MANIPAL INSTITUTE OF TECHNOLOGY

MANIPAL

A Constituent Unit of MAHE, Manipal

School of Computer Engineering

CERTIFICATE

This is to certify that Ms./Mr. ……...……

Reg. No. …..…… Section: …… Roll No .....has satisfactorily completed the lab exercises prescribed for …… Laboratory of …… Year B. Tech. Degree at MIT, Manipal, in the academic year 2025-26.

Date: …….....

Signature of the faculty

CONTENTS

LAB NO.	TITLE	PAGE NO.	MARKS	REMARKS	SIGN
	COURSE OBJECTIVES, OUTCOMES AND EVALUATION PLAN	5	---	---	---
	INSTRUCTIONS TO THE STUDENTS	6	---	---	----
	SAMPLE LAB OBSERVATION NOTE PREPARATION USING C EDITOR	8	---	---	---
1	BASIC SEARCHING AND SORTING METHODS USING ARRAY CONCEPTS	12			
2	POINTERS, RECURSION AND DYNAMIC MEMORY ALLOCATION FUNCTIONS	14			
3	STRINGS AND STRUCTURE -CONCEPTS AND APPLICATIONS	19			
4	SINGLY LINKED LIST- CREATION AND OPERATIONS	26			
5	DOUBLY LINKED LIST- INSERTION, DELETION AND TRAVERSAL	32			
6	POLYNOMIALS USING LINKED LIST AND CIRCULAR LIST	37			
7	STACKS - ARRAY AND LINKED LIST IMPLEMENTATION AND ITS APPLICATIONS	40			
8	QUEUES - ARRAY AND LINKED LIST IMPLEMENTATION AND ITS APPLICATIONS	43			
9	BINARY TREE CREATION AND TRAVERSALS	47			
10	BINARY SEARCH TREES AND AVL TREE OPERATIONS	50			
11	GRAPH REPRESENTATION AND TRAVERSALS	52			
	C QUICK REFERENCE SHEET	I			
	REFERENCES	II			

Course Objectives

1. To develop proficiency in using arrays, pointers, structures, and dynamic memory allocation for solving real-time computational problems.
2. To design and apply linear data structures such as stacks, queues, and linked lists for organizing and processing data efficiently.
3. To implement and analyze non-linear data structures such as trees and graphs for solving hierarchical based real-world problems.

Course Outcomes

At the end of this course, students will be able to

CO Code	Course Outcome Statement
CO1	Apply dynamic memory management, recursion, pointer-based techniques, and efficient searching and sorting algorithms to build optimized and flexible data-driven applications.
CO2	Design and implement applications using basic linear data structures such as arrays and linked lists to organize and process data in real-world problems.
CO3	Implement and apply restricted access linear data structures such as stacks and queues, and non-linear data structures such as trees and graphs, to perform structured data processing, traversal, and application-specific problem solving.

Evaluation plan

Internal (Continuous) Evaluation – Total: 60 Marks

Component	Marks	Remarks
Lab Record / Observation	15	Completion, clarity, and correctness
Lab Test / Internal Assessment	20	Scheduled lab test (mid semester)
Quiz	11	Based on completed experiments
Program Check & Execution	14	Regular implementation during lab sessions
Total	60	

End Semester Practical Evaluation – Total: 40 Marks

Component	Marks	Remarks
Program Write-Up	15	Algorithm, logic, clarity
Program Execution	25	Successful execution, output
Total	40	

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students must carry the Lab Manual and required stationery to every lab session.
2. Be punctual, follow institutional rules, and maintain decorum.
3. Keep mobile phones, pen drives, and other electronic devices in your bag and place the bag in the designated area in the lab.
4. Sign the log register provided.
5. Occupy your allotted seat and respond to the attendance call.

In- Lab Session Instructions

- Follow the instructions for the allotted exercises.
- Show the program and results to the instructor upon completion.
- After receiving approval from the instructor, copy the program and results into the Lab Record.
- Prescribed textbooks and class notes may be used for reference if required.

General Instructions for the exercises in Lab

- Implement the given exercise **individually**; group work is not allowed.
- Programs should meet the following criteria:
    - Be interactive with appropriate prompt, error, and output messages.
    - Perform input validation (data type, range errors, etc.) and provide meaningful error messages with corrective suggestions.
    - Include comments stating the problem and describing each function's purpose, inputs, and outputs.
    - Use proper indentation and formatting.
    - Use meaningful names for variables and functions.
    - Use constants and type definitions where appropriate.
- Plagiarism (copying from others) is strictly prohibited and will result in penalties during evaluation.
- Each week's exercises are divided into:
    - **Solved Exercise**
    - **Lab Exercise** – to be completed during lab hours
    - **Additional Exercise** – to be completed outside lab hours or for skill enhancement

- If a student misses a lab, they must complete the experiment during a repetition class with faculty permission. Credit will be given only for one day's experiment(s).
- Students missing lab due to official reasons (conference, sports, etc.) must take prior permission from the Dean/Associate Dean and complete the lab with another batch. Marks may be awarded for the write-up if submitted in the next lab session.
- Students who are sick must get permission from the Dean/Associate Dean for lab record evaluation. However, attendance will not be granted for that session.
- Students will be evaluated only by the faculty with whom they are registered, even if they complete additional labs with another batch.
- Attendance for the **lab end-semester exam is mandatory**, regardless of internal marks.
- A **minimum of 75% attendance** is required to be eligible for the final exam.
- If a student loses their lab record, they must rewrite all lab details in a new record.
- Lab test and exam questions may include variations or combinations of manual exercises.

THE STUDENTS SHOULD NOT

- Bring mobile phones or other electronic gadgets into the lab.
- Leave the lab without permission.

SAMPLE LAB OBSERVATION NOTE PREPARATION USING C EDITOR

Introduction

This lab course introduces computer programming using the C language. In this lab, students will learn to write, compile, and debug their first C program.

Running a sample C program

Let's understand the steps involved in writing, storing, compiling, and executing a sample program.

Steps to Execute the Program in Code::Blocks

1. **Open Code::Blocks.**
2. **Create a New Project:**
   - Go to File > New > Project...
   - Select **Console Application** and click **Go**.
   - Choose **C** as the language and click **Next**.
   - Name your project and select the folder to save it in.
   - Click **Next**, then **Finish**.
3. **Write Your Program:**
   - Open the main.c file under **Sources** in the project explorer.
   - Replace the default code with your program.
4. **Build and Run:**
   - Click the **Build** icon (hammer) or press **F9** to compile.
   - Click the **Run** icon (green arrow) or press **F10** to execute.
5. **Enter Input:**
   - Follow the on-screen prompts to enter input and view the output.

Sample Program (*InchToCm.c*)**:**

```
// Student's Name
// InchToCm.c
// This program converts inches to centimeters
#include <stdio.h>
int main() {
   float centimeters, inches;
   printf("This program converts inches to centimeters\n");
   printf("Enter a number in inches: ");
   scanf("%f", &inches);
```

```
    centimeters = inches * 2.54;
    printf("%.2f inches is equivalent to %.2f centimeters\n", inches, centimeter
s);
    return 0;
}
```

- Run the program as per the instructions given by the lab teacher.
  - Compile the saved program and run it either by using keyboard short cuts or through the menu.

