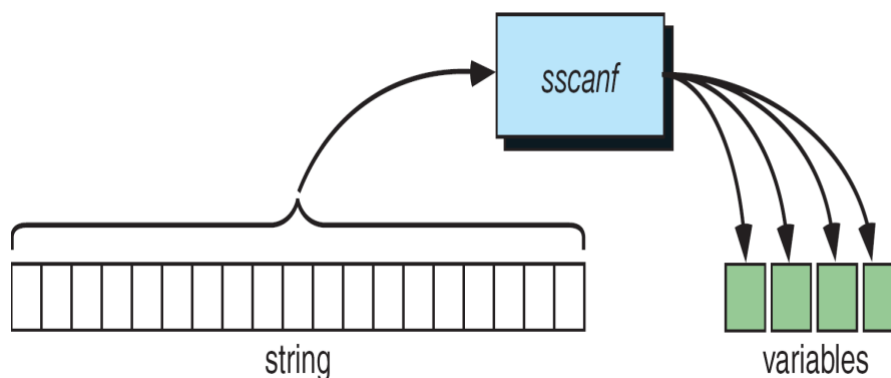
Following is the declaration for sscanf() function.

```
int sscanf(const char *str, const char *format, ...)
```

Parameters

- str – This is the C string that the function processes as its source to retrieve the data.
- format – This is the C string that contains one or more of the following items: Whitespace character, Non-whitespace character and Format specifiers

A format specifier follows this prototype: [=%[*][width][modifiers]type=]



Sr.No.	Argument & Description

1	<p>*</p> <p>This is an optional starting asterisk, which indicates that the data is to be read from the stream but ignored, i.e. it is not stored in the corresponding argument.</p>
2	<p>width</p> <p>This specifies the maximum number of characters to be read in the current reading operation.</p>
3	<p>modifiers</p> <p>Specifies a size different from int (in the case of d, i and n), unsigned int (in the case of o, u and x) or float (in the case of e, f and g) for the data pointed by the corresponding additional argument: h : short int (for d, i and n), or unsigned short int (for o, u and x) l : long int (for d, i and n), or unsigned long int (for o, u and x), or double (for e, f and g) L : long double (for e, f and g)</p>
4	<p>type</p> <p>A character specifying the type of data to be read and how it is expected to be read.</p>

EXAMPLE PROGRAM FOR SSCANF() FUNCTION IN C PROGRAMMING LANGUAGE:

```
/* sscanf example */
#include <stdio.h>
int main ()
{
    char buffer[30]="Geethanjali 5 ";
    char name [20];
    int age;
    sscanf (buffer,"%s %d",name,&age);
    printf ("Name : %s \n Age : %d \n",name,age);
    return 0;
}
```

OUTPUT:

```
Name : Geethanjali
Age : 5
```

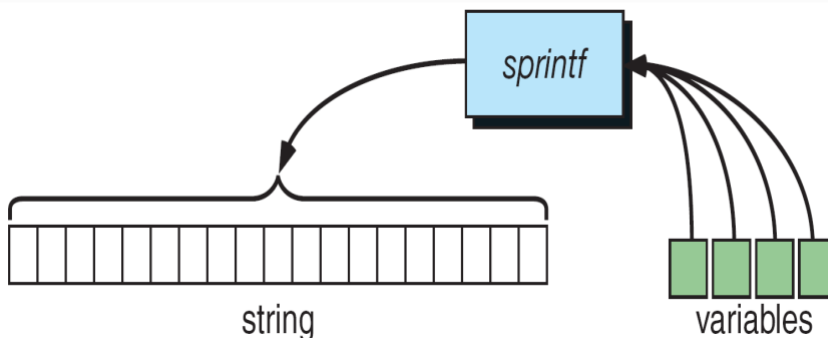
Data/String Conversion

sprintf() function is a file handling function in C programming language which is used to write formatted output

to the string.

- The string function sprintf() follows rules of fprintf.
- General form of the function:

```
int sprintf( char * out_string,const char * format_string,---);
```



File operation	Declaration & Description
sprintf()	<p>Declaration: int sprintf(char *string, const char *format, ...)</p> <p>sprintf function is used to write formatted output to the string, In a C program, we use fgets function as below. sprintf (string, “%d %c %f”, value, c, flt) ;</p> <p>where, string – buffer to put the data in. value – int variable, c – char variable and flt – float variable. There are for example only. You can use any specifiers.</p>

EXAMPLE PROGRAM FOR SPRINTF() FUNCTION IN C PROGRAMMING LANGUAGE:

```
#include <stdio.h>
#include <string.h>
int main( )
{
    int value = 50 ;
    float flt = 7.25 ;
    char c = 'Z' ;
    char string[40] = {‘\0’} ;
    printf ( "int value = %d \n char value = %c \n " \
            "float value = %f", value, c, flt ) ;
```

```

/*Now, all the above values are redirected to string
instead of stdout using sprint*/
printf("\n Before using sprint, data in string is %s", string);
sprintf ( string, "%d %c %f", value, c, flt );
printf("\n After using sprint, data in string is %s", string);
return 0;
}

```

OUTPUT:

```

int value = 50
char value = Z
float value = 7.25
Before using sprint, data in string is NULL
After using sprint, data in string is "50 Z 7.25"

```

Topic:Arrays of strings

An array of strings is a special form of a two-dimensional array. • The size of the left index determines the number of strings. • The size of the right index specifies the maximum length of each string. For example, the following declares an array of 30 strings, each having a maximum length of 80 characters (with one extra character for the null terminator): Arrays of strings are commonly used for handling tables of information.

Two Dimensional Array Of Strings

It is also referred as table of strings. This can be initialized as follows:

```
type variable-name[][];
```

- char a[10][10]; // declaration of 2-D array
- char b[2][2] = { "Andhra Pradesh", "Karnataka"}; // initialization of 2-D array

The first subscript gives the number of names in the array. The second subscript gives the length of each item of the array.

Example:

```
char list[6][10]={ "akshay", "parag", "raman", "srinivas", "gopal", "rajesh"};
```

The names would be stored in the memory as shown in the below figure

Example: Two dimensional arrays of strings

65454	a	k	s	h	a	y	\0			
65464	p	a	r	a	g	\0				
65474	r	a	m	a	n	\0				
65484	s	r	i	n	i	v	a	s	\0	
65494	g	o	p	a	l	\0				
65504	r	a	j	e	s	h	\0			

C program to read N names, store them in the form of an array and sort them in alphabetical order. Output the given names and the sorted names in two columns side by side.

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

void main()
{
char name[10][8], tname[10][8], temp[8];
int i, j, n;

printf("Enter the value of n \n");
scanf("%d",&n);
printf("Enter %d names n", \n);
for(i =0; i < n; i++)
{
scanf("%s", name[i]);
strcpy(tname[i], name[i]);
}
for(i =0; i < n -1; i++)
{
for(j = i +1; j < n; j++)
{
if(strcmp(name[i], name[j])>0)
{
```



```

strcpy(temp, name[i]);
strcpy(name[i], name[j]);
strcpy(name[j], temp);
}
}
}

printf("\n-----\n");
printf("Input NamesSorted names\n");
printf("-----\n");
for(i =0; i < n; i++)
{
printf("%s\t\t%s\n", tname[i], name[i]);
}
printf("-----\n");
}

```

C program examples.

1.C program to read line of text character by character.

```

#include<stdio.h>

int main()
{
char name[30], ch;
int i =0;
printf("Enter name: ");
while(ch !='\n')// terminates if user hit enter
{
ch= getchar();
name[i]= ch;
i++;
}
name[i]='\0';// inserting null character at end

printf("Name: %s", name);

```

```
return0;
```

```
}
```

2. C Program to Replace Lowercase Characters by Uppercase & Vice-Versa

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

```
#include <ctype.h>
```

```
void main()
```

```
{
```

```
char sentence[100];
```

```
int count, ch, i;
```

```
printf("Enter a sentence \n");
```

```
for(i =0;(sentence[i]=getchar())!='\n'; i++)
```

```
    sentence[i]='\0';
```

```
/* shows the number of chars accepted in a sentence */
```

```
    count = i;
```

```
printf("The given sentence is  : %s", sentence);
```

```
printf("\n Case changed sentence is: ");
```

```
for(i =0; i < count; i++)
```

```
{
```

```
ch=islower(sentence[i])?toupper(sentence[i]):
```

```
tolower(sentence[i]);
```

```
putchar(ch);
```

```
}
```

```
}
```

3. C program to count the number of vowels, consonants

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

```
int main()
```

```
{
```

```
char line[150];
```

```
int i, vowels, consonants, digits, spaces;
```

```
vowels= consonants = digits = spaces =0;
```

```

printf("Enter a line of string: ");
scanf("%[^\\n]", line);

for(i=0; line[i]!='\\0';++i)
{
if(line[i]=='a' || line[i]=='e' || line[i]=='i' ||
line[i]=='o' || line[i]=='u' || line[i]=='A' ||
line[i]=='E' || line[i]=='I' || line[i]=='O' ||
line[i]=='U')
{
++vowels;
}
elseif((line[i]>='a' && line[i]<='z') || (line[i]>='A' && line[i]<='Z'))
{
++consonants;
}
elseif(line[i]>='0' && line[i]<='9')
{
++digits;
}
elseif(line[i]==' ')
{
++spaces;
}
}

printf("Vowels: %d",vowels);
printf("\\nConsonants: %d",consonants);
printf("\\nDigits: %d",digits);
printf("\\nWhite spaces: %d", spaces);
return 0;
}

```

4. C Program to Find the Length of a String

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

```

int main()
{
char s[1000], i;

printf("Enter a string: ");
scanf("%s", s);

for(i =0; s[i]!='\0';++i);

printf("Length of string: %d", i);
return 0;
}

```

4. C Program to Copy String Without Using strcpy()

```

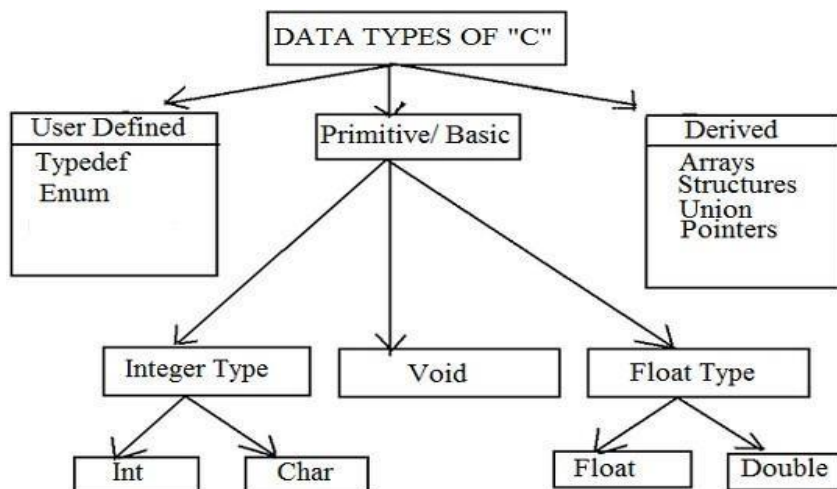
#include<stdio.h>

int main()
{
char s1[100], s2[100], i;

printf("Enter string s1: ");
scanf("%s",s1);

for(i =0; s1[i]!='\0';++i)
{
s2[i]= s1[i];
}
s2[i]='\0';
printf("String s2: %s", s2);
return 0;
}

```



User Defined Data Types

Typedef

The typedef is an advance feature in C language which allows us to create an alias or new name for an existing type or user defined type.

The syntax of typedef is as follows:

Syntax: typedef data_type new_name;

- typedef: It is a keyword.
- data_type: It is the name of any existing type or user defined type created using structure/union.
- new_name: alias or new name you want to give to any existing type or user defined type.

Let's take an example:

```
typedef int myint;
```

Now myint is an alias of int. From now on we can declare new int variables using myint instead of int keyword.

```
myint i = 0; // this statement is equivalent to int i = 0;
```

This statement declares and initializes a variable `i` of type `int`.

We can even create more than one alias for the same type. For example:

```
typedef int myint, integer;
```

This statement creates two aliases for type `int` namely `myint` and `integer`.

Here are some more examples:

```
typedef unsigned long int ulint;
```

```
typedef float real;
```

After these two declarations, `ulint` is an alias of unsigned long int and `real` is an alias of float.

We can write typedef declaration anywhere other declarations are allowed. However, it is important to note that the scope of the declarations depends on the location of the typedef statement. If the definition is placed outside all functions then the scope is global and any function can use an alias instead of the original name. On the other hand, if the definition is declared inside a function then the scope is local and the only the function which contains the typedef statement can use an alias.

Enumerated types(enum)

enum is a user defined enumerated data type supported by ANSI C.

Syntax:

```
enum identifier {value1,value2,...value n};
```

The identifier is a user defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces. After this definition we can declare variables of new type as

```
enum identifier v1,v2,...vn;
```

The enumerated variables `v1,v2,...vn` can only have only one of the values `value1,value2,...valuen`.

```
v1=value3;
```

```
v5=value1;
```

Example:

```
enum day {Monday,Tuesday,...Sunday};
```

```
enum day week_st,week_end;
```

```
week_st=Monday;
```

```
week_end=Friday;
```

```
If(week_st==Tuesday)
```

```
week_end=Saturday;
```

C provides the **enum** construction for the special case where you want to have a sequence of named constants of type int, but you don't care what their actual values are, as in

```
enum color { RED, BLUE, GREEN, MAUVE, TURQUOISE };
```

This will assign the value 0 to RED, 1 to BLUE, and so on. These values are effectively of type int, although you can declare variables, arguments, and return values as type **enum** color to indicate their intended interpretation.

Despite declaring a variable **enum** color c (say), the compiler will still allow c to hold arbitrary values of type int.

So the following code works just fine:

```
#include<stdio.h>
#include<stdlib.h>
enum foo { FOO };
enum apple { MACINTOSH, CORTLAND, RED_DELICIOUS };
enum orange { NAVEL, CLEMENTINE, TANGERINE };
int main(int argc, char **argv)
{
enum foo x;
if(argc != 1)
{
printf( "Usage: %s\n", argv[0]);
return 1;
}
printf("FOO = %d\n", FOO);
printf("sizeof(enum foo) = %d\n", sizeof(enum foo));
x = 127;
printf("x = %d\n", x); /* note we can add apples and oranges */
printf("%d\n", RED_DELICIOUS + TANGERINE);
return 0;
}
```

Specifying particular values

It is also possible to specify particular values for particular enumerated constants, as in

```
enum color { RED = 37, BLUE = 12, GREEN = 66, MAUVE = 5, TURQUOISE };
```

Anything that doesn't get a value starts with one plus the previous value; so the above definition would set TURQUOISE to 6. This may result in two names mapping to the same value.

In practice, enums are seldom used, and we will more commonly see a stack of #defines:

```
#define RED (0)
#define BLUE (1)
#define GREEN (2)
#define MAUVE (3)
#define TURQUOISE (4)
```

The reason for this is partly historical—enum arrived late in the evolution of C—but partly practical: a table of #defines makes it much easier to figure out which color is represented by 3, without having to count through a list. But if you never plan to use the numerical values, enum may be a better choice, because it guarantees that all the values will be distinct.

Structures and Union

STRUCTURES

A structure is a user defined data type. Structure is a collection of different data types referenced under one name, which keeps related information together. The variables which make up a structure are called structure elements.

Structure Declaration

C keyword struct is used for defining structure

```
struct addr
{
char name[30];
char street[40];
int postcode;
};
```

In this example, addr is known as the structure tag. A structure definition is terminated by a semicolon.

A structure variable declaration for type addr is:

```
struct addr addr_info;
```

Tagged Structure:

Variables may also be declared as the structure is defined. This can be done as follows:

```

struct addr
{
char name[30];
char street[40];
int postcode;
} addr_info, binfo, cinfo;

```

Structure Variable

The structure tag is not needed, if only one variable is declared for example:

```

struct
{
char name[30];
char street[40];
int postcode;
} addr_info;

```

Tagged Structure

```

struct book
{
    char title[20];
    char publisher[20];
    char author[20];
    int year;
    int pages;
};
typedef struct book Book;

```

After this declaration, Book is an alias of struct book. So instead of using struct book to declare new structure variables we can use just use Book.

```
Book b1 = {"The Alchemist", "TDM Publication", "Paulo Coelho", "1978", 331 };
```

We can also combine structure definition and typedef declaration.

The syntax to do so is:

```

typedef struct tagname
{
    data_type member1;
    data_type member1;
    ...
} newname;

```

Let's rewrite structure book definition using this new syntax of typedef.

```

typedef struct book
{

```

```
char title[20];
char publisher[20];
char author[20];
int year;
int pages;
} Book;
```

Initialization

A structure is initialized in a way as any other data type in C

```
struct addr addr_info = {-XYZ, -Paradise, 500056};
```

C compiler will automatically assign

```
name=XYZ
```

```
street=Paradise
```

```
postalcode=500056
```

Accessing Structures

The only operation that can be applied to struct variable is assignment. If a and b are structure variables and if both of them consist of same member name then one variables member value can be copied to another variables member.

Complex structures

A Structure can be defined to be a group of logically related data items, which may be of different types, stored in contiguous memory locations, sharing a common name, but distinguished by its member. So far structures used built-in data types char, int, float, etc. Let us now explore the possibility of making a group of data items of user-defined types also.

```
struct date
```

```
{
-
-
-
};
```

```
struct student
```

```
{
-
-
-
}
```

```
struct date doa;
```

```
};
```

Here the structure data is also a variable of structure student.

Nested Structure

Nested structure in C is nothing but structure within structure. One structure can be declared inside other structure as we declare structure members inside a structure.

The structure variables can be a normal structure variable or a pointer variable to access the data. You can learn below concepts in this section.

1. Structure within structure in C using normal variable
2. Structure within structure in C using pointer variable

1. Structure Within Structure In C Using Normal Variable:

This program explains how to use structure within structure in C using normal variable. -student_college_detail' structure is declared inside -student_detail structure in this program. Both structure variables are normal structure variables.

Please note that members of -student_college_detail structure are accessed by 2 dot(.) operator and members of -student_detail structure are accessed by single dot(.) operator.

