

PROGRAMMING FOR PROBLEM SOLVING LABORATORY

LAB MANUAL

Subject Name	Programming for Problem Solving Laboratory
Subject Code	ICT-101P
Academic Session	2026–27
Total Practicals	16 (4 from each Unit)

How to Use This Lab Manual

This manual contains 16 practical exercises aligned with the ICT-101P Programming for Problem Solving Laboratory syllabus. Each practical includes an objective, concepts covered, algorithm/pseudocode, C source code, sample input, sample output and result. Students should understand the logic before typing the program and should test the program with additional inputs.

General Laboratory Guidelines

- Write the algorithm or pseudocode before writing the program.
- Use meaningful variable names and proper indentation.
- Compile the program and correct syntax errors before execution.
- Test each program with normal as well as boundary or invalid inputs where applicable.
- Do not depend only on the sample input; try at least two additional test cases.
- Record the output in the practical file and explain the logic during the viva.

List of Practicals

1. Student Marks Analysis
2. Online Shopping Bill
3. Leap Year Checker
4. Number Analysis

5. Function-Based Calculator
6. Array Statistics
7. Searching and Sorting
8. String Analysis
9. Student Record Using Structure
10. Pointer and Array Operations
11. Dynamic Memory for Numbers
12. Matrix Operations and File Handling
13. Singly Linked List
14. Stack and Expression Checking
15. Circular Queue Implementation
16. Binary Tree Traversal

Practical 1: Student Marks Analysis [L3 – Apply]

Objective: To calculate total, average and percentage of a student using basic C data types, input/output and arithmetic operators.

Concepts Covered: Data types, variables, scanf(), printf(), arithmetic operators and expressions.

Algorithm / Pseudocode

```
START
1. Read marks m1, m2 and m3.
2. total = m1 + m2 + m3.
3. average = total / 3.
4. percentage = (total / 300) * 100.
5. Display total, average and percentage.
END
```

C Source Code

```
#include <stdio.h>

int main(void) {
    float m1, m2, m3, total, average, percentage;

    printf("Enter marks in three subjects: ");
    scanf("%f %f %f", &m1, &m2, &m3);

    total = m1 + m2 + m3;
    average = total / 3.0f;
    percentage = (total / 300.0f) * 100.0f;

    printf("Total = %.2f\n", total);
    printf("Average = %.2f\n", average);
    printf("Percentage = %.2f%%\n", percentage);

    return 0;
}
```

Sample Input

Enter marks in three subjects: 78 85 92

Sample Output

```
Total = 255.00
Average = 85.00
Percentage = 85.00%
```

Result: The program was executed successfully and the expected output was obtained.

Practical 2: Online Shopping Bill [L3 – Apply]

Objective: To apply arithmetic, relational, logical, bitwise and conditional operators in a simple online shopping case study.

Concepts Covered: Arithmetic, relational, logical, bitwise and conditional operators.

Algorithm / Pseudocode

```
START
1. Read the prices of two products.
2. Read the quantity of each product.
3. Calculate each product amount and the subtotal.
4. If subtotal is at least 2000, set discount to 10%; otherwise set it to 0.
5. Calculate final bill = subtotal - discount.
6. Compare the two prices using a relational operator.
7. Check discount eligibility using a logical condition.
8. Perform bitwise AND, OR and XOR on integer parts of the two prices.
9. Use the conditional operator to display the costlier product.
10. Display all results.
END
```

C Source Code

```
#include <stdio.h>

int main(void) {
    float p1, p2, q1, q2, a1, a2, subtotal, discount, bill;
    int x, y;

    printf("Enter price and quantity of Product 1: ");
    scanf("%f %f", &p1, &q1);
    printf("Enter price and quantity of Product 2: ");
    scanf("%f %f", &p2, &q2);

    a1 = p1 * q1;
    a2 = p2 * q2;
    subtotal = a1 + a2;

    discount = (subtotal >= 2000.0f && q1 > 0 && q2 > 0)
        ? subtotal * 0.10f : 0.0f;
    bill = subtotal - discount;

    printf("\nProduct 1 Amount = %.2f\n", a1);
    printf("Product 2 Amount = %.2f\n", a2);
    printf("Subtotal = %.2f\n", subtotal);
    printf("Discount = %.2f\n", discount);
    printf("Final Bill = %.2f\n", bill);
    printf("Product 1 price > Product 2 price: %d\n", p1 > p2);
    printf("Eligible for discount: %d\n", subtotal >= 2000.0f && q1 > 0 && q2 > 0);

    x = (int)p1; y = (int)p2;
    printf("Bitwise AND = %d\n", x & y);
    printf("Bitwise OR = %d\n", x | y);
    printf("Bitwise XOR = %d\n", x ^ y);

    printf("Costlier product = %s\n", (p1 > p2) ? "Product 1" :
        (p2 > p1) ? "Product 2" : "Both have the same price");
}
```

```
        return 0;
    }
```