PROGRAM STRUCTURE AND PARTS

Comments

The first line of the file is:

// InchToCm

This line is a comment. Let's add a comment above the name of the program that contains the student's name.

- Edit the file. **Add the student's name on the top line** so that the first two lines of the file now look like:

// student's name

// InchToCm

Comments informs the people what the program/line is intended to do. The compiler ignores these lines.

Preprocessor Directives

After the initial comments, the student should be able to see the following line:

#include <stdio.h>

This is called a preprocessor directive. It tells the compiler to include the information from the specified header file as part of the program. Preprocessor directives always start with a # sign. In this case, the preprocessor directive includes the information from the stdio.h file, which contains standard input and output functions. Most programs will have at least one include file.

The function main ()

The next non-blank line *void* ***main*** *()*specifies the name of a function as **main**. There must be exactly one function named *main* in each C program and this is where program execution starts. The *void* before *main ()* indicates the function is not returning any value and also to indicate empty argument list to the function. Essentially functions are units of C code that do a particular task. Large programs will have many functions just as large

organizations have many functions. Small programs, like smaller organizations, have fewer functions. The parentheses following the words *void main* contains a list of arguments to the function. In the present case, there is no *argument*. Arguments to functions indicate to the function what objects are provided to the function to perform any task. The curly braces ({) on the next line and on the last line (})of the program determine the beginning and ending of the function.

Variable Declarations

The line after the opening curly brace, *float centimeters*, *inches;* is called a variable declaration. This line tells the compiler to reserve two places in memory with adequate size for a real number (the *float* keyword indicates the variable as a real number). The memory locations will have the names *inches* and *centimeters* associated with them. The programs often have many different variables of many different types.

EXECUTABLE STATEMENTS

Output and Input

The statements following the variable declaration up to the closing curly brace are executable statements. The executable statements are statements that will be executed when the program runs. printf, the output function, tells the compiler to generate instructions that will display information on the screen when the program runs, and scanf, the input function, reads information from the keyboard when the program runs.

Assignment Statements

The statement *centimeters = inches * 2.54;* is an assignment statement. It calculates what is on the right-hand side of the equation (in this case *inches * 2.54*) and stores it in the memory location that has the name specified on the left hand side of the equation (in this case, *centimeters*). So *centimeters = inches * 2.54* takes whatever was read into the memory location *inches*, multiplies it by 2.54, and stores the result in *centimeters*. The next statement outputs the result of the calculation.

Return Statement

The last statement of this program, *return 0;* returns the program control back to the operating system. The value 0 indicates that the program ended normally. The last line of

every main function written should be **return 0**; for **int main()**; this is indicated alternatively to **void main()** which may have a simple **return** statement

Syntax

Syntax is the way that a language must be phrased in order for it to be understandable. The general form of a C program is given below:

```
// program name
// other comments like what program does and student's name
# include <appropriate files>
int main()
{
//Variable declarations
//Executable statements
Return 0;
} // end main
```

LAB 1: BASIC SEARCHING AND SORTING METHODS USING ARRAY CONCEPTS

Objectives:

In this lab, students will be able to understand and implement basic searching and sorting algorithms using arrays in C.

Introduction to Arrays in C

In C programming, an array is a collection of elements of the same data type stored in contiguous memory locations. Arrays allow programmers to store multiple values under a single variable name, using an index to access each element. The first element of an array is stored at index 0, the second at index 1, and so on.

Using arrays simplifies tasks such as storing and processing large sets of data like marks of students, temperatures, or sensor readings. Arrays can be one-dimensional (a simple list), two-dimensional (like a matrix), or even multi-dimensional. Arrays work closely with loops, making it easy to perform operations like reading input, displaying output, finding sums, or sorting data.

In C, arrays must be declared with a fixed size, and the type of data (such as int, float, or char) must be specified. Here's a simple example of an integer array:

int numbers[5] = {10, 20, 30, 40, 50};

This creates an array named numbers that can hold 5 integers. Each element can be accessed or modified using its index, e.g., numbers[0] refers to 10, and numbers[4] refers to 50

Solved Exercise

Find the Largest Element in an Array

```
#include <stdio.h>
int main() {
int arr[100], n, i, max;
printf("Enter number of elements: ");
scanf("%d", &n);
printf("Enter %d integers:\n", n);
for(i = 0; i < n; i++)
{
```

```
scanf("%d", &arr[i]);
}
max = arr[0]; // Assume first is max
for(i = 1; i < n; i++) {
if(arr[i] > max)
max = arr[i];
}
printf("Maximum element: %d\n", max);
return 0;
}
```

Lab exercises:

1. Given an array of n integers and a key element, write a C program to search the element using linear search.
2. Given an array of integers, implement binary search to find the position of a given key.
3. Write a C program to sort a given list of elements using

   i.) Bubble Sort

   ii.) Selection Sort

   iii.) Insertion Sort

Additional Exercise:

1. Write a C program to compare the number of comparisons and swaps made in Bubble Sort, Selection Sort, and Insertion Sort for the same input array.
2. Write a C program to read two matrices A & B, create and display a third matrix C such that C (i, j) = max (A(i, j) , B(i, j))
3. Write a C program to read two matrices, A and B and perform the following operations:
   (i) Multiply the two matrices (A × B)
   (ii) Add the two matrices (A + B)
   (iii) Read a square matrix and check whether it is a magic square or not

   (Hint: A magic square is a square matrix in which the sum of every row, column, and both diagonals is the same.)

Lab 2: POINTERS, RECURSION AND DYNAMIC MEMORY ALLOCATION FUNCTIONS

Objective

By the end of this lab, students will be able to:

- Understand the concept and syntax of pointers in C.
- Use pointers to access and manipulate memory locations.
- Understand and apply the concept of recursion.
- Allocate and deallocate memory dynamically using malloc(), calloc(), realloc(), and free()

Introduction

In C programming, **pointers** are variables that store the **memory addresses** of other variables. They are powerful tools that allow direct memory access and efficient array, string, and structure manipulation. Understanding pointers is crucial for working with dynamic memory and advanced data structures.

Recursion is a programming technique where a function calls itself to solve smaller instances of a problem. It is particularly useful in problems like factorial calculation, Fibonacci sequences, and tree traversals.

Dynamic Memory Allocation (DMA) allows you to allocate memory at runtime using functions like malloc(), calloc(), and realloc() from the <stdlib.h> library. This is essential when the size of data is not known in advance.