```
#include <stdio.h>
#include <string.h>
struct student_college_detail
{
    int college_id;
    char college_name[50];
};
struct student_detail
{
    int id;
    char name[20];
    float percentage;
    // structure within structure
    struct student_college_detail clg_data;
}stu_data;

int main()
{
    struct student_detail stu_data = { 1, "Raju", 90.5, 71145,"Osmania University"};
    printf(" Id is: %d \n", stu_data.id);
    printf(" Name is: %s \n", stu_data.name);
    printf(" Percentage is: %f \n\n", stu_data.percentage);
    printf(" College Id is: %d \n", stu_data.clg_data.college_id);
    printf(" College Name is: %s \n", stu_data.clg_data.college_name);

    return 0;
}
```

Output

Id is: 1
Name is: Raju
Percentage is: 90.500000
College Id is: 71145
College Name is: Osmania University

2. Structure Within Structure In C Using Pointer Variable:

This program explains how to use structure within structure in C using pointer variable. -student_college_detail' structure is declared inside -student_detail structure in this program. one normal structure variable and one pointer structure variable is used in this program.

Please note that combination of .(dot) and ->(arrow) operators are used to access the structure member which is declared inside the structure.

```
#include <stdio.h>
#include <string.h>
struct student_college_detail
{
    int college_id;
    char college_name[50];
};
struct student_detail
{
    int id;
    char name[20];
    float percentage;
    // structure within structure
    struct student_college_detail clg_data;
} stu_data, *stu_data_ptr;
int main()
{
    struct student_detail stu_data = { 1, "Raju", 90.5, 71145, "Osmania University" };
    stu_data_ptr = &stu_data;
    printf(" Id is: %d \n", stu_data_ptr->id);
    printf(" Name is: %s \n", stu_data_ptr->name);
    printf(" Percentage is: %f \n\n", stu_data_ptr->percentage);
    printf(" College Id is: %d \n", stu_data_ptr->clg_data.college_id);
    printf(" College Name is: %s \n", stu_data_ptr->clg_data.college_name);
    return 0;
}
```

Output

Id is: 1
Name is: Raju
Percentage is: 90.500000
College Id is: 71145
College Name is: Osmania University

Structures and Functions

For structures to be fully useful, we should pass them to functions and return them.

The members of the structure can be accessed by function in 3 ways:

1. Passing individual members to the function
2. Passing whole structure to the function where the members of the structure can be accessed using pass by value.
3. Passing the address of structure and function can access members through indirection and indirect selection operators using pass by reference or passing structures through pointers

Passing Individual Members to the Function

In order to refer the individual members for Actual parameters, we must use the direct selection operator (.) to refer the individual members of the structure.

struct fraction

```
{
int nr,dr;
.....
.....
};
main()
{
struct fraction fr1,fr2,res;
.....
.....
res.nr=multiply(fr1.nr,fr2.nr);
res.dr=multiply(fr1.dr,fr2.dr);
}
multiply(int x,int y)
{
return x*y;
}
```

Example:

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

```
/*
```

structure is defined above all functions so it is global.

```
*/
```

```
struct student
```

```
{
char name[20];
int roll_no;
int marks;
};
```

```
void print_struct(char name[], int roll_no, int marks);
```

```

int main()
{
    struct student stu = {"Tim", 1, 78};
    print_struct(stu.name, stu.roll_no, stu.marks);
    return 0;
}

void print_struct(char name[], int roll_no, int marks)
{
    printf("Name: %s\n", name);
    printf("Roll no: %d\n", roll_no);
    printf("Marks: %d\n", marks);
    printf("\n");
}

```

Passing Whole Structure To The Function

Passing the whole structure is a much better solution to complete the job in one call. since structure is a type , we simply specify the type in formal parameters of the called function. Similarly the function can return the structure. It is necessary to specify the structure as return type.

```

struct student
{
    int sno;
    float per;
};

Main()
{
    struct student s1;
    .....
    .....
    Display(s1);
}

Display(struct student a);
{
    .....
    .....
}

Example:
#include<stdio.h>

/*
structure is defined above all functions so it is global.
*/

```

```

struct student

```

```

{
    char name[20];
    int roll_no;
    int marks;
};

void print_struct(struct student stu);

int main()
{
    struct student stu = {"George", 10, 69};
    print_struct(stu);
    return 0;
}

void print_struct(struct student stu)
{
    printf("Name: %s\n", stu.name);
    printf("Roll no: %d\n", stu.roll_no);
    printf("Marks: %d\n", stu.marks);
    printf("\n");
}

```

Passing Structures Through Pointers

Pointer variable holds the address of other variables of basic data types such as integer, float, character etc.

A pointer also holds the address of structure variables. Pointers along with structures are used to make linked list, binary trees etc. Following declaration shows the pointer as an object of a structure.

```

struct data
{
    char name[30];
    ---
    ---
};
struct data *p1;

```

*p1 is a pointer variable which holds the address of the structure named data. Pointer is declared as any object of a structure. Pointer structure variable can be accessed and processed by two methods.

- (*p1).fieldname=variable
- p1->fieldname=variable

Example

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

```

/*
structure is defined above all functions so it is global.
*/

struct employee
{
    char name[20];
    int age;
    char doj[10]; // date of joining
    char designation[20];
};

void print_struct(struct employee *);

int main()
{
    struct employee dev = {"Jane", 25, "25/2/2015", "Developer"};
    print_struct(&dev);

    return 0;
}

void print_struct(struct employee *ptr)
{
    printf("Name: %s\n", ptr->name);
    printf("Age: %d\n", ptr->age);
    printf("Date of joining: %s\n", ptr->doj);
    printf("Age: %s\n", ptr->designation);
    printf("\n");
}

```

Self Referential Structures

A self referential structure creates data structures like linked lists, stacks, etc.

Following is an example :

```

struct tag_name
{
    datatype datatype_name;
    struct tag_name * pointer_name;
};

```

A self-referential structure is one which refers to another structure of same type. For example, a linked list is a self-referential data structure in which next node of a node is being pointed to same struct type. For example,

```

struct list_node

```

```
{
int data;
struct list_node *next;
} linked_list;
```

In the above example, the list_node is a self-referential structure – because the *next is of the type struct list_node.

Example

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

```
struct node {
    int data1;
    char data2;
    struct node* link;
};
```

```
int main()
```

```
{
    struct node ob1; // Node1
```

```
    // Intialization
```

```
    ob1.link = NULL;
```

```
    ob1.data1 = 10;
```

```
    ob1.data2 = 20;
```

```
    struct node ob2; // Node2
```

```
    // Initialization
```

```
    ob2.link = NULL;
```

```
    ob2.data1 = 30;
```

```
    ob2.data2 = 40;
```

```
    // Linking ob1 and ob2
```

```
    ob1.link = &ob2;
```

```
    // Accessing data members of ob2 using ob1
```

```
    printf("%d", ob1.link->data1);
```

```
    printf("\n%d", ob1.link->data2);
```

```
    return 0;
```

```
}
```

Output:

30

40

UNIONS

Unions are similar to structures and have same syntax as structures. But the difference between them is in terms of storage. Structure members have their its own storage locations, whereas all members of union use the same location. That is, union can handle only one member at a time and that depends on highest size. Union can be declared sing the keyword union

Declaring Union

```
union union-name
{
    data_type var-name;
    data_type var-name;
};
```

Example:

```
union student
{
    int id;
    float marks;
}code;
```

This declares a variable code of type union item. Union contains three members. For accessing a union member, we use the same syntax that we use for structure variables.

That is code.id , code.marks

Accessing a Member of a Union

```
#include <stdio.h>
#include <string.h>
union Data
{
    int i;
    float f;
    char str[20];
};
int main( )
{
    union Data data;
    data.i = 10;
    data.f = 220.5;
    strcpy( data.str, "C Programming");
    printf( "data.i : %d\n", data.i);
    printf( "data.f : %f\n", data.f);
    printf( "data.str : %s\n", data.str);
    return 0;
}
```

The **union tag** is optional and each member definition is a normal variable definition, such as `int i;` or `float f;` or any other valid variable definition. At the end of the union's

definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named `Data` which has the three members `i`, `f`, and `str`. Now, a variable of **`Data`** type can store an integer, a floating-point number, or a string of characters. This means that a single variable ie., same memory location can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the example `Data` type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by character string.

Dot operator can be used to access a member of the union . The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use **union** keyword to define variables of union type.

Difference between unions and structures

Let's take an example to demonstrate the difference between unions and structures:

```
#include <stdio.h>
union unionJob
{
    //defining a union
    char name[32];
    float salary;
    int workerNo;
} uJob;

struct structJob
{
    char name[32];
    float salary;
    int workerNo;
} sJob;

int main()
{
    printf("size of union = %d bytes", sizeof(uJob));
    printf("\nsize of structure = %d bytes", sizeof(sJob));
    return 0;
}
```

Output

```
size of union = 32
size of structure = 40
```

Why this difference in size of union and structure variables?

The size of structure variable is 40 bytes. It's because:

size of name[32] is 32 bytes

size of salary is 4 bytes

size of workerNo is 4 bytes

However, the size of union variable is 32 bytes. It's because the size of union variable will always be the size of its largest element. In the above example, the size of largest element (name[32]) is 32 bytes.

Only one union member can be accessed at a time

You can access all members of a structure at once as sufficient memory is allocated for all members. However, it's not the case in unions. Let's see an example:

```
#include <stdio.h>
union Job
{
    float salary;
    int workerNo;
} j;
int main()
{
    j.salary = 12.3;
    j.workerNo = 100;
    printf("Salary = %.1f\n", j.salary);
    printf("Number of workers = %d", j.workerNo);
    return 0;
}
```

When you run the program, the output will be:

Salary = 0.0

Number of workers = 100

Notice that, we couldn't store 12.3 in j.salary.

BIT FIELDS

Till now we have used integer fields of size 16 bits to store data. But there are some situations where data variables require much less than 16 bits of space. In such cases we waste memory space. To avoid this we use bit fields whose size can be from 1 to 16 bits in length.

Syntax:

```
struct tagname
{
    datatype var1:bit_length;
    ...
    datatype var n:bit_length
}
```

The data type is either int or unsigned int or signed int and the bit length is number of bits used.

Example: Personal information of employees

The range of values for each field is as follows :

Bit field	bit length	range of values
sex	1	0 or 1
age	7	0 or 127(2^7-1)
m_status	1	0 or 1
children	3	0 to 7(2^3-1)

To assign values to bitfields :

```
emp.sex=1  
emp.age=50;
```

We cannot use scanf() to read values into a bit field. We have to read into a temporary variable and then assign its value to bit field.

For example

```
scanf("%d%d",&eage,&echildren);  
emp.age=eage;  
emp.children=echildren;
```

One restriction in accessing bit fields is that a pointer cannot be used. It is possible to combine normal structure elements with bit field elements.

For example:

```
struct personal  
{  
char name[20];  
unsigned sex : 1;  
unsigned age : 7;  
....  
....  
}emp[100];
```

This declares emp as a 100 element array of type struct personal.

Example

Declaration of date without use of bit fields.

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

```
// A simple representation of date
```

```
struct date  
{  
    unsigned int d;  
    unsigned int m;  
    unsigned int y;  
};
```

```

int main()
{
    printf("Size of date is %d bytes\n", sizeof(struct date));
    struct date dt = {31, 12, 2014};
    printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
}

```

Output:

Size of date is 12 bytes

Date is 31/12/2014

Declaration of date with use of bit fields.

```

#include <stdio.h>
// A space optimized representation of date
struct date
{
    // d has value between 1 and 31, so 5 bits are sufficient
    unsigned int d: 5;
    // m has value between 1 and 12, so 4 bits are sufficient
    unsigned int m: 4;
    unsigned int y;
};

```

```

int main()
{
    printf("Size of date is %d bytes\n", sizeof(struct date));
    struct date dt = {31, 12, 2014};
    printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
    return 0;
}

```

Output:

Size of date is 8 bytes

Date is 31/12/2014

Following are some interesting facts about bit fields in C.

1) A special unnamed bit field of size 0 is used to force alignment on next boundary. For example consider the following program.

```

#include <stdio.h>
// A structure without forced alignment
struct test1
{
    unsigned int x: 5;
    unsigned int y: 8;
};

```

```
// A structure with forced alignment
struct test2
{
    unsigned int x: 5;
    unsigned int: 0;
    unsigned int y: 8;
};
int main()
{
    printf("Size of test1 is %d bytes\n", sizeof(struct test1));
    printf("Size of test2 is %d bytes\n", sizeof(struct test2));
    return 0;
}
```

Output:

Size of test1 is 4 bytes

Size of test2 is 8 bytes

2) We cannot have pointers to bit field members as they may not start at a byte boundary.

```
#include <stdio.h>
struct test
{
    unsigned int x: 5;
    unsigned int y: 5;
    unsigned int z;
};
int main()
{
    struct test t;
    // Uncommenting the following line will make the program compile and run
    printf("Address of t.x is %p", &t.x);
    // The below line works fine as z is not a bit field member
    printf("Address of t.z is %p", &t.z);
    return 0;
}
```

Output:

error: attempt to take address of bit-field structure member 'test::x'

3) It is implementation defined to assign an out-of-range value to a bit field member.

```
#include <stdio.h>
struct test
{
    unsigned int x: 2;
    unsigned int y: 2;
    unsigned int z: 2;
```

```
};
int main()
{
    struct test t;
    t.x = 5;
    printf("%d", t.x);
    return 0;
}
```

Output:Implementation-Dependent

4) Array of bit fields is not allowed. For example, the below program fails in compilation.

```
struct test
{
    unsigned int x[10]: 5;
};
int main()
{

}
```

Output:error: bit-field 'x' has invalid type

COMMAND –LINE ARGUMENTS

Sometimes it is very useful to pass information into a program when we run it from the command prompt. The general method to pass information into main() function is through the use of command line arguments. It follows the program's name on the command prompt of the operating system.

For example: TC program_name

There are three special built_in_arguments to main().

They are:

The first argument is argc (argument count) must be an integer value, which represents the number arguments in the command prompt. It will always be at least one because the name of the program qualifies as the first argument.

- The second argument argv (argument vector) is a pointer to an array of strings.
- The third argument env (environment data) is used to access the DOS environmental parameters active at the time the program begins execution.

When an array is used as an argument to function, only the address of the array is passed, not a copy of the entire array. When a function is called with an array name, a pointer to the first element in the array is passed into a function. (In C, an array name without as index is a pointer to the first element in the array).Each of the command line arguments must be separated by a space or a tab.

Declaration of argv must be done properly, A common method is:

```
char *argv[];
```

That is, as a array of undetermined length.

The env parameter is declared the same as the argv parameter, it is a pointer to an array of strings that contain environmental setting.

Recall that argv in main is declared as char **; this means that it is a pointer to a pointer to a char, or in this case the base address of an array of pointers to char, where each such pointer references a string. These strings correspond to the command-line arguments to your program, with the program name itself appearing in argv[0]

The count argc counts all arguments including argv[0]; it is 1 if your program is called with no arguments and larger otherwise.

Example: Program that prints its arguments.

