Problem Solver (DI03011011) - Problem Solver

Antigravity AI

June 4, 2026

Problem Solver

First Edition: 2026

Author: Antigravity AI
Website: milav.in
YouTube: @milav.dabgar

To my students,
who constantly inspire me to find new analogies,
break down complex theories, and make learning
an engineered science.

Preface

About this Book

Welcome to **Computer Networks**! This textbook is meticulously designed to serve as a comprehensive problem solver and study companion. It provides a robust and intuitive foundation in the core principles of this subject, bridging the gap between theoretical models and practical applications. A unique feature of this book is its focus on decoding complex concepts using clear, step-by-step methodologies and real-world analogies.

Target Audience

This book is intended for students and professionals looking to master the concepts of Computer Networks. Whether you are revising for exams, seeking to solidify your fundamental understanding, or looking for practical examples to clarify abstract concepts, this book has been structured to support your learning journey. It serves as an accessible guide designed to remove the fear of complex topics, replacing it with an appreciation for the underlying mechanics.

Scope and Coverage

The coverage is divided logically to follow standard academic curriculums. Each chapter focuses on key topics, breaking them down into digestible sections:

- **Theoretical Insights** – Core concepts explained intuitively.
- **Method Blocks** – Standardized, step-by-step procedures for solving specific types of problems.
- **Worked Examples** – Fully solved problems that apply the methods in practice.

We hope this resource proves invaluable in your studies and helps you achieve excellence in **Computer Networks**. Happy learning!

Acknowledgments

Writing a comprehensive book that connects the fundamental forces of Chemistry with practical engineering applications requires more than just academic knowledge—it requires a community of support. I would like to express my sincere gratitude to everyone who contributed to the realization of this project.

Specially, I extend my heartfelt gratitude to the **Department of Technical Education (DTE), Government of Gujarat**, and **Government Polytechnic, Palanpur**, for providing an environment that fosters innovation, academic rigor, and continuous professional development.

I am deeply thankful to my family for their unwavering patience and support during the countless hours spent researching, formatting, and refining these chapters.

Finally, my deepest appreciation goes to my students. Your curiosity, challenging questions, and continuous feedback are the true driving force behind the unique analogies and streamlined structure of this book. This textbook was engineered for you.

Antigravity AI

About the Author

Milav Jayeshkumar Dabgar is an engineering educator and R&D professional with over 9 years of experience in electronics hardware development, embedded systems, AI/ML, and full-stack software engineering. He currently serves as a **Lecturer (Class - II) & Innovation Leader** at Government Polytechnic, Palanpur, under the Education Department of the Government of Gujarat.

Milav holds a Master of Engineering (ME) in Communication Systems from L.D. College of Engineering and a Bachelor of Engineering (BE) in Electronics & Communication. He is also currently pursuing a **BS in Data Science and Applications from IIT Madras**, where he has completed both the **Diploma in Programming** and the **Diploma in Data Science** with distinction.

Most recently, he completed the **AICTE QIP PG Certificate Program in Deep Learning from SVNIT Surat** with a **perfect 10/10 CGPA**, topping his batch.

A dedicated mentor, Milav has guided numerous student teams in securing over **INR 45 lakhs** in innovation funding, including INR 25 lakhs from Shark Tank India and INR 20 lakhs through iHub Gujarat, resulting in two student patents. He is recognized as an **NPTEL Evangelist** and **NPTEL Discipline Star** in Computer Science, reflecting his commitment to continuous learning and academic excellence.

His technical expertise spans from embedded systems (8051, STM32, Arduino) to modern web technologies (Next.js, Python, PostgreSQL) and Data Science (TensorFlow, PyTorch). Through this textbook, he aims to simplify complex Engineering Chemistry concepts by integrating relatable, real-world analogies drawn from modern tech and engineering landscapes.

Milav is also an active content creator, running a popular **YouTube channel** (@milav.dabgar) where he shares technical tutorials and academic resources. He maintains a personal blog and resource hub at milav.in and can be reached for professional collaborations on LinkedIn.

Contents

List of Figures

List of Tables

CHAPTER 1

Basics of C Programming

1.1 Algorithms and Flowcharts

Methodology:

Developing an Algorithm and Flowchart An algorithm is a step-by-step procedure to solve a computational problem. A flowchart is its graphical representation. Use the following steps to construct them:

1. **Understand the Problem:** Identify the required inputs, the mathematical or logical processing needed, and the expected outputs.
2. **Write the Algorithm:** Express the logic in simple English-like statements. Always start with "Start" and end with "Stop".
3. **Draw the Flowchart:** Translate the algorithm steps into standard graphical symbols connected by flow lines (arrows).

Standard Flowchart Symbols:

Symbol Shape	Purpose
Oval	Start or Stop the process
Parallelogram	Input or Output operations
Rectangle	Processing or Mathematical Calculation
Diamond	Decision making and branching
Arrows	Flow of control

Worked Example:

Design a solution to find the largest of two numbers **Step 1: Write the Algorithm**

1. Start
2. Read two numbers A and B
3. If A $>$ B, print "A is largest" and go to Step 5
4. Else, print "B is largest"
5. Stop

Step 2: Construct the Flowchart Logic

- **Oval:** Start
- **Arrow down** to **Parallelogram:** Input A and B
- **Arrow down** to **Diamond:** Is A $>$ B?
- **True branch** to **Parallelogram:** Print A

- **False branch** to **Parallelogram:** Print B
- Both branches merge down to **Oval:** Stop

1.2 Structure of a C Program

Methodology:

Standard Structure of a C Program A C program is a collection of functions and declarations. Building a program requires arranging code in this standard sequence:

1. **Documentation Section:** Comments `/* ... */` explaining the program logic.
2. **Link Section:** Preprocessor directives (`#include`) to link standard library files.
3. **Definition Section:** Symbolic constants defined using `#define`.
4. **Global Declaration Section:** Variables accessible throughout the entire program.
5. **Main Function:** `int main()` block where execution begins. Contains local declarations and executable statements.
6. **Subprogram Section:** User-defined functions placed after `main()`.