Sample Input

```
Enter price and quantity of Product 1: 1200 1
Enter price and quantity of Product 2: 1000 1
```

Sample Output

```
Product 1 Amount = 1200.00
Product 2 Amount = 1000.00
Subtotal = 2200.00
Discount = 220.00
Final Bill = 1980.00
Product 1 price > Product 2 price: 1
Eligible for discount: 1
Bitwise AND = 992
Bitwise OR  = 1208
Bitwise XOR = 216
Costlier product = Product 1
```

Result: The program was executed successfully and the expected output was obtained.

Practical 3: Leap Year Checker [L4 – Analyze]

Objective: To use conditional statements to determine whether a given year is a leap year.

Concepts Covered: if, else-if, modulus operator and logical decision making.

Algorithm / Pseudocode

```
START
1. Read year.
2. If year is divisible by 400, it is a leap year.
3. Otherwise, if year is divisible by 100, it is not a leap year.
4. Otherwise, if year is divisible by 4, it is a leap year.
5. Otherwise, it is not a leap year.
6. Display the result.
END
```

C Source Code

```
#include <stdio.h>

int main(void) {
    int year;

    printf("Enter year: ");
    scanf("%d", &year);

    if (year % 400 == 0)
        printf("%d is a leap year.\n", year);
    else if (year % 100 == 0)
        printf("%d is not a leap year.\n", year);
    else if (year % 4 == 0)
        printf("%d is a leap year.\n", year);
    else
        printf("%d is not a leap year.\n", year);

    return 0;
}
```

Sample Input

Enter year: 2024

Sample Output

2024 is a leap year.

Result: The program was executed successfully and the expected output was obtained.

Practical 4: Number Analysis [L4 – Analyze]

Objective: To use loops to find the number of digits, sum of digits, reverse and palindrome status of a number.

Concepts Covered: while loop, modulus, integer division, tracing and palindrome logic.

Algorithm / Pseudocode

```
START
1. Read a positive integer n.
2. Store n in original.
3. Set count = 0, sum = 0, reverse = 0.
4. Repeat while n > 0:
    a. digit = n % 10.
    b. count = count + 1.
    c. sum = sum + digit.
    d. reverse = reverse * 10 + digit.
    e. n = n / 10.
5. Compare reverse with original.
6. Display count, sum, reverse and palindrome result.
END
```

C Source Code

```
#include <stdio.h>

int main(void) {
    int n, original, digit, count = 0, sum = 0, reverse = 0;

    printf("Enter a positive integer: ");
    scanf("%d", &n);

    original = n;

    while (n > 0) {
        digit = n % 10;
        count++;
        sum += digit;
        reverse = reverse * 10 + digit;
        n /= 10;
    }

    printf("Number of digits = %d\n", count);
    printf("Sum of digits = %d\n", sum);
    printf("Reverse = %d\n", reverse);

    if (reverse == original)
        printf("The number is a palindrome.\n");
    else
        printf("The number is not a palindrome.\n");

    return 0;
}
```

Sample Input

Enter a positive integer: 1221

Sample Output

```
Number of digits = 4  
Sum of digits = 6  
Reverse = 1221  
The number is a palindrome.
```

Result: The program was executed successfully and the expected output was obtained.

Practical 5: Function-Based Calculator [L3 – Apply]

Objective: To create a menu-driven calculator using user-defined functions.

Concepts Covered: Functions, parameters, return values, switch and validation.

