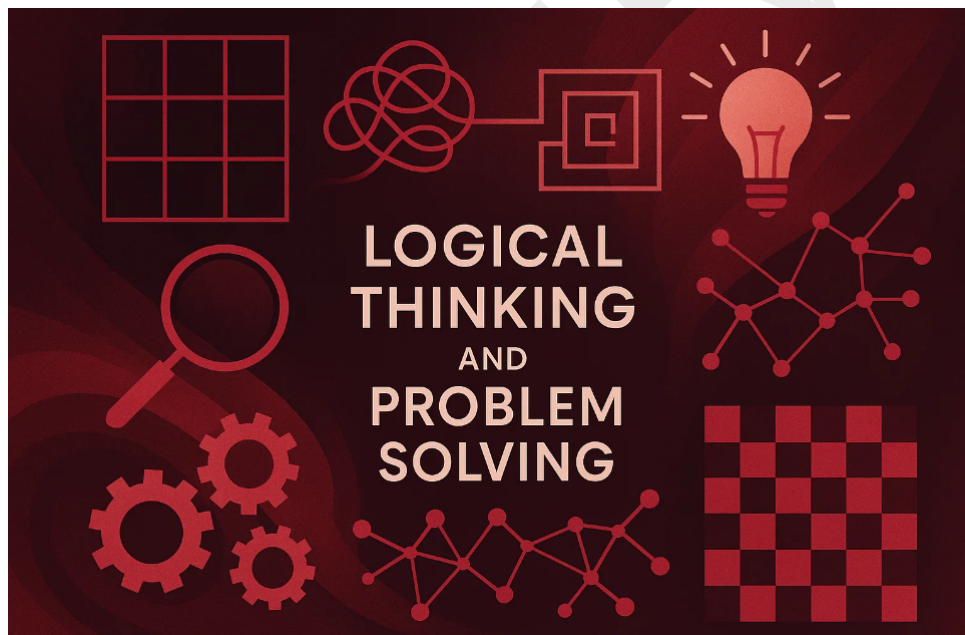


Logical Thinking and Problem Solving (25CSH-107)

Compiled by : Subhayu





Course Content

Unit-1 : BASIC CONCEPTS OF C

Ch 1 : Introduction to Problem Solving - Problem Solving phases: Understanding the problem, Making a plan of solution, Carrying out the plan, Looking back i.e. verifying, Flow charts, Algorithms.

Ch 2 : Fundamentals of C - Features & Applications of C language, Structure of writing a C Program, Header Files, Tokens, Comments, Data Types, Format Specifiers, Constants and Variables, I/O functions, Operators, Expressions, Evaluation of expressions, Type Casting, Precedence and Associativity

Ch 3 : Decision Control structure in C - Decision making statements (if, if-else, if-else-if, switch), Nesting of decision control structures.

Unit-2 : CONTROL STRUCTURE, DATA ORGANIZATION, MODULAR PROGRAMING

Ch 4 : Loop Control - structure in C, Looping statements (for, while, do-while), Nested loop, Use of jumping statements (goto, break, continue)

Ch 5 : Arrays and Strings -Introduction to array, One and Two-dimensional arrays, Declaration and Initialization of arrays, accessing elements (calculating address and length), Array Traversing, Linear Search, Strings inbuilt and Predefined Functions.

Ch 6 : Functions -Introduction to functions, Need, Function declaration/prototypes, Function definition, function call and its types, Built-in functions, User defined functions, Types of functions based on parameters and return-type, Parameter passing, Recursive functions.



Unit-3 : HANDLING HETEROGENEOUS DATA AND MEMORY MANAGEMENT

Ch 7 : Pointers: Pointers: Basics of pointers, Double pointer, Pointers and Arrays, passing array as pointer, Array of pointers, Functions returning a pointer.

Ch 8 : Structures- Structure: Basics of structure, accessing structure members, Nested structures, Array of structures, Structure and Functions, Passing structure through pointers, Union, Structure vs Union

Ch 9 : Dynamic Memory Allocation - Introduction to Dynamic memory allocation, Malloc, Calloc, Realloc.

Chapter 1

Introduction to Problem Solving

1.1 Introduction

Problem solving is a systematic approach to finding solutions to complex or simple problems. It involves understanding the problem, planning a solution, executing the solution, and verifying the results.

Key Idea

Effective problem solving requires logical thinking, planning, and iterative verification to ensure correctness.

1.2 Phases of Problem Solving

1.2.1 1. Understanding the Problem

- Carefully read and analyze the problem statement.
- Identify given information, constraints, and what needs to be found.
- Example: Calculate the average of 5 numbers. Identify inputs (5 numbers) and output (average).

1.2.2 2. Making a Plan of Solution

- Decide on the steps needed to solve the problem.
- Choose appropriate methods, formulas, or algorithms.
- Example Plan:
 1. Sum all 5 numbers.
 2. Divide the sum by 5.



1.2.3 3. Carrying Out the Plan

- Execute the steps in a logical sequence.
- Maintain accuracy and track intermediate results.
- Example: Sum = 50, Average = $50/5 = 10$.

1.2.4 4. Looking Back (Verifying)

- Check the results for correctness.
- Review each step to see if a better solution exists.
- Example: Verify average by summing numbers and dividing correctly.