Worked Example:

Write a standard C program to calculate the area of a rectangle **Step-by-step Code Construction:**

```
/* Documentation: Program to find area of a rectangle */
#include <stdio.h>  /* Link Section */

int main() {        /* Main Function */
    /* Local Declarations */
    int length;
    int width;
    int area;

    /* Executable Statements */
    length = 10;
    width = 5;
    area = length * width;

    printf("Area is %d\n", area);

    return 0;
}
```

Explanation: The program begins execution at `main()`. Three integer variables are declared to hold the data. Values are assigned to `length` and `width`, multiplied to compute the `area`, and outputted to the screen using the standard I/O library function `printf`.

1.3 Character Set and C Tokens

Methodology:

Rules for C Tokens and Identifiers The C character set consists of letters, digits, and special characters. Tokens are the smallest individual units built from these characters. They include:

1. **Keywords:** Reserved words with predefined meanings (e.g. `int`, `return`, `void`).
2. **Identifiers:** User-defined names for variables and functions.
3. **Constants:** Fixed values that do not change during execution.
4. **Strings:** Sequence of characters enclosed in double quotes.
5. **Operators:** Symbols triggering mathematical or logical actions.
6. **Special Symbols:** Punctuation marks like brackets and semicolons.

Rules for Naming Variables (Identifiers):

1. Must begin with a letter (uppercase or lowercase) or an underscore (`_`).
2. Subsequent characters can be letters, digits, or underscores.
3. Cannot be a reserved keyword.
4. Cannot contain spaces or special characters (other than underscore).
5. Case-sensitive (e.g. `Sum` and `sum` are treated as distinct variables).

Worked Example:

Evaluate the validity of given variable names Given variables: `student_name`, `1st_rank`, `total sum`, `int`, `_count`

Variable Name	Status	Reason
`student_name`	Valid	Starts with a letter and uses only valid characters.
`1st_rank`	Invalid	Cannot start with a digit.
`total sum`	Invalid	Cannot contain spaces.
`int`	Invalid	`int` is a reserved keyword in C.
`_count`	Valid	Starting with an underscore is allowed.

1.4 Data Types and Constants

Methodology:

Selecting Data Types in C Data types specify the type of data a variable can hold and the memory size allocated. Selecting the right type is critical for problem-solving.

1. **Predefined (Primary) Data Types:**
   - **Integer (`int`):** Whole numbers. Modifiers include `signed`, `unsigned`, `short`, and `long`.
   - **Constants:** Integer constants can be written in decimal, octal (prefix `0`), or hexadecimal (prefix `0x`).
   - **Floating-Point (`float`):** Single-precision fractional numbers.
   - **Double (`double`):** Double-precision fractional numbers for higher mathematical accuracy.
   - **Character (`char`):** Single characters enclosed in single quotes. Can also be `signed` or `unsigned`.

2. **User-Defined (Derived) Data Types:**
    - **Arrays:** A collection of elements of the same data type stored sequentially in memory.
    - **Structures (`struct`):** A collection of variables of varying data types grouped under a single custom name.

Worked Example:

Declare variables for processing employee data Write a C program snippet declaring variables for an employee's ID, salary, department code, and a list of 5 monthly performance scores. Use octal and hexadecimal constants as an example.

Solution:

```
#include <stdio.h>

/* User-defined Structure */
struct Employee {
    long id;              /* Predefined: long integer */
    float salary;         /* Predefined: single-precision floating-point */
    double bonus;         /* Predefined: double-precision floating-point */
    char deptCode;        /* Predefined: character */
    int scores[5];        /* User-defined: Array of integers */
};

int main() {
    /* Utilizing the structure and various data types */
    struct Employee emp1;
    emp1.id = 1042001L;
    emp1.salary = 45500.50f;
    emp1.bonus = 1250.75;
    emp1.deptCode = 'A';

    /* Using hexadecimal constant for a specific memory identifier */
    unsigned int mem_loc = 0x1A4F;

    /* Using octal constant */
    int permissions = 0755;

    return 0;
}
```

Explanation: The code uses `long`, `float`, `double`, and `char` for basic attributes. Modifiers like `unsigned` represent positive-only values. An array `scores[5]` groups multiple integers, and a `struct` encapsulates all these diverse types into a single logical unit. Octal (starting with 0) and hexadecimal (starting with 0x) integer constants are successfully initialized.

CHAPTER 2

Operator and Expressions and Input Output Functions

2.1 Types of Operators

Operators are symbols that tell the compiler to perform specific mathematical or logical functions.

Methodology:

Arithmetic Operators Arithmetic operators perform mathematical operations such as addition, subtraction, multiplication, division, and modulus.

1. **Addition (+)**: Adds two operands.
2. **Subtraction (-)**: Subtracts the second operand from the first.
3. **Multiplication (*)**: Multiplies two operands.
4. **Division (/)**: Divides numerator by denominator. For integer division, the fractional part is truncated.
5. **Modulus (%)**: Outputs the remainder of integer division.

Worked Example:

Integer Division and Modulus Evaluation Given integers $a = 14$ and $b = 4$. Evaluate a/b and $a\%b$.
Step 1: Evaluate integer division (a/b)

$$14/4 = 3.5$$

Since both operands are integers, truncate the fractional part. Result: 3.
Step 2: Evaluate modulus $(a\%b)$ The modulus operator finds the remainder of the division.

$$14 = 3 \times 4 + 2$$

The remainder is 2. Result: 2.

Methodology:

Relational and Logical Operators Relational operators compare two values. Logical operators combine multiple relational conditions.

1. **Relational Operators**: == (Equal to), ! = (Not equal to), > (Greater than), < (Less than), >= (Greater than or equal to), <= (Less than or equal to). Result is 1 (True) or 0 (False).
2. **Logical AND (&&)**: Returns 1 if both conditions are true. Returns 0 otherwise. Short-circuits if the first condition is false.
3. **Logical OR (||)**: Returns 1 if at least one condition is true. Returns 0 otherwise. Short-circuits if

the first condition is true.

4. **Logical NOT (!)**: Reverses the logical state of the operand.