Solved Sample Programs

Basic Pointer Example – Access and Modify a Variable

```
#include <stdio.h>
int main() {
int x = 10;
int *ptr;
ptr = &x; // ptr stores the address of x
printf("Value of x = %d\n", x);
printf("Value of x using pointer = %d\n", *ptr);
*ptr = 20; // Changing value using pointer
printf("New value of x = %d\n", x);
return 0; }
```

Recursive Function – Factorial of a Number

```
#include <stdio.h>

int factorial(int n) {
   if(n == 0)
      return 1;
   else
      return n * factorial(n - 1);
}

int main() {
   int num = 5;
   printf("Factorial of %d = %d\n", num, factorial(num));
   return 0;
}
```

Dynamic Memory Allocation – Array Using malloc()

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

int main() {
   int *arr;
   int n, i;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   arr = (int *)malloc(n * sizeof(int)); // Dynamic allocation

   if(arr == NULL) {
      printf("Memory not allocated.\n");
      return 1;
   }

   printf("Enter %d elements:\n", n);
   for(i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
   printf("Entered elements are:\n");
```

```
for(i = 0; i < n; i++) {
      printf("%d ", arr[i]);
  }

  free(arr); // Deallocate memory

  return 0;   }
```

Lab Exercise

1. Write a small function to find the smallest element in an array using pointers.
   In the main function, create a dynamically allocated array, read the values from the keyboard, and pass the array to the function.
   Display the result (smallest element) in the main function.

2. Write a recursive C program to implement Selection Sort using pointers.
   - The recursive function should sort the array using the Selection Sort algorithm.
   - Access and manipulate the array elements using pointers (i.e., avoid using arr[i] style directly).
   - The program should read the array from the user in the main function, call the recursive sorting function, and display the sorted array.

3. Implement a C program to read, display, and find the product of two matrices using functions with appropriate parameters.
   - The matrices must be created using dynamic memory allocation (malloc or calloc).
   - Access matrix elements using array dereferencing (i.e., *(*(mat + i) + j) style).

Additional Exercise

1. Write a C program to simulate the working of the Tower of Hanoi problem using recursion for n disks.
2.

2. Design and implement a C program using suitable data structures and memory management techniques to automate the process of **Class Representative Election**.

The application should follow the steps below:

- Accept input from the user for:
  - Number of students (voters) present in the class
  - Number of candidates contesting the election
  - Names of each candidate

- Allocate memory dynamically using calloc() to store:
  - An integer array for counting votes (index represents candidate number; index 0 is reserved for foul votes)
  - An array of strings (2D character array) for storing candidate names

- The voting process must be interactive:
  - A faculty member initiates the voting process (e.g., by pressing a specific key)
  - Each student enters the **serial number** of the candidate they wish to vote for
  - If an invalid number is entered (i.e., not matching any candidate), it should be counted as a **foul vote**
  - After every vote, a confirmation (e.g., a beep sound or message) is given

- After all students have cast their votes:
  - Display the **total votes** received by each candidate
  - Display the **total number of foul votes**
  - Determine and display the **winner (candidate with maximum votes)**

Constraints:

- Do **not** use struct or structure concepts
- Use **pointers** and **dynamic memory allocation functions** such as calloc()
- Optionally, use a **recursive function** to display results or find the winner.

Lab 3: STRINGS AND STRUCTURE CONCEPTS

Objectives:

In this lab, students should be able to:

• Understand the basics of strings.

• Write and execute programs using structure concepts.

Introduction to Strings and Structures:

Strings:

- A string is an array of characters. Any group of characters (except double quote sign) defined between double quotations is a constant string.
- Character strings are often used to build meaningful and readable programs.

The common operations performed on strings are

- Reading and writing strings
- Combining strings together
- Copying one string to another
- Comparing strings with one another
- Extracting a portion of a string

Declaration:

Syntax: `char string_name[size];`

- The size determines the number of characters in the string_name.

Structures:

In C, a structure is a user-defined data type that allows you to combine different variables of different data types into a single unit. It is used to group related data together so that it can be treated as a single entity. A structure is a composite data type, meaning it can hold multiple values of various types.

The basic syntax of defining a structure in C is as follows:

```
struct structure_name {
    data_type member1;
    data_type member2;
    // ... more members ...
};
```

A brief explanation of the elements:

struct: This keyword is used to define a structure.

structure_name: It is the name given to the structure, which is used to declare variables of that structure type.

data_type: Each member of the structure has its own data type, and it can be any valid C data type like int, float, char, or even another structure.

member1, member2, etc.: These are the names of the individual members (variables) that compose the structure.

Here's an example of a simple structure in C:

```
#include <stdio.h>
// Defining the structure
struct Person {
char name[50];
int age;
float height; };
int main() {
```

```
// Declaring and initializing a variable of the "Person" structure
struct Person person1 = {"John Doe", 30, 1.75};
// Accessing the members of the structure using the dot (.) operator
printf("Name: %s\n", person1.name);
printf("Age: %d\n", person1.age);
printf("Height: %.2f meters\n", person1.height);
return 0;  }
```

In this example, we define a structure named Person that holds information about a person's name, age, and height. We then declare and initialize a variable person1 of type struct Person and access its members using the dot (.) operator.

Structures in C are useful for organizing and managing related data under a single name, especially when dealing with complex data that has multiple properties.

While C is not an object-oriented language, you can use structures along with functions to simulate encapsulation. This means grouping data (inside structures) and related operations (as functions that take structures as parameters) in a modular and meaningful way.

Though C does not support member functions (like C++ or Java), such functions are often referred to informally as "structure-related functions" in C. Using this approach helps maintain clean and reusable code.

To work with functions and structures together, you can define functions that operate on the data members of the structure. Here's an example demonstrating how to create a structure with functions:

```
#include <stdio.h>
// Define the structure
struct Rectangle {
   int length;
   int width;
};
// Function to calculate the area of a rectangle
int calculateArea(struct Rectangle rect) {
   return rect.length * rect.width;
}
// Function to calculate the perimeter of a rectangle
int calculatePerimeter(struct Rectangle rect) {
   return 2 * (rect.length + rect.width);
}
int main() {
   // Declare and initialize a variable of the "Rectangle" structure
   struct Rectangle myRectangle = {5, 10};

   // Call the functions associated with the structure
   int area = calculateArea(myRectangle);
   int perimeter = calculatePerimeter(myRectangle);

   // Display the results
   printf("Rectangle area: %d\n", area);
   printf("Rectangle perimeter: %d\n", perimeter);

   return 0;
}
```

In this example, we define a structure named Rectangle that holds two data members: length and width.

We then define two separate functions:

- calculateArea() – computes the area of the rectangle.
- calculatePerimeter() – computes the perimeter of the rectangle.

Both functions take a Rectangle structure as an argument, allowing us to work with the complete data representation of a rectangle in a clean and organized manner.

While C doesn't support object-oriented features like member functions or encapsulation inherently, this approach mimics **encapsulation** by grouping data and associated operations logically. It makes the code more modular and easier to maintain.

Keep in mind that in C, structures do not inherently support the concept of **private** or **public** members, as seen in languages like C++ or Java. All structure members are **accessible directly** from outside the structure. However, it is considered good practice to **treat structure members as private** by convention and provide **functions (getters and setters)** to access or modify them. This approach, known as **encapsulation**, helps to control how data is accessed and manipulated, promoting better data integrity and modularity.

Solved exercise

Code snippet to read a string

```
#include <stdio.h>
int main()
   { char
   str[100];
   printf("Enter a string: ");
   scanf(" %[^\n]", str);
   printf("String read: %s\n",
   str); return 0;
}
```

Pointer with Strings

Pointers can be used to manipulate strings efficiently. Example:

```
#include <stdio.h>
int main() {
   char *str = "Hello, World!";
   printf("String: %s\n", str);
   return 0;
}
```

Pointer with Structures

Pointers can also be used to access structure members using the **arrow (->)** operator. Example:

```
#include <stdio.h>

struct Student {
   char name[50];
   int roll;
};

int main() {
   struct Student s1 = {"Alice", 101};
   struct Student *ptr = &s1;

   printf("Name: %s\n", ptr->name);
   printf("Roll: %d\n", ptr->roll);
   return 0; }
```

Lab exercises

1. Write a program to perform following string operations **without using string handling functions:**
   a.) length of the string
   b.) string concatenation
   c.) string comparison
   d.) to insert a sub string
   e.) to delete a substring

2. Write a C program to define a student **structure** with the data members to store name, roll no and grade of the student. Also write the required functions to read, display, and sort student information according to the roll number of the student. All the member functions will have array of objects as arguments.
3. Define a structure Student with the following members:
   char name [50] – to store student name as a string
   int roll_no – to store roll number
   float marks – to store marks
   Write a C program that:
   i. Reads the details of 'n' students using a function that uses pointer to structure as an argument.
   ii. Displays the details of all students using a separate function.
   iii. Finds and displays the student with the highest marks using pointer-based access.

Additional Questions:

1. Write a program in C to find the fast transpose of a sparse matrix represented using array of objects.
2. Write a program in C to find the transpose of a sparse matrix represented using array of objects.

Lab 4: SINGLY LINKED LIST- CREATION AND OPERATIONS

Objectives

In this lab, students will be able to:

- Understand and implement the concept of Linked Lists.
- Implement various applications and operations on Linked Lists.
- Manipulate pointers dynamically using C.
- Build menu-driven programs to simulate real-time list operations.

Introduction

The linked list is a linear data structure and an alternative to arrays for storing a collection of elements. It is particularly useful when the number of data elements is unknown in advance or frequently changes.

In a singly linked list, each node contains:

- A data field (to store the actual value).
- A pointer (or link) to the next node.

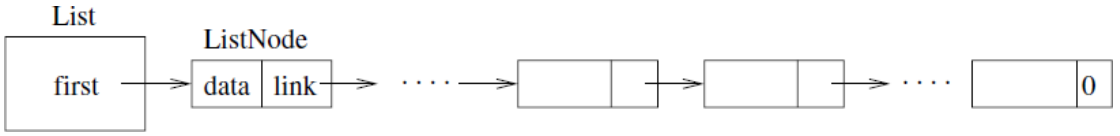


Figure: Structure of a Linked List

The linked list is implemented using pointers. Unlike arrays, the elements (nodes) in a linked list need not be stored in contiguous memory locations.

Structure of a Node

```
struct ListNode {
   int data;
   struct ListNode *link;
};
```

Each node forms part of a chain where the link field of a node points to the next node in the list. The last node's link is set to NULL, indicating the end of the list.

Solved Exercise

Write a **menu-driven program** to perform the following basic operations on a singly linked list:

- Create a list
- Display the list
- Delete an element

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
int data;
struct Node* next;
} Node;
// Function prototypes
Node* createNode(int data);
Node* insert(Node* head, int data);
Node* delete(Node* head, int data);
void display(Node* head);
int main() {
Node* head = NULL;
int choice, data;
while (1) {
printf("\nMenu:\n");
printf("1. Insert an element\n");
printf("2. Delete an element\n");
printf("3. Display the linked list\n");
printf("4. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:
printf("Enter the element to insert: ");
scanf("%d", &data);
```

```
head = insert(head, data);
break;
case 2:
printf("Enter the element to delete: ");
scanf("%d", &data);
head = delete(head, data);
break;
case 3:
display(head);
break;
case 4:
printf("Exiting the program.\n");
return 0;
default:
printf("Invalid choice! Please try again.\n");
}
}
return 0;
}
// Create new node
Node* createNode(int data) {
Node* newNode = (Node*)malloc(sizeof(Node));
if (!newNode) {
printf("Memory allocation failed\n");
exit(1);
}
newNode->data = data;
newNode->next = NULL;
return newNode;
}
// Insert at end
Node* insert(Node* head, int data) {
Node* newNode = createNode(data);
if (head == NULL) {
head = newNode;
}
```

```
else {
Node* current = head;
while (current->next != NULL)
current = current->next;
current->next = newNode;
}
return head;
}
// Delete an element
Node* delete(Node* head, int data) {
if (head == NULL) {
printf("Linked list is empty.\n");
return NULL;
}
Node* current = head;
Node* prev = NULL;
while (current != NULL && current->data != data) {
prev = current;
current = current->next;
}
if (current == NULL) {
printf("Element not found in the linked list.\n");
return head;
}
if (prev == NULL) {
head = current->next;
} else {
prev->next = current->next;
}
free(current);
printf("Element deleted successfully.\n");
return head;
}
// Display list
void display(Node* head) {
if (head == NULL)
```

```
 {
printf("Linked list is empty.\n");
return;
}
Node* current = head;
printf("Linked list elements: ");
while (current != NULL) {
printf("%d -> ", current->data);
current = current->next;
}
printf("NULL\n");
}
```

Lab Exercises

Write a menu-driven C program using structures to implement the following operations on a singly linked list:

- **Insert an element before another specified element in the list**
  (Example: Insert 10 before 25)
- **Insert an element after another specified element in the list**
  (Example: Insert 40 after 25)
- **Delete a specified element from the list**
  (Example: Delete node containing 15)
- **Traverse the list and display all elements**
- **Reverse the linked list**
  (Modify the links such that the list is reversed)
- **Sort the list in ascending order**
  (Using Bubble Sort or any appropriate algorithm on linked list)
- **Delete every alternate node in the list**
  (Starting from the second node)
- **Insert an element into a sorted linked list while maintaining the sorted order**
  (Example: Insert 28 into a list that is already sorted)

Requirements

Use dynamic memory allocation (malloc/free) for node creation and deletion.

Additional Exercises

1. **Recursive functions**:
   i) Create a linked list recursively
   ii) Traverse a linked list recursively
2. **Merge two sorted linked lists** (X and Y in non-decreasing order) into a new sorted list Z. After merge, X and Y should not exist separately. Do **not use new nodes**.
3. **Interleave two lists**:
   Merge list1 = (x1, x2…..xn) and list2 = (y1, y2…..ym) into list3 such that:
   - list3 = (x1, y1, x2, y2….xm, ym, xm+1…xn) if $m \leq n$
   - list3 = (x1, y1, x2, y2…xn, yn, xn+1…ym) if $m > n$

Lab 5: DOUBLY LINKED LIST- INSERTION, DELETION AND TRAVERSAL

Objectives

In this lab, students will be able to:

- Write and execute programs on doubly linked lists.
- Implement applications using both singly and doubly linked lists.
- Gain practical understanding of dynamic memory management using pointers.

Introduction to Doubly Linked List:

A doubly linked list (DLL) is a linear data structure where each node points to both its next and previous node. This enables bi-directional traversal, unlike a singly linked list which allows traversal only in one direction.