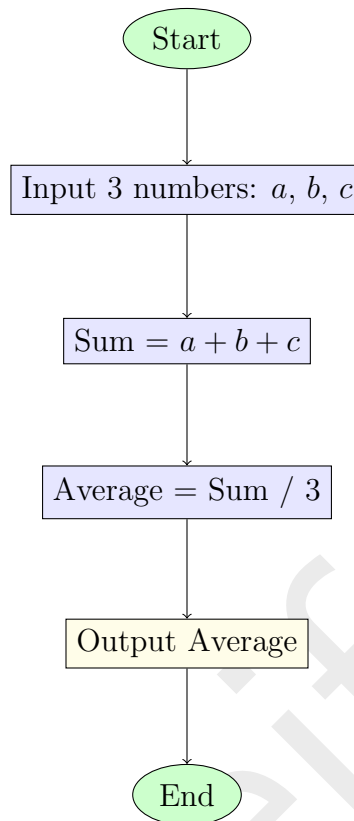
1.3 Flow Charts

Definition

A flowchart is a graphical representation of steps or processes using symbols such as:

- **Oval:** Start/End
- **Rectangle:** Process/Operation
- **Diamond:** Decision
- **Arrow:** Flow of control

Example Flowchart: Calculating Average of 3 Numbers



1.4 Algorithms

Definition

An algorithm is a step-by-step procedure to solve a specific problem. It must be clear, finite, and effective.

Example Algorithm: Average of 3 Numbers

1. Start
2. Read three numbers a, b, c
3. Compute $\text{sum} = a + b + c$
4. Compute $\text{average} = \text{sum} / 3$
5. Display average
6. Stop



1.5 Practice Questions

1. Explain the four phases of problem solving with an example.

1. **Understanding the Problem:** Identify inputs, outputs, constraints. *Example:* Find the average of 3 numbers. Inputs: 3 numbers; Output: average.
2. **Making a Plan:** Decide steps or algorithm to solve the problem. *Example Plan:* Sum the numbers and divide by 3.
3. **Carrying Out the Plan:** Execute the algorithm step by step. *Example Execution:* Sum = $10+20+30=60$, Average = $60/3=20$
4. **Looking Back (Verification):** Check correctness of results. *Example Verification:* Sum numbers again or compute average using a different method to confirm.

2. Draw a flowchart to find the largest of 3 numbers.

Steps:

1. Start
2. Input three numbers: A, B, C
3. Compare A and B
 - If $A > B$, compare A and C
 - If $A > C \rightarrow A$ is largest
 - Else $\rightarrow C$ is largest
 - Else, compare B and C
 - If $B > C \rightarrow B$ is largest
 - Else $\rightarrow C$ is largest
4. Output the largest number
5. End



3. Write an algorithm to calculate the factorial of a given number n .

1. Start
2. Read number n
3. Initialize factorial = 1
4. For $i = 1$ to n
 - factorial = factorial * i
5. Print factorial
6. Stop

—

4. Differentiate between an algorithm and a flowchart.

- **Algorithm:** Step-by-step written procedure to solve a problem; textual; language-independent.
- **Flowchart:** Graphical representation of algorithm; uses symbols like rectangles, diamonds, ovals to show processes, decisions, and flow of control.

—

5. Why is the verification phase important in problem solving? Provide an example.

Importance:

- Ensures solution is correct and meets requirements.
- Helps detect errors in logic or computation.

Example: Calculating average of 3 numbers: Sum = $10+20+30=60 \rightarrow$ Average = $60/3=20$. Verification: Re-sum numbers or check with calculator to confirm 20 is correct.

Chapter 2

Fundamentals of C

2.1 Introduction

C is a general-purpose, procedural programming language developed by Dennis Ritchie in 1972. It provides low-level memory access, structured programming, and is highly portable.

Key Idea

C is the foundation for many modern languages like C++, Java, and Python.

2.2 Features and Applications of C

2.2.1 Features of C

- Structured language with modular programming.
- Rich set of built-in operators.
- Efficient memory management via pointers.
- Portable across different platforms.
- Supports low-level hardware manipulation.

2.2.2 Applications of C

- System programming: Operating Systems, Compilers.
- Embedded systems.
- Game development.
- Application software like databases.



2.3 Structure of a C Program

2.3.1 Basic Structure

```
#include <stdio.h>          // Header file
int main() {
    // Variable declarations
    // Statements / logic
    return 0;
}
```

2.3.2 Header Files

- Contain function declarations, macros, constants.
- Examples: 'stdio.h' (I/O), 'math.h' (math functions).

2.4 Tokens in C

- Smallest units of C programs.
- Types: Keywords, Identifiers, Constants, Strings, Operators, Special symbols.

2.5 Comments in C

- Single-line: '//' This is a comment'.
- Multi-line: '/* This is a comment */'.

2.6 Data Types and Variables

2.6.1 Basic Data Types

- int, float, double, char, void

2.6.2 Constants

- Literal values that do not change.
- Examples: #define PI 3.14, const int MAX = 100;



2.6.3 Variables

- Named storage locations in memory.
- Must be declared before use.
- Example: 'int age; float salary;'

2.7 Format Specifiers

- %d → int
- %f → float
- %lf → double
- %c → char
- %s → string

2.8 Input and Output Functions

- 'printf()' → output
- 'scanf()' → input
- Example:

```
int a;
printf("Enter a number: ");
scanf("%d", &a);
printf("You entered %d", a);
```

2.9 Operators in C

2.9.1 Arithmetic Operators

'+', '-', '*', '/', '

2.9.2 Relational Operators

'==', '!=', '<', '>', '<=', '>='



2.9.3 Logical Operators

- `&&` → Logical AND
- `|` → Logical OR
- `!` → Logical NOT

2.9.4 Assignment Operators

`'='`, `'+='`, `'-='`, `'*='`, `'/='`, `'%='`

2.9.5 Increment/Decrement Operators

`'++'` (increment), `'--'` (decrement)

2.10 Expressions and Evaluation

2.10.1 Expression

A combination of operators and operands that evaluates to a value. Example: `'a + b * c'`

2.10.2 Type Casting

Converting one data type into another:

```
int a = 5;
float b = (float)a; // Explicit casting
```

2.10.3 Precedence and Associativity

- Determines order of evaluation.
- Example: `'a + b * c'` → multiplication before addition.
- Associativity resolves operators of same precedence (left-to-right or right-to-left).