Worked Example:

Evaluating Logical Expressions Evaluate the expression `!(x > y) && (x != 0)` for $x = 5$ and $y = 10$.
Step 1: Evaluate relational expressions inside parentheses Condition 1: $x > y \Rightarrow 5 > 10 \Rightarrow 0$ (False) Condition 2: $x! = 0 \Rightarrow 5! = 0 \Rightarrow 1$ (True)
Step 2: Apply Logical NOT `!(0)` $\Rightarrow 1$ (True)
Step 3: Apply Logical AND `1 && 1` $\Rightarrow 1$ (True)
Final Output: 1

Methodology:

Assignment and Conditional Operators

1. **Simple Assignment (=)**: Assigns the right-side value to the left-side variable.

2. **Compound Assignment (+= -= *= /= %=)**: Performs the arithmetic operation and assigns the result. $A+ = B$ is equivalent to $A = A + B$.

3. **Conditional Operator (? :)**: A ternary operator taking three operands: `condition ? expr1 : expr2`. If condition is true (non-zero), evaluates to `expr1`. Otherwise, evaluates to `expr2`.

Worked Example:

Compound Assignment and Ternary Evaluation Given integer $x = 10$. Execute the following sequence: 1. `x += 5;` 2. `int y = (x % 2 == 0) ? x * 2 : x * 3;` Find the final values of x and y.
Step 1: Evaluate compound assignment `x += 5` is $x = x + 5 \Rightarrow x = 10 + 5 \Rightarrow 15$. So, $x = 15$.
Step 2: Evaluate the ternary condition Condition: `x % 2 == 0` $\Rightarrow$ `15 % 2 == 0` $\Rightarrow$ `1 == 0` $\Rightarrow 0$ (False).
Step 3: Select ternary branch Since condition is false, select `expr2`: `x * 3`. $y = 15 \times 3 = 45$.
Final values: $x = 15$, $y = 45$.

Methodology:

Increment and Decrement Operators These unary operators add or subtract 1 from a variable.

1. **Prefix (++A or −A)**: The value is incremented/decremented first, and then the new value is used in the expression.

2. **Postfix (A++ or A−)**: The current value is used in the expression first, and then the value is incremented/decremented.

Worked Example:

Evaluating Prefix and Postfix Operations Given integer $a = 5$. Evaluate the expression `int b = ++a + a++;` and determine final values of a and b.
Note: Standard C leaves multiple modifications of a variable unsequenced but we evaluate purely based on strictly ordered hypothetical steps for conceptual understanding.
Step 1: Evaluate left operand (++a) Pre-increment increases a before use. a becomes 6. Left operand evaluates to 6.
Step 2: Evaluate right operand (a++) Post-increment uses current value, then increases. Current value is 6. Right operand evaluates to 6. a then becomes 7.
Step 3: Evaluate addition $b = 6 + 6 = 12$.

Final values: $a = 7$, $b = 12$.

2.2 Operator Precedence and Associativity

Methodology:

Operator Precedence and Associativity Rules When multiple operators appear in an expression, Precedence determines which operation is performed first. Associativity determines the direction of evaluation when operators have the same precedence level.

1. **Parentheses** (): Highest precedence. Left-to-Right.
2. **Unary** (! ++ − + -): Right-to-Left.
3. **Multiplicative** (* / %): Left-to-Right.
4. **Additive** (+ -): Left-to-Right.
5. **Relational** (< <= > >=): Left-to-Right.
6. **Equality** (== !=): Left-to-Right.
7. **Logical AND** (&&): Left-to-Right.
8. **Logical OR** (||): Left-to-Right.
9. **Conditional** (? :): Right-to-Left.
10. **Assignment** (= += -= *=): Right-to-Left.

Worked Example:

Precedence and Associativity Application Evaluate the expression: `5 + 2 * 3 - 8 / 4`
Step 1: Identify highest precedence operators Multiplication (`*`) and Division (`/`) have higher precedence than Addition (`+`) and Subtraction (`-`).
Step 2: Apply left-to-right associativity for multiplicative operators First evaluate `2 * 3` = 6. Expression becomes: `5 + 6 - 8 / 4`. Next evaluate `8 / 4` = 2. Expression becomes: `5 + 6 - 2`.
Step 3: Apply left-to-right associativity for additive operators First evaluate `5 + 6` = 11. Expression becomes: `11 - 2`. Next evaluate `11 - 2` = 9.
Final Result: 9.

2.3 Evaluation of Arithmetic and Logical Expression

Methodology:

Algorithm for Expression Evaluation To compute the final value of a mixed expression:

1. **Substitute** all variable values into the expression.
2. **Resolve Parentheses** from innermost to outermost.
3. **Evaluate Unary Operators** right-to-left.
4. **Evaluate Binary Operators** descending order of precedence.
5. **Apply Associativity** for operators at the same precedence level.

Worked Example:

Complex Logical Expression Evaluation Given $a = 4$, $b = 5$, $c = 6$. Evaluate: `ans = (a < b) || (b > c) && (a == 4);`
Step 1: Substitute values `(4 < 5) || (5 > 6) && (4 == 4)`
Step 2: Evaluate relational operators in parentheses `(4 < 5)` $\Rightarrow 1$ (True) `(5 > 6)` $\Rightarrow 0$ (False) `(4 == 4)` $\Rightarrow 1$ (True) Expression reduces to: `1 || 0 && 1`
Step 3: Apply precedence of logical operators Logical AND (`&&`) has higher precedence than Logical OR (`||`). Evaluate `0 && 1` $\Rightarrow 0$. Expression reduces to: `1 || 0`
Step 4: Evaluate Logical OR Evaluate `1 || 0` $\Rightarrow 1$.
Final result: `ans = 1`.

2.4 Input and Output Functions

Methodology:

Formatted Input and Output Functions Formatted I/O functions allow reading and displaying data in a specific format using format specifiers like `%d` (integer), `%f` (float), `%c` (character), and `%s` (string).