```
#include<stdio.h>
int main(int argc, char **argv)
{
    int i;
    printf("argc = %d\n\n", argc);
    for(i = 0; i < argc; i++)
    {
        printf("argv[%d] = %s\n", i, argv[i]);
    }
    return 0;
}
```

Preprocessor commands

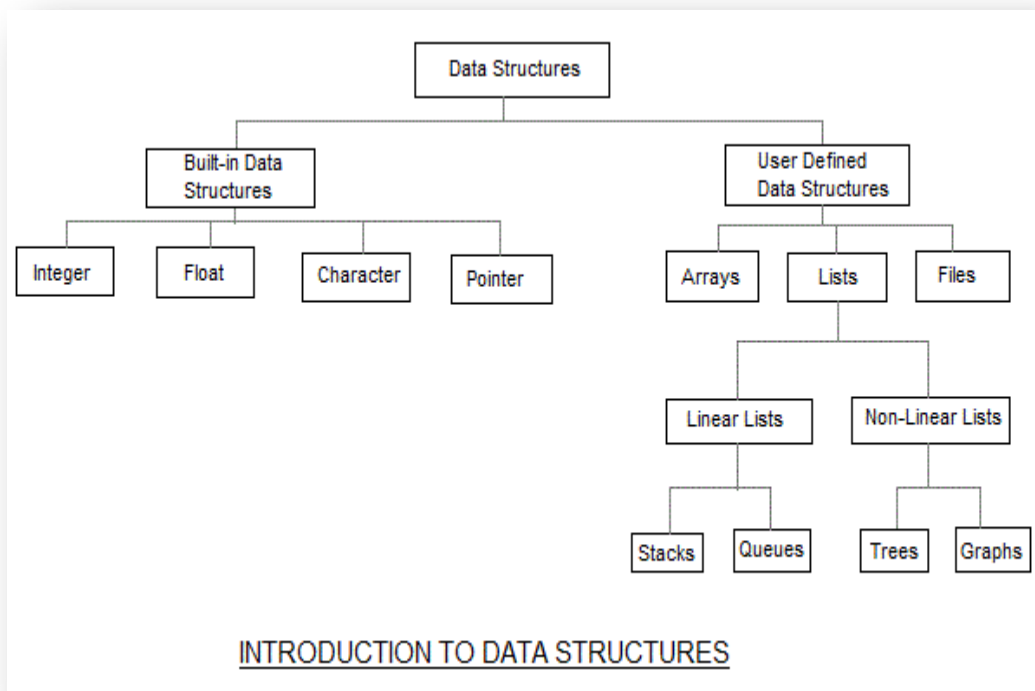
- Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.
- Commands used in preprocessor are called preprocessor directives and they begin with —# symbol.

Below is the list of preprocessor directives that C programming language offers.

Preprocessor	Syntax/Description
Macro	Syntax: #define This macro defines constant value and can be any of the basic data types.

Header file inclusion	Syntax: #include<file_name> The source code of the file -file_name is included in the main program at the specified place.
Conditional compilation	Syntax: #ifdef, #endif, #if, #else, #ifndef Set of commands are included or excluded in source program before compilation with respect to the condition.
Other directives	Syntax: #undef, #pragma #undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program.

UNIT-II



INTRODUCTION TO DATA STRUCTURES

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of **String** data type and 26 is of **integer** data type.

We can organize this data as a record like **Player** record. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.

Basic types of Data Structures

As we discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

- Linked List
- Tree
- Graph

- Stack,
- Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.

Abstract Data Type:

It can be defined as a collection of data items together with the operations on the data. The word -abstract refers to the fact that the data and the basic operations defined on it are being studied independently of how they are implemented. It involves what can be done with the data, not how has to be done.

An implementation of ADT consists of storage structures to store the data items and algorithms for basic operation. All the data structures i.e. array, linked list, stack, queue etc are examples of ADT.

DATA STRUCTURE

In computer science, a data structure is a particular way of storing and organizing data in a computer's memory so that it can be used efficiently. Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a data structure. The choice of a particular data model depends on the two considerations first; it must be rich enough in structure to mirror the actual relationships of the data in the real world. On the other hand, the structure should be simple enough that one can effectively process the data whenever necessary.

Need of data structure ·

It gives different level of organization data. ·

It tells how data can be stored and accessed in its elementary level. ·

Provide operation on group of data, such as adding an item, looking up highest priority item. ·

Provide a means to manage huge amount of data efficiently. ·

Provide fast searching and sorting of data.

Selecting a data structure

Selection of suitable data structure involve following steps –

- Analyze the problem to determine the resource constraints a solution must meet.
- Determine basic operation that must be supported.
- Quantify resource constraint for each operation
- Select the data structure that best meets these requirements.
- Each data structure has cost and benefits. Rarely is one data structure better than other in all situations.

A data structure requires:

- Space for each item it stores
- Time to perform each basic operation
- Programming effort.

Each problem has constraints on available time and space. Best data structure for the task requires careful analysis of problem characteristics.

Type of data structure

Static data structure

A data structure whose organizational characteristics are invariant throughout its lifetime. Such structures are well supported by high-level languages and familiar examples are arrays and records. The prime features of static structures are

- (a) None of the structural information need be stored explicitly within the elements – it is often held in a distinct logical/physical header;
- (b) The elements of an allocated structure are physically contiguous, held in a single segment of memory;
- (c) All descriptive information, other than the physical location of the allocated structure, is determined by the structure definition;
- (d) Relationships between elements do not change during the lifetime of the structure.

Dynamic data structure

A data structure whose organizational characteristics may change during its lifetime. The adaptability afforded by such structures, e.g. linked lists, is often at the expense of decreased efficiency in accessing elements of the structure. Two main features distinguish dynamic structures from static data structures. Firstly, it is no longer possible to infer all structural information from a header; each data element will have to contain information relating it logically to other elements of the structure. Secondly, using a single block of contiguous storage is often not appropriate, and hence it is necessary to provide some storage management scheme at run-time.

Linear Data Structure

A data structure is said to be linear if its elements form any sequence. There are basically two ways of representing such linear structure in memory.

- a) One way is to have the linear relationships between the elements represented by means of sequential memory location. These linear structures are called arrays.
- b) The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists. The common examples of linear data structure are arrays, queues, stacks and linked lists.

Non-linear Data Structure

This structure is mainly used to represent data containing a hierarchical relationship between elements. E.g. graphs, family trees and table of contents.

SINGLY LINKED LISTS

Linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. It is the second most-used data structure after array.

Following are the important terms to understand the concept of Linked List.

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **LinkedList** – A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

- **Single Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

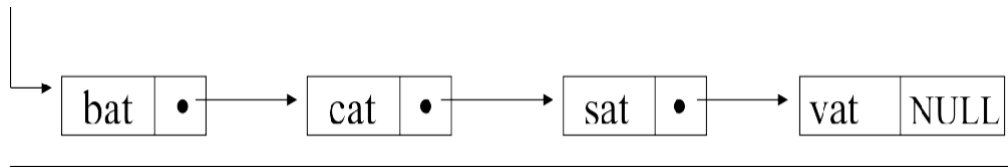
Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning, end and after a specific node of the list.
- **Deletion** – Deletes an element at the beginning and at position of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.

In a linked list the ordered sequence of nodes with links do not reside in sequential locations. The locations of the nodes may change on different runs.

ptr



Usual way to draw a linked list

Create A Linked List Of Words

```

typedef struct list_node *list_pointer;
typedef struct list_node {
    char data [4];
    list_pointer link;
};
  
```

Creation

```
list_pointer ptr =NULL;
```

Testing

```
#define IS_EMPTY(ptr) (!ptr)
```

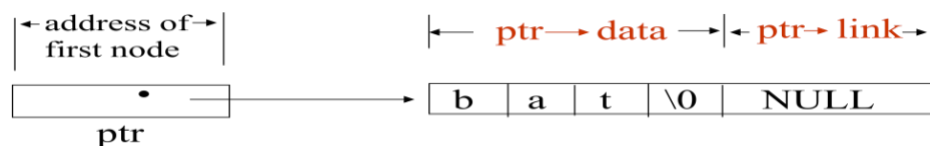
Allocation

```
ptr=(list_pointer) malloc (sizeof(list_node));
```

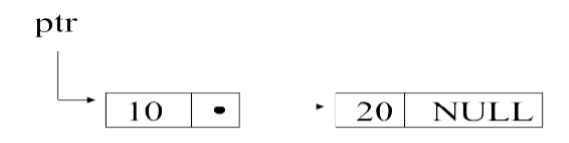
```
e -> name -> (*e).name
```

```
strcpy(ptr -> data, -ball);
```

```
ptr -> link = NULL;
```



Referencing the fields of a node



```

typedef struct list_node *list_pointer;
typedef struct list_node {
    int data;
    list_pointer link;
};
  
```



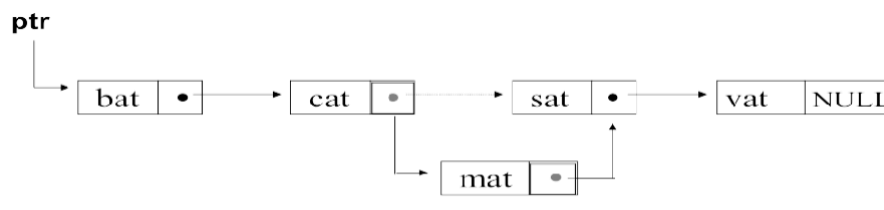
```

    };
list_pointer ptr = NULL

list_pointer create2( )
{
/* create a linked list with two nodes */
list_pointer first, second;
first = (list_pointer) malloc(sizeof(list_node));
second = (list_pointer) malloc(sizeof(list_node));
second -> link = NULL;
second -> data = 20;
first -> data = 10;
first -> link = second;
return first;
}

```

Insert mat after cat



1. Get a node that is currently unused ; let its address be paddr.
2. Set the data field of this node to mat.
3. Set paddr's link field to point to the address found in the link field of the node containing cat.
4. Set the link field of the node containing cat to point to paddr.

Insert a node after a specific node

```
#define IS_FULL(p) (!p)
```

```

void insert(list_pointer *ptr, list_pointer node)
{
/* insert a new node with data = 50 into the list ptr after node */

list_pointer temp;
temp = (list_pointer) malloc(sizeof(list_node));
if (IS_FULL(temp)) //if, temp==NULL
{
fprintf(stderr, -The memory is full\n);
exit (1);
}

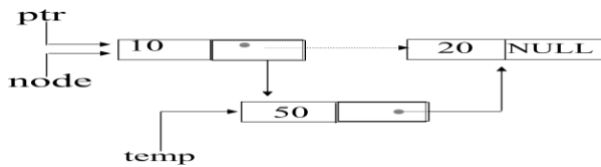
temp->data = 50;
if (*ptr) { // nonempty list
temp->link = node ->link;

```

```

    node->link = temp;
}
else { // empty list
    temp->link = NULL;
    *ptr = temp;
}
}

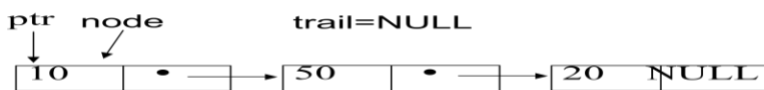
```



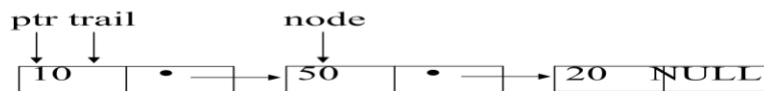
List Deletion

a) Before

Delete the first node.



b) After deletion



Delete other than first node

```

void delete(list_pointer *ptr, list_pointer trail,
            list_pointer node)
{
    /* delete node from the list, trail is the preceding node
    ptr is the head of the list */
    if (trail)
        trail->link = node->link;
    else
        *ptr = (*ptr) ->link;
    free(node);
}

```

Print out a list (traverse a list)

```

void print_list(list_pointer ptr)
{
    printf(-The list contains: -);
    for ( ; ptr; ptr = ptr->link)
        printf(-%4d\|, ptr->data);
}

```

```
printf(-\n);
}
```

Easy Way Of Linked List Representation

Singly Linked list

Single Linked List is a collection of nodes. Each node contains 2 fields: I) info where the information is stored and ii) link which points to the next node in the list.

The node is like this:

Node

Info (or) data	Link (or) next
----------------	----------------

The operations that can be performed on single linked lists includes: insertion, deletion and traversing the list.

Inserting a node in a single linked list:

Insertion can be done in 3 ways:

- 1.Inserting an element in the beginning of the list.
- 2.Inserting in the middle and
- 3.Inserting at end.

The algorithm for inserting an element in the beginning of a list:

Procedure insertbeg(x, first)

/* x – is an element first is the Pointer to the first element in a SLL */

begin

if avail = NULL then

Write(‘_ Availability Stack underflow’)

return (first)

else

new← avail;

avail← link(avail)

```

info(new) ← x

link(new) ← first

return (new);

