Pc (4331105) - Problem Solver

Antigravity AI

June 4, 2026

Pc

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.

Contents

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.

CHAPTER 1
Basics of C

1.1 Structure and Execution of a C Program

Methodology:

Writing a Basic C Program To write and execute a C program successfully follow these numbered steps:

1. **Include Header Files**: Use `#include <stdio.h>` to include standard input/output library functions.
2. **Define Main Function**: Every C program begins execution from the `int main()` function.
3. **Declare Variables**: Define variables with appropriate data types (`int float char`) at the beginning of a block.
4. **Write Executable Statements**: Write the core logic sequentially. End every statement with a semicolon (`;`).
5. **Return Value**: End the `main` function with `return 0;` to indicate successful execution to the operating system.

Worked Example:

Simple Interest Calculation **Problem Statement:** Write a C program to calculate the simple interest for a given principal amount rate of interest and time period.
Step-by-step Solution:

1. **Identify Variables**: We need `p` (Principal) `r` (Rate) `t` (Time) and `si` (Simple Interest). Since interest can be fractional use `float`.
2. **Construct the Logic**: The standard formula is si $= (p \times r \times t)/100$.
3. **Write the Code**:

```
#include <stdio.h>
int main() {
    float p = 1000.0;
    float r = 5.0;
    float t = 2.0;
    float si;

    si = (p * r * t) / 100.0;
    printf("Simple Interest = %f\n", si);

    return 0;
}
```

4. **Execution Result**: The output will be `Simple Interest = 100.000000`.

1.2 Input and Output Operations

Methodology:

Formatted Input and Output C uses `printf` and `scanf` for standard terminal I/O. Follow these steps:

1. **Identify Format Specifiers**: Use `%d` for integers `%f` for floats and `%c` for characters.
2. **Using scanf (Input)**: Provide the format string followed by the memory address of the variables using the ampersand (`&`) operator. Example: `scanf("%d" &age);`
3. **Using printf (Output)**: Provide the format string followed by the variable values directly (no ampersand). Example: `printf("Age is %d" age);`

Worked Example:

Reading Mixed Data Types **Problem Statement:** Write a program snippet to read an integer Roll Number and a floating-point Percentage from the user and print them.
Step-by-step Solution:

1. **Variable Declaration**:

```
int rollNo;
float percentage;
```

2. **Accepting Input**: Request input and use `scanf` with matching format specifiers.

```
printf("Enter Roll Number: ");
scanf("%d", &rollNo);
printf("Enter Percentage: ");
scanf("%f", &percentage);
```

3. **Displaying Output**: Pass the variables to `printf` to display the data. Using `%.2f` restricts output to two decimal places.

```
printf("Roll Number: %d\nPercentage: %.2f\n", rollNo, percentage);
```

1.3 Expression Evaluation and Operators

Methodology:

Evaluating Arithmetic Expressions To accurately evaluate complex arithmetic expressions in C follow the strict precedence and associativity rules:

1. **Parentheses**: Evaluate expressions inside `()` first.
2. **Unary Operators**: Evaluate unary plus `+` minus `-` increment `++` decrement `-` from Right to Left.
3. **Multiplicative Operators**: Evaluate `* /` and `%` from Left to Right.
4. **Additive Operators**: Evaluate `+` and `-` from Left to Right.
5. **Assignment**: Assign the final evaluated result to the variable on the left using `=` (Right to Left).
6. **Integer Division**: Note that dividing two integers yields an integer (the fractional part is discarded). Example: `5 / 2 = 2`.

Worked Example:

Evaluating a Complex Arithmetic Expression **Problem Statement:** Trace the evaluation of the expression `a * b / c + d - e % f` where `a=4 b=3 c=2 d=5 e=7 f=3`.
Step-by-step Solution:

1. **Substitute values**: `4 * 3 / 2 + 5 - 7 % 3`
2. **Identify highest precedence operators**: The operators `* /` and `%` share the highest precedence here and are evaluated Left to Right.
3. **Evaluate 4 * 3**: `12 / 2 + 5 - 7 % 3`
4. **Evaluate 12 / 2**: `6 + 5 - 7 % 3`
5. **Evaluate 7 % 3**: `7` divided by `3` leaves a remainder of `1`. Expression becomes `6 + 5 - 1`.
6. **Identify remaining operators**: `+` and `-` are left. Evaluate Left to Right.
7. **Evaluate 6 + 5**: `11 - 1`
8. **Evaluate 11 - 1**: `10`

Final Answer: The expression evaluates to 10.

1.4 Type Conversion and Casting

Methodology:

Applying Explicit Type Casting Type casting converts a value from one data type to another bypassing implicit conversion logic to avoid data loss or enforce floating-point division.

1. **Identify the operation**: Check if the operation involves operands of the same type that might cause unwanted truncation (e.g. integer division on sums).
2. **Apply explicit cast syntax**: Prefix the variable or expression with the target data type enclosed in parentheses: `(type) expression`.
3. **Evaluate**: The value is temporarily converted to the specified `type` for the evaluation of that specific arithmetic operation.

Worked Example:

Calculating Accurate Averages **Problem Statement:** A student has integer marks 85 and 90. Calculate the exact float average.
Step-by-step Solution:

1. **Declare variables**:

```
int m1 = 85;
int m2 = 90;
float avg;
```

2. **Naive Calculation (Incorrect)**: `avg = (m1 + m2) / 2;`
   Here `(85 + 90) / 2` gives `175 / 2`. Because both 175 and 2 are integers integer division occurs resulting in `87`. The value `87.0` is assigned causing precision loss.
3. **Using Explicit Casting (Correct)**:

```
avg = (float)(m1 + m2) / 2;
```

Here `(m1 + m2)` is cast to `float` resulting in `175.0`. Then `175.0 / 2` triggers floating-point division evaluating to `87.5`.

4. **Execution Result**: The average calculated and stored in `avg` is accurately `87.5`.

1.5 Logical and Relational Operators

Methodology:

Evaluating Conditional Logic C uses relational operators to compare values and logical operators to combine conditions. They return `1` for True and `0` for False.