1. **printf()**: Used for formatted output to the console. Syntax: `printf("format string", arg1, arg2, ...);`
2. **scanf()**: Used for formatted input from the console. Requires memory addresses (`&`) for non-array variables. Syntax: `scanf("format string", &arg1, &arg2, ...);`

Worked Example:

Calculating Area Using Formatted IO Write the C code fragment to read the radius of a circle as a float, calculate its area, and print the result up to 2 decimal places.
Step 1: Declare variables `float radius, area;`
Step 2: Read input using scanf `scanf("%f", &radius);` (The `&` operator passes the memory address of `radius`.)
Step 3: Perform computation `area = 3.14159 * radius * radius;`
Step 4: Print formatted output Use `%.2f` to restrict output to two decimal places. `printf("Area = %.2f", area);`

Methodology:

Unformatted Input and Output Functions Unformatted I/O functions process single characters or raw strings without format specifiers.

1. **getch()**: Reads a single character from the keyboard immediately without waiting for the Enter key. It does not echo the character on the screen.
2. **putch()**: Writes a single character directly to the console screen.
3. **gets()**: Reads an entire string (including spaces) from standard input until the Enter key is pressed. (Note: Unsafe in modern C due to buffer overflow risks, but often taught in fundamental courses).
4. **puts()**: Writes a string to standard output and automatically appends a newline character (`\n`).

Worked Example:

Character and String Manipulation using Unformatted IO Write a C code fragment that takes a user's name as input using `gets()` and displays it using `puts()`, followed by reading a secret single-character confirmation code using `getch()` and displaying it using `putch()`.

Step 1: Declare character array and variable `char name[50]; char code;`
Step 2: String IO `puts("Enter your name:");` // Outputs string and adds newline `gets(name);` // Reads until newline `puts("Hello:"); puts(name);` // Outputs string and adds newline
Step 3: Character IO `puts("Enter secret confirmation code:"); code = getch();` // Reads one char, no screen echo `puts("Code accepted:"); putch(code);` // Prints the secret code manually

CHAPTER 3

Decision Control and Looping

This chapter focuses on the computational implementation of decision-making and iterative processes in programming. We will cover various control structures and their algorithmic applications.

3.1 Decision Control Statements

Methodology:

The if Statement The `if` statement is used to execute a block of code only if a specified condition evaluates to true.

1. Evaluate the boolean test expression.
2. If the condition is true (non-zero), execute the statement block inside the `if` construct.
3. If the condition is false (zero), bypass the statement block.

Syntax:

```
if (condition) {
    // Statements to execute if condition is true
}
```

Worked Example:

Checking for Positive Number Write a program snippet to check if a given number is positive and increment it.

Solution:

```
int num = 5;
if (num > 0) {
    num = num + 1;
    printf("The number is positive. Incremented value: %d\n", num);
}
```

Step-by-step Execution:

1. Initialize `num = 5`.
2. Evaluate condition `num > 0` $\Rightarrow$ `5 > 0` (True).
3. Execute block: `num = 5 + 1 = 6`.
4. Output the result.

Methodology:

The if else Statement The `if-else` statement executes one block of code if the condition is true and another block if it is false.

1. Evaluate the boolean condition.
2. If true, execute the `if` block and skip the `else` block.
3. If false, skip the `if` block and execute the `else` block.

Syntax:

```
if (condition) {
    // True block
} else {
    // False block
}
```

Worked Example:

Checking Even or Odd Determine if an integer is even or odd using computational logic.
Solution:

```
int num = 14;
if (num % 2 == 0) {
    printf("Even Number\n");
} else {
    printf("Odd Number\n");
}
```

Execution steps:

1. `num = 14`.
2. Evaluate `14 % 2 == 0` $\Rightarrow$ `0 == 0` (True).
3. The condition is true; the `if` block executes printing "Even Number".
4. The `else` block is skipped.

Methodology:

The if else if Ladder Used to evaluate multiple conditions sequentially until one is found to be true.

1. Evaluate `condition1`. If true, execute its block and exit the ladder.
2. If false, evaluate `condition2`. If true, execute its block and exit.
3. Repeat until a true condition is found or the final `else` block is reached.

Syntax:

```
if (condition1) {
    // Block 1
} else if (condition2) {
    // Block 2
} else {
    // Default block
}
```

Worked Example:

Determining Grade from Marks Calculate the grade based on marks: $\geq 80 \rightarrow$ A, $\geq 60 \rightarrow$ B, $\geq 40 \rightarrow$ C, else Fail.
Solution:

```
int marks = 65;
if (marks >= 80) {
    printf("Grade A\n");
} else if (marks >= 60) {
    printf("Grade B\n");
} else if (marks >= 40) {
    printf("Grade C\n");
} else {
    printf("Fail\n");
}
```

Execution steps:

1. `marks = 65`.
2. `marks >= 80` $\Rightarrow$ False.
3. `marks >= 60` $\Rightarrow$ True. Execute Block 2 ("Grade B").
4. Exit the ladder.

Methodology:

Nested if else Statements Placing an `if-else` statement inside another `if` or `else` block for hierarchical conditions.

1. Evaluate the outer condition.
2. If true, enter the block and evaluate the inner condition.
3. Depending on the inner condition execute the inner true or false blocks.

Worked Example:

Finding the Greatest of Three Numbers Find the maximum among three numbers a, b, and c.
Solution:

```
int a = 10, b = 25, c = 15;
if (a > b) {
    if (a > c) {
        printf("a is greatest\n");
    } else {
        printf("c is greatest\n");
    }
} else {
    if (b > c) {
        printf("b is greatest\n");
    } else {
        printf("c is greatest\n");
    }
}
```

Execution steps:

1. Outer condition `a > b` ($10 > 25$) is False. Jump to the outer `else`.
2. Inner condition `b > c` ($25 > 15$) is True.
3. Print "b is greatest".

Methodology:

The switch case Statement A multi-way branch statement that provides an efficient way to dispatch execution to different parts of code based on the value of an expression.

1. Evaluate the control expression (must result in an integer or character).
2. Compare the result against the constant values in the `case` labels.
3. If a match is found execute the corresponding block of statements.
4. Use `break` to exit the `switch` block; otherwise execution falls through to subsequent cases.
5. If no match is found execute the `default` block (if provided).