Algorithm / Pseudocode

```
START
1. Display a menu for addition, subtraction, multiplication and division.
2. Read the user's choice and two numbers.
3. Call the corresponding function.
4. Display the returned result.
5. For division, check that the second number is not zero.
6. Stop.
END
```

C Source Code

```
#include <stdio.h>

float add(float a, float b) { return a + b; }
float subtract(float a, float b) { return a - b; }
float multiply(float a, float b) { return a * b; }
float divide(float a, float b) { return a / b; }

int main(void) {
    int choice;
    float a, b;

    printf("1. Add\n2. Subtract\n3. Multiply\n4. Divide\n");
    printf("Enter choice: ");
    scanf("%d", &choice);
    printf("Enter two numbers: ");
    scanf("%f %f", &a, &b);

    switch (choice) {
        case 1: printf("Result = %.2f\n", add(a,b)); break;
        case 2: printf("Result = %.2f\n", subtract(a,b)); break;
        case 3: printf("Result = %.2f\n", multiply(a,b)); break;
        case 4:
            if (b != 0)
                printf("Result = %.2f\n", divide(a,b));
            else
                printf("Division by zero is not allowed.\n");
            break;
        default: printf("Invalid choice.\n");
    }
    return 0;
}
```

Sample Input

```
Enter choice: 3
Enter two numbers: 12 5
```

Sample Output

```
Result = 60.00
```

Result: The program was executed successfully and the expected output was obtained.

Practical 6: Array Statistics [L3 – Apply]

Objective: To store numbers in an array and calculate the largest, smallest, sum and average using a function.

Concepts Covered: Arrays, functions, traversal and pointer parameters.

Algorithm / Pseudocode

```
START
1. Read n.
2. Read n values into an array.
3. Pass the array and n to a function.
4. Set largest and smallest to the first element.
5. Traverse the array and update largest, smallest and sum.
6. Calculate average = sum / n.
7. Display all results.
END
```

C Source Code

```
#include <stdio.h>

void statistics(int a[], int n, int *largest, int *smallest, int *sum) {
    int i;
    *largest = *smallest = a[0];
    *sum = 0;

    for (i = 0; i < n; i++) {
        *sum += a[i];
        if (a[i] > *largest) *largest = a[i];
        if (a[i] < *smallest) *smallest = a[i];
    }
}

int main(void) {
    int a[100], n, i, largest, smallest, sum;
    float average;

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

    printf("Enter %d numbers: ", n);
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);

    statistics(a, n, &largest, &smallest, &sum);
    average = (float)sum / n;

    printf("Largest = %d\nSmallest = %d\n", largest, smallest);
    printf("Sum = %d\nAverage = %.2f\n", sum, average);

    return 0;
}
```

Sample Input

```
Enter number of elements: 5
Enter 5 numbers: 12 7 25 10 16
```

Sample Output

```
Largest = 25  
Smallest = 7  
Sum = 70  
Average = 14.00
```

Result: The program was executed successfully and the expected output was obtained.

Practical 7: Searching and Sorting [L4 – Analyze]

Objective: To sort an array and search for a required element using suitable algorithms.

Concepts Covered: Arrays, bubble sort, binary search and algorithmic analysis.

Algorithm / Pseudocode

```
START
1. Read n and the array.
2. Sort the array in ascending order using bubble sort.
3. Read the value to search.
4. Set low = 0 and high = n - 1.
5. Find mid = (low + high) / 2.
6. If a[mid] equals the key, report its position.
7. If key is greater, search the right half; otherwise search the left half.
8. Repeat until found or low > high.
9. Display the sorted array and search result.
END
```

C Source Code

```
#include <stdio.h>

int main(void) {
    int a[100], n, i, j, temp, key;
    int low, high, mid, found = -1;

    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter elements: ");
    for (i = 0; i < n; i++) scanf("%d", &a[i]);

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

    printf("Sorted array: ");
    for (i = 0; i < n; i++) printf("%d ", a[i]);

    printf("\nEnter number to search: ");
    scanf("%d", &key);

    low = 0; high = n - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        if (a[mid] == key) { found = mid; break; }
        else if (a[mid] < key) low = mid + 1;
        else high = mid - 1;
    }

    if (found != -1)
        printf("%d found at position %d.\n", key, found + 1);
    else
        printf("%d not found.\n", key);

    return 0;
}
```

Sample Input

```
Enter number of elements: 6
Enter elements: 25 10 40 5 30 15
Enter number to search: 30
```

Sample Output

```
Sorted array: 5 10 15 25 30 40
30 found at position 5.
```

Result: The program was executed successfully and the expected output was obtained.

Practical 8: String Analysis [L3 – Apply]

Objective: To perform common string operations using standard C string functions.

Concepts Covered: Strings, arrays of characters, strlen(), strcpy(), strcat(), strcmp().