1. **Relational Operators**: Evaluate `< <= > >=` first followed by equality operators `==` and `!=`.
2. **Logical AND (&&)**: Returns True (1) only if both operands are non-zero. Short-circuits (skips right operand) if the left operand evaluates to False (0).
3. **Logical OR (||)**: Returns True (1) if at least one operand is non-zero. Short-circuits if the left operand evaluates to True (non-zero).
4. **Logical NOT (!)**: Reverses truth value. `!0` becomes `1` and any non-zero value negated becomes `0`.

Worked Example:

Tracing Logical Expressions **Problem Statement:** Given integer variables `x = 5 y = 10 z = 15`. Evaluate the truth value of the expression `(x < y) && (y == z) || !(x > 0)`.
Step-by-step Solution:

1. **Substitute values**: `(5 < 10) && (10 == 15) || !(5 > 0)`
2. **Evaluate Relational Operators (inside parentheses)**:
   - `5 < 10` is True (`1`)
   - `10 == 15` is False (`0`)
   - `5 > 0` is True (`1`)

   The expression simplifies to: `1 && 0 || !1`
3. **Evaluate Logical NOT (!)**: Highest precedence among logical operators. `!1` evaluates to `0`. Expression becomes: `1 && 0 || 0`
4. **Evaluate Logical AND (&&)**: Higher precedence than OR. `1 && 0` evaluates to `0`. Expression becomes: `0 || 0`
5. **Evaluate Logical OR (||)**: `0 || 0` evaluates to `0`.

Final Answer: The expression evaluates to 0 (False).

1.6 Ternary Conditional Operator

Methodology:

Using the Conditional Operator The conditional operator `?:` is a shorthand for simple if-else logic.

1. **Identify components**: The syntax requires three operands: `Condition ? Expression1 : Expression2`.

2. **Evaluate condition**: Check if the boolean `Condition` is non-zero (True) or zero (False).
3. **Return result**: If True evaluate and return `Expression1`. If False evaluate and return `Expression2`.

Worked Example:

Finding Maximum of Two Numbers **Problem Statement:** Write a single statement using the ternary operator to find the maximum of variables `a = 15` and `b = 20` and assign it to `max`.
Step-by-step Solution:

1. **Construct the Condition**: Compare `a` and `b` using `a > b`.
2. **Determine Expressions**: If `a > b` is true return `a`. If false return `b`.
3. **Write the Statement**: `max = (a > b) ? a : b;`
4. **Execution Trace**: Since `15 > 20` is False (0) the condition fails. The operator skips `a` and evaluates the second expression `b`. The value `20` is assigned to `max`.

CHAPTER 2

Operators Expressions and Input/Output Functions

2.1 Operators in C

Methodology:

Arithmetic Operators Arithmetic operators are used to perform mathematical calculations. C provides unary and binary arithmetic operators.

1. **Unary Operators**: Require one operand. E.g., `+` (unary plus), `-` (unary minus).
2. **Binary Operators**: Require two operands.
   - `+`: Addition
   - `-`: Subtraction
   - `*`: Multiplication
   - `/`: Division (Truncates decimal part for integers)
   - `%`: Modulo (Returns remainder of integer division. Cannot be used with floating-point types)

Worked Example:

Evaluate Arithmetic Expressions Evaluate the following expressions where `int a = 15; int b = 4; float c = 4.0;`. 1. `a / b` 2. `a % b` 3. `a / c`
Solution:

1. `a / b` $\rightarrow 15/4$. Since both are integers, integer division is performed. The fractional part is discarded. Result: 3.
2. `a % b` $\rightarrow 15\%4$. Modulo returns the remainder. $15 = 4 \times 3 + 3$. Result: 3.
3. `a / c` $\rightarrow 15/4.0$. Implicit type conversion occurs. `a` is converted to float. 15.0/4.0. Result: 3.75.

Methodology:

Relational and Logical Operators Relational operators compare two values. Logical operators combine multiple relational expressions. Both return 1 for True and 0 for False.

1. **Relational Operators**: `==` (Equal to), `!=` (Not equal to), `>` (Greater than), `<` (Less than), `>=` (Greater than or equal to), `<=` (Less than or equal to).
2. **Logical Operators**:
   - `&&` (Logical AND): Returns 1 if *both* operands are non-zero (true). Evaluates left-to-right with short-circuiting.

- `||` (Logical OR): Returns 1 if *at least one* operand is non-zero (true). Evaluates left-to-right with short-circuiting.
- `!` (Logical NOT): Unary operator. Returns 1 if operand is zero, and 0 if operand is non-zero.

Worked Example:

Evaluate Relational and Logical Expressions Evaluate `(x > y) && !(z == 5)` given `int x = 10, y = 5, z = 5`.
Solution:

1. Substitute the values: `(10 > 5) && !(5 == 5)`.
2. Evaluate relational expressions first. `10 > 5` is True (1). `5 == 5` is True (1).
3. The expression becomes: `(1) && !(1)`.
4. Evaluate Logical NOT: `!(1)` is False (0).
5. The expression becomes: `1 && 0`.
6. Evaluate Logical AND: True AND False is False (0).
7. Result: 0.

Methodology:

Assignment and Compound Assignment Operators The assignment operator `=` assigns the value of the right operand to the left operand (variable). Compound assignment operators perform an operation and an assignment in one step.

1. **Simple Assignment**: `var = expression;`
2. **Compound Assignment**: `var op= expression;` is equivalent to `var = var op (expression);`.
3. Examples: `+=`, `-=`, `*=`, `/=`, `%=`.

Worked Example:

Evaluate Compound Assignment Given `int x = 5;` and `int y = 2;`, evaluate `x *= y + 3;`.
Solution:

1. Expand the compound assignment: `x = x * (y + 3);`. Note the parentheses around the right-hand side.
2. Substitute values: `x = 5 * (2 + 3);`.
3. Evaluate parenthesis: `2 + 3 = 5`.
4. Evaluate multiplication: `x = 5 * 5 = 25`.
5. Result: `x` is updated to 25.

Methodology:

Increment and Decrement Operators These are unary operators used to increase or decrease a variable's value by 1.

1. **Increment (`++`)**: Adds 1 to the operand.
2. **Decrement (`--`)**: Subtracts 1 from the operand.

3. **Prefix Form (`++x` or `-x`)**: The value is modified *first*, and then the new value is used in the expression.

4. **Postfix Form (`x++` or `x-`)**: The current value is used in the expression *first*, and then the value is modified.