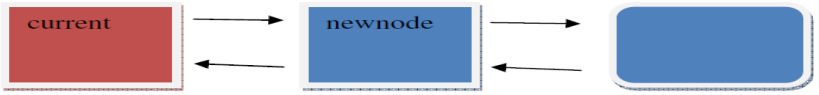
A generic doubly linked list node can be defined as:

```
struct dnode {
int info;
struct dnode *prev;
struct dnode *next;
};
```

This structure gives the flexibility to insert or delete elements from both ends efficiently. Doubly linked lists are widely used in memory management, navigation systems, and undo functionality, and many other applications involving dynamic memory operations.

Inserting to a Doubly Linked Lists

Suppose a new node, *new node* needs to be inserted after the node *current,*



The following code can then be written

```
newnode➔ next = current➔next;    current➔next = newnode;
newnode➔prev = current;  (newnode➔next)➔prev = newnode;
```

Deleting a Node from a Doubly Linked Lists

Suppose a new node, **current** needs to be deleted

The following code can then be written

node* N = current➔prev

N➔ next = current➔next;

(N➔next)➔prev = N;

free(current);

Solved Exercise

Write a program to create and display a doubly linked list using a header node.

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

typedef struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
} Node;

// Function prototypes
Node* createNode(int data);
void insert(Node* header, int data);
void display(Node* header);
```

```
int main() {
    Node* header = createNode(-1); // Header node with dummy data
    int choice, data;
    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert an element\n");
        printf("2. Display the doubly linked list\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the element to insert: ");
                scanf("%d", &data);
                insert(header, data);
                break;
            case 2:
                display(header);
                break;
            case 3:
                printf("Exiting the program.\n");
                return 0;
            default:
                printf("Invalid choice! Please try again.\n");     }    }
    return 0;  }

// Create a new node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}
```

```
// Insert at the end
void insert(Node* header, int data) {
    Node* newNode = createNode(data);
    Node* current = header;

    while (current->next != NULL)
        current = current->next;

    current->next = newNode;
    newNode->prev = current;  }

// Display the list
void display(Node* header) {
    if (header->next == NULL) {
        printf("Doubly linked list is empty.\n");
        return;
    }

    Node* current = header->next;
    printf("Doubly linked list elements: ");
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}
```

Lab Exercises

1. Write a menu-driven C program using structures to implement the following operations on a Doubly Linked List.
    - **Insert an element at the rear end of the list**
      (Append a new node to the end of the list)
    - **Delete an element from the rear end of the list**
      (Remove the last node in the list)
    - **Insert an element at a given position in the list**
      (e.g., Insert at position 3. Positioning starts from 1.)
    - **Delete an element from a given position in the list**

- **Insert an element *after* a node containing a specific value**
  (e.g., Insert 40 after 25)
- **Insert an element *before* a node containing a specific value**
  (e.g., Insert 10 before 25)
- **Traverse the list in forward direction**
  (From head to tail)
- **Traverse the list in reverse direction**
  (From tail to head – i.e., reverse traversal)

Requirements:

- Use dynamic memory allocation (malloc and free).
- Maintain both head and tail pointers for efficient operations.
- Use appropriate functions for modular implementation of each operation.

2.Write a program to concatenate two doubly linked lists X1 and X2. After concatenation, X1 should point to the first node of the resulting list.

Additional Exercises

1. Union and Intersection
   Write a program to implement the union and intersection of two doubly linked lists.
2. Addition of Long Integers
   Write a program to implement the addition of two long positive integers using doubly linked lists.
   (Each digit can be stored in one node; addition is done from the least significant digit using reverse traversal.)

Lab 6: POLYNOMIALS USING LINKED LIST AND CIRCULAR LIST

Objectives

In this lab, students will be able to:

- Write and execute programs using circular linked lists and circular doubly linked lists.
- Understand how to represent and manipulate polynomials using linked lists.
- Perform operations like insertion, deletion, addition, and multiplication of polynomials.

Circular Linked List

In a traditional (linear) linked list, the last node points to NULL, signifying the end of the list. However, in a circular linked list, the last node points back to the first node, making the list circular in structure.

In a circular singly linked list, only the next pointer of the last node points to the first node.

In a circular doubly linked list, both:

The next of the last node points to the first node, and

The prev of the first node points to the last node.

Circular linked lists are useful in applications such as task scheduling, multiplayer games, and queue-based systems where cyclic behavior is desired.

Polynomials and Linked Lists

A polynomial is a mathematical expression consisting of variables and coefficients, involving operations like addition and multiplication. Example:

$$P(x) = 4x^3 + 6x^2 + 7x + 9$$

Array Representation:

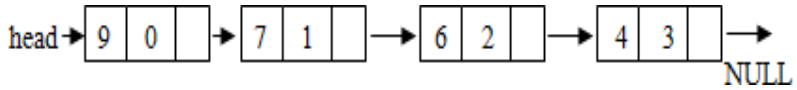
Each index represents the exponent, and the value at that index represents the coefficient.

Linked List Representation:

A node contains:

- coefficient
- exponent
- next pointer

```
struct polynomial {
   int coefficient;
   int exponent;
   struct polynomial *next;
};
```



Lab Exercises

1.Write a C program to implement a Circular Singly Linked List using First and Last pointers.

Implement the following operations:

i. Insertion at the end of the list using First and Last pointers.
ii. Deletion from the beginning or end using First and Last pointers.
iii. Display the list after each operation.

2. Add Two Polynomials Represented as Doubly Linked Lists

i) Represent each polynomial using a doubly linked list, where each node contains the coefficient and exponent of a term.
ii) Write a function to add two polynomials by merging terms with equal exponents. The resulting polynomial should be stored in a new doubly linked list, maintaining the order of terms in descending powers of exponents.
iii) Display all three polynomials: the two input polynomials and their sum.
Ensure dynamic memory allocation is used for all node operations and that both prev and next pointers are maintained correctly.

Additional Exercises

1. Write a menu-driven C program to perform the following operations on a Circular Doubly Linked List:

i) Insert an element into the list
ii) Delete an element from the list
iii) Display the elements of the list

2. Multiply Two Polynomials Using Circular Doubly Linked List (with Header Node)
    i. Represent each polynomial using a circular doubly linked list with a header node.
    ii. Multiply each term of the first polynomial with every term of the second polynomial.
    iii. Merge like terms during or after multiplication.

3. Washing Machine Rental System Simulation

Develop a C application to simulate a washing machine rental queue using a circular doubly linked list.

Requirements:

i. Each user books the washing machine for a specific time.
ii. After time expires, the machine automatically passes to the next user in the queue.
iii. Use the circular nature of the list to model the continuous cycle of usage.
iv. Optional: simulate time using delays or counters for testing logic.

Lab 7: STACKS – ARRAY AND LINKED LIST IMPLEMENTATION AND ITS APPLICATIONS

Objectives:

In this lab, students will be able to:

- Understand the concept and applications of stacks.
- Implement stack operations using **arrays** and **linked lists**.
- Write and execute stack-based application programs.

Introduction:

Stacks are linear data structures that follow the **LIFO (Last In, First Out)** principle. This means the element inserted last is the one removed first.

Stack supports two primary operations:

- push: Inserts an element on the top of the stack.
- pop: Removes the top element from the stack.

Other useful operations:

- peek or top: Returns the top element without removing it.
- isEmpty: Checks if the stack is empty.
- isFull: Checks if the stack is full (applicable for array-based implementation).

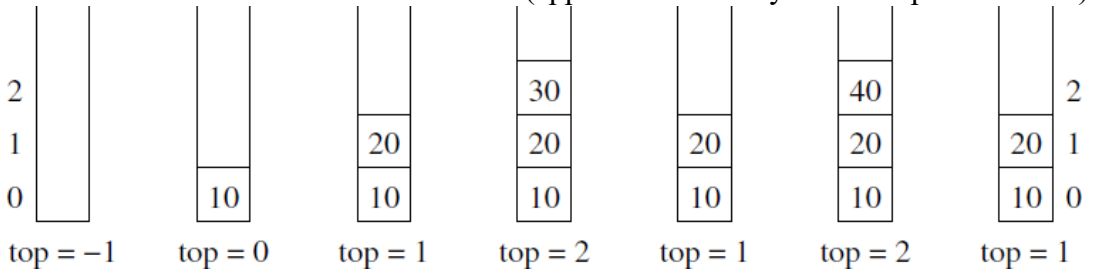


Figure: Stack Operations

Stacks can be implemented using:

- Arrays (fixed size)
- Linked lists (dynamic size)

STACK IMPLEMENTATION USING ARRAYS