Algorithm / Pseudocode

```
START
1. Read two strings.
2. Find the length of the first string.
3. Copy the first string into another string.
4. Join the two strings.
5. Compare the two strings.
6. Display the results.
END
```

C Source Code

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

int main(void) {
    char s1[100], s2[100], copy[100], joined[210];
    int cmp;

    printf("Enter first string: ");
    fgets(s1, sizeof(s1), stdin);
    printf("Enter second string: ");
    fgets(s2, sizeof(s2), stdin);

    s1[strcspn(s1, "\n")] = '\0';
    s2[strcspn(s2, "\n")] = '\0';

    printf("Length of first string = %zu\n", strlen(s1));

    strcpy(copy, s1);
    printf("Copied string = %s\n", copy);

    strcpy(joined, s1);
    strcat(joined, " ");
    strcat(joined, s2);
    printf("Joined string = %s\n", joined);

    cmp = strcmp(s1, s2);
    if (cmp == 0)
        printf("Both strings are equal.\n");
    else if (cmp < 0)
        printf("First string comes before second string.\n");
    else
        printf("First string comes after second string.\n");

    return 0;
}
```

Sample Input

```
Enter first string: Hello
Enter second string: World
```

Sample Output

```
Length of first string = 5  
Copied string = Hello  
Joined string = Hello World  
First string comes before second string.
```

Result: The program was executed successfully and the expected output was obtained.

Practical 9: Student Record Using Structure [L3 – Apply]

Objective: To use structures to store student records and find the student with the highest total marks.

Concepts Covered: Structures, arrays of structures, member access and comparison.

Algorithm / Pseudocode

```
START
1. Read number of students n.
2. Define a Student structure with roll number, name and three marks.
3. Read details of all students.
4. Calculate each student's total.
5. Compare totals and keep the highest.
6. Display the student with the highest total.
END
```

C Source Code

```
#include <stdio.h>

struct Student {
    int roll;
    char name[50];
    float m1, m2, m3;
    float total;
};

int main(void) {
    struct Student s[50];
    int n, i, best = 0;

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

    for (i = 0; i < n; i++) {
        printf("\nStudent %d roll number: ", i + 1);
        scanf("%d", &s[i].roll);
        printf("Name: ");
        scanf(" %49[^\n]", s[i].name);
        printf("Enter marks in 3 subjects: ");
        scanf("%f %f %f", &s[i].m1, &s[i].m2, &s[i].m3);
        s[i].total = s[i].m1 + s[i].m2 + s[i].m3;

        if (s[i].total > s[best].total)
            best = i;
    }

    printf("\nTop Student: %s\n", s[best].name);
    printf("Roll No. = %d\nTotal = %.2f\n", s[best].roll, s[best].total);

    return 0;
}
```

Sample Input

```
Enter number of students: 2
Student 1 roll number: 101
Name: Asha
Enter marks in 3 subjects: 80 75 90
```

```
Student 2 roll number: 102  
Name: Rohan  
Enter marks in 3 subjects: 85 92 88
```

Sample Output

```
Top Student: Rohan  
Roll No. = 102  
Total = 265.00
```

Result: The program was executed successfully and the expected output was obtained.

Practical 10: Pointer and Array Operations [L4 – Analyze]

Objective: To access and modify array elements using pointers and understand pointer notation.

Concepts Covered: Pointers, addresses, dereferencing, arrays and pointer arithmetic.

Algorithm / Pseudocode

```
START
1. Read n and array elements.
2. Set pointer p to the first array element.
3. Display each element using pointer arithmetic.
4. Add 5 to every element using the pointer.
5. Display the modified array.
END
```

C Source Code

```
#include <stdio.h>

int main(void) {
    int a[100], n, i;
    int *p;

    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter elements: ");
    for (i = 0; i < n; i++) scanf("%d", &a[i]);

    p = a;
    printf("Elements using pointer: ");
    for (i = 0; i < n; i++)
        printf("%d ", *(p + i));

    for (i = 0; i < n; i++)
        *(p + i) += 5;

    printf("\nAfter adding 5: ");
    for (i = 0; i < n; i++)
        printf("%d ", *(p + i));

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

Sample Input

```
Enter number of elements: 4
Enter elements: 10 20 30 40
```

Sample Output

```
Elements using pointer: 10 20 30 40
After adding 5: 15 25 35 45
```

Result: The program was executed successfully and the expected output was obtained.

Practical 11: Dynamic Memory for Numbers [L3 – Apply]

Objective: To allocate memory at run time, process the values and release the allocated memory.

Concepts Covered: malloc(), free(), pointers, dynamic memory and error checking.