Worked Example:

Evaluate Prefix and Postfix Operations Trace the values of `a`, `b`, and `c` in the following sequence: `int a = 5; int b = ++a; int c = b-;`
Solution:

1. `int a = 5;` → Initial state: $a = 5$.

2. `int b = ++a;` → Prefix increment. `a` becomes 6 first. Then `b` is assigned the new value of `a`. State: $a = 6$, $b = 6$.

3. `int c = b-;` → Postfix decrement. `c` is assigned the current value of `b` (which is 6). Then `b` is decremented to 5. State: $a = 6$, $b = 5$, $c = 6$.

Methodology:

Conditional Operator Also known as the ternary operator, it is a shorthand for an `if-else` statement.

1. **Syntax**: `condition ? expression1 : expression2;`

2. **Evaluation**: If `condition` is true (non-zero), `expression1` is evaluated and becomes the result. If `condition` is false (zero), `expression2` is evaluated and becomes the result.

Worked Example:

Find Maximum using Conditional Operator Use the conditional operator to find the maximum of two numbers `a = 15` and `b = 20`.
Solution:

1. Formulate the condition: `a > b`.

2. Construct the expression: `max_val = (a > b) ? a : b;`

3. Substitute values: `(15 > 20) ? 15 : 20`.

4. Evaluate condition: `15 > 20` is False.

5. Select expression: Since it's False, select the second expression (after the colon).

6. Result: 20.

2.2 Operator Precedence and Associativity

Methodology:

Operator Precedence and Associativity Rules When an expression contains multiple operators, precedence determines the order of evaluation. Associativity determines the direction (Left-to-Right or Right-to-Left) for operators with the same precedence.

1. **Precedence Hierarchy** (Highest to Lowest):

   - `()` (Parentheses)

- `++ - !` (Unary, Right-to-Left)
- `* / %` (Multiplicative, Left-to-Right)
- `+ -` (Additive, Left-to-Right)
- `< <= > >=` (Relational, Left-to-Right)
- `== !=` (Equality, Left-to-Right)
- `&&` (Logical AND, Left-to-Right)
- `||` (Logical OR, Left-to-Right)
- `?:` (Conditional, Right-to-Left)
- `= += -=` etc. (Assignment, Right-to-Left)

2. **Evaluation Algorithm**:
   - Step 1: Identify sub-expressions within parentheses and evaluate them first.
   - Step 2: Identify the operator with the highest precedence.
   - Step 3: If multiple operators have the highest precedence, apply associativity rules.
   - Step 4: Evaluate that operator and replace it with its result.
   - Step 5: Repeat until a single value remains.

Worked Example:

Evaluate Complex Expression Evaluate the expression: `res = a + b * c / d - e % f;` given `a=2, b=4, c=6, d=3, e=8, f=5`.
Solution:

1. Substitute values: `res = 2 + 4 * 6 / 3 - 8 % 5;`
2. Highest precedence operators are `*`, `/`, `%`. They associate Left-to-Right.
3. Evaluate `4 * 6`: `res = 2 + 24 / 3 - 8 % 5;`
4. Evaluate `24 / 3`: `res = 2 + 8 - 8 % 5;`
5. Evaluate `8 % 5`: `res = 2 + 8 - 3;`
6. Next highest precedence operators are `+` and `-`. They associate Left-to-Right.
7. Evaluate `2 + 8`: `res = 10 - 3;`
8. Evaluate `10 - 3`: `res = 7;`
9. Finally, assignment operator `=` evaluates Right-to-Left. The value 7 is assigned to `res`.

2.3 Input and Output Functions

Methodology:

Formatted Input and Output Functions Formatted I/O functions allow reading and writing data in a specific format defined by format specifiers (e.g., `%d` for int, `%f` for float, `%c` for char).

1. **`printf()` Function**: Outputs formatted data to the standard output (screen).
   - Syntax: `printf("control string", arg1, arg2, ...);`
   - The control string contains text to print and format specifiers.

2. **`scanf()` Function**: Reads formatted data from the standard input (keyboard).
    - Syntax: `scanf("control string", &arg1, &arg2, ...);`
    - The control string contains format specifiers matching the data types to read.
    - The `&` (address-of) operator is required before variable names to store the input at their memory locations.

Worked Example:

Use Formatted IO for Simple Calculation Write a C snippet to read the principal amount, rate of interest, and time from the user, and print the simple interest.
Solution:

1. **Declare variables**: `float p, r, t, si;`
2. **Input data**: Use `scanf` to read the three float values.

```
printf("Enter Principal Rate Time: ");
scanf("%f %f %f", &p, &r, &t);
```

3. **Compute**: `si = (p * r * t) / 100.0;`
4. **Output data**: Use `printf` to display the result formatted to 2 decimal places.

```
printf("Simple Interest = %.2f\n", si);
```

Methodology:

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

1. **Character I/O**:
    - `getch()`: Reads a single character from the keyboard without echoing it to the screen. (Requires `<conio.h>`). Returns the character read.
    - `putch(ch)`: Prints a single character `ch` to the screen. (Requires `<conio.h>`).
2. **String I/O**:
    - `gets(str)`: Reads a line of text (string) from standard input until a newline is encountered, and stores it in `str`. (Note: `gets` is inherently unsafe against buffer overflows, but often taught in diploma curricula. `fgets` is the modern alternative).
    - `puts(str)`: Prints the string `str` to standard output followed by a newline character.

Worked Example:

Process Strings using Unformatted IO Write a C snippet to read a user's full name (which may contain spaces) and display a greeting.
Solution:

1. **Declare string variable**: Create a character array to hold the name. `char name[50];`
2. **Input string**: `scanf("%s", name)` stops reading at the first space. Therefore, use `gets()` to read the entire line including spaces.

```
printf("Enter your full name: ");
gets(name);
```

3. **Output string**: Use `puts()` or `printf()` to display.

```
printf("Hello, ");
puts(name);
```

*Note: **`puts(name)`** will automatically append a newline after the name.*

CHAPTER 3

Decision and Control Statements

3.1 Conditional Branching Statements

3.1.1 Simple If Statement

Methodology:

Simple If Statement The `if` statement provides a way to execute a block of code conditionally based on a single expression.

1. Evaluate the boolean condition inside the `if` statement.
2. If the condition evaluates to true (non-zero), execute the block of code inside the `if` body.
3. If the condition evaluates to false (zero), bypass the `if` body and move to the next sequential statement.