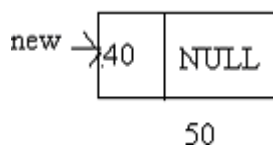
end.

```

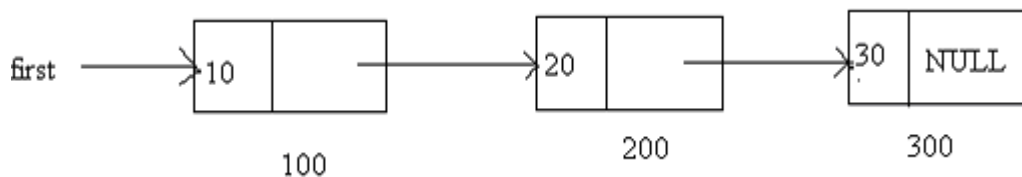
In the algorithm, first, the memory for a new node is available or not is checked out. Here avail is a pointer to the available memory.

If it is NULL then there is no memory. Otherwise a new node is created from available memory and it is stored in new. The new contains the address of the new node and avail moves forward to point to next available memory location. In the info field of new, x is stored and the link field of new points to the first element address of the list. Now, new becomes the pointer to the whole list.

This can be shown as:



(a)

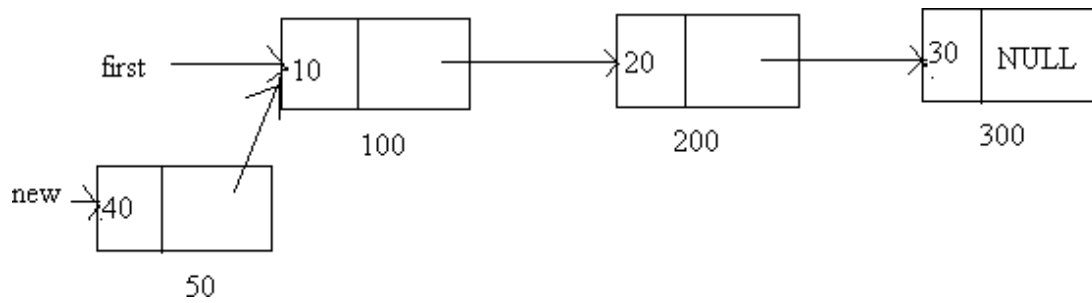


(b)

First is having value of 100, means it is Pointing the 100th memory location. The node at 100th location is having info as 10 and containing the address 200 of the next node. The second node at 200th location, contains 20 and address 300 of the next node. At,300th memory location, the node contains info as 30 and its link field is NULL. That means it is no more having any nodes. That is, the list is having 3 nodes with a starting address of 100.

Now, we created another node new that is having the address of 50 and its info is 40 and its link field is NULL.

After the call to insertbeg() function the list will look like.



The insertbeg() function, inserts a node by storing the address of the first node of list in the link field of new and making the new address as the pointer to the list.

The algorithm for inserting an element at the end of the list:

Procedure insertend(x, first)

begin

if avail = null then /* Checking for memory availability */

write (‘__ Availability Stack underflow’);

return(first);

else /* Obtaining the next free node */

new ← avail; /* Removing free node from available memory */

avail ← link(avail)

info(new) ← x /* Initializing the fields of new node */

link(new) ← NULL

if first = NULL then /* Is the list empty? */

return(new);

Save ← first /* Searching the last node */

Repeat while link (Save) ≠ NULL

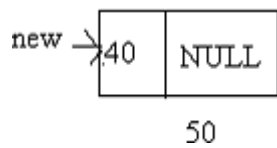
Save ← link(Save) /* Setting the link field of last node to new */

Link (Save) ← new

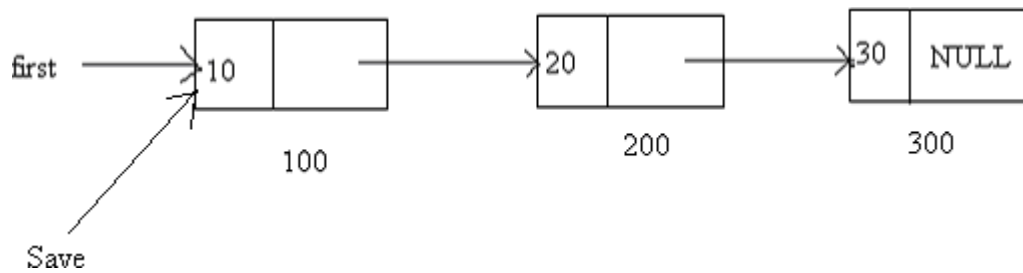
return (first)

end;

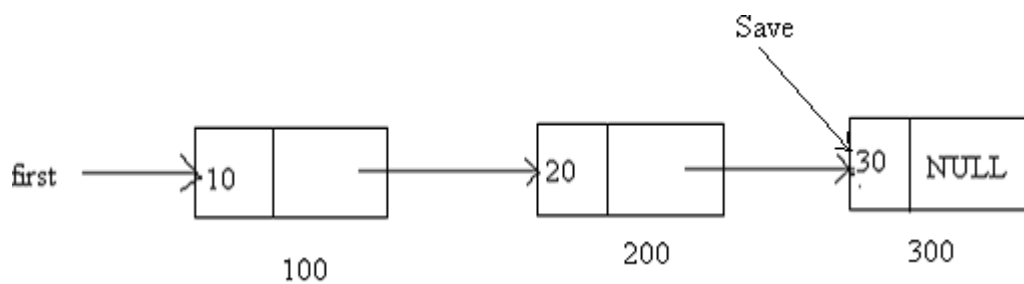
The Pictorial Representation:



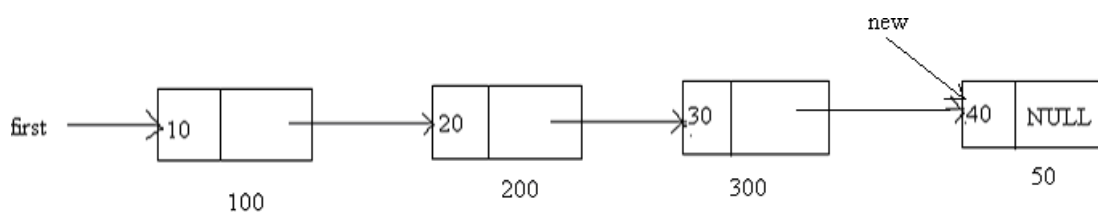
(a)



(b)



(c)



first (100) address is stored in Save to go through the end of the list, after reaching to the last node, set the address of new node in the link field of last node.

Inserting in the middle:

In this process, the address of first is stored in Save to go to the particular node at which the insertion is to be done. After reaching the particular node, set the link to point to new node, and set the link of new node to connect the remaining nodes.

ALGORITHM: The algorithm for inserting an element in the middle of a list:

Procedure insertmid(x, first)

begin

if avail = NULL then

write (‘__ Availability Stack Underflow’);

return (first)

else /* Obtain the address of next free node */

new ← avail

avail ← link(avail) /* Removing free node */

info(new) ← x /* Copying Information into new node */

if first = NULL then /* Checking whether the list is empty */

link(new) ← NULL /* list is empty */

return(new) /* if the new data precedes all other in the list */

if info(new) ≤ info(first) then

link(new) ← first

return(new) /* initialise temporary Pointer */

Save ← first

Repeat while link(Save) ≠ NULL and

Info(link(Save)) ≤ info(new)

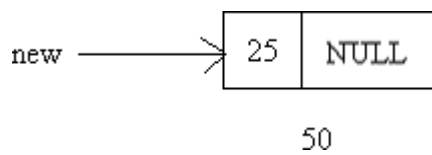
Save ← link(Save) /* Search for predecessor of new data */

link(new) ← link(Save) /* Setting the links of new and its
Predecessor*/

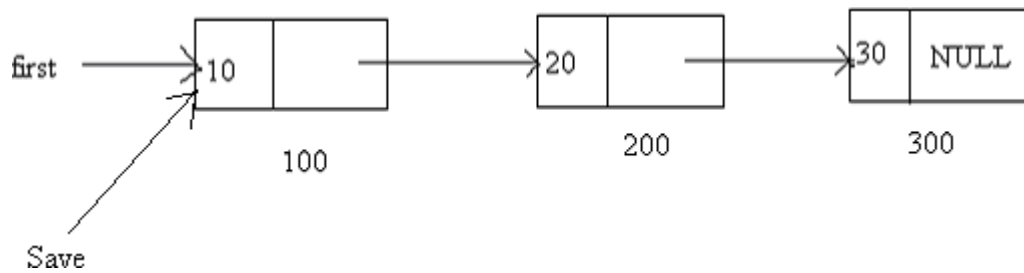
link(Save) ← new

return(first)

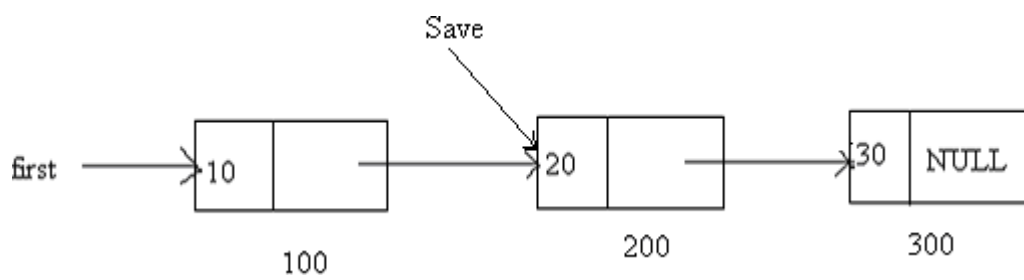
end



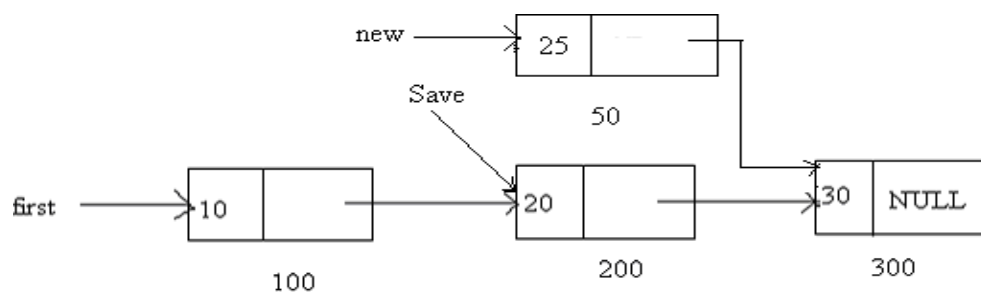
(a)



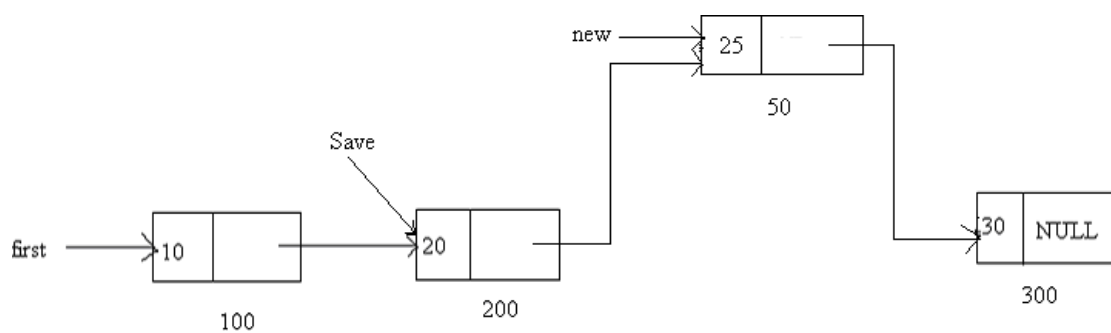
(b)



(c)



(d)



(e)

Deleting a node in a single linked list:

Deletion can be done in 3 ways:

1. Deleting an element in the beginning of the list.
2. Deleting in the middle and
3. Deleting at the end.

The algorithms for deleting an element in the list:

Procedure:

- If the linked list is empty then write underflow and return
- Repeat Step 3 while the end of the list has not been reached and the node has not been found.
- Obtain the next node in the list and record its predecessor node.
- If the end of the list has been reached then write node not found and return.
- delete the node from the list.
- Return the node to the availability area.

Algorithm:

Procedure deletelement(x, first)

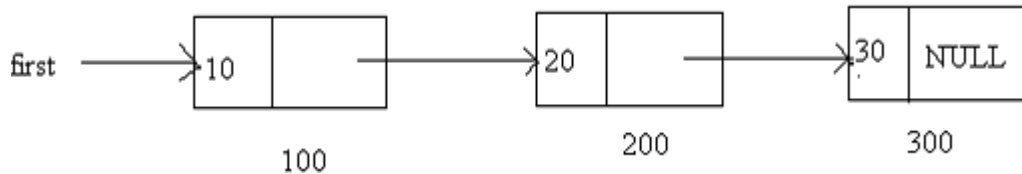
```
begin
    if first = NULL then /* Checking whether the list is empty */
        write(_ Underflow');
        return
    temp ← first /* Initializing search for x */
    while((temp ≠ x ) and
        (link(temp) ≠ NULL))
begin
    pred ← temp
    temp ← link(temp)
end
    if temp ≠ x then /* if the node not found */
        write(_node not found')
        return
    if (x = first) then / is x first node */
        first ← link(first) /* delete the node at x */
    else    link(pred) ← link(x)
    link(x) ← avail /* Return node to availability area */
    avail ← x
```

return:

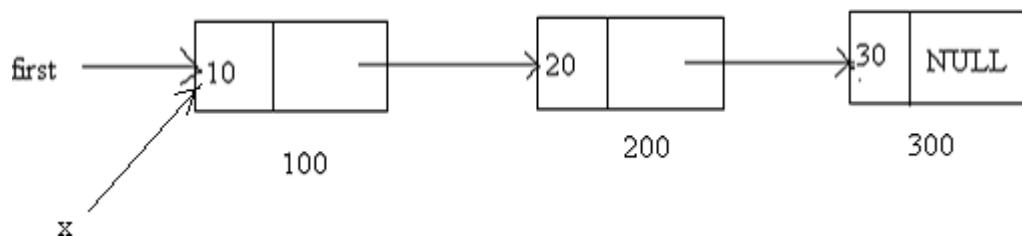
end.

In the algorithm, it first checks whether the list is empty, if it is empty it prints underflow message. Otherwise, it initializes a temporary pointer to first. Until the Predecessor of x node found, the temporary pointer is moved forward. If it reaches the end of the list, without finding the node of address x, it flashes an error message, stating '_node not found'. If x is the first address, then first node is deleted and now first points to second node. Otherwise it sets the links so that it deletes the node of address x.

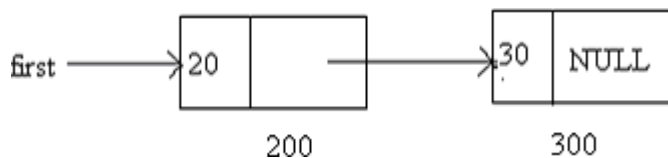
The original list is:



1. If x = 100 (First node)



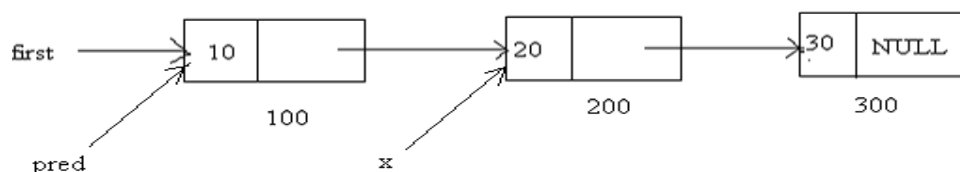
(a)

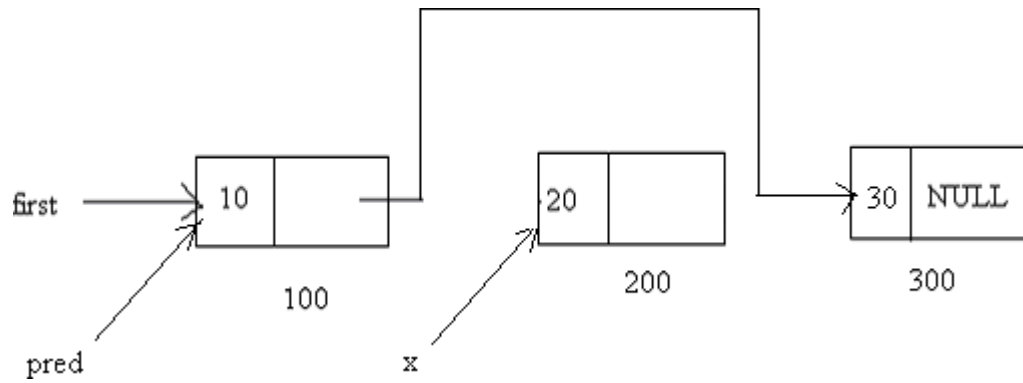


(b)

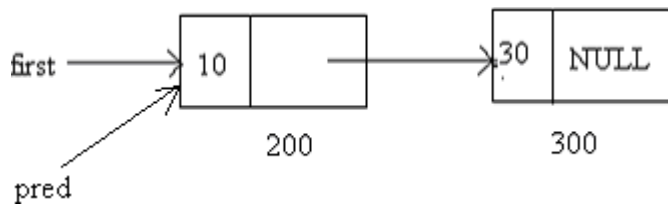
The node which is 100th memory address will be added to free space

(ii) if x = 200



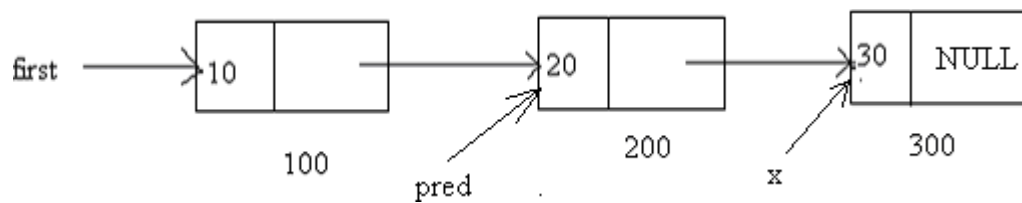


(b)

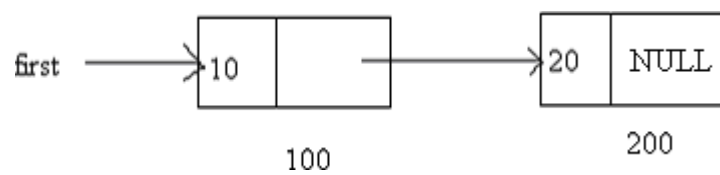


(c)

(iii) if $x = 300$ (Last node)



(a)



(b)

Traversing the List:

This includes, visiting the node and printing the data of that node one at a time and moving forward until it reaches end of the list. (i.e., link becomes NULL)

Procedure(first)

begin

for (temp = first; temp!= NULL; temp = temp → link)

write (temp → info)

end.

UNIT-III

Sorting –selection sort, Quick Sort, insertion sort techniques (Using Arrays)

Searching - linear search, binary search techniques (Using Arrays)

Introduction to Sorting

Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical, or any user-defined order. Sorting is a process through which the data is arranged in ascending or descending order. Sorting can be classified in two types;

Internal Sorts:-This method uses only the primary memory during sorting process. All data items are held in main memory and no secondary memory is required this sorting process. If all the data that is to be sorted can be accommodated at a time in memory is called internal sorting. There is a limitation for internal sorts; they can only process relatively small lists due to memory constraints. There are 3 types of internal sorts.

- | | |
|---------------------|--|
| (i) SELECTION SORT | Ex:- Selection sort algorithm, Heap Sort |
| :- | algorithm |
| (ii) INSERTION SORT | Ex:- Insertion sort algorithm, Shell Sort |
| :- | algorithm |
| (iii) EXCHANGE SORT | Ex:- Quick Sort Sort Algorithm, Quick |
| :- | sort algorithm |

External Sorts:-Sorting large amount of data requires external or secondary memory. This process uses external memory such as HDD, to store the data which does not fit into the main memory. So, primary memory holds the currently being sorted data only. All external sorts are based on process of merging. Different parts of data are sorted separately and merged together.
Ex:- Merge Sort

1. Selection Sort

In selection sort the list is divided into two sub-lists:- sorted and unsorted. These two lists are divided an imaginary wall. We find the smallest element from unsorted sub-list and swap it to the beginning. And the wall moves one element ahead, as the sorted list increases and unsorted list decreases.

Assume that we have a list on n elements. By applying selection sort, the first element is compared with all remaining (n-1) elements. The smallest element is placed at the first location. Again, the second element is compared with remaining (n-1) elements. At the time of comparison, the smaller element is swapped with larger element. Similarly, entire array is checked for smallest element and then swapping is done accordingly. Here we

need n-1 passes or iterations to completely rearrange the data.

Algorithm: Selection_Sort (A [], N)

Step 1 : Repeat For K = 0 to N – 2

Step 2 : Set POS = K

Step 3 : Repeat for J = K + 1 to N – 1

 If A[J] < A [POS]

 Set POS = J

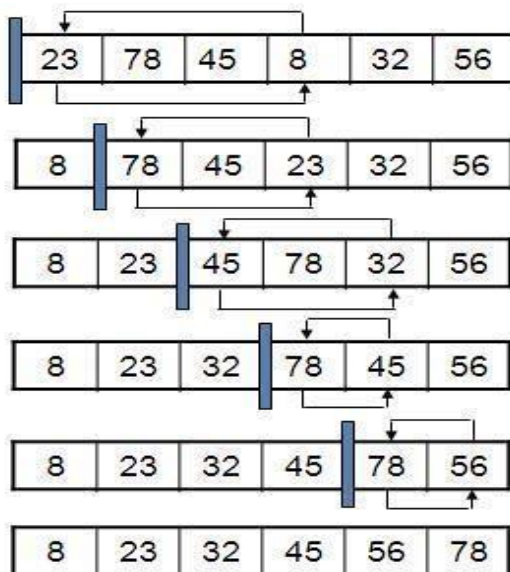
 End For

Step 5 : Swap A [K] with A [POS]

 End For

Step 6 : Exit

Ex:- A list of unsorted elements are: 23 78 45 8 32 56



A list of sorted elements now : 8 23 32 45 56 78

Write a program to explain selection sort?

/* program to sort elements of an array using selection sort*/

```
#include<stdio.h> void main( )
{
    int i,j,t,n,min,a[10]; clrscr( );
    printf("\n How many elements you want to sort? "); scanf("%d",&n);
    printf("\nEnter elements for an array:"); for(i=0;i<n;i++)
    scanf("%d",&a[i]);
```