2.11 Practice Questions with Solutions

2.11.1 1. Write a C program to find the sum of two numbers.

Solution:



```
#include <stdio.h>
int main() {
    int a, b, sum;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
    sum = a + b;
    printf("Sum = %d", sum);
    return 0;
}
```

2.11.2 2. Explain the difference between ‘float’ and ‘double’ data types.

Solution:

- **float:** 4 bytes, 7 digits precision.
- **double:** 8 bytes, 15 digits precision.
- Use ‘float’ for memory efficiency, ‘double’ for precision.

2.11.3 3. Output of the following code:

```
int a=5, b=2;
printf("%d", a/b);
```

Solution: ‘ $a/b = 5/2 = 2$ ’ (integer division truncates decimal). Output: ‘2’

2.11.4 4. Write a program to swap two numbers using a temporary variable.

Solution:

```
#include <stdio.h>
int main() {
    int x, y, temp;
    printf("Enter two numbers: ");
    scanf("%d %d", &x, &y);
    temp = x;
    x = y;
    y = temp;
    printf("After swapping: x=%d, y=%d", x, y);
}
```



```
    return 0;  
}
```

2.11.5 5. Explain the difference between ‘++i’ and ‘i++’.

Solution:

- ‘++i’ → Pre-increment: Increment first, then use value. Example:

```
int i=5;  
printf("%d", ++i); // Output: 6
```

- ‘i++’ → Post-increment: Use value first, then increment. Example:

```
int i=5;  
printf("%d", i++); // Output: 5  
printf("%d", i);   // Output: 6
```

Chapter 3

Decision Control Structures in C

3.1 Introduction

Decision control structures allow a program to **take different actions based on conditions**. They control the flow of a program according to logical decisions.

Key Idea

Decision control structures in C include if, if-else, if-else-if, and switch.

3.2 If Statement

```
if (condition) {  
    // statements executed if condition is true  
}
```

3.2.1 If Statement Example

```
#include <stdio.h>  
int main() {  
    int num;  
    printf("Enter a number: ");  
    scanf("%d", &num);  
    if (num > 0) {  
        printf("Positive number");  
    }  
    return 0;  
}
```



3.3 If-Else Statement

```
if (condition) {  
    // executed if condition is true  
} else {  
    // executed if condition is false  
}
```

3.3.1 If-Else Example

```
#include <stdio.h>  
int main() {  
    int num;  
    printf("Enter a number: ");  
    scanf("%d", &num);  
    if (num % 2 == 0) {  
        printf("Even number");  
    } else {  
        printf("Odd number");  
    }  
    return 0;  
}
```

3.4 If-Else-If Ladder

Used when there are **multiple conditions** to check.

3.4.1 Grading System Example

```
#include <stdio.h>  
int main() {  
    int marks;  
    printf("Enter marks: ");  
    scanf("%d", &marks);  
    if (marks >= 90) {  
        printf("Grade A");  
    } else if (marks >= 75) {  
        printf("Grade B");  
    } else if (marks >= 60) {  
        printf("Grade C");  
    }
```



```
    } else {  
        printf("Fail");  
    }  
    return 0;  
}
```

3.5 Switch Statement

Used when there are **fixed discrete values** to choose from.

3.5.1 Switch Example: Day of the Week

```
#include <stdio.h>  
int main() {  
    int day;  
    printf("Enter day number (1-7): ");  
    scanf("%d", &day);  
    switch(day) {  
        case 1: printf("Sunday"); break;  
        case 2: printf("Monday"); break;  
        case 3: printf("Tuesday"); break;  
        case 4: printf("Wednesday"); break;  
        case 5: printf("Thursday"); break;  
        case 6: printf("Friday"); break;  
        case 7: printf("Saturday"); break;  
        default: printf("Invalid day");  
    }  
    return 0;  
}
```

3.6 Nesting of Decision Structures

You can **nest if/else statements inside each other** to handle complex conditions.

3.6.1 Nested If Example: Positive Even/Odd Number

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



```
printf("Enter a number: ");
scanf("%d", &num);
if (num > 0) {
    if (num % 2 == 0) {
        printf("Positive Even");
    } else {
        printf("Positive Odd");
    }
} else {
    printf("Number is not positive");
}
return 0;
}
```

3.7 Practice Questions with Solutions

3.7.1 Check Positive, Negative or Zero

```
#include <stdio.h>
int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    if (num > 0) {
        printf("Positive");
    } else if (num < 0) {
        printf("Negative");
    } else {
        printf("Zero");
    }
    return 0;
}
```

3.7.2 Find Largest of Three Numbers

```
#include <stdio.h>
int main() {
    int a, b, c;
    printf("Enter three numbers: ");
```



```
scanf("%d %d %d", &a, &b, &c);
if (a > b && a > c) {
    printf("Largest: %d", a);
} else if (b > c) {
    printf("Largest: %d", b);
} else {
    printf("Largest: %d", c);
}
return 0;
}
```

3.7.3 Month Name Using Switch

```
#include <stdio.h>
int main() {
    int month;
    printf("Enter month number (1-12): ");
    scanf("%d", &month);
    switch(month) {
        case 1: printf("January"); break;
        case 2: printf("February"); break;
        case 3: printf("March"); break;
        case 4: printf("April"); break;
        case 5: printf("May"); break;
        case 6: printf("June"); break;
        case 7: printf("July"); break;
        case 8: printf("August"); break;
        case 9: printf("September"); break;
        case 10: printf("October"); break;
        case 11: printf("November"); break;
        case 12: printf("December"); break;
        default: printf("Invalid month");
    }
    return 0;
}
```