Worked Example:

Simple Calculator using switch Perform basic arithmetic operations based on a given operator.
Solution:

```
char op = '+';
int num1 = 10, num2 = 5, result;

switch (op) {
    case '+':
        result = num1 + num2;
        printf("Result: %d\n", result);
        break;
    case '-':
        result = num1 - num2;
        printf("Result: %d\n", result);
        break;
    case '*':
        result = num1 * num2;
        printf("Result: %d\n", result);
        break;
    case '/':
        result = num1 / num2;
        printf("Result: %d\n", result);
        break;
    default:
        printf("Invalid Operator\n");
}
```

Execution steps:

1. `op = '+'`.
2. The expression `op` evaluates to `'+'`.
3. Matches `case '+'`. `result = 10 + 5 = 15`.
4. The `break` statement terminates the `switch` block.

3.2 Looping Statements

Methodology:

The while Loop An entry-controlled loop that repeatedly executes a block of statements as long as a given condition remains true.

1. Evaluate the boolean condition.
2. If true execute the loop body.
3. Repeat step 1 until the condition evaluates to false.

Syntax:

```
while (condition) {
    // Loop body
    // Update statement
}
```

Worked Example:

Sum of First N Natural Numbers Calculate the sum of natural numbers from 1 to N using a while loop.
Solution:

```
int n = 5;
int sum = 0;
int i = 1;

while (i <= n) {
    sum = sum + i;
    i++;
}
printf("Sum: %d\n", sum);
```

Step-by-step Execution:

Iteration	Condition ($i \leq 5$)	Sum ($sum + i$)	Next i
1	$1 \leq 5$ (True)	$0 + 1 = 1$	2
2	$2 \leq 5$ (True)	$1 + 2 = 3$	3
3	$3 \leq 5$ (True)	$3 + 3 = 6$	4
4	$4 \leq 5$ (True)	$6 + 4 = 10$	5
5	$5 \leq 5$ (True)	$10 + 5 = 15$	6
6	$6 \leq 5$ (False)	-	-

Methodology:

The do while Loop An exit-controlled loop that executes its body at least once before testing the condition.

1. Execute the loop body.
2. Evaluate the boolean condition at the end of the loop.
3. If true repeat step 1.
4. If false terminate the loop.

Syntax:

```
do {
```

```
    // Loop body
    // Update statement
} while (condition);
```

Worked Example:

Counting Digits of an Integer Count the number of digits in an integer using a do-while loop.
Solution:

```
int num = 3452;
int count = 0;

do {
    count++;
    num = num / 10;
} while (num != 0);
printf("Digits: %d\n", count);
```

Step-by-step Execution:

1. Initial: `num = 3452, count = 0`.
2. Iteration 1: `count = 1, num = 345`. Condition `345 != 0` (True).
3. Iteration 2: `count = 2, num = 34`. Condition `34 != 0` (True).
4. Iteration 3: `count = 3, num = 3`. Condition `3 != 0` (True).
5. Iteration 4: `count = 4, num = 0`. Condition `0 != 0` (False). Loop terminates. Result is 4.

Methodology:

The for Loop An entry-controlled loop with built-in initialization condition checking and update statements.

1. Execute the initialization expression once.
2. Evaluate the condition expression.
3. If true execute the loop body.
4. Execute the update expression.
5. Repeat from step 2 until the condition is false.

Syntax:

```
for (initialization; condition; update) {
    // Loop body
}
```

Worked Example:

Generating Factorial of a Number Compute $N! = N \times (N-1) \times \cdots \times 1$.
Solution:

```
int n = 4;
int fact = 1;

for (int i = 1; i <= n; i++) {
    fact = fact * i;
```

```
}
printf("Factorial: %d\n", fact);
```

Step-by-step Execution:

Iteration	Init/Update i	Condition ($i \leq 4$)	fact	Next i
1	$i = 1$	$1 \leq 4$ (True)	$1 \times 1 = 1$	2
2	$i = 2$	$2 \leq 4$ (True)	$1 \times 2 = 2$	3
3	$i = 3$	$3 \leq 4$ (True)	$2 \times 3 = 6$	4
4	$i = 4$	$4 \leq 4$ (True)	$6 \times 4 = 24$	5
5	$i = 5$	$5 \leq 4$ (False)	-	-

Methodology:

Nested for Loops Placing one `for` loop inside another. The inner loop fully completes its execution for each single iteration of the outer loop.

1. Start outer loop initialization.
2. Check outer condition.
3. If true enter the outer loop body and start inner loop initialization.
4. Execute inner loop until its condition becomes false.
5. Update outer loop and repeat from step 2.

Worked Example:

Printing a Number Pattern Print the following triangular pattern of numbers for $N = 3$:

```
1
1 2
1 2 3
```

Solution:

```
int n = 3;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        printf("%d ", j);
    }
    printf("\n");
}
```

Step-by-step Execution:

- **Outer Loop ($i = 1$):** Inner loop runs for $j = 1$ to 1. Prints `1`. Prints newline.
- **Outer Loop ($i = 2$):** Inner loop runs for $j = 1$ to 2. Prints `1 2`. Prints newline.
- **Outer Loop ($i = 3$):** Inner loop runs for $j = 1$ to 3. Prints `1 2 3`. Prints newline.

3.3 Jump Statements

Methodology:

The break Statement Used to immediately exit from a loop or a `switch` block.

1. Encounter `break` inside a loop or `switch`.
2. Immediately terminate the innermost enclosing loop or `switch`.
3. Transfer control to the statement following the terminated block.

Worked Example:

Finding the First Multiple of 7 Iterate from 1 to 20 and break when the first multiple of 7 is found.
Solution:

```
for (int i = 1; i <= 20; i++) {
    if (i % 7 == 0) {
        printf("First multiple of 7 is: %d\n", i);
        break;
    }
}
```

Execution: Loop checks $1, 2, \ldots, 6$. When $i = 7$, condition $7\%7 == 0$ is True. It prints the message, executes `break`, and the loop ends immediately without checking numbers 8 through 20.

Methodology:

The continue Statement Used to skip the remaining statements in the current iteration of a loop and proceed to the next iteration.

1. Encounter `continue` inside a loop.
2. Skip the rest of the loop body for the current iteration.
3. Jump to the update expression (in `for` loops) or condition evaluation (in `while`/`do-while` loops).