```

for(i=0;i<n;i++)
{
    min=i;
    for(j=i+1;j<n;j++)
    {
        if(a[j] > a[min])
            min=j;
    }
    t=a[i];
    a[i]=a[min];
    a[min]=t;
}

printf("\nAfter sorting the elements are:");
for(i=0;i<n;i++)
    printf("%d ",a[i]); getch( );
}

```

OUTPUT

How many elements you want to

sort? : 5

Enter elements for an array : 2 6 4 8 5

After Sorting the elements are
: 8 6 5 4 2

2. Quick Sort:

Sorting an array of items is clearly a fundamental problem, directly linked to efficient searching with numerous applications. The problem is that given an array of keys, we want to rearrange these in non-decreasing order. Note that the order may be numerical, alphabetical or any other transitive relation defined on the keys . The analysis deals with numerical order, where the keys are decimal numbers and we particularly focus on Quicksort algorithm and variants of it. Quicksort was invented by C. A. R. Hoare [29, 30]. Here is the detailed definition.

The steps taken by the Quicksort algorithm are:

Choose an element from the array, called pivot. . Rearrange the array by comparing every element to the pivot, so all elements smaller than or equal to the pivot come before the pivot and all elements greater than or equal to the pivot come after the pivot. Recursively apply steps 1 and 2 to the subarray of the elements smaller than or equal to the pivot and to the subarray of the elements greater than or equal to the pivot. Note that the original problem is divided into smaller ones, with (initially) two subarrays, the keys smaller than the pivot, and those bigger than it. Then recursively these are divided into smaller subarrays by further pivoting, until we get trivially sorted subarrays, which contain one or no elements. Given an array of n distinct keys $A = \{a_1, a_2, \dots, a_n\}$ that we want to quick sort, with all the $n!$ permutations equally likely, the aim is to finding the unique permutation out of all the $n!$ possible, such that the keys are in increasing order. The essence of Quicksort is the partition operation, where by a

series of pairwise comparisons, the pivot is brought to its final place, with smaller elements on its left and greater elements to the right. Elements equal to pivot can be on either or both sides.

As we shall see, there are numerous partitioning schemes, and while the details of them are not central to this thesis, we should describe the basic ideas. A straightforward and natural way (see e.g. [1]) uses two pointers – a left pointer, initially at the left end of the array and a right pointer, initially at the right end of the array. We pick the leftmost element of the array as pivot and the right pointer scans from the right end of the array for a key less than the pivot. If it finds such a key, the pivot is swapped with that key. Then, the left pointer is increased by one and starts its scan, searching for a key greater than the pivot: if such a key is found, again the pivot is exchanged with it. When the pointers are crossed, the pivot by repeated exchanges will “float” to its final position and the keys which are on its left are smaller and keys on its right are greater.

3. Insertion Sort:

Both the selection and Quick Sort sorts exchange elements. But insertion sort does not exchange elements. In insertion sort the element is inserted at an appropriate place similar to card insertion. Here the list is divided into two parts sorted and unsorted sub-lists. In each pass, the first element of unsorted sub list is picked up and moved into the sorted sub list by inserting it in suitable position. Suppose we have n elements, we need $n-1$ passes to sort the elements.

Insertion sort works this way:

It works the way you might sort a hand of playing cards:

1. We start with an empty left hand [sorted array] and the cards face down on the table [unsorted array].
2. Then remove one card [key] at a time from the table [unsorted array], and insert it into the correct position in the left hand [sorted array].
3. To find the correct position for the card, we compare it with each of the cards already in the hand, from right to left.

INSERTION_SORT (A)

FOR $j \leftarrow 2$ TO $\text{length}[A]$

DO $\text{key} \leftarrow A[j]$

. {Put $A[j]$ into the sorted sequence $A[1 \dots j-1]$ }

$i \leftarrow j-1$

WHILE $i > 0$ and $A[i] > \text{key}$

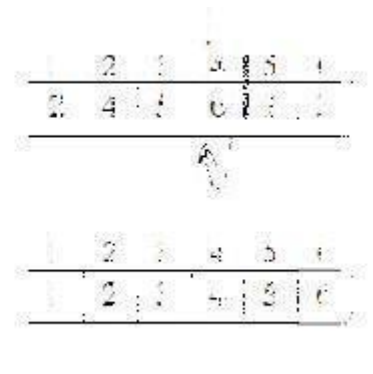
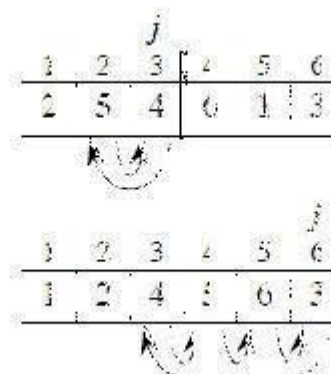
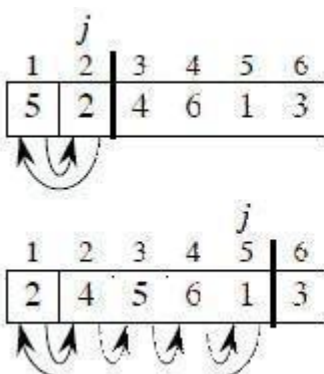
DO $A[i+1] \leftarrow A[i]$

$i \leftarrow i-1$

$A[i+1] \leftarrow \text{key}$

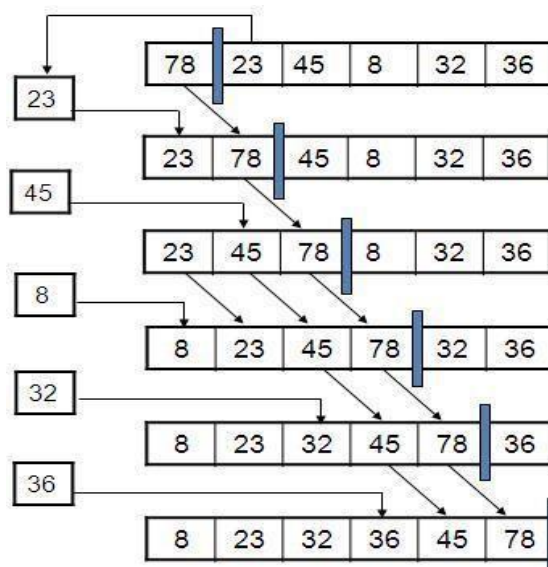
Example: Following figure (from CLRS) shows the operation of INSERTION-SORT on the array $A = (5, 2, 4, 6, 1, 3)$. Each part shows what happens for a particular iteration with the value

of j indicated. j indexes the "current card" being inserted into the hand.



Read the figure row by row. Elements to the left of $A[j]$ that are greater than $A[j]$ move one position to the right, and $A[j]$ moves into the evacuated position.

Ex:- A list of unsorted elements are: 78 23 45 8 32 36 . The results of insertion sort for each pass is as follows:-



A list of sorted elements now : 8 23 32 36 45 78

Write a program to explain insertion sort . Which type of technique does it belong.

(or)

Write a C-program for sorting integers in ascending order using insertion sort.

/*Program to sort elements of an array using insertion sort method*/

```
#include<stdio.h>
void main( )
{
    int
    a[10],i,j,k,n;
    printf("How many elements you want to sort?\n");
    scanf("%d",&n);
    printf("\nEnter the Elements into an array:\n");
    for (i=0;i<n;i++)
    scanf("%d",&a[i]);
    for(i=1;i<n;i++)
    {
        k=a[i];
```

```

    for(j= i-1; j>=0 && k<a[j]; j--)
        a[j+1]=a[j];
    a[j+1]=k;
}
printf("\n\n Elements after sorting: \n");
for(i=0;i<n;i++)

    printf("%d\n", a[i]);
}

```

OUTPUT:

How many elements you want to sort ? :

6

Enter elements for an array : 78 23 45 8 32 36

After Sorting the elements are : 8 23 32 36 45 78

A list of sorted elements now : 8 23 32 36 45 78

Linear Search

Linear search technique is also known as sequential search technique. The linear search is a method of searching an element in a list in sequence. In this method, the array is searched for the required element from the beginning of the list/array or from the last element to first element of array and continues until the item is found or the entire list/array has been searched.

Algorithm:

Step 1: set-up a flag to indicate -element not found

Step 2: Take the first element in the list

Step 3: If the element in the list is equal to the desired element

- Set flag to -element found
- Display the message -element found in the list Go
- to step 6

Step 4: If it is not the end of list,

- Take the next element in the
- list
- Go to step 3

Step 5: If the flag is —element not found

Display the message -element not found Step

6: End of the Algorithm

Advantages:

1. It is simple and conventional method of searching data. The linear or sequential name implies that the items are stored in a systematic manner.
2. The elements in the list can be in any order. i.e. The linear search can be applied on sorted or unsorted linear data structure.

Disadvantage:

1. This method is insufficient when large number of elements is present in list.
2. It consumes more time and reduces the retrieval rate of the system.

Time complexity: $O(n)$

Write a C program that searches a value in a stored array using linear search.

```
#include<stdio.h>
int linear(int [ ],int,int);
void main( )
{
    int a[20], pos = -1, n, k, i;
    printf("\nEnter the n value:");
    scanf("%d",&n);
    printf("\nEnter elements for an array:");
    for(i=0; i<n ;i++)
        scanf("%d",&a[i]);
    printf("\nEnter the element to be searched:");
    scanf("%d",&k);
    pos=linear(a,n,k);
    if(pos != -1)
        printf("\n Search successful element found at position %d",pos);
    Else
        printf("\n Search unsuccessful, element not found");
    getchar( );
}
int linear(int a[ ],int n,int k)
{
    int i;
    for(i=0;i<n;i++)
    {
        if(a[i]==k)
```

```

    return(i);
}
return -1;
}

```

Output:-

```

Enter the n value           :    5
Enter elements for an array :    11    2    23    14    55
Enter the element to be searched:    14
Search successful element found at position :    3

```

Write a C program that searches a value in a stored array using recursive linear search.

```

/* recursive program for Linear Search*/

#include<stdio.h>
int linear(int [ ],int,int);
void main( )

{
    int a[20],pos=-1,n,k,i;
    printf("\nEnter n value:");
    scanf("%d",&n);
    printf("\nEnter elements for an array:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("\n Enter the element to be searched:");
    scanf("%d",&k);
    pos=linear(a,n,k);
    if(pos!=-1)
        printf("\n Search successful, Element found at Position %d",pos);
    else
        printf("Search unsuccessful, element not found ")
}

int linear(int a[ ],int n,int k)
{
    int i;
    for(i=n-1;i>=0;i--)
    {

```

```

if(a[i]==k)
    return(i);
else
{
    n = n-1;
    return (linear(a,n,k);
}
}
return -1;
}

```

Output:-

```

Enter 'n' value :          6
Enter elements for an array      :      10      32      22      84      55      78
Enter the element to be searched :          55

Search successful, Element found at Position      :      4

```

Binary search

Binary search is the most popular Search algorithm. It is efficient and also one of the most commonly used techniques that is used to solve problems.

If all the names in the world are written down together in order and you want to search for the position of a specific name, binary search will accomplish this in a maximum of 35 iterations.

Binary search works only on a sorted set of elements. To use binary search on a collection, the collection must first be sorted.

When binary search is used to perform operations on a sorted set, the number of iterations can always be reduced on the basis of the value that is being searched.

Let us consider the following array:

Arr	0	1	2	3	4	5	6	7	8	9
-----	---	---	---	---	---	---	---	---	---	---

By using linear search, the position of element 8 will be determined in the 9th iteration.

Let's see how the number of iterations can be reduced by using binary search. Before we start the search, we need to know the start and end of the range. Let's call them Low and High.

Low = 0

High = n-1

Now, compare the search value K with the element located at the median of the lower and upper bounds. If the value K is greater, increase the lower bound, else decrease the upper bound.



Referring to the image above, the lower bound is 0 and the upper bound is 9. The median of the lower and upper bounds is $(\text{lower_bound} + \text{upper_bound}) / 2 = 4$. Here $a[4] = 4$. The value $4 > 3$, which is the value that you are searching for. Therefore, we do not need to conduct a search on any element beyond 4 as the elements beyond it will obviously be greater than 3.

Therefore, we can always drop the upper bound of the array to the position of element 4. Now, we follow the same procedure on the same array with the following values:

Low: 0

High: 3

Repeat this procedure recursively until $\text{Low} > \text{High}$. If at any iteration, we get $a[\text{mid}] = \text{key}$, we return value of mid. This is the position of key in the array. If key is not present in the array, we return -1.

Write a C program that searches a value in a stored array using iterative binary search.

```
#include<stdio.h>
int bsearch(int [ ],int,int);
void main( )
{
    int a[20],pos,n,k,i;
    printf("\nEnter the n value:");
    scanf("%d",&n);
    printf("\nEnter elements for an array:");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("\nEnter the key value:");
    scanf("%d",&k);
    pos=bsearch(a,n,k);
    if(pos!= -1)
        printf("Search successful, element found at position %d",pos);
    else
        printf("Search unsuccessful, element not found");
}
```

```

        getch( );
    }

    int bsearch(int a[ ],int n, int k)
    {
        int
        lb=0,ub,mid;
        lb=0;
        ub=n-1;
        while(ub>=l
        b)
        {
            mid=(lb+ub)/2;

            i(k<a[mid])

            ub=mid-1;

            else if(k>a[mid])

            lb=mid+1;

            else if(k==a[mid])

            return(mid);

        }

        return -1;

    }

```

OUTPUT

```

Enter _n_ value :          67

Enter elements for an array      :    35    10    32    25    84    55    78
Enter the element to be
searched                      :          25
Search successful, Element found at
Position                      :          3

```

Write a C program that searches a value in a stored array using recursive binary search.

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

```
// A recursive binary search function. It returns location of x in
```

```
// given array arr[l..r] is present, otherwise -1
```

```
int binarySearch(int arr[], int l, int r, int x)
```

```
{
```

```

if (r >= l)
{
    int mid = l + (r - l)/2;

    // If the element is present at the middle itself
    if (arr[mid] == x) return mid;

    // If element is smaller than mid, then it can only be present
    // in left subarray
    if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);

    // Else the element can only be present in right subarray
    return binarySearch(arr, mid+1, r, x);
}

// We reach here when element is not present in array
return -1;
}

int main(void)
{
    int arr[] = {2, 3, 4, 10, 40};
    int n = sizeof(arr)/ sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n-1, x);
    (result == -1)? printf("Element is not present in array")
                  : printf("Element is present at index %d", result);
    return 0;
}

```

UNIT -IV

STACKS & QUEUES:

Stacks and Queues are very useful in numerous applications. For eg. Stacks are used in computers in parsing an expression by recursion; in memory management in operating system; etc. Queues find their use in CPU scheduling in printer spooling, in message queuing in computer network etc.

STACKS:

Stack is an ordered collection of data in which data is inserted and deleted at one end (same end). In stack the most recent elements are inserted first. For eg. plates are pushed on to the top and popped off the top. Such a processing strategy is referred to as Last-in-First-out or LIFO. A Stack is a data structure which operates on a LIFO basis, it is used to provide temporary storage space for values. Stack is most commonly used as a place to store local variables, parameters and return addresses. When a function is called the use of stack is critical where recursive functions, that are function that call them are defined.

Stack uses a single pointer `_top` to keep track of the information in the stack.

Basic operations of a stack are.

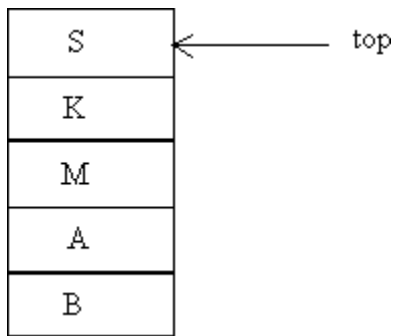
1. Insert (push) an item onto stack.
2. Remove (pop) an item from the stack.

A stack pointer keeps track of the current position on the stack. When an element is placed on the stack, it is said to be pushed on the stack when an object is removed from the stack, it is said to be popped from the stack. Stack `_Overflow` occurs when one tries to push more information on a stack, and `_Underflow` occurs when we try to pop an item off a stack which is empty.

Representation Of Stack:

Two standard methods of representing a stack are in terms of an array and a pointer linked list which is a way of representation using pointer.

The figure below illustrates an array implementation of a stack of 5 elements B, A, M, K, S.



The Stack pointer is pointing to the top of the stack (called the top of the stalk). The size of the array is fixed at the time of its declaration itself. But, as the definition shows, stacks are dynamic data structures. They can grow and shrink during the operation. However, stack can be implemented using arrays by specifying a maximum size. We require two things to represent stacks. First one is array to hold the item and second an integer variable to hold the index of the top element conveniently in C, we can declare stack as a structure to represent these two

```
#define max 50
struct stack
{
    int top;
    int items [max];
};
```

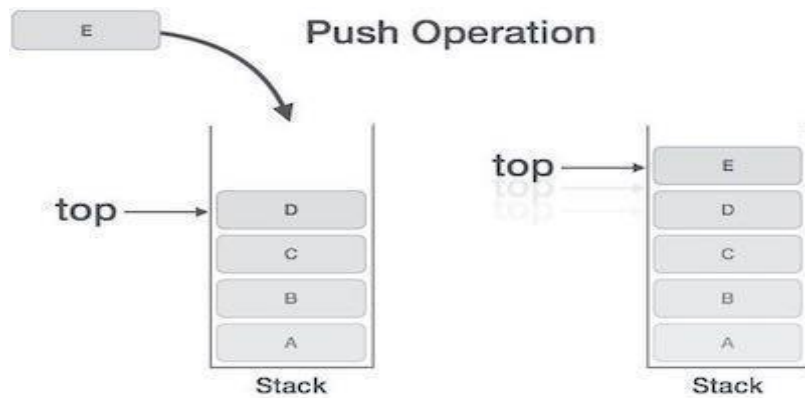
Now the stack s can be declared as struct stack S;

Note that the type of array elements in the above declaration is integer, but actually it can be of any other type. Since array subscripts starts from 0, the 50 stack elements in the above example reside in the location S.item[0] through S.items[49]. The top element can be accessed as S.item[top].

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- **Step 1** – Checks if the stack is full.
- **Step 2** – If the stack is full, produces an error and exit.
- **Step 3** – If the stack is not full, increments **top** to point next empty space.
- **Step 4** – Adds data element to the stack location, where top is pointing.
- **Step 5** – Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows –

```

begin procedure push: stack, data
if stack is full
return null
endif
top ← top + 1
stack[top] ← data
end procedure

```

Implementation of this algorithm in C, is very easy. See the following code –

Example

```

void push(int data){
if(!isFull()){
top= top +1;
stack[top]= data;
}else{
printf("Could not insert data, Stack is full.\n");
}
}

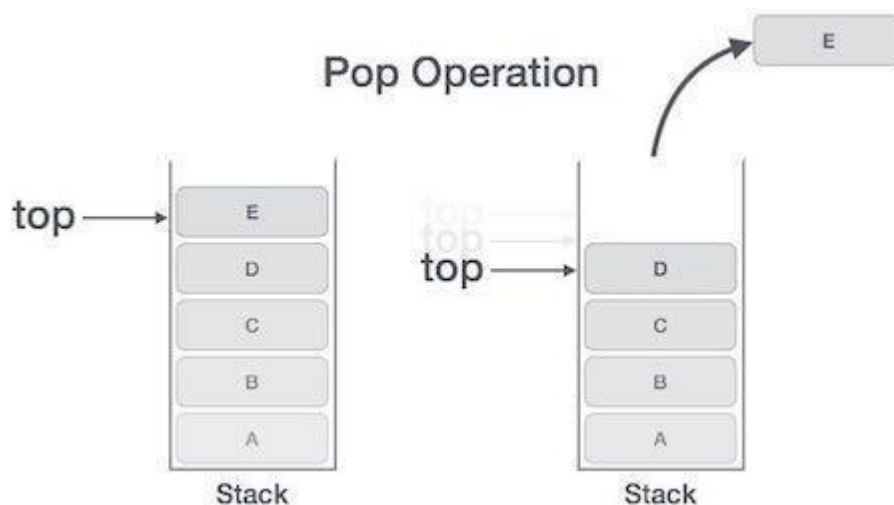
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps –

- **Step 1** – Checks if the stack is empty.
- **Step 2** – If the stack is empty, produces an error and exit.
- **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** – Decreases the value of top by 1.
- **Step 5** – Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows –
begin procedure pop: stack

```
if stack is empty
return null
endif
data ← stack[top]
top ← top - 1
return data
end procedure
```

Implementation of this algorithm in C, is as follows –

Example

```
int pop(int data){  
  
    if(!isempty()){  
        data= stack[top];  
        top= top -1;  
        return data;  
    }else{  
        printf("Could not retrieve data, Stack is empty.\n");  
    }  
}
```

Since a stack can contain only a finite number of elements, an error called a ‘_stack overflow’ occurs when the stack is full and a push operation is performed. Similarly, an error called a stack underflow occurs when the stack is empty and a pop operation is performed.