3.7.4 Positive and Even Using Nested If

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



```
int num;
printf("Enter a number: ");
scanf("%d", &num);
if (num > 0) {
    if (num % 2 == 0) {
        printf("Positive Even");
    } else {
        printf("Positive Odd");
    }
} else {
    printf("Number is not positive");
}
return 0;
}
```

3.7.5 Difference Between If-Else-If Ladder and Switch

- **If-Else-If Ladder:** Can handle complex conditions, ranges, or expressions.
- **Switch:** Only for discrete constant values; usually faster for many cases.

Chapter 4

Loop Control in C

4.1 Introduction

Loops allow a program to execute a block of statements ****repeatedly**** as long as a specified condition is true. They are essential for performing repetitive tasks efficiently.

Key Idea

C provides three primary looping statements: **for**, **while**, and **do-while**.

4.2 For Loop

Syntax:

```
for(initialization; condition; increment/decrement) {  
    // statements  
}
```

4.2.1 Example: Print numbers 1 to 5

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

—



4.3 While Loop

Syntax:

```
while(condition) {  
    // statements  
}
```

4.3.1 Example: Print numbers 1 to 5

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

4.4 Do-While Loop

Syntax:

```
do {  
    // statements  
} while(condition);
```

4.4.1 Example: Print numbers 1 to 5

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



4.5 Nested Loops

Loops can be **nested** to perform tasks that require multiple levels of iteration.

4.5.1 Example: Print a 3x3 multiplication table

```
#include <stdio.h>
int main() {
    int i, j;
    for(i = 1; i <= 3; i++) {
        for(j = 1; j <= 3; j++) {
            printf("%d ", i*j);
        }
        printf("\n");
    }
    return 0;
}
```

4.6 Jumping Statements in Loops

4.6.1 Break Statement

Terminates the loop immediately.

```
for(int i=1; i<=5; i++) {
    if(i == 3) break;
    printf("%d ", i);
}
```

// Output: 1 2

4.6.2 Continue Statement

Skips the current iteration and continues with the next.

```
for(int i=1; i<=5; i++) {
    if(i == 3) continue;
    printf("%d ", i);
}
```

// Output: 1 2 4 5



4.6.3 Goto Statement

Transfers control to a labeled statement. Use sparingly.

```
#include <stdio.h>
int main() {
    int i = 1;
start:
    printf("%d ", i);
    i++;
    if(i <= 5) goto start;
    return 0;
}
// Output: 1 2 3 4 5
```

4.7 Practice Questions with Solutions

4.7.1 Print all even numbers from 1 to 10 using a for loop

```
#include <stdio.h>
int main() {
    int i;
    for(i=1; i<=10; i++) {
        if(i % 2 == 0) printf("%d ", i);
    }
    return 0;
}
// Output: 2 4 6 8 10
```

4.7.2 Print the first 5 natural numbers using a while loop

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



```
}
// Output: 1 2 3 4 5
```

4.7.3 Print a 4x4 square pattern of asterisks using nested loops

```
#include <stdio.h>
int main() {
    int i, j;
    for(i=1; i<=4; i++) {
        for(j=1; j<=4; j++) {
            printf("* ");
        }
        printf("\n");
    }
    return 0;
}
```

Output:

```
* * * *
* * * *
* * * *
* * * *
```

4.7.4 Use break to stop a loop when a number 7 is reached

```
#include <stdio.h>
int main() {
    int i;
    for(i=1; i<=10; i++) {
        if(i == 7) break;
        printf("%d ", i);
    }
    return 0;
}
```

Output: 1 2 3 4 5 6

4.7.5 Use continue to skip multiples of 3 from 1 to 10

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



```
    for(i=1; i<=10; i++) {  
        if(i % 3 == 0) continue;  
        printf("%d ", i);  
    }  
    return 0;  
}
```

Output: 1 2 4 5 7 8 10

4.7.6 Use goto to print numbers 1 to 5

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

Output: 1 2 3 4 5

Chapter 5

Arrays and Strings

5.1 Introduction

Arrays and strings are **data structures** that store multiple values of the same type under a single variable name.

Key Idea

Array: A collection of elements of the same data type stored in **contiguous memory locations**.

String: An array of characters terminated by a null character (`\0`)

5.2 Arrays in C

5.2.1 Declaration and Initialization

```
data_type array_name[size];  
int numbers[5];           //Declaration  
int nums[5] = {1,2,3,4,5}; // Initialization
```

5.2.2 Accessing Array Elements

```
int x = nums[2]; // Accessing 3rd element (index starts from 0)
```

5.2.3 Array Traversing

```
for(int i=0; i<5; i++) {  
    printf("%d ", nums[i]);  
}
```



5.2.4 One-Dimensional Array Example

```
#include <stdio.h>

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int i;
    for(i=0; i<5; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Output: 10 20 30 40 50

5.3 Two-Dimensional Arrays

5.3.1 Declaration and Initialization

```
data_type array_name[rows][columns];
int matrix[2][3] = { {1,2,3}, {4,5,6} };
```

5.3.2 Accessing Elements

```
int x = matrix[1][2]; // Accessing 2nd row, 3rd column (6)
```

5.3.3 Traversing a 2D Array Example

```
#include <stdio.h>

int main() {
    int matrix[2][3] = { {1,2,3}, {4,5,6} };
    int i,j;
    for(i=0; i<2; i++) {
        for(j=0; j<3; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```