Worked Example:

Printing Odd Numbers Print odd numbers between 1 and 5 using a `continue` statement.
Solution:

```
for (int i = 1; i <= 5; i++) {
    if (i % 2 == 0) {
        continue;
    }
    printf("%d ", i);
}
```

Execution:

- $i = 1$: `1 % 2 != 0`. Prints 1.
- $i = 2$: `2 % 2 == 0`. `continue` executes. Skip print.
- $i = 3$: `3 % 2 != 0`. Prints 3.
- $i = 4$: `4 % 2 == 0`. `continue` executes. Skip print.
- $i = 5$: `5 % 2 != 0`. Prints 5.

Methodology:

The goto Statement An unconditional jump statement that transfers control to a labeled statement within the same function.

1. Define a label identifier followed by a colon (e.g. `labelName:`).
2. Use the `goto` keyword followed by the label name (e.g. `goto labelName;`).
3. Execution immediately jumps to the line marked by the label.

Note: Overuse of goto is discouraged as it makes code difficult to trace.

Worked Example:

Simple Loop using goto Print numbers from 1 to 3 using `goto` instead of a standard loop.
Solution:

```
int i = 1;
startLoop:
if (i <= 3) {
    printf("%d ", i);
    i++;
    goto startLoop;
}
```

Execution:

1. Initialize $i = 1$.
2. At `startLoop`, check $1 \leq 3$. True. Print 1, update $i = 2$.
3. Execute `goto startLoop`.
4. Check $2 \leq 3$. True. Print 2, update $i = 3$.
5. Execute `goto startLoop`.
6. Check $3 \leq 3$. True. Print 3, update $i = 4$.
7. Execute `goto startLoop`.
8. Check $4 \leq 3$. False. Bypass the block.

CHAPTER 4

Array and Pointer

4.1 One-Dimensional Array

A one-dimensional array is a linear collection of elements of the same data type stored in contiguous memory locations. It is used to store multiple values of similar types under a single name.

Methodology:

Operations on One Dimensional Arrays

1. **Declaration**: Specify the data type, the array name, and the size enclosed in square brackets.
   - Integer array: `int arr[5];`
   - Float array: `float marks[10];`
   - Character array: `char name[20];`
2. **Initialization**: Assign values to the array elements at the time of declaration.
   - `int arr[5] = {10, 20, 30, 40, 50};`
   - `char vowels[] = {'a', 'e', 'i', 'o', 'u'};` or `char str[] = "hello";`
3. **Storing in Memory**: Array elements are stored sequentially. The index starts from 0 to size-1. If `arr[0]` is at memory address 1000 and the type is `int` (typically 4 bytes), the next element `arr[1]` is stored at 1004, `arr[2]` at 1008, and so on.
4. **Displaying**: Use a loop from index 0 to size-1 to access and print each element using standard output functions.

Worked Example:

Storing and Displaying Integer and Float Arrays **Problem Statement**: Write an algorithm or program snippet to declare, initialize, and display an integer array of 5 elements and a float array of 3 elements.
Step 1: Declaration and Initialization Declare an integer array `arr` with 5 values and a float array `prices` with 3 values.

```
int arr[5] = {10, 20, 30, 40, 50};
float prices[3] = {15.50, 25.00, 9.99};
```

Step 2: Displaying the Integer Array Loop from `i = 0` to 4 to access each element.

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

Step 3: Displaying the Float Array Loop from `i = 0` to 2 to access each element.

```
for(int i = 0; i < 3; i++) {
    printf("prices[%d] = %.2f\n", i, prices[i]);
}
```

Worked Example:

Memory Mapping of Character Arrays **Problem Statement**: Declare a character array to store the word "HELLO" and display each character along with its hypothetical memory address. Assume the base address is 2000.

Step 1: Initialization

```
char word[] = "HELLO";
```

Note: A null character `'\0'` is automatically appended at the end of double-quoted strings, making the total size 6 bytes.

Step 2: Memory Mapping and Display Characters occupy 1 byte each. The memory allocation proceeds sequentially:

Index	0	1	2	3	4	5
Character	'H'	'E'	'L'	'L'	'O'	'\0'
Address	2000	2001	2002	2003	2004	2005

4.2 Two-Dimensional Array

A two-dimensional array is an array of arrays. It is commonly used to represent grids, tables, and matrices.

Methodology:

Handling Two Dimensional Integer Arrays

1. **Declaration**: Specify the data type, array name, number of rows, and number of columns.
   - `int matrix[3][4];` (Declares a grid with 3 rows and 4 columns)
2. **Initialization**: Use nested curly braces to group row elements together.
   - `int mat[2][3] = {{1, 2, 3}, {4, 5, 6}};`
3. **Storing in Memory**: Elements are typically stored in "row-major order" in C. This means all elements of the first row are stored in sequential memory locations, followed directly by all elements of the second row, and so forth.
4. **Displaying**: Use nested loops. The outer loop iterates over the rows, and the inner loop iterates over the columns.

Worked Example:

Matrix Initialization and Display **Problem Statement**: Create a 2×3 integer matrix, initialize it with predefined values, and display it in a tabular grid format.

Step 1: Declaration and Initialization

```
int matrix[2][3] = {
    {10, 20, 30},
    {40, 50, 60}
};
```

Step 2: Displaying the Matrix Use an outer loop `i` for rows (0 to 1) and an inner loop `j` for columns (0 to 2).

```
for(int i = 0; i < 2; i++) {
    for(int j = 0; j < 3; j++) {
        printf("%d ", matrix[i][j]);
    }
```

```
    printf("\n"); // Move to the next line after each row
}
```

Step 3: Output Generation The printed output will appear as:

```
10 20 30
40 50 60
```

4.3 Pointer Concept and Variables

A pointer is a special variable that stores the memory address of another variable rather than storing a direct data value. Pointers are powerful tools for dynamic memory allocation and efficient array manipulation.