Syntax:

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

Worked Example:

Find if a given number is negative **Problem:** Write a computational procedure to read a number and print a warning if it is strictly less than zero.
Step-by-Step Solution:

1. Initialize the variable num. Let $num = -5$.
2. Check the condition $num < 0$. Here, $-5 < 0$ is true.
3. Since the condition is true, execute the block inside the `if` statement.
4. Print the warning message.

Implementation:

```
#include <stdio.h>
int main() {
    int num = -5;
    if (num < 0) {
        printf("Warning: The number is negative.\n");
    }
    return 0;
}
```

3.1.2 If-Else Statement

Methodology:

If Else Statement The `if-else` structure provides a two-way decision path.

1. Evaluate the test expression.
2. If the expression evaluates to true, execute the `if` block.
3. If the expression evaluates to false, execute the `else` block.
4. Execution resumes after the entire `if-else` construct.

Syntax:

```
if (condition) {
    // true branch
} else {
    // false branch
}
```

Worked Example:

Check whether a given year is a leap year **Problem:** Determine if a given year is a leap year using simple modulus arithmetic (simplified logic for standard 4-year cycle).
Step-by-Step Solution:

1. Input the year Y. Let $Y = 2024$.
2. Calculate the remainder of Y divided by 4.
3. If $Y \pmod 4 == 0$, the year is a leap year.
4. Otherwise, it is not a leap year.

Implementation:

```
#include <stdio.h>
int main() {
    int year = 2024;
    if (year % 4 == 0) {
        printf("%d is a Leap Year.\n", year);
    } else {
        printf("%d is not a Leap Year.\n", year);
    }
    return 0;
}
```

3.1.3 Nested If-Else Statement

Methodology:

Nested If Else Statement A nested `if-else` statement contains one or more `if` or `else` blocks within another `if` or `else` block, allowing for multi-layered decision making.

1. Evaluate the outer condition.
2. If true, enter the outer `if` block and evaluate the inner condition.
3. Depending on the inner condition, execute the corresponding inner block.

4. If the outer condition is false, skip to the outer `else` block (if present).

Syntax:

```
if (condition1) {
    if (condition2) {
        // executes when condition1 and condition2 are true
    } else {
        // executes when condition1 is true and condition2 is false
    }
} else {
    // executes when condition1 is false
}
```

Worked Example:

Find the largest of three numbers **Problem:** Determine the maximum value among three numbers A, B, and C.

Step-by-Step Solution:

1. Let $A = 15$, $B = 25$, and $C = 10$.
2. Check if $A \geq B$. Here, $15 \geq 25$ is false.
3. Move to the outer `else` block.
4. Inside the `else` block, check if $B \geq C$. Here, $25 \geq 10$ is true.
5. Conclude that B is the largest.

Implementation:

```
#include <stdio.h>
int main() {
    int a = 15, b = 25, c = 10;
    if (a >= b) {
        if (a >= c) {
            printf("%d is the largest.\n", a);
        } else {
            printf("%d is the largest.\n", c);
        }
    } else {
        if (b >= c) {
            printf("%d is the largest.\n", b);
        } else {
            printf("%d is the largest.\n", c);
        }
    }
    return 0;
}
```

3.1.4 If-Else-If Ladder

Methodology:

If Else If Ladder The `if-else-if` ladder evaluates multiple conditions sequentially.

1. Evaluate the first condition. If true, execute its block and exit the ladder.
2. If false, evaluate the second condition. If true, execute its block and exit the ladder.
3. Repeat this process for subsequent conditions.
4. If all conditions fail, execute the default `else` block at the end.

Worked Example:

Calculate Student Grade based on Percentage **Problem:** Assign a grade based on a student's percentage: ≥ 70: Distinction, ≥ 60: First Class, ≥ 50: Second Class, otherwise Pass or Fail.
Step-by-Step Solution:

1. Given percentage $P = 65$.
2. Check $P \geq 70$. ($65 \geq 70$ is False).
3. Check $P \geq 60$. ($65 \geq 60$ is True).
4. Execute the corresponding block, assigning "First Class", and exit the ladder.

Implementation:

```
#include <stdio.h>
int main() {
    int p = 65;
    if (p >= 70) {
        printf("Distinction\n");
    } else if (p >= 60) {
        printf("First Class\n");
    } else if (p >= 50) {
        printf("Second Class\n");
    } else {
        printf("Pass or Fail\n");
    }
    return 0;
}
```

3.1.5 Switch Statement

Methodology:

Switch Statement The `switch` statement tests the value of a variable against a list of case values.

1. Evaluate the integer or character expression in the `switch` parenthesis.
2. Compare the evaluated value sequentially against each `case` constant.
3. When a match is found, execute the code block associated with that `case`.
4. Use the `break` statement to exit the `switch` block; otherwise, execution falls through to the next case.
5. If no case matches, execute the optional `default` block.

Worked Example:

Perform basic arithmetic using Switch **Problem:** Create a simple calculator to perform addition, subtraction, multiplication, or division based on an operator symbol.
Step-by-Step Solution:

1. Let operator $op =' *'$ and operands $a = 10, b = 5$.
2. The `switch` expression evaluates the value of op.
3. Compare op to '+', '-', '*', '/'. It matches '*'.
4. Execute the multiplication operation: $10 \times 5 = 50$.
5. Execute `break` to exit the switch structure.

Implementation:

```
#include <stdio.h>
int main() {
    char op = '*';
    int a = 10, b = 5, result;

    switch (op) {
        case '+':
            result = a + b;
            printf("Sum = %d\n", result);
            break;
        case '-':
            result = a - b;
            printf("Difference = %d\n", result);
            break;
        case '*':
            result = a * b;
            printf("Product = %d\n", result);
            break;
        case '/':
            result = a / b;
            printf("Quotient = %d\n", result);
            break;
        default:
            printf("Invalid operator\n");
    }
    return 0;
}
```

3.2 Unconditional Branching

3.2.1 Goto Statement

Methodology:

Goto Statement The `goto` statement transfers control unconditionally to a specific labeled statement within the same function.

1. Define a label identifier followed by a colon (e.g., `start:`).
2. When `goto start;` is executed, program control immediately jumps to the line marked by `start:`.