Output:

```
1 2 3
4 5 6
```

5.4 Linear Search in Arrays

5.4.1 Example: Search a number in an array

```
#include <stdio.h>
int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int key = 30, i, found=0;
    for(i=0; i<5; i++) {
        if(arr[i] == key) {
            printf("Element found at index %d", i);
            found = 1;
            break;
        }
    }
    if(!found) printf("Element not found");
    return 0;
}
```

Output: Element found at index 2

5.5 Strings in C

- Strings are arrays of characters ending with `\0`.
- Can be declared as:

```
char str1[10] = "Hello";
char str2[] = "World";
```

5.5.1 Common String Functions

- `strlen(str)` – returns length of string
- `strcpy(dest, src)` – copies src to dest
- `strcat(dest, src)` – concatenates src at the end of dest
- `strcmp(str1, str2)` – compares two strings



5.5.2 Example: String Length and Concatenation

```
#include <stdio.h>
#include <string.h>
int main() {
    char str1[20] = "Hello";
    char str2[20] = " World";
    printf("Length of str1: %lu\n", strlen(str1));
    strcat(str1, str2);
    printf("Concatenated string: %s", str1);
    return 0;
}
```

Output:

Length of str1: 5

Concatenated string: Hello World

5.6 Practice Questions with Solutions

1. Write a program to print all elements of a 1D array

```
#include <stdio.h>
int main() {
    int arr[5] = {1,2,3,4,5};
    for(int i=0; i<5; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Output: 1 2 3 4 5

2. Write a program to print elements of a 2D array

```
#include <stdio.h>
int main() {
    int matrix[2][3] = { {1,2,3}, {4,5,6} };
    for(int i=0; i<2; i++) {
        for(int j=0; j<3; j++) {
            printf("%d ", matrix[i][j]);
        }
    }
}
```



```

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

```

Output:

```

1 2 3
4 5 6

```

3. Search an element in an array using linear search

```

#include <stdio.h>
int main() {
    int arr[5] = {10,20,30,40,50};
    int key = 40, found=0;
    for(int i=0; i<5; i++) {
        if(arr[i] == key) {
            printf("Element found at index %d", i);
            found=1;
            break;
        }
    }
    if(!found) printf("Element not found");
    return 0;
}

```

Output: Element found at index 3

4. Find the length of a string and concatenate another string

```

#include <stdio.h>
#include <string.h>
int main() {
    char str1[20] = "C";
    char str2[20] = "Programming";
    printf("Length: %lu\n", strlen(str1));
    strcat(str1, str2);
    printf("Concatenated: %s", str1);
    return 0;
}

```



```
}
```

Output:

Length: 1

Concatenated: CProgramming

5. Compare two strings

```
#include <stdio.h>
#include <string.h>
int main() {
    char str1[] = "Hello";
    char str2[] = "World";
    if(strcmp(str1, str2) == 0)
        printf("Strings are equal");
    else
        printf("Strings are not equal");
    return 0;
}
```

Output: Strings are not equal

Chapter 6

Functions

6.1 Introduction

A function is a **block of code** that performs a specific task. Functions help in **modular programming**, **code reusability**, and **easy debugging**.

Key Idea

Functions divide a program into **smaller manageable modules**. C supports both **built-in functions** and **user-defined functions**.

6.2 Need for Functions

- Reduces code duplication
- Makes code modular
- Easier to maintain and debug

6.3 Function Declaration (Prototype)

A function prototype **informs the compiler** about the function name, return type, and parameters before its actual definition.

```
return_type function_name(parameter_list);  
int add(int, int);
```

6.4 Function Definition

The **actual implementation** of the function.



```
int add(int a, int b) {  
    return a + b;  
}
```

6.5 Function Call

A function is executed when called in the program.

```
int result = add(5, 10);  
printf("%d", result); // Output: 15
```

6.6 Types of Functions Based on Parameters and Return Type

- No parameters, No return value
- Parameters, No return value
- No parameters, Return value
- Parameters and Return value

6.6.1 Example: Function with Parameters and Return Value

```
#include <stdio.h>  
int square(int x) {  
    return x * x;  
}  
  
int main() {  
    int num = 5;  
    printf("Square: %d", square(num));  
    return 0;  
}
```

Output: Square: 25

6.7 Parameter Passing

6.7.1 Call by Value

A **copy of the actual parameter** is passed to the function. Changes inside the function do not affect the original variable.



6.7.2 Example: Call by Value

```
#include <stdio.h>
void addTen(int x) {
    x = x + 10;
    printf("Inside function: %d\n", x);
}
int main() {
    int num = 5;
    addTen(num);
    printf("Outside function: %d", num);
    return 0;
}
```

Output:

Inside function: 15

Outside function: 5

6.7.3 Call by Reference

The **address of the variable** is passed. Changes inside the function affect the original variable.

6.7.4 Example: Call by Reference

```
#include <stdio.h>
void addTen(int *x) {
    *x = *x + 10;
}
int main() {
    int num = 5;
    addTen(&num);
    printf("Value: %d", num);
    return 0;
}
```

Output: Value: 15

6.8 Recursive Functions

A function that **calls itself** to solve a smaller version of the original problem.



6.8.1 Example: Factorial Using Recursion

```
#include <stdio.h>

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

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

Output: Factorial: 120



6.9 Practice Questions with Solutions

1. Write a function to find the square of a number

```
#include <stdio.h>
int square(int n) {
    return n * n;
}
int main() {
    int num = 6;
    printf("Square: %d", square(num));
    return 0;
}
```