Methodology:

Pointer Declaration and Initialization Steps

1. **Declaration**: Use the asterisk (`*`) operator before the pointer variable name. The data type of the pointer must match the data type of the variable whose address it will store.
   - `int *ptr;` (Pointer to an integer)
   - `char *cptr;` (Pointer to a character)
2. **Initialization**: Use the address-of operator (`&`) to assign the memory address of an existing variable to the pointer.
   - `int num = 50;`
   - `int *ptr = &num;`
3. **Dereferencing**: Use the `*` operator on the pointer variable to access or modify the value stored at the memory address it points to.
   - `printf("%d", *ptr);` accesses the value of `num` and outputs 50.

Worked Example:

Basic Pointer Address and Value Operations **Problem Statement**: Declare an integer variable, assign it a value, use a pointer to access its value, and trace both the address and the dereferenced value.
Step 1: Variable Declaration

```
int x = 100;
```

Assume the operating system allocates memory address `4000` for variable `x`.
Step 2: Pointer Declaration and Initialization

```
int *ptr;
ptr = &x;
```

Now, the pointer `ptr` holds the address `4000`.
Step 3: Accessing Value and Address By applying different operators, we can observe the relationships:

- Direct value of `x`: `x` $\rightarrow$ `100`
- Address of `x`: `&x` $\rightarrow$ `4000`
- Value stored inside `ptr`: `ptr` $\rightarrow$ `4000`
- Dereferenced value: `*ptr` $\rightarrow$ `100`

Worked Example:

Traversing Arrays using Pointer Arithmetic **Problem Statement**: Use a pointer to traverse and display the elements of a one-dimensional integer array `arr[3] = {5, 10, 15}`.

Step 1: Initialization

```
int arr[3] = {5, 10, 15};
int *ptr = arr; // Equivalent to ptr = &arr[0]
```

Assume the base address of `arr` is `5000` and the size of an integer is 4 bytes.

Step 2: Traversal using Pointer Operations

- **Access Element 1**: `*ptr` evaluates to `5`. The address in `ptr` is `5000`.
- **Increment Pointer**: `ptr++`. The pointer moves to the next integer block. New address is $5000 + 4 = 5004$.
- **Access Element 2**: `*ptr` evaluates to `10`. The address in `ptr` is `5004`.
- **Increment Pointer**: `ptr++`. The pointer moves to the next integer block. New address is $5004 + 4 = 5008$.
- **Access Element 3**: `*ptr` evaluates to `15`. The address in `ptr` is `5008`.

CHAPTER 5

Introduction to Strings, Structures, Functions, and File Operations

5.1 Strings

Methodology:

String Declaration Initialization and Display

1. **Declaration**: A string is a 1-D array of characters terminated by a null character `'\0'`.
   `char str_name[size];`
2. **Initialization**: Strings can be initialized at the time of declaration.
   `char str[] = "Hello";` or
   `char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};`
3. **Display**: Use `printf()` with the `%s` format specifier.
   `printf("%s", str);`

Methodology:

String Handling Functions Include the `<string.h>` header file to use these standard string library functions:

1. **`strlen(s1)`**: Returns the length of string `s1` (excluding the null character).
2. **`strcpy(s1, s2)`**: Copies the content of string `s2` into string `s1`.
3. **`strcat(s1, s2)`**: Concatenates (appends) string `s2` at the end of string `s1`.
4. **`strcmp(s1, s2)`**: Compares string `s1` and `s2` character by character. Returns 0 if they are identical.

Worked Example:

String Length and Concatenation **Problem**: Write a program to find the length of a string and concatenate it with another string.
Solution:

1. Include `<stdio.h>` and `<string.h>`.
2. Declare two strings `s1` and `s2`.
3. Initialize `s1` to "GTU " and `s2` to "Diploma".
4. Print the length of `s1` using `strlen()`.
5. Concatenate `s2` to `s1` using `strcat()` and display the result.

Code Implementation:

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

int main() {
    char s1[20] = "GTU ";
    char s2[] = "Diploma";

    printf("Length of s1: %lu\n", strlen(s1));
    strcat(s1, s2);
    printf("Concatenated String: %s\n", s1);

    return 0;
}
```

Worked Example:

String Copy and Compare **Problem**: Write a program to copy one string to another and compare two strings.
Solution:

1. Declare strings `src`, `dest`, and `test`.
2. Copy `src` to `dest` using `strcpy()`.
3. Compare `dest` with `test` using `strcmp()`.

Code Implementation:

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

int main() {
    char src[] = "Programming";
    char dest[20];
    char test[] = "Programming";

    // Copying string
    strcpy(dest, src);
    printf("Copied String: %s\n", dest);

    // Comparing strings
    if (strcmp(dest, test) == 0) {
        printf("Strings are equal.\n");
    } else {
        printf("Strings are not equal.\n");
    }

    return 0;
}
```

5.2 Structures

Methodology:

Declaring Initializing and Accessing Structures A structure is a user-defined data type that groups related variables of different data types under a single name.

1. **Declaring**: Use the `struct` keyword.

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

2. **Initialization**: Assign values at the time of declaring a structure variable.
   `struct Student s1 = {101, "John", 85.5};`
3. **Accessing**: Use the dot (`.`) operator to access structure members.
   `s1.id = 102;`

Worked Example:

Storing and Displaying Structure Data **Problem**: Create a structure to store the ID, name, and marks of a student and display them.
Solution:

1. Define the `Student` structure globally.
2. In `main()`, initialize a structure variable with data.
3. Print the individual members using the dot operator.

Code Implementation:

```
#include <stdio.h>

struct Student {
    int id;
    char name[20];
    float marks;
};

int main() {
    struct Student s1 = {1, "Alice", 92.5};

    printf("Student ID: %d\n", s1.id);
    printf("Student Name: %s\n", s1.name);
    printf("Marks: %.2f\n", s1.marks);

    return 0;
}
```

5.3 Functions

Functions are reusable blocks of code. They are broadly categorized into User-Defined Functions (created by the programmer) and Library Functions (pre-defined in C standard libraries).

Methodology:

User-Defined Functions

1. **Declaration (Prototype)**: Specifies the return type, function name, and parameters.
   `int add(int a, int b);`
2. **Definition**: Contains the actual logic and code inside the function block.
   `int add(int a, int b) { return a + b; }`
3. **Calling**: Invokes the function from `main()` or another function.
   `int sum = add(5, 10);`

Worked Example:

Simple User-Defined Function **Problem**: Write a program with a user-defined function to multiply two numbers.
Solution:

1. Declare the `multiply` function.
2. Call the function from `main()` by passing two integer values.
3. Print the returned result.