```
#define MAX_SIZE 100
struct Stack {
   int items[MAX_SIZE];
   int top;
};
```

STACK IMPLEMENTATION USING LINKED LISTS

```
struct Node {
   int data;
   struct Node* next;
};
```

Lab Exercise

1. Array-Based Stack
   i. Check whether a given string is a palindrome using stack.
      ➤ Use character stack to compare original and reversed string.
   ii. Check for matching parentheses in each expression.
      ➤ Push opening brackets, pop for matching closing brackets.
2. Linked List-Based Stack:
   i. Write a program to input an infix expression and convert into its equivalent post fix form and display. Operands can be single characters.
   ii. Evaluate a postfix expression using stack.
      ➤ Push operands, pop two for operator, and push result back.

Additional Exercise:

1. Implement multiple stacks in a single array.
   ➤ Use fixed partitioning or k-stacks algorithm (optional advanced).
2. Write a C program to convert an infix expression to its equivalent prefix form.
   - Use a stack-based algorithm, and handle operator precedence and associativity appropriately.
   - Operands can be single characters.
   - Display the resulting prefix expression
3. Write a program that converts a post fix expression to a fully parenthesized infix expression.
4. Write a program to evaluate prefix expressions. The input to the program is a prefix expression.

Lab: 8 QUEUES - ARRAY AND LINKED LIST IMPLEMENTATION AND ITS APPLICATIONS

Objectives:

By the end of this lab, students will be able to:

- Understand the concept and operations of Queues.
- Implement queues using arrays and linked lists.
- Apply queue operations to solve real-world computational problems.
- Write and execute programs involving queue-based logic.

Introduction:

A queue is a linear data structure that follows the FIFO (First In, First Out) principle. The element inserted first is the one to be removed first, similar to a line of people at a ticket counter.

Types of Queues:

1. Simple Queue – Insertion at rear, deletion from front.
2. Circular Queue – Overcomes space limitation in simple queues.
3. Double-Ended Queue (Deque) – Insertion and deletion at both ends.
4. Priority Queue – Elements are served based on priority.

Queue Operations:

- enqueue(x): Insert an element at the rear.
- dequeue(): Remove and return the front element.
- peek(): View the front element without removing.
- isEmpty(): Check if the queue is empty.
- isFull(): Check if the queue is full (array implementation only).

Implementations:

- Array-Based Queue: Fixed-size implementation.
- Linked List-Based Queue: Dynamic memory allocation using malloc/free.

Solved Exercise

```
#include <stdio.h>
#define MAX 100
struct Queue {
   int items[MAX];
   int front, rear;
};
// Initialize the queue
void initialize(struct Queue* q) {
   q->front = -1;
   q->rear = -1;
}
// Check if queue is empty
int isEmpty(struct Queue* q) {
   return q->front == -1;
}
// Check if queue is full
int isFull(struct Queue* q) {
   return q->rear == MAX - 1;
}
// Enqueue operation
void enqueue(struct Queue* q, int value) {
   if (isFull(q)) {
      printf("Queue is full.\n");
      return;
   }
   if (isEmpty(q)) q->front = 0;
   q->items[++q->rear] = value;
   printf("Enqueued: %d\n", value);
}
// Dequeue operation
int dequeue(struct Queue* q) {
   if (isEmpty(q)) {
      printf("Queue is empty.\n");
      return -1;
   }
```

```
    int value = q->items[q->front];
    if (q->front == q->rear)
        q->front = q->rear = -1;
    else
        q->front++;
    return value;
}
// Peek operation
int peek(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return -1;
    }
    return q->items[q->front];
}
// Main menu-driven function
int main() {
    struct Queue q;
    initialize(&q);
    int choice, value;
    while (1) {
        printf("\n--- Queue Menu ---\n");
        printf("1. Enqueue\n2. Dequeue\n3. Peek\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
        case 1:
            printf("Enter value to enqueue: ");
            scanf("%d", &value);
            enqueue(&q, value);
            break;
        case 2:
            value = dequeue(&q);
            if (value != -1)
                printf("Dequeued: %d\n", value);
            break;
```

```
        case 3:
            value = peek(&q);
            if (value != -1)
                printf("Front of queue: %d\n", value);
            break;
        case 4:
            return 0;
        default:
            printf("Invalid choice.\n");  } } }
```

Lab Exercises

Array-Based Queue Implementation:

1. Write a C program to simulate a printer queue where tasks arrive randomly and are processed in order.
    i. Tasks (with a document ID and name) arrive at random (simulate using random function or user input).
    ii. Enqueue each print job.
    iii. Dequeue in FIFO order to simulate printing
2. Write a C program to implement a circular queue using arrays.

Linked List-Based Queue Implementation:

3. Implement a queue using a singly linked list with enqueue and dequeue operations.
4. Create a queue of structures (e.g., queue of patients with name, age, and priority).

Additional Exercises

1. Write a C program to implement a queue using two stacks.
   Simulate the queue operations (enqueue, dequeue, and optionally peek) using the stack data structure. Use either:
    - Two stacks and recursion
    - Or two stacks and iterative logic

   Demonstrate FIFO behavior using LIFO structures.
2. Design a data representation to sequentially map n queues into a single array A(1:m). Represent each queue as a **circular queue** within A. Write the following algorithms:
    - ADDQ(i, x) – Add element x to the i-th queue
    - DELETEQ(i) – Delete an element from the i-th queue
    - QUEUE_FULL() – Check if the array is full and cannot accommodate further insertions

   Illustrate the working of your design using a **menu-driven C program** with appropriate input/output operations for n queues.

Lab:9 BINARY TREE CREATION AND TRAVERSALS

Objectives

By the end of this lab, students should be able to:

- Understand the concept and structure of binary trees.
- Implement binary trees using linked nodes.
- Perform tree traversals: **Inorder**, **Preorder**, and **Postorder**.
- Solve problems related to tree structures in C.

Introduction

A **Binary Tree** is a hierarchical data structure in which each node has at most **two children**, referred to as the **left child** and **right child**.

Key Terms:

- **Root** – The top node in a tree.
- **Leaf** – A node with no children.
- **Subtree** – A tree formed by a node and its descendants.

Tree Traversals:

Traversing a tree means visiting all its nodes in a specific order.

1. **Inorder (Left → Root → Right)**
2. **Preorder (Root → Left → Right)**
3. **Postorder (Left → Right → Root)**

These recursive processes are essential for many tree-based algorithms and applications.

Solved Exercise

Inorder Traversal (Left → Root → Right)