Output: Square: 36

2. Swap two numbers using call by reference

```
#include <stdio.h>
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int main() {
    int x = 5, y = 10;
    swap(&x, &y);
    printf("x = %d, y = %d", x, y);
    return 0;
}
```

Output: x = 10, y = 5

3. Print first n Fibonacci numbers using recursion

```
#include <stdio.h>
int fib(int n) {
    if(n==0) return 0;
    if(n==1) return 1;
    return fib(n-1) + fib(n-2);
}
int main() {
```



```

    int n = 5;
    for(int i=0; i<n; i++)
        printf("%d ", fib(i));
    return 0;
}

```

Output: 0 1 1 2 3

4. Write a function without parameters that prints Hello

```

#include <stdio.h>
void greet() {
    printf("Hello\n");
}
int main() {
    greet();
    return 0;
}

```

Output: Hello

5. Write a function that returns the maximum of two numbers

```

#include <stdio.h>
int max(int a, int b) {
    return (a > b) ? a : b;
}
int main() {
    printf("Max: %d", max(10, 20));
    return 0;
}

```

Output: Max: 20

Chapter 7

Pointers

7.1 Introduction

A **pointer** is a variable that stores the **memory address** of another variable.

Why pointers are important:

- They allow direct access to memory.
- They make **arrays and strings easier to manipulate**.
- They allow **dynamic memory allocation**, i.e., memory can be requested at run-time.
- They enable **passing variables by reference** to functions, so changes inside functions affect the original variables.

Key Idea

A pointer stores the **address of a variable** and uses the ***** operator to access the value stored at that address.

7.2 Basics of Pointers

Pointer Declaration: To declare a pointer, specify the type of variable it will point to, followed by `*`:

```
int *ptr; // ptr is a pointer to an int
```

Assigning Address to a Pointer: Use the `&` operator to get the address of a variable:

```
int a = 10;  
ptr = &a; // store the address of 'a' in ptr
```



Accessing Value Through a Pointer: Use “*” (dereference operator) to get the value stored at the address:

```
printf("%d", *ptr);
```

7.2.1 Example: Pointer to an Integer

```
#include <stdio.h>

int main() {
    int a = 20;          // Declare an integer variable
    int *p = &a;         // Declare a pointer p and assign address of a
    printf("%d", *p);    // Print value at the address stored in p
    return 0;
}
```

Explanation: - ‘int a = 20;’ creates a variable ‘a’ with value 20. - ‘int *p = &a;’ declares a pointer ‘p’ and stores the address of ‘a’. - ‘*p’ accesses the value at the address, which is 20.

Output: 20

7.3 Double Pointers

A **double pointer** stores the address of another pointer.

Use cases: - Dynamic 2D arrays - Functions that need to modify the pointer itself

7.3.1 Example: Double Pointer

```
#include <stdio.h>

int main() {
    int a = 50;
    int *p = &a;    // p points to a
    int **pp = &p;  // pp points to p
    printf("%d", **pp); // Dereference twice to get value of a
    return 0;
}
```

Explanation: - ‘*p’ gives value of ‘a’. - ‘**pp’ gives value at the address stored in ‘p’, which is also ‘a’.

Output: 50



7.4 Pointers and Arrays

- The name of an array is a pointer to its first element. - Access array elements using pointer arithmetic:

```
*(arr + i) // equivalent to arr[i]
```

7.4.1 Example: Access Array Using Pointer

```
#include <stdio.h>
int main() {
    int arr[5] = {1,2,3,4,5};
    int *p = arr; // points to first element
    for(int i=0; i<5; i++) {
        printf("%d ", *(p+i));
    }
    return 0;
}
```

Explanation: - 'p = arr;' means 'p' points to first element 'arr[0]'. - '*(p+i)' moves 'i' steps forward in memory, giving each array element.

Output: 1 2 3 4 5

7.5 Passing Array as Pointer to Function

Passing arrays as pointers saves memory and allows functions to modify original elements.

7.5.1 Example: Sum of Array Elements

```
#include <stdio.h>
int sum(int *arr, int n) {
    int s = 0;
    for(int i=0; i<n; i++)
        s += *(arr + i);
    return s;
}
int main() {
    int arr[4] = {1,2,3,4};
    printf("%d", sum(arr, 4));
    return 0;
}
```



Explanation: - 'arr' is passed as a pointer. - '*(arr+i)' accesses each element for summation.

Output: 10

7.6 Pointer Arithmetic

Pointers can:

- Be incremented/decremented: 'p++', 'p--'
- Traverse arrays efficiently
- Compute distance between two elements

7.7 Practice Questions with Solutions

7.7.1 Pointer Basics

```
#include <stdio.h>
int main() {
    int x = 100;
    int *p = &x;
    printf("%d", *p);
    return 0;
}
```

Output: 100

7.7.2 Double Pointer

```
#include <stdio.h>
int main() {
    int x = 5;
    int *p = &x;
    int **pp = &p;
    **pp = 50;
    printf("%d", x);
    return 0;
}
```

Output: 50



7.7.3 Array Traversal Using Pointer

```
#include <stdio.h>
int main() {
    int arr[5] = {10,20,30,40,50};
    int *p = arr;
    for(int i=0; i<5; i++) {
        printf("%d ", *(p+i));
    }
    return 0;
}
```

Output: 10 20 30 40 50

7.7.4 Pass Array to Function and Find Maximum

```
#include <stdio.h>
int max(int *arr, int n) {
    int m = arr[0];
    for(int i=1; i<n; i++)
        if(arr[i] > m) m = arr[i];
    return m;
}
int main() {
    int arr[4] = {12,7,25,18};
    printf("%d", max(arr, 4));
    return 0;
}
```

Output: 25

7.7.5 Increment Array Elements Using Pointer

```
#include <stdio.h>
void increment(int *arr, int n) {
    for(int i=0; i<n; i++)
        *(arr+i) += 1;
}
int main() {
    int arr[3] = {1,2,3};
    increment(arr, 3);
}
```



```
    for(int i=0; i<3; i++)  
        printf("%d ", arr[i]);  
    return 0;  
}
```

Output: 2 3 4

Notefity

Chapter 8

Structures

8.1 Introduction

A **structure** in C is a user-defined data type that allows grouping variables of different types under a single name.