Algorithm / Pseudocode

```
START
1. Read n.
2. Allocate memory for n integers using malloc().
3. Check whether memory was allocated successfully.
4. Read n integers.
5. Calculate sum and average.
6. Display the results.
7. Free the allocated memory.
END
```

C Source Code

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

int main(void) {
    int *a, n, i, sum = 0;
    float average;

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

    a = malloc(n * sizeof(int));
    if (a == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Enter %d numbers: ", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &a[i]);
        sum += a[i];
    }

    average = (float)sum / n;
    printf("Sum = %d\nAverage = %.2f\n", sum, average);

    free(a);
    printf("Memory released successfully.\n");

    return 0;
}
```

Sample Input

```
Enter number of elements: 5
Enter 5 numbers: 10 20 30 40 50
```

Sample Output

```
Sum = 150
Average = 30.00
Memory released successfully.
```

Result: The program was executed successfully and the expected output was obtained.

Practical 12: Matrix Operations and File Handling [L4 – Analyze]

Objective: To add two matrices and write the result to a file, then read and display it.

Concepts Covered: Two-dimensional arrays, matrix addition, files, fopen(), fprintf(), fscanf(), fclose().

Algorithm / Pseudocode

```
START
1. Read rows and columns.
2. Read Matrix A and Matrix B.
3. Add corresponding elements to form Matrix C.
4. Open a file in write mode.
5. Write Matrix C to the file.
6. Close the file.
7. Open the file in read mode.
8. Read and display Matrix C.
9. Close the file.
END
```

C Source Code

```
#include <stdio.h>

int main(void) {
    int a[10][10], b[10][10], c[10][10];
    int r, col, i, j;
    FILE *fp;

    printf("Enter rows and columns: ");
    scanf("%d %d", &r, &col);

    printf("Enter Matrix A:\n");
    for (i = 0; i < r; i++)
        for (j = 0; j < col; j++) scanf("%d", &a[i][j]);

    printf("Enter Matrix B:\n");
    for (i = 0; i < r; i++)
        for (j = 0; j < col; j++) scanf("%d", &b[i][j]);

    fp = fopen("matrix_result.txt", "w");
    if (fp == NULL) {
        printf("Unable to open file.\n");
        return 1;
    }

    for (i = 0; i < r; i++) {
        for (j = 0; j < col; j++) {
            c[i][j] = a[i][j] + b[i][j];
            fprintf(fp, "%d ", c[i][j]);
        }
        fprintf(fp, "\n");
    }
    fclose(fp);

    fp = fopen("matrix_result.txt", "r");
    if (fp == NULL) {
        printf("Unable to read file.\n");
        return 1;
    }
}
```

```
printf("Matrix stored in file:\n");
for (i = 0; i < r; i++) {
    for (j = 0; j < col; j++)
        fscanf(fp, "%d", &c[i][j]);
    for (j = 0; j < col; j++)
        printf("%d ", c[i][j]);
    printf("\n");
}
fclose(fp);

return 0;
}
```

Sample Input

```
Enter rows and columns: 2 2
Matrix A: 1 2 3 4
Matrix B: 5 6 7 8
```

Sample Output

```
Matrix stored in file:
6 8
10 12
```

Result: The program was executed successfully and the expected output was obtained.

Practical 13: Singly Linked List [L3 – Apply]

Objective: To create a singly linked list and perform insertion, deletion and display operations.

Concepts Covered: Structures, dynamic nodes, pointers, insertion, deletion and traversal.

Algorithm / Pseudocode

```
START
1. Set head = NULL.
2. For insertion, create a new node and store the value.
3. Link the new node to the list and update head.
4. For deletion, locate the required node and adjust the links.
5. Traverse from head to NULL and display all values.
6. Repeat menu operations until exit.
END
```

C Source Code

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

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

void insertEnd(struct Node **head, int value) {
    struct Node *newNode = malloc(sizeof(struct Node));
    struct Node *temp;

    newNode->data = value;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    temp = *head;
    while (temp->next != NULL) temp = temp->next;
    temp->next = newNode;
}

void deleteValue(struct Node **head, int value) {
    struct Node *temp = *head, *prev = NULL;

    while (temp != NULL && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Value not found.\n");
        return;
    }

    if (prev == NULL) *head = temp->next;
    else prev->next = temp->next;
}
```

```

        free(temp);
    }

void display(struct Node *head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

int main(void) {
    struct Node *head = NULL;
    int choice, value;

    do {
        printf("\n1.Insert  2.Delete  3.Display  4.Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                insertEnd(&head, value);
                break;
            case 2:
                printf("Enter value to delete: ");
                scanf("%d", &value);
                deleteValue(&head, value);
                break;
            case 3:
                display(head);
                break;
            case 4:
                break;
            default:
                printf("Invalid choice.\n");
        }
    } while (choice != 4);

    return 0;
}