These two conditions need to be checked before doing the respective operations.

STACK USING LINKED LIST

ALGORITHM

PUSH OPERATION: insert an element on top of the stack

STEP 1 : Create a newNode with given value.

STEP2: Check whether stack is empty(top==NULL)

STEP3: if it is empty then set newNode->next=NULL.

STEP4: if it is not empty then newNode->next=top;


STEP5: Finally set top=newNode.

POP OPERATION: delete an element from a stack

STEP1: check whether the stack is empty(top=NULL)

STEP2: if it is empty then display stack is empty.

STEP3: if it is not empty then define a node pointer and set it to top

STEP4: then set top=top  next

STEP5: Finally delete temp(free(temp)).

Display():

STEP1: check whether the stack is empty(top=NULL)

STEP2: if it is empty then display stack is empty.

STEP3: if it is not empty then define a node pointer and initialize with the top.

STEP4: Display temp->data and move it to the next node.repet this until temp reaches the first node in the stack(temp->next=NULL).

STEP5: finally display temp data NULL

Infix to postfix conversion algorithm:

There is an algorithm to convert an infix expression into a postfix expression. It uses a stack; but in this case, the stack is used to hold operators rather than numbers. The purpose of the stack is to reverse the order of the operators in the expression. It also serves as a storage structure, since no operator can be printed until both of its operands have appeared.

In this algorithm, all operands are printed (or sent to output) when they are read. There are more complicated rules to handle operators and parentheses.

Example:

1. $A * B + C$ becomes $A B * C +$

The order in which the operators appear is not reversed. When the '+' is read, it has lower precedence than the '*', so the '*' must be printed first.

We will show this in a table with three columns. The first will show the symbol currently being read. The second will show what is on the stack and the third will show the current contents of the postfix string. The stack will be written from left to right with the 'bottom' of the stack to the left.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	+	+	A B * {pop and print the '*' before pushing the '+'}
5	C	+	A B * C

The rule used in lines 1, 3 and 5 is to print an operand when it is read. The rule for line 2 is to push an operator onto the stack if it is empty. The rule for line 4 is if the operator on the top of the stack has higher precedence than the one being read, pop and print the one on top and then push the new operator on. The rule for line 6 is that when the end of the expression has been reached, pop the operators on the stack one at a time and print them.

2. $A + B * C$ becomes $A B C * +$

Here the order of the operators must be reversed. The stack is suitable for this, since operators will be popped off in the reverse order from that in which they were pushed.

	current symbol	operator stack	postfix string
1	A		A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6			A B C * +

In line 4, the '*' sign is pushed onto the stack because it has higher precedence than the '+' sign which is already there. Then when they are both popped off in lines 6 and 7, their order will be reversed.

3. $A * (B + C)$ becomes $A B C + *$

A subexpression in parentheses must be done before the rest of the expression.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(* (A B
4	B	* (A B
5	+	* (+	A B

6	C	* (+	A B C
7)	*	A B C +
8			A B C + *

Since expressions in parentheses must be done first, everything on the stack is saved and the left parenthesis is pushed to provide a marker. When the next operator is read, the stack is treated as though it were empty and the new operator (here the '+' sign) is pushed on. Then when the right parenthesis is read, the stack is popped until the corresponding left parenthesis is found. Since postfix expressions have no parentheses, the parentheses are not printed.

4. $A - B + C$ becomes $A B - C +$

When operators have the same precedence, we must consider association. Left to right association means that the operator on the stack must be done first, while right to left association means the reverse.

	current symbol	operator stack	postfix string
1	A		A
2	-	-	A
3	B	-	A B
4	+	+	A B -
5	C	+	A B - C
6			A B - C +

In line 4, the '-' will be popped and printed before the '+' is pushed onto the stack. Both operators have the same precedence level, so left to right association tells us to do the first one found before the second.

5. $A * B ^ C + D$ becomes $A B C ^ * D +$

Here both the exponentiation and the multiplication must be done before the addition.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A

3	B	*	A B
4	^	* ^	A B
5	C	* ^	A B C
6	+	+	A B C ^ *
7	D	+	A B C ^ * D
8			A B C ^ * D +

When the '+' is encountered in line 6, it is first compared to the '^' on top of the stack. Since it has lower precedence, the '^' is popped and printed. But instead of pushing the '+' sign onto the stack now, we must compare it with the new top of the stack, the '*'. Since the operator also has higher precedence than the '+', it also must be popped and printed. Now the stack is empty, so the '+' can be pushed onto the stack.

6. $A * (B + C * D) + E$ becomes A B C D * + * E +

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(* (A
4	B	* (A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

A summary of the rules follows:

1. Print operands as they arrive.

2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain).

QUEUES:

A queue is an ordered collection of elements, where the elements can be inserted at one end and deleted from another end. It works on the principle called First-In-First-Out (FIFO).

The information we retrieve from a queue comes in the same order that it was placed on the queue. The examples of queues are checkout line at supermarket cash register, line of cars waiting to proceed in some fixed direction at an intersection of streets.

The queue can be implemented using arrays and linked lists.

Operations On Queues:

Insertion: The operation of adding new items on the queue occurs only at one end of the queue called the **rear end**.

Deletion: The operation of removing items of the queue occurs at the other end called the **front end**.

The following is an algorithm for inserting an element into the queue.

The variable FRONT holds the index of the front item and REAR variable holds the index of the last item. Array elements begin at 1 and the maximum elements of the Queue is MAX. Q is the Queue. ITEM is the element to be inserted.

ALGORITHM: Inserting an element into Queue
Queue_insert(ITEM)

If REAR \geq MAX then write `_Queue Overflow _`

Return

REAR \leftarrow REAR +1

Q[REAR] \leftarrow ITEM

IF FRONT ==0 THEN

FRONT \leftarrow 1

Return

The following is the algorithm for detecting an element from the queue.

ALGORITHM: Deleting an elements from the Queue

Queue_delete()

If FRONT =0 then

Write(`_Queue under flow'`).

Return

K \leftarrow Q[FRONT]

If FRONT = REAR then

Begin

FRONT \leftarrow 0

REAR \leftarrow 0

End

Else

FRONT \leftarrow FRONT +1

Return K

UNIT – V

File is a collection of bytes that is stored on secondary storage devices like disk.

A File is an external collection of related data treated as a unit.

The primary purpose of a file is to keep record of data. Record is group of related fields, Field is a group of characters they convey meaning.

Files are stored in auxiliary of secondary storage devices. The two common forms of secondary storage are disk (hard disk, CD and DVD) and tape.

Each file ends with an end of file (EOF) at a specified byte number, recorded in file structure.

There are two kinds of files in a system. They are,

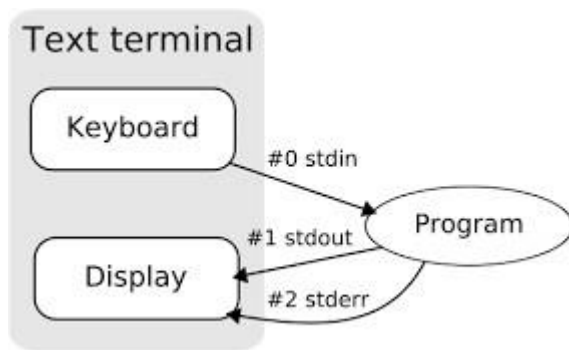
1. Text files (ASCII)
 2. Binary files
- Text files contain ASCII codes of digits, alphabetic and symbols.
 - Binary file contains collection of bytes (0's and 1's). Binary files are compiled version of text files.

Streams:

- In C all input and Output is done with streams.
- Stream is nothing but the sequence of bytes of data.
- A sequence of bytes flowing into program is called input stream
- A sequence of bytes flowing out of the program is called output stream
- Use of stream make I/O machine independent.

Predefined Streams :

stdin	Standard Input
stdout	Standard Output
stderr	Standard Error



Standard Input stream Device:

1. Stdin stands for standard Input
2. Keyboard is standard input device.
3. Standard input data going into a program.
4. The program requests data transfers by use of the read operation.
5. Not all programs require input.

Standard output stream Device:

1. Stdout stands for Standard output
2. Screen is standard output device.
3. Standard output is data going out from a program.
4. The program sends data to output device by using write operation.

Difference Between Std. Input and Output Stream Devices :

Point	Std i/p Stream Device	Standard o/p Stream Device
Stands For	Standard Input	Standard Output
Example	Keyboard	Screen/Monitor
Data Flow	Data (Often Text) going into a program	data (Often Text) going out from a program
Operation	Read Operation	Write Operation

Some Important Summary :

Point	Input Stream	Output Stream
-------	--------------	---------------

Point	Input Stream	Output Stream
Standard Device 1	Keyboard	Screen
Standard Device 2	Scanner	Printer
IO Function	scanf and gets	printf and puts
IO Operation	Read	Write
Data	Data goes from stream	data comes into stream

FILE OPERATIONS:

There are 4 basic operations that can be performed on any files in C programming language. They are,

1. Opening a file
2. Closing a file
3. Reading a file
4. Writing in a file

Let us see the syntax for each of the above operations in a table:

File operation/ Syntax	Description
file open FILE *fp; fp=fopen(—filename[, _mode]);	fopen function is used to open a file. Where, fp is file pointer to the data type -FILE.
file close: fclose(fp);	fclose function closes the file that is being pointed by file pointer fp.
file read: fgets(buffer, size, fp);	fgets is used to read a file line by line. where, buffer – buffer to put the data in. size – size of the buffer fp – file pointer
file write: fprintf(fp, -some data); fprintf(fp, -text %d[, variable_name]);	fprintf writes the data into a file pointed by fp.

Mode of operations

There are many modes in opening a file. Based on the mode of file, it can be opened for reading or writing or appending the texts. They are listed below.

- **r** – Opens a file in read mode and sets pointer to the first character in the file. It returns null if file does not exist.
- **w** – Opens a file in write mode. It returns null if file could not be opened. If file exists, data are overwritten.
- **a** – Opens a file in append mode. It returns null if file couldn't be opened.
- **r+** – Opens a file for read and write mode and sets pointer to the first character in the file.
- **w+** – opens a file for read and write mode and sets pointer to the first character in the file.
- **a+** – Opens a file for read and write mode and sets pointer to the first character in the file. But, it can't modify existing contents.

Inbuilt file handling functions in C language:

C programming language offers many inbuilt functions for handling files. They are given below.

File handling functions	Description
fopen ()	fopen () function creates a new file or opens an existing file.
fclose ()	fclose () function closes an opened file.
getw ()	getw () function reads an integer from file.
putw ()	putw () functions writes an integer to file.
getc (), fgetc ()	getc () and fgetc () functions read a character from file.
putc (), fputc ()	putc () and fputc () functions write a character to file.
gets ()	gets () function reads line from keyboard.
puts ()	puts () function writes line to o/p screen.
fgets ()	fgets () function reads string from a file, one line at a time.
fputs ()	fputs () function writes string to a file.
feof ()	feof () function finds end of file.
fgetchar ()	fgetchar () function reads a character from keyboard.
fgetc ()	fgetc () function reads a character from file.
fprintf ()	fprintf () function writes formatted data to a file.
fscanf ()	fscanf () function reads formatted data from a file.
fgetchar ()	fgetchar () function reads a character from keyboard.
fputchar ()	fputchar () function writes a character from keyboard.

fseek ()	fseek () function moves file pointer position to given location.
SEEK_SET	SEEK_SET moves file pointer position to the beginning of the file.
SEEK_CUR	SEEK_CUR moves file pointer position to given location.
SEEK_END	SEEK_END moves file pointer position to the end of file.
ftell ()	ftell () function gives current position of file pointer.
rewind ()	rewind () function moves file pointer position to the beginning of the file.
getc ()	getc () function reads character from file.
getch ()	getch () function reads character from keyboard.
getche ()	getche () function reads character from keyboard and echoes to o/p screen.
getchar ()	getchar () function reads character from keyboard.
putc ()	putc () function writes a character to file.
putchar ()	putchar () function writes a character to screen.
printf ()	printf () function writes formatted data to screen.
sprintf ()	sprintf () function writes formatted output to string.
scanf ()	scanf () function reads formatted data from keyboard.
sscanf ()	sscanf () function Reads formatted input from a string.
remove ()	remove () function deletes a file.
fflush ()	fflush () function flushes a file.

Opening a file – for creation and edit

Opening a file is performed using the library function in the "**stdio.h**" header file:

fopen():

The syntax for opening a file in standard I/O is:

```
ptr= fopen( -file open, -mode);
```

Closing a File:

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using library function `fclose()`.

```
fclose(fp);
```

Reading and writing to a text file:

For reading and writing to a text file, we use the functions `fprintf()` and `fscanf()`.

They are just the file versions of `printf()` and `scanf()`. The only difference is that, `fprint` and `fscanf` expects a pointer to the structure `FILE`.

Other functions like `fgetchar()`, `fputc()` etc. can be used in similar way.

Reading and writing to a binary file:

Functions `fread()` and `fwrite()` are used for reading from and writing to a file on the disk respectively in case of binary files.

Writing to a binary file:

To write into a binary file, you need to use the function `fwrite()`. The function takes four arguments: Address of data to be written in disk, Size of data to be written in disk, number of such type of data and pointer to the file where you want to write.

```
fwrite(address_data,size_data,numbers_data,pointer_to_file);
```

Reading from a binary file:

Function `fread()` also take 4 arguments similar to `fwrite()` function as above.

```
fread(address_data,size_data,numbers_data,pointer_to_file);
```

Getting data using `fseek()`

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record.

This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using `fseek()`.

As the name suggests, `fseek()` seeks the cursor to the given record in the file.

Syntax:

```
fseek(FILE * stream, long int offset, int whence)
```

The first parameter `stream` is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

Different Whence in fseek

Whence	Meaning
SEEK_SET	Starts the offset from the beginning of the file.
SEEK_END	Starts the offset from the end of the file.
SEEK_CUR	Starts the offset from the current location of the cursor in the file.

File System Functions:

You can erase a file using **remove()**.

Its prototype is

int remove(char *file-name);

You can position a file's current location to the start of the file using **rewind()**. Its prototype is

void rewind(FILE *fp);

File Status Function:

C provides function for error handling during I/O operations. Typical error situations include:

- Trying to read beyond the end-of-file mark.
- Device overflow
- Trying to use a file that has not been opened.
- Trying to perform an operation on a file, when the file is opened for another type of operation.
- Opening a file with an invalid filename.

We have two status-inquiry library functions, **feof ()** and **ferror()** that can help us detect I/O error in the files.

EOF:

The **feof** function can be used to test for an end of file condition.

n = feof(fp);

Error handling function:

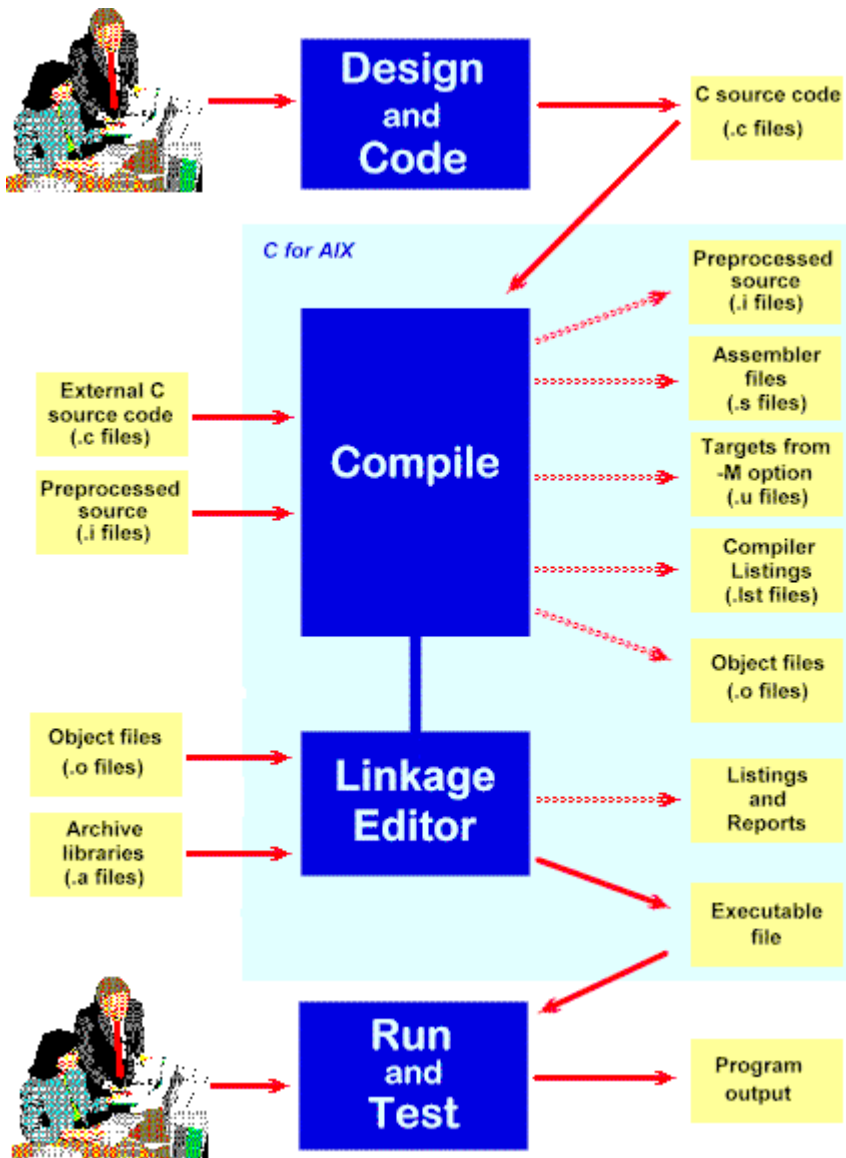
errno, perror and strerror()

The C programming language provides **perror()** and **strerror()** functions which can be used to display the text message associated with **errno**.