Why structures are useful:

- To represent a record (e.g., a student with name, age, marks).
- To manage heterogeneous data together.
- To pass multiple related variables to functions easily.

Key Idea

A structure lets you combine variables of different types under a single name. Access individual elements using the dot operator ‘.’.

8.2 Basics of Structure

Declaration:

```
struct Student {  
    char name[50];  
    int age;  
    float marks;  
};
```

Accessing Members:

```
struct Student s1;  
s1.age = 20;  
s1.marks = 95.5;
```



8.2.1 Example: Simple Structure

```
#include <stdio.h>
struct Student {
    char name[50];
    int age;
    float marks;
};
int main() {
    struct Student s1;
    s1.age = 18;
    s1.marks = 90.5;
    printf("Age: %d, Marks: %.2f", s1.age, s1.marks);
    return 0;
}
```

Explanation: - 'struct Student' defines the structure with three members. - 's1' is a variable of type 'struct Student'. - Members are accessed using '.' (dot operator).

Output: Age: 18, Marks: 90.50

8.3 Nested Structures

Structures can contain other structures as members.

8.3.1 Example: Nested Structure

```
#include <stdio.h>
struct Date {
    int day, month, year;
};
struct Student {
    char name[50];
    struct Date dob;
};
int main() {
    struct Student s1;
    s1.dob.day = 15;
    s1.dob.month = 10;
    s1.dob.year = 2005;
    printf("DOB: %d-%d-%d", s1.dob.day, s1.dob.month, s1.dob.year);
}
```



```
    return 0;
}
```

Output: DOB: 15-10-2005

8.4 Array of Structures

You can declare arrays of structures to store multiple records.

8.4.1 Example: Array of Students

```
#include <stdio.h>
struct Student {
    char name[50];
    int age;
};
int main() {
    struct Student s[2];
    s[0].age = 18;
    s[1].age = 20;
    printf("%d %d", s[0].age, s[1].age);
    return 0;
}
```

Output: 18 20

8.5 Structure and Functions

Structures can be passed to functions:

8.5.1 Passing Structure by Value

```
#include <stdio.h>
struct Student {
    int age;
};
void display(struct Student s) {
    printf("%d", s.age);
}
int main() {
```



```
    struct Student s1;
    s1.age = 19;
    display(s1);
    return 0;
}
```

8.5.2 Passing Structure by Pointer

```
#include <stdio.h>
struct Student {
    int age;
};
void display(struct Student *s) {
    printf("%d", s->age); // use '->' with pointer
}
int main() {
    struct Student s1;
    s1.age = 21;
    display(&s1);
    return 0;
}
```

8.6 Union

A **union** is similar to a structure but all members share the same memory location.

8.6.1 Example: Union

```
#include <stdio.h>
union Data {
    int i;
    float f;
};
int main() {
    union Data d;
    d.i = 10;
    d.f = 20.5;
    printf("%f", d.f);
    return 0;
}
```



Explanation: Only one member can store value at a time. Assigning 'd.f' overwrites 'd.i'.

Output: 20.500000

8.7 Structure vs Union

- **Structure:** Each member has its own memory; size = sum of all members.
- **Union:** All members share memory; size = largest member.
- Use structure when you need to store all members simultaneously.
- Use union when only one member is needed at a time (saves memory).

8.8 Practice Questions with Solutions

8.8.1 Structure Basics

```
#include <stdio.h>
struct Student {
    int age;
    float marks;
};
int main() {
    struct Student s1;
    s1.age = 20;
    s1.marks = 95.5;
    printf("%d %.2f", s1.age, s1.marks);
    return 0;
}
```

Output: 20 95.50

8.8.2 Nested Structure

```
#include <stdio.h>
struct Date { int day, month, year; };
struct Student { char name[50]; struct Date dob; };
int main() {
    struct Student s1;
    s1.dob.day = 15;
```



```

        s1.dob.month = 10;
        s1.dob.year = 2005;
        printf("%d-%d-%d", s1.dob.day, s1.dob.month, s1.dob.year);
        return 0;
    }

```

Output: 15-10-2005

8.8.3 Array of Structures

```

#include <stdio.h>
struct Student { int age; };
int main() {
    struct Student s[2];
    s[0].age = 18;
    s[1].age = 20;
    printf("%d %d", s[0].age, s[1].age);
    return 0;
}

```

Output: 18 20

8.8.4 Passing Structure to Function by Pointer

```

#include <stdio.h>
struct Student { int age; };
void display(struct Student *s) { printf("%d", s->age); }
int main() {
    struct Student s1;
    s1.age = 21;
    display(&s1);
    return 0;
}

```

Output: 21

8.8.5 Union Example

```

#include <stdio.h>
union Data { int i; float f; };
int main() {

```



```
    union Data d;  
    d.i = 10;  
    d.f = 20.5;  
    printf("%f", d.f);  
    return 0;  
}
```

Output: 20.500000

Notefity

Chapter 9

Dynamic Memory Allocation

9.1 Introduction

In C, normally variables are stored in memory at compile-time (static memory allocation). **Dynamic memory allocation** allows allocating memory at **runtime**, which is useful when:

- You don't know the exact size of data beforehand.
- You want efficient memory usage.