```

Sample Input

```

Choice: 1, Value: 10
Choice: 1, Value: 20
Choice: 1, Value: 30
Choice: 3
Choice: 2, Value: 20
Choice: 3
Choice: 4

```

Sample Output

```

10 -> 20 -> 30 -> NULL
10 -> 30 -> NULL

```

Result: The program was executed successfully and the expected output was obtained.

Practical 14: Stack and Expression Checking [L4 – Analyze]

Objective: To implement a stack using an array and use it to check balanced parentheses.

Concepts Covered: Stack, push, pop, arrays and expression validation.

Algorithm / Pseudocode

```
START
1. Set top = -1.
2. Read the expression character by character.
3. If an opening bracket is found, push it.
4. If a closing bracket is found, check the top:
   a. If matching, pop it.
   b. Otherwise, expression is unbalanced.
5. After scanning, if top is -1, brackets are balanced.
6. Otherwise, brackets are unbalanced.
END
```

C Source Code

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

#define MAX 100

int main(void) {
    char exp[MAX], stack[MAX];
    int top = -1, i, balanced = 1;

    printf("Enter expression: ");
    fgets(exp, sizeof(exp), stdin);

    for (i = 0; exp[i] != '\0'; i++) {
        char ch = exp[i];

        if (ch == '(' || ch == '[' || ch == '{')
            stack[++top] = ch;
        else if (ch == ')' || ch == ']' || ch == '}') {
            if (top == -1) {
                balanced = 0;
                break;
            }
            if ((ch == ')' && stack[top] != '(') ||
                (ch == ']' && stack[top] != '[') ||
                (ch == '}' && stack[top] != '{')) {
                balanced = 0;
                break;
            }
            top--;
        }
    }

    if (top != -1) balanced = 0;

    if (balanced)
        printf("Parentheses are balanced.\n");
    else
        printf("Parentheses are not balanced.\n");
}
```

```
        return 0;  
    }
```

Sample Input

Enter expression: (a+b) * [c-{d/e}]

Sample Output

Parentheses are balanced.

Result: The program was executed successfully and the expected output was obtained.

Practical 15: Circular Queue Implementation [L4 – Analyze]

Objective: To implement a circular queue using an array and perform insertion, deletion and display.

Concepts Covered: Queue, circular indexing, insertion, deletion and overflow/underflow conditions.

Algorithm / Pseudocode

```
START
1. Set front = -1 and rear = -1.
2. For insertion:
   a. Check whether queue is full.
   b. If empty, set front = rear = 0.
   c. Otherwise move rear circularly.
   d. Insert the value.
3. For deletion:
   a. Check whether queue is empty.
   b. Remove the front value.
   c. Move front circularly.
4. Display elements from front to rear circularly.
5. Repeat menu operations until exit.
END
```

C Source Code

```
#include <stdio.h>

#define MAX 5

int main(void) {
    int q[MAX], front = -1, rear = -1;
    int choice, value, i;

    do {
        printf("\n1.Insert  2.Delete  3.Display  4.Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        if (choice == 1) {
            if ((rear + 1) % MAX == front) {
                printf("Queue is full.\n");
            } else {
                printf("Enter value: ");
                scanf("%d", &value);
                if (front == -1) front = rear = 0;
                else rear = (rear + 1) % MAX;
                q[rear] = value;
            }
        } else if (choice == 2) {
            if (front == -1) {
                printf("Queue is empty.\n");
            } else {
                printf("Deleted = %d\n", q[front]);
                if (front == rear) front = rear = -1;
                else front = (front + 1) % MAX;
            }
        } else if (choice == 3) {
            if (front == -1) {
                printf("Queue is empty.\n");
            } else {
                for (i = front; i <= rear; i++)
                    printf("%d\t", q[i]);
                if (i < front)
                    for (i = 0; i < front; i++)
                        printf("%d\t", q[i]);
                printf("\n");
            }
        } else if (choice == 4) {
            printf("Exit\n");
            break;
        }
    } while (1);
}
```