- The **perror()** function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.
- The **strerror()** function, which returns a pointer to the textual representation of the current errno value.

Program Development:

A C program typically passes through four steps of development.



The solid lines show inputs into each step of the development cycle. Compile and Linkage Editor Operations are performed by the C for AIX product, which also lets you specify what optional outputs are produced. Optional outputs are shown in the diagram by the broken lines. Descriptions of the steps follow below:

Design and Code Involves designing a program to meet a specified requirement, and creating the programming language text files that will comprise the program source.

Compile After checking for syntactical correctness, converts the programming language source files into machine readable instructions, where C variables are associated with memory addresses, and C statements are turned into a series of machine language instructions. The compiler can produce various forms of output, depending on the compiler options selected.

Linkage Editor Links compiler output with external modules requested by the compiled program. C programs can use routines from C libraries or any object or archive file from the IBM XL family of languages. C programs can also use modules produced by the current or previous compilations. As well as linking the external modules, the linkage editor resolves addresses within the object module.

Run and Test This stage can be both the final step in program development, or it can be an intermediate point in the program design and implementation process. A program's design commonly is further refined as a result of information gathered during testing.

Multiple File Compilation

Multiple File Projects:

Most of the time, full programs are not contained in a single file. Many small programs are easy to write in a single file, but larger programs involve separate files containing different modules. Usually, files are separated by related content.

A class module normally consists of:

- A header file - contains the declaration of the class (without implementation details)
- An implementation file - contains implementations of the class members

Filenames:

Header files are usually in the format *filename.h*, and implementation files are usually in the format *filename.cpp*. It is a good idea to use the same base filename for corresponding header and implementation files. Example:

```
circle.h // header file for a class called Circle
circle.cpp // implementation file for Circle class
```

Filenames do not have to be the same as the class name, but well-chosen filenames can help identify the contents or purpose of a file.

When classes are used in a program, the main program would usually be written in a separate file.

The "compilation" of a program actually consists of two major stages.

1. Compile stage

- Syntax checked for correctness.
- Variables and function calls checked to insure that correct declarations were made and that they match. (Note: The compiler doesn't need to match function definitions to their calls at this point).
- Translation into **object code**. Object code is just a translation of your code file -- it is not an executable program, at this point. (Note: the word "object" in object code does not refer to the definition of "object" that we use to define object-oriented programming. These are different terms.)

2. Linking stage

- Links the object code into an executable program.
- May involve one or more object code files.
- The linking stage is the time when function calls are matched up with their definitions, and the compiler checks to make sure it has one, and only one, definition for every function that is called.
- The end result of linking is usually an executable program.

Putting together a multiple-file project:

For a simple example like our Fraction example, it may be tempting to simply use the following statement inside the main.cpp file:

```
#include "frac.cpp"
```

and then just compile the main.cpp file with a single command. This will work in this example, because it's a linear sequence of #includes -- this essentially causes the whole thing to be put together into one file as far as the compiler is concerned. This is **not** a good idea in the general case. Sometimes the line-up of files is not so linear. The separate ideas of **compiling** and **linking** allow these steps to be done separately and there are some good reasons and benefits:

- Changes to a file require only that file to be re-compiled (rather than everything), along with the re-linking
- Often, libraries are distributed in pre-compiled format, so trying to #include the .cpp file would not even be feasible. (A pre-compiled library would still give you the actual .h file for the #include statements, to satisfy declare-before-use in your own code).

Separate Compilation of functions:

All of the programs that we have looked at to this point have been contained within a single file. This is the exception rather than the rule. As programs become larger, it is important to spread them across files of a reasonable size. In this lesson we will look at why this is important, and how it can be done.

Compiling Files:

Up to this point, whenever you wanted to compile a program you compiled the file displayed in an Emacs buffer. The Compile This File... option of Emacs' Compile menu (or equivalently, **M-x compile**) is a handy feature when your entire program is contained with one file. It is, of course, possible to compile a file completely independently of Emacs.

Download hello.c. It is a simple program that prints out "Hello world", and you can compile and run it without ever displaying it in Emacs. To compile it, go to a Unix Shell window, connect to the directory into which you downloaded the program, and enter the following command:

```
gcc -Wall -g -c hello.c
```

This command will result in a new file being created. Use the ls command to list your directory. What file is there that wasn't before?

Answer: You should see a file called "hello" in your directory. This is the executable. The name of the executable came from the command that you just typed--it is the name that follows the -o in the command.

Now that you have an executable you can run it in the familiar way. Simply type "hello" into the UNIX Shell window.

To summarize, the steps involved in compiling, linking, and running a program are:

1. Compile the ".c" file containing the source code with a command such as

```
gcc -Wall -g -c hello.c
```

The -c in the command is important--it tells C to produce an object file called (in this case) "hello.o". The -Wall part is optional but is strongly encouraged--it tells C to produce all possible warning messages. A warning message will usually lead you to a bug in your program. The -g tells the compiler to include information needed by the debugger.

2. Link the ".o" file to produce an executable with a command such as

```
gcc -o hello hello.o -lm
```

The -o hello in the command is important--it tells C what to name the executable. The -lm part tells C to link in the math libraries. If your program doesn't use any of the math functions from "math.h", you can leave this part out.

3. Run the executable in the usual way.

If your entire program is contained within a *single* file, you can do the compilation and linking in one step by entering the command

```
gcc -Wall -g -o hello hello.c -lm
```

This is exactly what Emacs does when it compiles a program for you. We have been careful to this point to do the compilation in two separate steps so that you will be able to deal with programs made up of more than one file.

Basic Non-Linear Data Structures

Basic Non-Linear Data Structures: Introduction, Definition and terminology of Trees, Graphs.

A **data structure** is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.

Data structures are used in almost every program or software system. Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables. Data structures are widely applied in the following areas:

- Compiler design
- Statistical analysis package
- Numerical analysis
- Artificial intelligence
- Operating system
- DBMS
- Simulation
- Graphics

When selecting a data structure to solve a problem, the following steps must be performed.

1. Analysis of the problem to determine the basic operations that must be supported. For example, basic operation may include inserting/deleting/searching a data item from the data structure.
2. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements

Data structures are building blocks of a program. A program built using improper data structures may not work as expected. So as a programmer it is mandatory to choose most appropriate data structures for a program. The term data means a value or set of values. It specifies either the value of a variable or a constant (e.g., marks of students, name of an employee, address of a customer, value of pi, etc.).

While a data item that does not have subordinate data items is categorized as an elementary item, the one that is composed of one or more subordinate data items is called a group item. A record is a collection of data items. For example, the name, address, course, and marks obtained are individual data items. But all these data items can be grouped together to form a record. A file is a collection of related records

15. ADDITIONAL TOPICS

16. UNIVERSITY QUESTION PAPERS OF PREVIOUS YEARS

17. QUESTION BANK

UNIT-I

1. Define a enumerated data type?
2. Write about type definition (typedef)?
3. How do you declare and initialize a structure?
4. Define Union and Structure?
5. Define a Bit field with an example?
6. What are various Command line arguments?
7. Define a structure and state how the members of a structure are accessed with example?
8. Write the major differences between arrays and structures?
9. Write an example of nested structure?
10. State the difference between a structure and union?
11. Write an example of array of structures?
12. Write the general format of sending a copy of a structure to the called function?
13. What are various operators supported by structures? Explain with a example
14. Explain about Complex structures?
15. Write briefly about Nested Structures in C?
16. Explain about passing structures through pointers in C?
17. Explain the process of initialization and declaration of structures
18. State the differences between structures and unions
19. Illustrate self referential structures
20. Explain different kinds of operations on structures in c
21. What is the need of bit fields? Illustrate with an example the uses of bit fields in C?
22. Explain the process of passing arrays through structures?
23. Explain with an example the process of Dynamic memory allocation using structures in C?
24. Explain in detail about Structures and Functions in c?
25. Explain about Structures and Unions and differentiate between them? What are the advantages of Structures over unions?
26. Predict the output of following programs. Assume that unsigned int takes 4 bytes and long int takes 8 bytes.

a).#include <stdio.h>
struct test
{
 unsigned int x;
 unsigned int y: 33;
 unsigned int z;
};
int main()

```

{
    printf("%d", sizeof(struct test));
    return 0;
}

```

b) #include <stdio.h>

```

struct test

```

```

{
    unsigned int x;
    long int y: 33;
    unsigned int z;
};

```

```

int main()

```

```

{
    struct test t;
    unsigned int *ptr1 = &t.x;
    unsigned int *ptr2 = &t.z;
    printf("%d", ptr2 - ptr1);
    return 0;
}

```

c) #include <stdio.h>

```

union test

```

```

{
    unsigned int x: 3;
    unsigned int y: 3;
    int z;
};

```

```

int main()

```

```

{
    union test t;
    t.x = 5;
    t.y = 4;
    t.z = 1;
    printf("t.x = %d, t.y = %d, t.z = %d",
        t.x, t.y, t.z);
    return 0;
}

```

27. Use bit fields in C to figure out a way whether a machine is little endian or big endian.

UNIT-II

1. Narrate following operations on single linked list
 - A. Traversing
 - B. Insertion before head node
 - C. Insertion after tail node
2. Describe following operations on single linked list
 - A. deleting head node
 - B. copying
 - C. deleting tail node
3. Explain following operations on single linked list with algorithm
 - A. Deleting a specified node
 - B. Insertion after specified node
4. Write about the following operations on double linked list
 - a. Deleting head node
 - b. Insertion after specified node
 - c. Insertion before head node
5. Draw the node structure of a double linked list
 - a) Differentiate static data structure and dynamic data structure
 - b) What are the applications of -linked lists?

UNIT-III

1. Define sorting.
2. What is the importance of sorting?
3. List different sorting techniques.
4. Define searching.
5. Define linear search.
6. Define binary search.
7. Explain the working of selection sort.
8. Explain the working of Quick Sort sort.
9. Explain the working of insertion sort.
10. Which technique of searching an element in an array would you prefer to use and in which situation?
11. What are the different types of sorting techniques? Which sorting technique has the least worst case?
12. Explain the difference between Quick Sort sort and selection sort. Which one is more efficient?
13. Sort the elements 77, 49, 25, 12, 9, 33, 56, 81 using insertion sort.
14. Sort the elements 77, 49, 25, 12, 9, 33, 56, 81 using selection sort.
15. Sort the elements 77, 49, 25, 12, 9, 33, 56, 81 using Quick Sort sort.
16. A certain sorting technique was applied to the following data set, 45, 1, 27, 36, 54, 90. After two passes, the rearrangement of the data set is given as below: 1, 27, 45, 36, 54, 90. Identify the sorting algorithm that was applied.
17. A certain sorting technique was applied to the following data set, 81, 72, 63, 45, 27, 36. After two passes, the rearrangement of the dataset is given as: 27, 36, 80, 72, 63, 45. Identify the sorting algorithm that was applied.

18. A certain sorting technique was applied to the following data set, 45, 1, 63, 36, 54, 90. After two passes, the rearrangement of the data set is given as below: 1, 45, 63, 36, 54, 90. Identify the sorting algorithm that was applied.
19. Discuss the advantages of insertion sort.
20. Write a recursive function to perform selection sort.
21. Compare the running time complexity of different sorting algorithms.
22. Describe insertion sort with a proper algorithm. What is the complexity of insertion sort in the worst case?
23. How many key comparisons and assignments an insertion sort makes in its worst case?
24. Quick Sort sort algorithm is inefficient because it continues execution even after an array is sorted by performing unnecessary comparisons. Therefore, the number of comparisons in the best and worst cases is the same. Modify the algorithm in such a fashion that it will not make the next pass when the array is already sorted.
25. How many comparisons are required to find 73 in the following array using binary search 12,25,32,37,41,48,58,60,66,73,74,79,83,91,95.
26. Write a program to implement linear search.
27. Differentiate between linear and binary search.
28. Write a program to implement Quick Sort sort.
29. Write a program to implement selection sort.

UNIT -IV

1. What is ADT? Write stack ADT?
2. When the stack is overflow?
3. What is the difference between push and pop?
4. How does variable declaration affect memory allocation?
5. What is the difference between FIFO and LIFO?
6. Convert the following expression to postfix $A+b-c$.
7. Explain stack operations with an example
8. Explain dequeue operations using linked list in c
9. Explain Enqueue operations in a queue.
10. Write a c program to implement stack using arrays?
11. Convert the following expression from infix to postfix $(a+b)*c/(a+b*c)$
12. Write a c program to implement queue using linked list
13. Find the smallest element in the stack elements using array?

UNIT V

1. How is a file pointer declared?
2. Explain difference between EOF and feof() in c?
3. What is difference between file opening mode r+ and w+?
4. What functions are used for character I/O?
5. What is meant by flushing of a file?
6. What is the use of rewind() function?
7. State the functions for direct file I/O.
8. Define *fputc* function and *putc* macro.

9. Describe *fgetc* function and *getc* macro.
 10. Explain about the functions for reading data from a file.
 11. What are different types of files? Explain.
 12. What is the use of command line arguments?
 13. Define a Tree.
 14. List the applications of trees.
 15. Define the terms node, degree, siblings, depth/height and level?
 16. Define a graph.
 17. Differentiate between file, buffer and stream.
 18. Differentiate between *printf()* and *fprintf()*.
 19. Differentiate between *scanf()* and *fscanf()* functions with examples.
 20. State the purpose of *fseek()*.
 21. Explain *fopen()* for an existing file in write mode?
 22. How can we know read/write permission any given file?
 23. Differentiate between text and binary file? Explain with suitable example?
 24. Explain about file i/o operations in C with example.
 25. Discuss about the formatted I/O.
 26. Illustrate the character input/output functions with suitable examples.
 27. Demonstrate the use of *fread()* and *fscanf()* for reading sequentially from a disk.
 28. Explain about the *fprintf* and *fscanf* functions with examples.
 29. Explain about the *fopen* and *fclose* functions with examples.
 30. What functions are used for file positioning? State the *SEEK_* Constants used in *fseek* function and explain the meaning.
 31. State the functions used for accessing files randomly? Explain with examples.
 32. Mention the different file opening modes that can be used with *fopen()*.
 33. Discuss about unformatted I/O with suitable examples.
 34. How many ways are used to represent a tree? Discuss with examples.
 35. Discuss representation of graph with examples
 36. Write a program to read a text file and to print the count the no of tab characters in a given file.
 37. Write a program to read a text file and to count the no of uppercase letters in a given file.
 38. Write a 'C' program to append the contents of one file to another.
 39. Write a program to generate prime numbers in a given range and append them to primes data file and display the file.
 40. Write a C program to read name and marks of n number of students from user and store them in a file. If the file previously exists, add the information of n students.
 41. Write a program to print file contents in reverse order
 42. Write a program to copy one file contents into another file in reverse order.
 43. Write a program to display contents of the file on screen.
 44. Write a program to merge two files into single file.
 45. Write a C program to read a text file and to count Number of characters, Number of words and Number of sentences and write in an output file.
 46. Write a C program to read the text file containing some paragraph. Use *fseek()* and read the text after skipping n characters from the beginning of the file.
 47. Write a C program to read the text file containing some paragraph. Print the first n characters from the beginning of the file.
 48. Explain the terms used in Trees.
 49. Explain the terminology used in Graphs.
-

18. ASSIGNMENT QUESTIONS

UNIT-I

1. Define a structure and explain declaration and initialization of a structure with examples.
2. Define Union and explain the declaration of a Union with examples.
3. Distinguish between structure and Union.
4. What are nested structures. Explain various ways in which we can nest structure with appropriate examples.

UNIT II

1. Write a C program to implement the following operations using Singlylinked list.
 - a. Create a single linked list with N number of nodes.
 - b. Insert a new node at the end.
 - c. Delete a node from the beginning.
 - d. Insert a node at the middle
2. Write a C program to reverse the elements stored in a single linked list.
3. Analyze the function and find the output of following function for start pointing to first node of following linked list?
1->2->3->4->5->6
void fun(struct node* start)
{
if(start == NULL)
return;
printf("%d ", start->data);
if(start->next != NULL)
fun(start->next->next);
printf("%d ", start->data);
}
4. The following C function takes a single-linked list of integers as a parameter and rearranges the elements of the list. The function is called with the list containing the integers 1, 2, 3, 4, 5, 6, 7 in the given order. Analyze the code and find the contents of the list after the function completes execution?

```
struct node
{
    int value;
    struct node *next;
};
void rearrange(struct node *list)
{
    struct node *p, *q;
    int temp;
    if ((!list) || !list->next)
        return;
    p = list;
    q = list->next;
```



```

while(q)
{
    temp = p->value;
    p->value = q->value;
    q->value = temp;
    p = q->next;
    q = p ? p->next:0;
}
}

```

5. You are given pointers to first and last nodes of a singly linked list, Identify the following operations which are dependent on the length of the linked list?
 - a. Delete the first element
 - b. Insert a new element as a first element
 - c. Delete the last element of the list
 - d. Add a new element at the end of the list

UNIT-III

1. Write a recursive function to perform selection sort.
2. Compare the running time complexity of different sorting algorithms.
3. Describe insertion sort with a proper algorithm. What is the complexity of insertion sort in the worst case?
4. Write a program to implement Quick Sort sort.
5. Sort the elements 8, 22, 7, 9, 31, 19, 5, 13 using selection sort, Quick Sort sort, insertion sort

UNIT-IV

1. What is a Data Structure? Explain the concept of a linear list, queue and stack.
2. What is post fix and pre fix notation. Explain stack implementation for both with appropriate example.
3. Write a C program to evaluate a prefix notation
4. Illustrate the difference between Stack and Queue

UNIT-V

1. Write a C program to read a text file and to count
 - a) number of characters
 - b) number of words and
 - c) number of sentences and write in an output file.
2. What are the file I/O functions in C. Give a brief note about the task performed by each function with appropriate example.
3. (a) How does an append mode differ from a write mode.