3. Execution proceeds sequentially from the label onwards.

Worked Example:

Print numbers from 1 to 3 using Goto **Problem:** Simulate a loop using `goto` to print the numbers 1, 2, and 3.
Step-by-Step Solution:

1. Initialize variable $i = 1$.
2. Define a label `start:`.
3. Print i and increment it by 1.
4. If $i \leq 3$, execute `goto start;` to jump back to the label.
5. When $i = 4$, the condition fails and the jump is bypassed.

Implementation:

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

3.3 Looping Control Statements

3.3.1 While Loop

Methodology:

While Loop The `while` loop repeatedly executes a target statement as long as a given condition is true. It is an entry-controlled loop.

1. Evaluate the boolean condition.
2. If the condition is true, execute the loop body.
3. After executing the body, re-evaluate the condition.
4. If the condition becomes false, terminate the loop and proceed to the next statement.

Worked Example:

Sum of digits of a given number **Problem:** Calculate the sum of the digits of the number $N = 123$.
Step-by-Step Solution:

1. Initialize $N = 123$, $sum = 0$.
2. **Iteration 1:** $N > 0$ (True). Extract last digit $d = 123 \pmod{10} = 3$. Add to sum: $sum = 0 + 3 = 3$.

Update $N = 123/10 = 12$.

3. **Iteration 2:** $N > 0$ (True). $d = 12 \pmod{10} = 2$. $sum = 3 + 2 = 5$. Update $N = 12/10 = 1$.
4. **Iteration 3:** $N > 0$ (True). $d = 1 \pmod{10} = 1$. $sum = 5 + 1 = 6$. Update $N = 1/10 = 0$.
5. **Iteration 4:** $N > 0$ (False). Terminate loop. Final sum is 6.

Implementation:

```
#include <stdio.h>
int main() {
    int n = 123;
    int sum = 0, digit;

    while (n > 0) {
        digit = n % 10;
        sum = sum + digit;
        n = n / 10;
    }
    printf("Sum of digits = %d\n", sum);
    return 0;
}
```

3.3.2 Do-While Loop

Methodology:

Do While Loop The `do-while` loop is an exit-controlled loop. It executes the loop body first, and then evaluates the condition.

1. Execute the loop body unconditionally for the first time.
2. Evaluate the `while` condition at the end of the loop body.
3. If the condition is true, repeat step 1.
4. If the condition is false, exit the loop.

Worked Example:

Count the number of digits in an integer **Problem:** Count how many digits are present in the number $N = 456$.
Step-by-Step Solution:

1. Initialize $N = 456$, $count = 0$.
2. **Pass 1:** Enter loop body. $N = 456/10 = 45$. Increment $count$ to 1. Check condition $N > 0$ (True).
3. **Pass 2:** Execute body. $N = 45/10 = 4$. Increment $count$ to 2. Check condition $N > 0$ (True).
4. **Pass 3:** Execute body. $N = 4/10 = 0$. Increment $count$ to 3. Check condition $N > 0$ (False).
5. Loop terminates. The final digit count is 3.

Implementation:

```
#include <stdio.h>
int main() {
    int n = 456;
```

```
    int count = 0;

    do {
        n = n / 10;
        count++;
    } while (n > 0);

    printf("Number of digits = %d\n", count);
    return 0;
}
```

3.3.3 For Loop

Methodology:

For Loop The `for` loop provides a compact way to iterate by combining initialization, condition checking, and increment/decrement into a single line.

1. **Initialization:** Executed exactly once at the beginning to set up the loop control variable.
2. **Condition:** Evaluated before each iteration. If true, the loop body executes; if false, the loop terminates.
3. **Update:** Executed at the end of each iteration to modify the loop control variable, followed by re-evaluation of the condition.

Worked Example:

Generate Fibonacci series **Problem:** Print the first 5 terms of the Fibonacci sequence: 0, 1, 1, 2, 3.
Step-by-Step Solution:

1. Initialize first term $t_1 = 0$ and second term $t_2 = 1$.
2. For $i = 1$, print t_1 (0). Next term $next = t_1 + t_2 = 1$. Update $t_1 = t_2 = 1$, $t_2 = next = 1$.
3. For $i = 2$, print t_1 (1). Next term $next = t_1 + t_2 = 2$. Update $t_1 = 1$, $t_2 = 2$.
4. For $i = 3$, print t_1 (1). Next term $next = t_1 + t_2 = 3$. Update $t_1 = 2$, $t_2 = 3$.
5. Continue this process until $i = 5$.

Implementation:

```
#include <stdio.h>
int main() {
    int n = 5;
    int t1 = 0, t2 = 1, nextTerm;

    for (int i = 1; i <= n; i++) {
        printf("%d ", t1);
        nextTerm = t1 + t2;
        t1 = t2;
        t2 = nextTerm;
    }
    printf("\n");
    return 0;
}
```

3.4 Jump Statements

3.4.1 Break Statement

Methodology:

Break Statement The `break` statement terminates the execution of the nearest enclosing loop or `switch` statement immediately.

1. When a `break` statement is encountered inside a loop or switch, execution of the current block stops.
2. Control is transferred directly to the first statement following the terminated loop or switch construct.
3. It is commonly used for early loop termination when a specific condition is met.

Worked Example:

Check if a number is prime **Problem:** Determine if $N = 7$ is a prime number by checking for divisors.
Step-by-Step Solution:

1. A prime number has no divisors other than 1 and itself.
2. Loop variable i goes from 2 to $N - 1$.
3. Inside the loop, check if $N \pmod{i} == 0$.
4. If a divisor is found, set a flag to 1 and immediately execute `break` to avoid unnecessary iterations.
5. If the loop finishes without the flag changing, N is prime.

Implementation:

```
#include <stdio.h>
int main() {
    int n = 7;
    int isPrime = 1; // Assume prime initially

    for (int i = 2; i <= n / 2; i++) {
        if (n % i == 0) {
            isPrime = 0; // Found a divisor
            break;       // Terminate the loop early
        }
    }

    if (isPrime == 1 && n > 1) {
        printf("%d is a Prime Number.\n", n);
    } else {
        printf("%d is not a Prime Number.\n", n);
    }
    return 0;
}
```

3.4.2 Continue Statement

Methodology:

Continue Statement The `continue` statement skips the remaining code inside the current loop iteration and immediately jumps to the next iteration.

1. When `continue` is encountered, any subsequent statements in the loop body are bypassed.
2. For a `while` and `do-while` loop, control jumps to the condition evaluation.
3. For a `for` loop, control jumps to the update/increment expression.