```

        printf("Queue: ");
        i = front;
        while (1) {
            printf("%d ", q[i]);
            if (i == rear) break;
            i = (i + 1) % MAX;
        }
        printf("\n");
    }
} while (choice != 4);

return 0;
}

```

Sample Input

Insert: 10, Insert: 20, Insert: 30, Display, Delete, Display

Sample Output

```

Queue: 10 20 30
Deleted = 10
Queue: 20 30

```

Result: The program was executed successfully and the expected output was obtained.

Practical 16: Binary Tree Traversal [L4 – Analyze]

Objective: To create a binary search tree and display its elements using inorder, preorder and postorder traversal.

Concepts Covered: Binary trees, dynamic nodes, recursion, BST insertion and tree traversal.

Algorithm / Pseudocode

```
START
1. Set root = NULL.
2. Read values one by one.
3. Insert each value into the binary search tree:
   a. Smaller values go to the left.
   b. Larger values go to the right.
4. Perform inorder traversal: left, root, right.
5. Perform preorder traversal: root, left, right.
6. Perform postorder traversal: left, right, root.
7. Display the three traversal results.
END
```

C Source Code

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

struct Node {
    int data;
    struct Node *left, *right;
};

struct Node* createNode(int value) {
    struct Node *n = malloc(sizeof(struct Node));
    n->data = value;
    n->left = n->right = NULL;
    return n;
}

struct Node* insert(struct Node *root, int value) {
    if (root == NULL) return createNode(value);

    if (value < root->data)
        root->left = insert(root->left, value);
    else if (value > root->data)
        root->right = insert(root->right, value);

    return root;
}

void inorder(struct Node *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

void preorder(struct Node *root) {
    if (root != NULL) {
        printf("%d ", root->data);
    }
}
```

```

        preorder(root->left);
        preorder(root->right);
    }
}

void postorder(struct Node *root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

int main(void) {
    struct Node *root = NULL;
    int n, i, value;

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

    printf("Enter values: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &value);
        root = insert(root, value);
    }

    printf("Inorder: ");
    inorder(root);
    printf("\nPreorder: ");
    preorder(root);
    printf("\nPostorder: ");
    postorder(root);
    printf("\n");

    return 0;
}

```

Sample Input

```

Enter number of nodes: 7
Enter values: 50 30 70 20 40 60 80

```

Sample Output

```

Inorder: 20 30 40 50 60 70 80
Preorder: 50 30 20 40 70 60 80
Postorder: 20 40 30 60 80 70 50

```

Result: The program was executed successfully and the expected output was obtained.

Bloom's Taxonomy Mapping

Bloom's Taxonomy is used in this lab manual to indicate the level of thinking and practical skill expected from students. The practicals progress from applying basic programming concepts to analyzing and implementing data-structure based solutions.

Level	Bloom's Category	Meaning in Programming Lab
L1	Remember	Recall syntax, definitions, commands and basic programming concepts.
L2	Understand	Explain programming concepts and describe how a solution works.
L3	Apply	Use learned programming concepts to solve a given problem.
L4	Analyze	Break a problem into smaller parts, trace execution, identify logic and examine a solution.
L5	Evaluate	Test, verify and judge a solution using suitable inputs and criteria.
L6	Create	Design and develop a new or improved programming solution.

Practical-wise Bloom's Taxonomy Mapping

S. No.	Practical	Bloom's Level	Mapping / Expected Skill
1	Student Marks Analysis	L3 – Apply	Use data types, input/output and arithmetic operators to solve a marks calculation problem.
2	Online Shopping Bill	L3 – Apply	Apply arithmetic, relational, logical, bitwise and conditional operators to a real-life billing problem.
3	Leap Year Checker	L4 – Analyze	Analyze the conditions required to correctly identify a leap year.
4	Number Analysis	L4 – Analyze	Analyze digits of a number using loops and trace the logic for reverse and palindrome checking.
5	Function-Based Calculator	L3 – Apply	Apply functions, parameters, return values and selection statements to build a calculator.
6	Array Statistics	L3 – Apply	Apply arrays, traversal and functions to calculate statistical values.
7	Searching and Sorting	L4 – Analyze	Analyze the ordering and searching process and trace the sorting/searching logic.
8	String Analysis	L3 – Apply	Apply standard string functions to perform common string

			operations.
9	Student Record Using Structure	L3 – Apply	Apply structures and arrays of structures to manage student records.
10	Pointer and Array Operations	L4 – Analyze	Analyze the relationship between arrays and pointers and trace pointer-based access.
11	Dynamic Memory for Numbers	L3 – Apply	Apply dynamic memory allocation and pointers to process a run-time sized array.
12	Matrix Operations and File Handling	L4 – Analyze	Analyze matrix data and the process of writing and reading results using files.
13	Singly Linked List	L3 – Apply	Apply linked-list concepts, pointers and node operations to implement a list.
14	Stack and Expression Checking	L4 – Analyze	Analyze an expression using stack operations and determine whether brackets are balanced.
15	Circular Queue Implementation	L4 – Analyze	Analyze queue states and circular indexing while performing insertion and deletion.
16	Binary Tree Traversal	L4 – Analyze	Analyze tree structure and trace inorder, preorder and postorder traversals.