- (b) Compare between printf and fprintf functions.
- (c) Write a program to copy up to 100 characters from a file to an output array.
4. Write a C program to replace every fifth character of the data file using fseek() command.
5. Write a program to read the following data to find the value of each item and display the contents of the file.

Item	Code	Price	Quantity
Pen	101	Rs.20	5
Pencil	103	Rs.3	100

6. Explain about error handling in C with examples.

19. UNIT WISE QUIZ QUESTIONS

UNIT-I

1. The member variable of structure are accessed by using
 - a) dot (.) operator
 - b) arrow () operator
 - c) asterisk (*) operator
 - d) ampersand (&) operator
 2. Identify the most appropriate of structure are accessed by using
 - a) Union contain member of different data types which share the same storage area in memory
 - b) Union are like structures
 - c) union are less frequently used in the program
 - d) union are used for set operation
 3. The typedef statement is used for
 - a) declaring user-defined data types
 - b) declaring different variables
 - c) for typecasting of variable
 - d) none of the above
 4. The structure combines variable of
 - a) dissimilar data types
 - b) similar data types
 - c) unsigned data types
 - d) none of the above
 5. Bit fields are used only with
 - a) unsigned int data type
 - b) float data type
 - c) char data type
 - d) int data type
 6. The union holds
 - a) one object at time
 - b) multiple objects
 - c) both (a)&(b)
 - d) none of the above
 7. Interrupt 0*21 is a
 - a) software interrupt
 - b) hardware interrupt
 - c) both (a)&(b)
 - d) none of the above
 8. The 86() function invokes
-

- a) ROM-BIOS service
- b) DOS services
- c) Both (a) & (b)
- d) none of the above

9. The intdos() function invokes interrupt number

- a) 0*21
- b) 0*17
- c) 0 * 18
- d) none of the above

10. The number of bytes required for enumerated data type in memory are

- a) 2 bytes
- b) 4 bytes
- c) 1 bytes
- d) 3 bytes

11. The service number is always initialized in the register

- a) AH
- b) AL
- c) BH
- d) AX

12. Observe the following program neatly and choose the appropriate printf() statement

from the options

```
struct month
```

```
{
```

```
Char *month;
```

```
};
```

```
void main()
```

```
{
```

```
struct month m={—Marchll};
```

```
-----
```

```
}
```

- a) printf(-\n Month :%s|m.month);
- b) printf(-\n Month :%s|m **7** month);
- c) printf(-\n Month :%s|m.*month);
- d) printf(-\n Month :%s|,*m.month);

13. what will be the value of m displayed on axecution of the following programe?

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

```
struct bit
```

```
{
```

```
unsigned int m:3;
```

```
}
```

```
void main()
```

```
{
```

```
struct bit b={ 8};
```

```
printf(-\n m=%d|b.m);
```

```
}
```

- a) m=0
- b) m=8
- c) m=3
- d) none of the above

14. The size of structure in bytes occupied in the following program will be

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

```

struct bit
{
    unsigned int m:4;
    unsigned int x:4;
    int k;
    float bit f;
};
struct bit b;
void main()
{
    printf(-size of structure in bytes =%d, sizeof(b));
}

```

a) 8 b) 10 c) 7 d) 4

UNIT II

1. Logical organization of data in computer memory is called []
 A. data style B. data manner C. data set D. all are wrong
2. A 2D array which contains majority of elements as null or zero []
 A. sparse matrix B. sparse matrix C. null matrix D all are correct
3. Address of the first element of an array is called []
 A. starting address B. base address C. both p & q are valid D. all are correct
4. Which of the following refers indexing formula []
 A. $a[i] = M + (i - L) * w$ B. $a[i] = M + (i + L) * w$ C. $a[i] = M - (i + L) * w$ D. $a[i] = M * (i - L) + w$
5. Arrays are not static data structures []
 A. True B. false C. can't determine D. may be
6. A pointer array variable contains ----- as its elements []
 A. integer values B. addresses of memory locations C. both p & q are valid D. none
7. What does link part of a tail node in a circular linked list hold? []
 A. address of head node B. address of header node C. address of it's previous node
 D. none
8. What is the value present in the address part of a header node in linked list []
 A. address of head node B. address of header node
 C. address of tail node D. all are wrong
9. Tail node in circular linked list always points []
 A. header node B. head node C. NULL value D. it's next node
10. How can we access the value present at nth- row ,pth- plane, mth- column of a multi - dimensional array -x []
 A. $x[n][p][m]$ B. $x[n][m][p]$ C. $x[p][n][m]$ D. $x[m][n][p]$
11. memory for elements in an array is not allocated in contiguous locations(addressed. []
 A. true B. false C. not possible D. can't predict

12. Each node in a linked list must contain at leastfields []
 A.3 B.2 C.4 D.5
13. LLINK is the pointer pointing to the[]
 A. successor node B. predecessor node C. head node D. last node
14. A linear list in which the pointer points only to the successive node is.....[]
 A. singly linked list B. double linked list C. circular linked list D. all are correct
15. If address part of the head node contains non null value, it means []
 A. list has exactly 1 node
 B. list has exactly 2 nodes address
 C. list has at least 2 nodes
 D. all are wrong
16. Let us suppose, header node address is:100,head node address is:200 and a new node(address is 300) is inserted after the header node. Then data part of the header node contains []
 A.100 B.200 C.300 D.NULL
17. Llink part of the header node in a circular double linked list contains []
 A. address of left most node B. address of right most node
 C. address of head node D. both p&r
18. Which of the following is disadvantage in using -linked lists? []
 A .requires more memory B. dynamic data structure
 C. elements are deleted and inserted easily D. none
19. Memory for elements in linked list are allocated in non-contiguous locations []
 A. true B. false C. can't determine D. may be
20. Which of the following data structures are convenient to perform insertion and deletion operations []
 A. arrays B. linked lists C. both A&B D. all are wrong
21. Which of the following is disadvantage of arrays
 A. static storage B. insertions require shifting of elements []
 C. deletions require shifting of elements D. all are correct
22. To traverse an array means []
 A. to process each element in an array B. to delete an element from an array
 C. to insert an element into an array D. to combine two arrays into a single array
23. Finding the location of the element with a given value is: []
 A. Traversal B. Search C. Sort D. None of above
24. If the base address of a character array is 200 then what is the address of 3rd element in that array []
 A.202 B.203 C.204 D. none
25. The memory address of first element in array is called? []
 A. base address B. first address C. stating address

D. both a & c

26. the address of first node in a linked list is?
A. can't say B. 100 C. 0 D. none
27. A doubly linked list haspointers with each node. []
A. 0 B. 1 C. 2 D. 3
28. A doubly linked list is also called as..... []
A. linked list B. one way chain C. two way chain D. right link
29. In a linked list, insertion can be done as []
A. beginning B. end C. middle D. all of the above
30. To implement Sparse matrix dynamically, the following data structure is used []
A. Trees B. Graphs C. Priority Queues D. Linked List
31. Header of a linked list is a special node at the []
A. end of the linked list B. at the middle of the linked list
C. beginning of the linked list D. none of these
32. Which of the following operations is performed more efficiently by a doubly linked list than by a linear linked list? []
A. Deleting nodes whose location is given
B. Searching an unsorted list for a given item
C. Inserting a node after the node with a given location
D. Traversing the list to process each node
33. Overflow condition in a linked list may occur when attempting to []
A. create a node when free space pool is empty
B. traverse the nodes when free space pool is empty
C. create a node when linked list is empty
D. none of these

UNIT-III

1. The worst case complexity is _____ when compared with the average case complexity of a binary search algorithm.
(a) Equal (b) Greater (c) Less (d) None of these
2. The complexity of binary search algorithm is
(a) $O(n)$ (b) $O(n^2)$ (c) $O(n \log n)$ (d) $O(\log n)$
3. Which of the following cases occurs when searching an array using linear search the value to be searched is equal to the first element of the array?
(a) Worst case (b) Average case (c) Best case (d) Amortized case
4. A card game player arranges his cards and picks them one by one. With which sorting technique can you compare this example?
-

(a) Quick Sort sort (b) Selection sort (c) Merge sort (d) Insertion sort

5. In which sorting, consecutive adjacent pairs of elements in the array are compared with each other?

(a) Quick Sort sort (b) Selection sort (c) Insertion sort (d) Radix sort

6. Time complexity of Quick Sort sort in best case is

a) $\theta(n)$ b) $\theta(n \log n)$ c) $\theta(n^2)$ d) $\theta(n(\log n)^2)$

6. Performance of the linear search algorithm can be improved by using a_____.

7. The complexity of linear search algorithm is_____.

8. Sorting means_____.

9. _____sort shows the best average-case behaviour.

10. _____deals with sorting the data stored in files.

11. $O(n^2)$ is the running time complexity of_____algorithm.

12. In the worst case, insertion sort has a_____running time

13. Linear Search is best for ... ?

a) Large arrays b) Small or Large, but unsorted arrays c) Small arrays d) Small or Large, but sorted arrays

14. The disadvantage of Binary Search is ?

A. It has the overhead of sorting

B. It may not work for floating point numbers

C. Its performance depends on the position of the search element in the array

D. It may not work for strings

15. It is not a good idea to use binary search to find a value in a sorted linked list of values. Why?

A. Binary search may not work for a linked list

B. Binary search using a linked list would usually be much slower than Linear search

C. It is not possible to sort a linked list, but sorted data is needed for binary search.

D. Linked list uses dynamic memory allocation. So the number of items are not known.

16. Which of the following case does not exist in complexity theory

A. Best case B. Worst case C. Average case D. Null case

17. The Worst case occur in linear search algorithm when
- A. Item is somewhere in the middle of the array
 - B. Item is not in the array at all
 - C. Item is the last element in the array
 - D. Item is the last element in the array or is not there at all
18. The Average case occur in linear search algorithm
- A. When Item is somewhere in the middle of the array
 - B. When Item is not in the array at all
 - C. When Item is the last element in the array
 - D. When Item is the last element in the array or is not there at all
19. The operation of processing each element in the list is known as
- A. Sorting
 - B. Merging
 - C. Inserting
 - D. Traversal
20. Finding the location of the element with a given value is:
- A. Traversal B. Search C. Sort D. None of above
21. If the given input array is sorted or nearly sorted, which of the following algorithm gives the best performance?
- A. Insertion sort B. Selection sort C. Quick Sort sort
22. The number of swappings needed to sort the numbers 8, 22, 7, 9, 31, 19, 5, 13 in ascending order using Quick Sort sort is
- A. 11 B. 14 C. 13 D. 12
23. You have to sort a list L consisting of a sorted list followed by a few "random" elements. Which of the following sorting methods would be especially suitable for such a task?
- A. Quick Sort sort B. Selection sort C. Quick sort D. Insertion sort

UNIT-IV

1. If the sequence of operations - push(1), push(2), pop, push(1), push(2), pop, pop, pop, push(2),pop are performed on a stack, the sequence of popped out values are:

a. 2 2 1 1 2

b. 2 2 1 2 2

c. 2 1 2 2 1

d. 2 1 2 2 2

2. Stacks cannot be used to

a. evaluate arithmetic expression in postfix form

b. implement recursion

c. convert given arithmetic expression in infix form to its equivalent postfix form.

d. allocate resources by OS

3. Stack is used for implementing

a. recursion

b. BFS

c. DFS

d both a & c

4. X is called self referential structure. What is X?

a, Linked list

b. Stack

c. Queues

d. Graph

5. The correct pop function is

a. $S \rightarrow \text{arr}[(S \rightarrow \text{top})--]$

b. $S \rightarrow \text{arr}[--(S \rightarrow \text{top})]$

c. $S \rightarrow \text{arr}[(S \rightarrow \text{top})]$

d. None

6. The Ackermann's function

a. has quadratic time complexity

b. has exponential time complexity

c. cant be solved iteratively

d. has logarithmic time complexity

7. Stack A has enteries a, b, c (with a on top). Stack B is empty. An entry popped out of stack B. An entry popped out of Stack B can only be printed. In this arrangement which of the following permutations of a, b, c is not possible?

a. b a c

b. b c a

c. c a b

d. a b c

8. Which one of the following is an application of Queue Data Structure?

- A. When a resource is shared among multiple consumers.
- B. When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes
- C. Load Balancing
- D. All of the above

9. How many stacks are needed to implement a queue. Consider the situation where no other data structure like arrays, linked list is available to you.

- A. 1
- B. 2
- C. 3
- D. 4

10. In a queue, the initial values of front pointer f and pointer r should be and respectively

- A. -1, -1
- B. 0, -1
- C. 0, 0
- D. -1, 0

UNIT-V

Predict the output or error(s) for the following:

1. What will be the position of the file marker?

a: fseek(ptr,0,SEEK_SET);

b: fseek(ptr,0,SEEK_CUR);

2. #include<stdio.h>

main()

{

FILE *ptr;

char i;

ptr=fopen("zzz.c","r");

while((i=fgetch(ptr))!=EOF)

```

        printf("%c",i);

    }

```

3. There were 10 records stored in -somefile.dat but the following program printed 11 names. What went wrong?

```

void main()
{
    struct student
    {
        char name[30], rollno[6];
    }stud;
    FILE *fp = fopen(-somefile.dat,r); while(!feof(fp))
    {
        fread(&stud, sizeof(stud), 1 , fp);
        puts(stud.name);
    }
}

```

4. #define assert(cond)

```

    if(!(cond)) \
    (fprintf(stderr, "assertion failed: %s, file %s, line %d \n",#cond,\
    __FILE__,__LINE__), abort())
void main()
{
    int i = 10;
    if(i==0)
    assert(i < 100);
    else
    printf("This statement becomes else for if in assert macro");
}

```

5. What is the problem with the following code segment?

```

while ((fgets(receiving array,50,file_ptr)) != EOF) ;

```

6. If a file is opened in r+ mode then

- a)reading is possible b)writing is possible c) it will be created if it does not exist
- d)appending is possible

7. If a file is opened in w+ mode then

- a)reading is possible b)writing is possible c) it will be created if it does not exist
- d)appending is possible

8. If a file is opened in r mode then

- a)reading is possible b)writing is possible c) it will be created if it does not exist
- d)appending is possible

9. If a file is opened in a mode then

- a) reading is possible b)writing is possible c) it will be created if it does not exist
- d)appending is possible

10. ftell()
- a) is a function b) gives the current file position indicator c) can be used to find the size of a file d) none of the above
11. The fseek() function
- a) needs 2 arguments b) makes rewind function unnecessary c) takes 3 arguments d) none of the above
12. rewind function takes ____ number of arguments
- a) 1 b) 2 c) 3 d) 0
13. fseek(fp,0,0) is equivalent to
- a) ftell b) rewind c) a & b d) none of the above
14. ferror function is used to find _____ errors
- a) logical b) file opening c) data d) all the above
15. The contents of the file are lost if it is opened in _____ mode
- b) a) a b) w c) w+ d) a+
16. The contents of the file are safe if it is opened in _____ mode
- c) a) a b) r c) a+b d) all the above
17. The valid binary modes of operation are
- d) a) ab b) rb+ c) wb+ d) ab+
18. rewind function is used to
- a) reset the file pointer b) point it to the end of the file c) stay at current position d) none of the above
19. feof function checks for
- a) file opening error b) data error c) end of file d) file closing error
20. The value returned by fopen() function when the file is not opened
- a) 0 b) garbage value c) NULL d) none of the above
21. The fcloseall() function performs
- e) a) closing of all the files b) closes all the files that are opened by that program c) closes only specified files d) none of the above
22. The function that is not used for random access to files is
- f) a) rewind b) ftell c) fseek d) fprintf

20. TUTORIAL PROBLEMS

Unit 1

- 1) Write a C Program to calculate the difference between two time periods.
- 2) Write a C Program to store information using structures for n elements dynamically

Unit 3

- 1) Write a c program to create an array of structures to store student information and sort them using any of the sorting algorithms.

Unit 4

- 1) Write a c program to convert an infix notation to postfix notation

Unit 5

- 1) Write a c program to display the contents of a file using command line arguments.
- 2) Write a c program to create a structure and pass the structure data to files.
- 3) Write a c program to perform arithmetic operations using multi file approach.

21. KNOWN GAPS ,IF ANY

None

22. DISCUSSION TOPICS , IF ANY

1. Use of preprocessor commands in selective compilation
-

2. Practical use of Unions and Bit-fields
3. Advantages and disadvantages of various searching techniques, conditions that drive selection of particular search technique.
4. Advantages and disadvantages of various sorting techniques, conditions that drive selection of particular sort technique.
5. Use of single, double and circular linked list.
6. How to break down a problem into modules and approach the solution

23. REFERENCES, JOURNALS, WEBSITES AND E-LINKS

References

1. The C Programming Language, B.W. Kernighan and Dennis M.Ritchie, PHI.
2. Programming in C. P. Dey and M Ghosh , Oxford University Press.
3. Programming with C, B.Gottfried, 3rd edition, Schaum's outlines, TMH.
4. Problem Solving and Program Design in C, J.R. Hanly and E.B. Koffman, 7th Edition, Pearson education.

Websites

- a. <http://www.programiz.com/c-programming>
- b. <https://www.tutorialspoint.com/cprogramming/>
- c. https://en.wikibooks.org/wiki/C_Programming

24. QUALITY MEASUREMENT SHEETS

- a. Course End Survey
- b. Teaching Evaluation

25. STUDENT LIST

26. GROUP-WISE STUDENTS LIST FOR DISCUSSION TOPICS