Worked Example:

Print odd numbers skipping even numbers **Problem:** Iterate from 1 to 5, but print only odd numbers by skipping even ones using `continue`.
Step-by-Step Solution:

1. Start loop with $i = 1$ to 5.
2. **i = 1:** $1 \pmod 2 \neq 0$, the `if` condition is false. Print 1.
3. **i = 2:** $2 \pmod 2 == 0$, the `if` condition is true. Execute `continue`. The `printf` is skipped, move to $i = 3$.
4. **i = 3:** Condition false. Print 3.
5. **i = 4:** Condition true. Execute `continue`, move to $i = 5$.
6. **i = 5:** Condition false. Print 5.

Implementation:

```
#include <stdio.h>
int main() {
    for (int i = 1; i <= 5; i++) {
        if (i % 2 == 0) {
            continue; // Skip the even number
        }
        printf("%d ", i);
    }
    printf("\n");
    return 0;
}
```

CHAPTER 4
Arrays and Pointers

4.1 One-Dimensional Arrays

Methodology:

Array Traversal and Element Access An array is a collection of elements of the same data type stored in contiguous memory locations.

1. **Declaration**: `data_type array_name[size];` (e.g. `int arr[5];`)
2. **Initialization**: `int arr[5] = {10, 20, 30, 40, 50};`
3. **Access**: Elements are accessed using a 0-based index: `arr[0]` to `arr[size-1]`.
4. **Traversal Algorithm**:
   (a) Initialize a loop counter `i = 0`.
   (b) Check condition `i < size`.
   (c) Process `arr[i]`.
   (d) Increment `i` and repeat steps 2-4.

Worked Example:

Finding the Maximum Element in an Array **Problem:** Given an array `A = {14, 56, 23, 89, 12}`, find the maximum element.
Step-by-step Solution:

1. **Initialize:** Let `max` be the first element. `max = A[0] = 14`.
2. **Iteration 1 (`i=1`):** Compare `A[1]` with `max`. Since $56 > 14$, update `max = 56`.
3. **Iteration 2 (`i=2`):** Compare `A[2]` with `max`. Since $23 \not> 56$, `max` remains 56.
4. **Iteration 3 (`i=3`):** Compare `A[3]` with `max`. Since $89 > 56$, update `max = 89`.
5. **Iteration 4 (`i=4`):** Compare `A[4]` with `max`. Since $12 \not> 89$, `max` remains 89.
6. **Result:** The maximum element is 89.

Methodology:

Bubble Sort Algorithm Bubble sort repeatedly steps through the list and swaps adjacent elements if they are in the wrong order.

1. Let the array be `A` of size `N`.
2. Loop `i` from 0 to $N - 1$ (represents the pass number).

3. Loop `j` from 0 to $N - i - 2$.
4. If `A[j] > A[j+1]`, swap `A[j]` and `A[j+1]`.
5. After each pass i, the largest unsorted element is placed at its correct sorted position at the end.

Worked Example:

Sorting an Array using Bubble Sort **Problem:** Sort the array `A = {34, 12, 25, 9}` in ascending order.
Step-by-step Solution:

1. **Initial Array:** {34, 12, 25, 9}. Here $N = 4$.
2. **Pass 1** ($i = 0$)**:**
   - Compare 34 and 12. $34 > 12$, so swap: {12, 34, 25, 9}.
   - Compare 34 and 25. $34 > 25$, so swap: {12, 25, 34, 9}.
   - Compare 34 and 9. $34 > 9$, so swap: {12, 25, 9, 34}.
3. **Pass 2** ($i = 1$)**:**
   - Compare 12 and 25. No swap needed.
   - Compare 25 and 9. $25 > 9$, so swap: {12, 9, 25, 34}.
4. **Pass 3** ($i = 2$)**:**
   - Compare 12 and 9. $12 > 9$, so swap: {9, 12, 25, 34}.
5. **Final Result:** The sorted array is {9, 12, 25, 34}.

4.2 Two-Dimensional Arrays

Methodology:

Matrix Multiplication Algorithm To multiply two matrices, Matrix `A` of size $m \times n$ and Matrix `B` of size $n \times p$, the resulting Matrix `C` will be of size $m \times p$.

1. Initialize all elements of Matrix `C` to zero.
2. Loop `i` from 0 to $m - 1$ (iterate over rows of `A`).
3. Loop `j` from 0 to $p - 1$ (iterate over columns of `B`).
4. Loop `k` from 0 to $n - 1$ (iterate over columns of `A` or rows of `B`).
5. Compute intermediate sum: `C[i][j] = C[i][j] + A[i][k] * B[k][j]`.

Worked Example:

Multiplying Two 2x2 Matrices **Problem:** Multiply matrices A and B.

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Step-by-step Solution:

1. **Element $C[0][0]$ (Row 0 of A and Col 0 of B):**

$$C[0][0] = (1 \times 5) + (2 \times 7) = 5 + 14 = 19$$

2. **Element $C[0][1]$ (Row 0 of A and Col 1 of B):**

$$C[0][1] = (1 \times 6) + (2 \times 8) = 6 + 16 = 22$$

3. **Element $C[1][0]$ (Row 1 of A and Col 0 of B):**

$$C[1][0] = (3 \times 5) + (4 \times 7) = 15 + 28 = 43$$

4. **Element $C[1][1]$ (Row 1 of A and Col 1 of B):**

$$C[1][1] = (3 \times 6) + (4 \times 8) = 18 + 32 = 50$$

5. **Resultant Matrix C:**

$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

4.3 Strings as Character Arrays

Methodology:

Computing String Length Manually A string in C is a one-dimensional array of characters terminated by a null character (`'\0'`).

1. Initialize a counter variable `len = 0`.
2. Iterate through the character array using a loop: `while (str[len] != '\0')`.
3. Inside the loop increment the counter: `len++`.
4. The final value of `len` is the length of the string (excluding the null terminator).