Code Implementation:

```
#include <stdio.h>

// User-defined function
int multiply(int x, int y) {
    return x * y;
}

int main() {
    int result = multiply(4, 5); // Calling the function
    printf("Product: %d\n", result);
    return 0;
}
```

5.3.1 Library Functions

Methodology:

Math and Character Library Functions Include `<math.h>` for mathematical operations and `<ctype.h>` for character operations.

1. **Math Functions** (`<math.h>`):
   - `sqrt(x)`: Computes the square root of `x`.
   - `pow(x, y)`: Computes `x` raised to the power `y`.
   - `exp(x)`: Computes the exponential value e^x.
   - `fmod(x, y)` (mod): Computes the floating-point remainder of `x/y`.

- `round(x)`: Rounds `x` to the nearest integer.
- *Note: `sum()` is generally not a built-in standard C function, but you compute sums simply via addition (`a + b`).*

2. **Character Functions** (`<ctype.h>`):
   - `islower(c)`: Returns non-zero if the character is lowercase.
   - `isupper(c)`: Returns non-zero if the character is uppercase.
   - `isdigit(c)`: Returns non-zero if the character is a digit (0-9).
   - `tolower(c)`: Converts an uppercase character to lowercase.
   - `toupper(c)`: Converts a lowercase character to uppercase.

Worked Example:

Using Math and Character Functions **Problem**: Demonstrate the usage of standard math and character functions in C.
Solution:

1. Include `<math.h>` and `<ctype.h>`.
2. Apply `sqrt()`, `pow()`, and `round()` on floating-point numbers.
3. Use `toupper()` to convert a lowercase letter, and `isdigit()` to test a character.

Code Implementation:

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

int main() {
    double num = 16.0;
    double val = 3.6;
    char ch = 'a';
    char digitChar = '7';

    // Math functions
    printf("Square root of %.1f is %.1f\n", num, sqrt(num));
    printf("2 raised to 3 is %.1f\n", pow(2.0, 3.0));
    printf("Rounding %.1f gives %.1f\n", val, round(val));

    // Character functions
    printf("Uppercase of '%c' is '%c'\n", ch, toupper(ch));

    if (isdigit(digitChar)) {
        printf("'%c' is a digit.\n", digitChar);
    }

    return 0;
}
```

5.4 File Operations

Methodology:

File Opening and Closing File operations allow programs to read from and write to external files on the disk using a file pointer.

1. **File Pointer**: Declare a pointer of type `FILE`.
   `FILE *fp;`
2. **`fopen()`**: Opens a file in a specific mode.
   - `"w"` (Write mode): Opens a file for writing. If the file already exists, its content is erased. If not, it is created.
   - `"r"` (Read mode): Opens an existing file for reading. Returns `NULL` if the file does not exist.

   `fp = fopen("data.txt", "w");`
3. **`fclose()`**: Closes the opened file and flushes buffers, ensuring data is safely saved.
   `fclose(fp);`

Worked Example:

Writing and Reading a File **Problem**: Write a program to open a file in write mode, write a string to it, close it, then open it in read mode and display its contents.
Solution:

1. Use `fopen()` with mode `"w"` to create `test.txt`.
2. Use `fprintf()` to write text to the file. Close the file.
3. Use `fopen()` with mode `"r"` to read the file.
4. Use `fgets()` in a loop to read contents line by line and print them to the console.

Code Implementation:

```
#include <stdio.h>

int main() {
    FILE *fp;
    char buffer[100];

    // Open in Write Mode
    fp = fopen("test.txt", "w");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    fprintf(fp, "Hello GTU Students!\n");
    fprintf(fp, "Welcome to File Operations.\n");
    fclose(fp); // Closing file

    // Open in Read Mode
    fp = fopen("test.txt", "r");
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
```

```
    printf("--- File Content ---\n");
    while (fgets(buffer, sizeof(buffer), fp) != NULL) {
        printf("%s", buffer);
    }
    fclose(fp); // Closing file

    return 0;
}
```

Appendix: Key Formulae

This appendix provides a quick reference to the key mathematical relationships and programming operators discussed in this book.

Unit 2: Operator Expressions and I/O Functions

Compound Assignment

Compound assignment operators provide a shorter syntax for updating a variable by performing an operation on it. For example, the addition assignment operator `+=` is defined as:

$$A \mathrel{+}= B \quad \equiv \quad A = A + B$$

This principle applies similarly to other operators such as `-=`, `*=`, `/=`, and `%=`.

Relational Operators

Relational operators evaluate a condition and return a boolean result (represented in C as `1` for true and `0` for false). The not-equal operator `!=` checks if two operands are different:

$$x \mathrel{!}= 0$$

If x is non-zero, the expression evaluates to 1 (true). If x is 0, it evaluates to 0 (false).

Unit 3: Decision Control and Looping

Factorial Function

The factorial of a non-negative integer N, denoted by $N!$, is frequently used to demonstrate looping constructs (such as `for` or `while` loops). It represents the product of all positive integers less than or equal to N:

$$N! = N \times (N-1) \times (N-2) \times \cdots \times 1$$

By definition, $0! = 1$.

Modulo Operator

The modulo operator `%` returns the remainder of an integer division. It is commonly used in conditional statements to check for divisibility (e.g., checking if a number is even or odd):

$$a \,\%\, n \equiv \text{remainder of } \frac{a}{n}$$

For example, if a is perfectly divisible by n, then $a \,\%\, n == 0$.

Unit 4: Array and Pointers

Pointer Arithmetic

When performing arithmetic on pointers, adding an integer i to a pointer does not simply add i bytes to the memory address. Instead, the pointer is advanced by i elements of its underlying data type:

$$\text{New Address} = \text{Base Address} + i \times \text{sizeof(data\_type)}$$

For instance, if a pointer `ptr` points to an integer (typically 4 bytes), the operation `ptr + 1` advances the memory address by 4 bytes.