Key Idea

Dynamic memory is allocated from the **heap segment** during program execution, and pointers are used to access it.

9.2 malloc()

malloc stands for "memory allocation". It allocates a block of memory of given size and returns a pointer to the first byte.

Syntax:

```
ptr = (castType*) malloc(size_in_bytes);
```

9.2.1 Example: Using malloc

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *p;
    p = (int*) malloc(3 * sizeof(int)); // allocate memory for 3 integers
```



```
p[0] = 10;
p[1] = 20;
p[2] = 30;
for(int i=0; i<3; i++)
    printf("%d ", p[i]);
free(p); // free allocated memory
return 0;
}
```

Explanation: - 'malloc' allocates memory but does not initialize it. - 'sizeof(int)' ensures correct memory for integers. - 'free(p)' releases memory back to heap.

Output: 10 20 30

9.3 calloc()

calloc stands for "contiguous allocation". It allocates memory for multiple elements and initializes them to zero.

Syntax:

```
ptr = (castType*) calloc(num_elements, size_of_element);
```

9.3.1 Example: Using calloc

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *p;
    p = (int*) calloc(3, sizeof(int)); // allocate and initialize to 0
    for(int i=0; i<3; i++)
        printf("%d ", p[i]);
    free(p);
    return 0;
}
```

Explanation: - 'calloc' automatically initializes all elements to 0. - Memory is contiguous, so pointer arithmetic works.

Output: 0 0 0

9.4 realloc()

realloc is used to change the size of previously allocated memory block.

**Syntax:**

```
ptr = (castType*) realloc(ptr, new_size_in_bytes);
```

9.4.1 Example: Using realloc

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *p;
    p = (int*) malloc(2 * sizeof(int));
    p[0] = 5; p[1] = 10;
    p = (int*) realloc(p, 4 * sizeof(int)); // increase size to 4
    p[2] = 15; p[3] = 20;
    for(int i=0; i<4; i++)
        printf("%d ", p[i]);
    free(p);
    return 0;
}
```

Explanation: - 'realloc' extends the memory block to hold more elements. - Old values are preserved, new memory may contain garbage if not initialized.

Output: 5 10 15 20

9.5 Practice Questions with Solutions**9.5.1 Allocate Memory for 5 Integers Using malloc**

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *p = (int*) malloc(5 * sizeof(int));
    for(int i=0; i<5; i++)
        p[i] = i+1;
    for(int i=0; i<5; i++)
        printf("%d ", p[i]);
    free(p);
    return 0;
}
```

Output: 1 2 3 4 5



9.5.2 Use calloc and Initialize Array

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *p = (int*) calloc(4, sizeof(int));
    for(int i=0; i<4; i++)
        printf("%d ", p[i]);
    free(p);
    return 0;
}
```

Output: 0 0 0 0

9.5.3 Resize Memory Using realloc

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *p = (int*) malloc(2 * sizeof(int));
    p[0] = 100; p[1] = 200;
    p = (int*) realloc(p, 4 * sizeof(int));
    p[2] = 300; p[3] = 400;
    for(int i=0; i<4; i++)
        printf("%d ", p[i]);
    free(p);
    return 0;
}
```

Output: 100 200 300 400

9.5.4 Check Behavior Without Free

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *p = (int*) malloc(3 * sizeof(int));
    p[0]=1; p[1]=2; p[2]=3;
    printf("%d %d %d", p[0], p[1], p[2]);
    // forgot free(p)
    return 0;
}
```



}

Explanation: Memory will be released automatically at program termination, but not freeing leads to memory leak in long-running programs.

Output: 1 2 3

Notefity



Final Examination – Sample Paper

Total Marks : 60

Time Allotted : 3 hours

Instructions:

- The question paper consists of 3 sections. It is compulsory for students to attempt all questions of Section A and Section B.
- Section A has 5 questions of 2 marks each, Section B has 4 questions of 5 marks each, and Section C has 4 questions of 10 marks each (out of which any 3 are to be attempted).
- Question no. 10 & 11 of Section C are compulsory to be attempted.
- Students to attempt ANY ONE question from question no. 12 & question no. 13 of Section C.

Section A (5x2 = 10 marks)

1. Define `const` and `#define` in C. Write a program that uses each to store the value of PI and calculate the circumference of a circle.
2. Write a program to find the largest of three numbers using `if-else-if`.
3. Declare a pointer to pointer (`int **p`) and explain how it can be used to store a 2D array dynamically.
4. Compare `sizeof` for a structure and union containing the same members and explain why the sizes differ.
5. Use `calloc` to allocate memory for 4 floats, read values from user, and print the average.

Section B (4x5 = 20 marks)

6. Give an example demonstrating operator precedence and associativity in an arithmetic expression.



7. Implement linear search on an array of 5 integers and display whether a given number is found or not.
8. Write a program using a structure `Student` with `name`, `roll`, and `marks`. Input details for one student and print them.
9. Define a union `Data` containing an `int`, `float`, and `char`. Assign values and observe which value is retained. Explain.

Section C (3x10 = 30 marks)

10. What is the difference between an algorithm and a flowchart? Draw a flowchart for finding the largest of three numbers.
11. What do you mean by arguments and parameters in a function? Write a function `max()` that takes two integers as parameters and returns the maximum. Call it from `main()` and display the result.
12. Describe why the need for pointers arises. How are they more beneficial in dynamic memory allocation? Explain with an example.

OR

13. Distinguish between the use cases of `malloc`, `calloc`, and `realloc`. Write a program that resizes a dynamically allocated array of integers from 3 to 6 elements using `realloc`.