Worked Example:

Finding Length and Reversing a String **Problem:** Given the string `"GTU"`, find its length and reverse it without using library functions.
Step-by-step Solution:

1. **Memory Representation:** `str[] = {'G', 'T', 'U', '\0'}`
2. **Find Length:**
   - `str[0] = 'G'` $\neq$ `'\0'` $\Rightarrow$ `len = 1`
   - `str[1] = 'T'` $\neq$ `'\0'` $\Rightarrow$ `len = 2`
   - `str[2] = 'U'` $\neq$ `'\0'` $\Rightarrow$ `len = 3`
   - `str[3] = '\0'` $==$ `'\0'` $\Rightarrow$ Loop terminates. Length is 3.
3. **Reverse String Algorithm:**
   - Use two indices: `start = 0` and `end = len - 1 = 2`.
   - Swap `str[start]` and `str[end]`.
   - Swap `str[0]` ('G') and `str[2]` ('U') $\Rightarrow$ `str[] = {'U', 'T', 'G', '\0'}`
   - Increment `start` to 1 and decrement `end` to 1. Since `start >= end`, stop the process.
4. **Result:** Length is 3 and the reversed string is `"UTG"`.

4.4 Pointers

Methodology:

Pointer Operations and Indirection A pointer is a variable that stores the memory address of another variable.

1. **Declaration**: `int *ptr;` (Pointer to an integer).
2. **Address-of Operator (&)**: Used to get the memory address. `ptr = &x;` assigns the address of `x` to `ptr`.
3. **Indirection/Dereference Operator (*)**: Used to access the value at the stored address. `*ptr = 10;` changes the value of `x` to 10.
4. **Pointer Arithmetic**: Adding 1 to a pointer (`ptr + 1`) increments its address by the size of the data type it points to (e.g. +4 bytes for a standard 32-bit `int`).

Worked Example:

Swapping Two Values using Pointers Call by Reference **Problem:** Swap the values of $a = 5$ and $b = 9$ using a swap function that accepts pointers.
Step-by-step Solution:

1. **Initialization:** Let `a` be at memory address 1000 (value 5) and `b` at address 1004 (value 9).
2. **Function Call:** `swap(&a, &b)`. Pointers `p1` and `p2` receive the addresses. `p1 = 1000`, `p2 = 1004`.
3. **Store value via pointer:** `temp = *p1`. The value at address 1000 is fetched. `temp = 5`.
4. **Overwrite first value:** `*p1 = *p2`. The value at address 1004 (which is 9) is copied to address 1000. Now $a = 9$.
5. **Overwrite second value:** `*p2 = temp`. The value of `temp` (which is 5) is copied to address 1004. Now $b = 5$.
6. **Result:** a and b are successfully swapped ($a = 9, b = 5$) in the calling function.

Methodology:

Pointers and Array Traversal The name of an array acts as a constant pointer to its first element.

1. For an array `int arr[5];`, the identifier `arr` is equivalent to `&arr[0]`.
2. To traverse using a pointer, initialize `int *p = arr;`.
3. Access the element using `*p`.
4. Move to the next element by incrementing the pointer: `p++`.
5. The i-th element can be accessed as `*(arr + i)`, which is perfectly equivalent to `arr[i]`.

Worked Example:

Summing Array Elements via Pointers **Problem:** Calculate the sum of elements in `arr = {4, 7, 2}` using pointer arithmetic.
Step-by-step Solution:

1. **Initialize:** `sum = 0`. Pointer `p = arr`. Let `arr[0]` be at address 2000. Thus `p = 2000`.

2. **Element 1:** Add `*p` to sum. `sum = 0 + 4 = 4`. Increment pointer `p++`. Now `p = 2004` (assuming 4-byte integers).
3. **Element 2:** Add `*p` to sum. The value at address 2004 is 7. `sum = 4 + 7 = 11`. Increment pointer `p++`. Now `p = 2008`.
4. **Element 3:** Add `*p` to sum. The value at address 2008 is 2. `sum = 11 + 2 = 13`.
5. **Result:** The total sum of the array is 13.

CHAPTER 5
Functions and Structures

5.1 User-Defined Functions

Functions allow decomposing large problems into smaller, manageable sub-problems. A function in C encapsulates a specific computational task.

Methodology:

Designing a User Defined Function

1. **Function Declaration**: Declare the function prototype before the `main()` function specifying the return type, function name and parameter types.
2. **Function Definition**: Write the function block with the exact same signature. The variables inside the parentheses are called formal parameters.
3. **Function Call**: Invoke the function from `main()` or another function by passing actual parameters (arguments).
4. **Return Value**: Use the `return` statement if the function calculates and sends a value back to the caller.

Worked Example:

Write a function to calculate the sum of digits of a given integer **Step 1: Function Declaration** Define a function that takes an integer and returns an integer.

```
int sumOfDigits(int n);
```

Step 2: Function Definition Extract digits using modulo arithmetic and sum them.

```
int sumOfDigits(int n) {
    int sum = 0;
    while (n != 0) {
        sum += n % 10;
        n /= 10;
    }
    return sum;
}
```

Step 3: Main Function Call the function and display the result.

```
#include <stdio.h>

int main() {
    int num = 1234;
    int result = sumOfDigits(num);
    printf("Sum of digits of %d is %d\n", num, result);
```

```
    return 0;
}
```

Output:

```
Sum of digits of 1234 is 10
```

5.2 Parameter Passing Mechanisms

Methodology:

Call by Value versus Call by Reference

1. **Call by Value**: The actual parameter values are copied into the formal parameters. Changes made inside the function do not affect the original variables.
2. **Call by Reference**: The memory addresses (pointers) of the actual parameters are passed. The function accesses the original variables using the dereference operator (`*`).
3. **Algorithm for Call by Reference**:
   (a) Pass the addresses using the `&` operator during the function call.
   (b) Catch the addresses in pointer variables in the function definition.
   (c) Use `*` to read or modify the values at those addresses.

Worked Example:

Write a C program to swap two numbers using Call by Reference **Step 1: Function Definition** Use pointer variables to receive addresses. Use a temporary variable to swap the values.

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Step 2: Main Function Declare two variables and pass their addresses to the `swap` function.

```
#include <stdio.h>

int main() {
    int x = 10;
    int y = 20;

    printf("Before swapping: x = %d y = %d\n", x, y);
    swap(&x, &y);
    printf("After swapping: x = %d y = %d\n", x, y);

    return 0;
}
```

Output:

```
Before swapping: x = 10 y = 20
After swapping: x = 20 y = 10
```

5.3 Recursion

Recursion occurs when a function calls itself to solve a smaller instance of the same problem.