```
void inorder(node *root)
{
   if(root == NULL) return;
   inorder(root->llink);
   cout << root->info << " ";
   inorder(root->rlink);
}
```

Preorder Traversal (Root → Left → Right)

```
void preorder(node *root)
{
   if(root == NULL) return;
   cout << root->info << " ";
   preorder(root->llink);
   preorder(root->rlink);
}
```

Postorder Traversal (Left → Right → Root)

```
void postorder(node *root)
{
   if(root == NULL) return;
   postorder(root->llink);
   postorder(root->rlink);
   cout << root->info << " ";
}
```

Lab Exercises

1. Write user-defined functions to perform the following operations on binary trees:
   i) Inorder traversal (Iterative)
   ii) Postorder traversal (Iterative)
   iii) Preorder traversal (Iterative)
   iv) Print the parent of a given element
   v) Print the depth (or height) of the tree
   vi) Print the ancestors of a given element
   vii) Count the number of leaf nodes in a binary tree

2. Write a recursive function to:
   i) Create a binary tree
   ii) Print the binary tree (in traversal order, typically level-order)

Additional Exercises

1. Write a program to check for the equality of two binary trees (same structure and data).
2. Write a program to check if one binary tree is the mirror image of another.
3. Write a program to copy one binary tree to another.

Lab 10: BINARY SEARCH TREES AND AVL TREE OPERATIONS

Objectives

By the end of this lab, students should be able to:

- Understand the concept and structure of Binary Search Trees (BSTs).
- Perform fundamental operations: Insertion, Search, Deletion in BST.
- Understand the need for self-balancing trees like AVL Trees.
- Implement AVL Tree operations including rotations for balancing.

Introduction

Binary Search Tree (BST)

A **BST** is a binary tree where each node satisfies the condition:

- Left child < Node < Right child

This property allows efficient **search**, **insert**, and **delete** operations with average time complexity of **O(log n)** for balanced trees and **O(n)** in the worst case (e.g., skewed trees).

AVL Tree (Adelson-Velsky and Landis Tree)

An **AVL Tree** is a **self-balancing binary search tree** where the difference between the heights of left and right subtrees (called the **balance factor**) is at most **1** for every node.
To maintain balance, **rotations** are used:

- Right Rotation (RR)
- Left Rotation (LL)
- Left-Right Rotation (LR)
- Right-Left Rotation (RL)

Lab Exercise

1. Write a program to create a BST and perform inorder, preorder, and postorder traversals.
2. Write a function to search an element in a BST.
3. Write a function to delete a node from a BST.
4. Write a function to find the minimum and maximum elements in a BST.
5. Write a program to create an AVL Tree and perform insertion.

Additional Exercise

1. Write a program to implement level order traversal on binary search tree.
2. Write a program to convert a BST into a sorted doubly linked list.
3. Write a program to create an AVL Tree and perform insertion.

Lab 11: GRAPH REPRESENTATION AND TRAVERSALS

Objectives

By the end of this lab, students will be able to:

- Understand the fundamentals of graph data structures.
- Represent graphs using adjacency matrix and adjacency list.
- Perform graph traversals: Depth First Search (DFS) and Breadth First Search (BFS).

Introduction

A graph is a collection of nodes (vertices) and edges (arcs) where each edge connects a pair of vertices.

Types of Graphs:

- Directed Graph (Digraph) – edges have direction.
- Undirected Graph – edges are bidirectional.
- Weighted Graph – edges have weights (costs).
-

Graph Representations:

1. Adjacency Matrix
    - A 2D array where matrix[i][j] = 1 (or weight) if an edge exists between vertex i and j.
    - Good for dense graphs.
2. Adjacency List
    - An array of linked lists, where each index represents a vertex and contains a list of all connected vertices.
    - Efficient for sparse graphs.

Lab Exercise:

1. Write a C program to represent directed and undirected graphs using adjacency matrix.
2. Write a C program to represent directed and undirected graphs using adjacency list.

3. Write a C program to implement Breadth First Search (BFS)
4. Write a C program to implement Depth First Search (DFS)

Additional Exercises

1. Detect a cycle in an undirected graph using DFS.
2. Detect a cycle in a directed graph using recursion stack.
3. Check if a graph is bipartite using BFS.

C QUICK REFERENCE

PREPROCESSOR

```
                        // Comment to end of line
                        /* Multi-line comment */
#include <stdio.h>      // Insert standard header file
#include "myfile.h"         // Insert file in current directory
#define X some text             // Replace X with some text
#define F(a,b) a+b      // Replace F(1,2) with 1+2
#define X \
        some text       // Line continuation
#undef X                // Remove definition
#if defined(X)          // Condional compilation (#ifdef X)
#else                   // Optional (#ifndef X or #if !defined(X))
#endif                  // Required after #if, #ifdef
```

LITERALS

```
255, 0377, 0xff                 // Integers (decimal, octal, hex)
2147463647L, 0x7fffffffl        // Long (32-bit) integers
123.0, 1.23e2           // double (real) numbers
‘a’, ‘\141’, ‘\x61’             // Character (literal, octal, hex)
‘\n’, ‘\\’, ‘\’’, ‘\”’,         // Newline, backslash, single quote, double quote
“string\n”              // Array of characters ending with newline and \0
“hello” “world”                 // Concatenated strings
true, false             // bool constants 1 and 0
```

DECLARATIONS

```
int x;                  // Declare x to be an integer (value undefined)
int x=255;              // Declare and initialize x to 255
short s; long 1;        // Usually 16 or 32 bit integer (int may be either)
char c= ‘a’;            // Usually 8 bit character
unsigned char u=255; signed char m=-1; // char might be either
unsigned long x=0xffffffffL; // short, int, long are signed
float f; double d;              // Single or double precision real (never unsigned)
bool b=true;            // true or false, may also use int (1 or 0)
```