Overall Bloom's Taxonomy Distribution

Bloom's Level	Number of Practicals	Practical Numbers
L3 – Apply	7 practicals	1, 2, 5, 6, 8, 9, 11
L4 – Analyze	9 practicals	3, 4, 7, 10, 12, 14, 15, 16 and 13

Note: The mapping represents the primary Bloom's level assessed by each practical. Individual laboratory evaluation may also include higher-level activities such as debugging, test-case design, code improvement and viva questions, which can assess L5 (Evaluate) and L6 (Create).

Bloom's Taxonomy: Relevance to Programming for Problem Solving (PPS)

Bloom's Taxonomy strengthens Programming for Problem Solving (PPS) by moving students beyond memorising C syntax and writing routine programs. It provides a structured progression from recalling programming concepts to understanding, applying, analysing, evaluating and creating solutions. This makes the laboratory more problem-oriented, competency-based and aligned with higher-order thinking.

Bloom's Level	Relevance to PPS	How It Is Developed in the Lab	How It Strengthens PPS
L1 – Remember	Recall C syntax, data types, operators, statements, functions and basic programming terms.	Short viva questions, syntax recall, identification of operators and basic C constructs.	Builds a strong foundation so students can correctly use fundamental C concepts.
L2 – Understand	Explain what a program, algorithm, flowchart, variable, function or data structure does.	Students explain the logic of a program, convert a problem statement into steps and interpret code/output.	Develops conceptual clarity and helps students understand why a solution works.
L3 – Apply	Use learned C concepts to solve a given computational problem.	Students write and execute programs using operators, functions, arrays, strings, structures and dynamic memory.	Builds programming competency and the ability to convert concepts into working solutions.
L4 – Analyze	Break a problem into smaller parts, trace execution, identify errors and examine program logic.	Dry runs, debugging, tracing, searching/sorting analysis, pointer tracing and data-structure operations.	Strengthens logical thinking, debugging ability and systematic problem solving.
L5 – Evaluate	Test and judge whether a solution is correct, efficient, safe and suitable for different cases.	Students design test cases, compare alternative solutions, identify limitations and improve incorrect programs.	Develops critical thinking, code review skills and the ability to validate solutions.
L6 – Create	Design a complete solution by combining programming concepts to solve a new or open-ended problem.	Mini case studies, menu-driven applications, modified problems and integrated programming tasks.	Develops creativity, solution design and the ability to build independent programming solutions.

How Bloom's Taxonomy Strengthens the PPS Laboratory

- Moves learning from 'remembering syntax' to 'solving problems'.
- Develops logical and computational thinking through algorithms, flowcharts, tracing and debugging.
- Connects theory with practice by requiring students to implement concepts in executable C programs.
- Encourages students to test programs with different and boundary inputs instead of relying on one sample output.
- Builds debugging and critical-thinking skills through analysis and evaluation of program behaviour.
- Prepares students to design larger programs by progressively combining functions, arrays, pointers, files and data structures.
- Supports competency-based assessment because each practical can be mapped to an observable learning level.

- Encourages independent problem solving and provides a clear progression toward higher-order programming skills.

Recommended Assessment Approach

For stronger PPS learning outcomes, practical assessment should not be limited to whether the final program produces the correct output. Students should also be assessed on the algorithm/pseudocode, program logic, coding practice, test cases, debugging, explanation and ability to modify the solution.

Assessment Component	Suggested Bloom's Focus	PPS Skill Developed
Algorithm / Pseudocode	L2–L3	Understanding and planning a solution
Program Implementation	L3	Applying C concepts
Dry Run / Tracing	L4	Logical analysis
Debugging / Error Finding	L4–L5	Analysis and evaluation
Test Cases	L5	Verification and validation
Problem Modification / Case Study	L5–L6	Critical thinking and solution creation
Viva / Explanation	L1–L4	Conceptual understanding and reasoning