Methodology:

Designing Recursive Algorithms

1. **Identify the Base Case**: Determine the simplest condition for which the answer is known and does not require further recursion. This prevents infinite loops.
2. **Define the Recursive Step**: Express the problem in terms of a smaller version of itself.
3. **Combine**: Ensure that each recursive call brings the execution closer to the base case.

Worked Example:

Calculate the factorial of a number using recursion **Step 1: Identify Base Case and Recursive Step**

- Base Case: Factorial of 0 is 1. $(0! = 1)$
- Recursive Step: Factorial of n is $n \times (n-1)!$

Step 2: Function Definition

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

Step 3: Main Function

```
#include <stdio.h>

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

Output:

```
Factorial of 5 is 120
```

5.4 Structures

Structures allow grouping variables of different data types under a single name.

Methodology:

Defining and Using Structures

1. **Structure Definition**: Use the `struct` keyword to define a new composite data type. List the member variables inside curly braces.
2. **Variable Declaration**: Declare variables of the structure type.

3. **Accessing Members**: Use the dot operator (`.`) to access individual members of a structure variable.
4. **Initialization**: Initialize members at the time of declaration using curly braces `{}`.

Worked Example:

Create a structure for a Point in a 2D Cartesian plane and find the distance between two points **Step 1: Define the Structure**

```
struct Point {
    float x;
    float y;
};
```

Step 2: Write a Distance Function Use the distance formula $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Note the function takes structure variables as arguments.

```
#include <math.h>

float calculateDistance(struct Point p1, struct Point p2) {
    float dx = p2.x - p1.x;
    float dy = p2.y - p1.y;
    return sqrt(dx * dx + dy * dy);
}
```

Step 3: Main Function

```
#include <stdio.h>

int main() {
    struct Point point1 = {0.0, 0.0};
    struct Point point2 = {3.0, 4.0};

    float distance = calculateDistance(point1, point2);
    printf("Distance between points: %.2f\n", distance);
    return 0;
}
```

Output:

```
Distance between points: 5.00
```

5.5 Array of Structures

An array of structures is used to store data for multiple entities that share the same attributes (e.g. details of 50 students).

Methodology:

Processing Arrays of Structures

1. **Declaration**: Declare an array where the data type is a previously defined `struct`. For example: `struct Student s[50];`
2. **Iteration**: Use a `for` loop to iterate through the array indices.
3. **Accessing Data**: For the i-th element, access members using `arrayName[i].memberName`.

Worked Example:

Store and display the records of 3 employees using an array of structures **Step 1: Structure Definition**

```
struct Employee {
    int empId;
    char name[50];
    float salary;
};
```

Step 2: Main Function with Array and Loop Processing

```
#include <stdio.h>

int main() {
    struct Employee emp[3] = {
        {101, "Alice", 55000.0},
        {102, "Bob", 62000.0},
        {103, "Charlie", 48000.0}
    };

    printf("Employee Records:\n");
    for (int i = 0; i < 3; i++) {
        printf("ID: %d | Name: %s | Salary: %.2f\n",
               emp[i].empId, emp[i].name, emp[i].salary);
    }

    return 0;
}
```

Output:

```
Employee Records:
ID: 101 | Name: Alice | Salary: 55000.00
ID: 102 | Name: Bob | Salary: 62000.00
ID: 103 | Name: Charlie | Salary: 48000.00
```

5.6 Nested Structures

A structure can contain another structure as its member. This is known as a nested structure.

Methodology:

Working with Nested Structures

1. **Define Inner Structure**: Define the structure that will be embedded first.
2. **Define Outer Structure**: Include a variable of the inner structure type inside the outer structure definition.
3. **Chaining Dot Operators**: Access deeply nested members by chaining the dot operator. Example: `outerVar.innerVar.member`

Worked Example:

Create a nested structure for a Student containing their Date of Birth **Step 1: Define Structures**

```
struct Date {
```

```
    int day;
    int month;
    int year;
};

struct Student {
    int rollNo;
    char name[50];
    struct Date dob;
};
```

Step 2: Main Function

```
#include <stdio.h>

int main() {
    struct Student s1 = {1, "David", {15, 8, 2005}};

    printf("Student Info:\n");
    printf("Roll No: %d\n", s1.rollNo);
    printf("Name: %s\n", s1.name);
    // Access nested members
    printf("Date of Birth: %02d/%02d/%d\n",
           s1.dob.day, s1.dob.month, s1.dob.year);

    return 0;
}
```

Output:

```
Student Info:
Roll No: 1
Name: David
Date of Birth: 15/08/2005
```

Formulae

Appendix: Key Formulae

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

Unit 1: Basic Computations

Simple Interest

$$SI = \frac{p \times r \times t}{100}$$

- SI: Simple Interest
- p: Principal amount
- r: Rate of interest per annum
- t: Time period

Unit 2: Bitwise Operations and Number Theory

Bitwise NOT (Two's Complement)

$$\sim x = -(x + 1)$$

- x: An integer value
- $\sim x$: Bitwise NOT of x

Division Algorithm

$$a = b \times q + r$$

- a: Dividend
- b: Divisor
- q: Quotient
- r: Remainder (where $0 \leq r < b$)

Unit 3: Iteration and Sequences

Fibonacci Sequence

$$F_n = F_{n-1} + F_{n-2}$$

- F_n: The n-th term of the sequence
- $F_0 = 0$, $F_1 = 1$: The base cases

Digit Extraction

$$d = N \pmod{10}$$
$$N_{next} = \lfloor N/10 \rfloor$$

- N: An integer
- d: The least significant digit of N
- N_{next}: The integer obtained after removing the last digit

Unit 4: Arrays and Matrices

Matrix Multiplication

For matrices A of size $m \times p$ and B of size $p \times n$, their product $C = AB$ is an $m \times n$ matrix with elements defined by:

$$C_{ij} = \sum_{k=1}^{p} A_{ik} B_{kj}$$

- C_{ij}: Element at the i-th row and j-th column of matrix C
- A_{ik}: Element at the i-th row and k-th column of matrix A
- B_{kj}: Element at the k-th row and j-th column of matrix B

Unit 5: Mathematical Functions

Euclidean Distance

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- d: Distance between two points
- (x_1, y_1): Coordinates of the first point
- (x_2, y_2): Coordinates of the second point

Factorial

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0 \end{cases}$$

- $n!$: Factorial of a non-negative integer n