```
int a, b, c;              // Multiple declarations
int a[10];                // Array of 10 ints (a[0] through a[9])
int a[]={0,1,2};          // Initialized array (or a[3]={0,1,2}; )
int a[2][3]={{1,2,3},{4,5,6}; // Array of array of ints
char s[]= "hello";        // String (6 elements including '\0')
int* p;                   // p is a pointer to (address of) int
char* s= "hello";         // s points to unnamed array containing "hello"
void* p=NULL;             // Address of untyped memory (NULL is 0)
int& r=x;                 // r is a reference to (alias of) int x
enum weekend {SAT, SUN};// weekend is a type with values SAT and SUN
enum weekend day;         // day is a variable of type weekend
enum weekend {SAT=0,SUN=1}; // Explicit representation as int
enum {SAT,SUN} day;       // Anonymous enum
typedef String char*;  // String s; means char* s;
constint c=3;             // Constants must be initialized, cannot assign
constint* p=a;            // Contents of p (elements of a) are constant
int* const p=a;           // p (but not contents) are constant
constint* const p=a;      // Both p and its contents are constant
constint&cr=x;            // cr cannot be assigned to change x
```

STORAGE CLASSES

```
int x;                    // Auto (memory exists only while in scope)
staticint x;              // Global lifetime even if local scope
externint x;              // Information only, declared elsewhere
```

STATEMENTS

```
x=y;                      // Every expression is a statement
int x;                    // Declarations are statements
;                         // Empty statement
{                         // A block is a single statement
int x;                    // Scope of x is from declaration to end of
block
a;                        // In C, declarations must precede statements
}
if (x) a;                 // If x is true (not 0), evaluate a
```

```
else if (y) b;          // If not x and y (optional, may be repeated)
else c;                 // If not x and not y (optional)
while (x) a;            // Repeat 0 or more times while x is true
for (x; y; z) a;        // Equivalent to:  x; while(y) {a; z;}
do a; while (x);                // Equivalent to:  a; while(x) a;
switch (x) {            // x must be int
        case X1: a;     // If x == X1 (must be a const), jump here
        case X2: b;     // Else if x == X2, jump here
        default: c;     // Else jump here (optional)
}
break;                  // Jump out of while, do, for loop, or switch
continue;               // Jump to bottom of while, do, or for loop
return x;                       // Return x from function to caller
try { a; }
catch (T t) { b; }              // If a throws T, then jump here
catch (...)  { c; }             // If a throws something else, jump here
```

FUNCTIONS

```
int f(int x, int);              // f is a function taking 2 ints and returning int
void f();                       // f is a procedure taking no arguments
void f(int a=0);                // f() is equivalent to f(0)
f();                    // Default return type is int
inline f();             // Optimize for speed
f( ) { statements; }    // Function definition (must be global)
```

Function parameters and return values may be of any type. A function must either be declared or defined before it is used. It may be declared first and defined later. Every program consists of a set of global variable declarations and a set of function definitions (possibly in separate files), one of which must be:

```
int main() { statements... }      or
int main(intargc, char* argv[]) { statements... }
```

argv is an array of argc strings from the command line. By convention, main returns status 0 if successful, 1 or higher for errors.

EXPRESSIONS

Operators are grouped by precedence, highest first. Unary operators and assignment evaluate right to left. All others are left to right. Precedence does not affect order of evaluation which is undefined. There are no runtime checks for arrays out of bounds, invalid pointers etc.

```
T::X            // Name X defined in class T
N::X            // Name X defined in namespace N
::X             // Global name X
t.x             // Member x of struct or class t
p → x           // Member x of struct or class pointed to by p
a[i]            // i'th element of array a
f(x, y)         // Call to function f with arguments x and y
T(x, y)             // Object of class T initialized with x and y
x++             // Add 1 to x, evaluates to original x (postfix)
x--             // Subtract 1 from x, evaluates to original x
sizeof x            // Number of bytes used to represent object x
sizeof(T)       // Number of bytes to represent type T
++x             // Add 1 to x, evaluates to new value (prefix)
--x             // Subtract 1 from x, evaluates to new value
~x              // Bitwise complement of x
!x              // true if x is 0, else false (1 or 0 in C)
-x              // Unary minus
+x              // Unary plus (default)
&x              // Address of x
*p              // Contents of address p (*&x equals x)
x * y           // Multiply
x / y           // Divide (integers round toward 0)
x % y           // Modulo (result has sign of x)
x + y           // Add, or &x[y]
x – y           // Subtract, or number of elements from *x to *y
x << y          // x shifted y bits to left (x * pow(2, y))
x >> y          // x shifted y bits to right (x / pow(2, y))
x < y           // Less than
x <= y          // Less than or equal to
x > y           // Greater than
```

```
x >= y          // Greater than or equal to
x == y          // Equals
x != y          // Not equals
x& y            // Bitwise and (3 & 6 is 2)
x ^ y           // Bitwise exclusive or (3 ^ 6 is 5)
x | y           // Bitwise or (3 | 6 is 7)
x&& y           // x and then y (evaluates y only if x (not 0))
x || r              // x or else y (evaluates y only if x is false(0))
x = y               // Assign y to x, returns new value of x
x += y              // x = x + y, also -= *= /= <<= >>= &= |= ^=
x ?y : z                    // y if x is true (nonzero), else z
throw x                     // Throw exception, aborts if not caught
x, y                // evaluates x and y, returns y (seldom used)
```

STRING (Variable sized character array)

```
string s1, s2= "hello";         //Create strings
sl.size(), s2.size();           // Number of characters: 0, 5
sl += s2 +  '  ' + "world";     // Concatenation
sl == "hello world";            // Comparison, also <, >, !=, etc.
s1[0];                          // 'h'
sl.substr(m, n);                    // Substring of size n starting at sl[m]
sl.c_str();                     // Convert to const char*
getline(cin, s);                // Read line ending in '\n'
```

STRING Handling functions

```
strcat(s1,s2)               // Concatenates s2 to s1
strcmp(s1,s2)               // Comparison of two strings and returns 0 if
equalstrstr(mainstr, substr)    // Search for the Substring in mainstr
strlen(s1)                  // string length of s1
gets(s1)                    // Read line ending in '\n'
```

Other functions

```
asin(x); acos(x); atan(x);      // Inverses
atan2(y, x);                    //
atan(y/x) sinh(x); cosh(x); tanh(x);
Hyperbolic
exp(x); log(x); log10(x);       // e to the x, log base e, log base
10pow(x, y); sqrt(x);           // x to the y, square root
ceil(x); floor(x);              // Round up or down (as a
double)fabs(x); fmod(x, y);     // Absolute value, x mod y
```

REFERENCES

1. Behrouz A. Forouzan, Richard F. Gilberg, A Structured Programming Approach Using C,3rd Edition, Cengage Learning India Pvt. Ltd, India, 2007.
2. Ellis Horowitz, Sartaj Sahani, Susan Anderson and Freed, Fundamentals of Data Structures in C, 2nd Edition, Silicon Press, 2007.
3. Richard F. Gilberg, Behrouz A. Forouzan, Data structures, A Pseudocode Approach with C, 2nd Edition, Cengage Learning India Pvt. Ltd, India , 2009.
4. Tenenbaum Aaron M., Langsam Yedidyah, Augenstein Moshe J., Data structures using C, Pearson Prentice Hall of India Ltd., 2007.
5. Debasis Samanta, Classic Data Structures, 2nd Edition, PHI Learning Pvt. Ltd., India, 2010. .