Q.1

Define problem-solving steps. List any three characteristics of a good algorithm.

Answer
Problem solving in programming is the process of turning a real-world requirement into a sequence of clear, testable steps that a computer (or a person) can follow. It begins with carefully understanding what is required, moves on to planning and designing a solution, and finishes with implementing and testing that solution so it reliably solves the original problem.

Key steps (explained):

1. **Understand the problem:** Read and restate the problem in your own words; identify inputs and desired outputs. This prevents wrong assumptions.
2. **Analyze and break down:** Divide the problem into smaller, manageable subproblems (divide and conquer). Smaller steps are easier to design and test.
3. **Design a solution:** Choose an approach (algorithm), decide data structures, and plan control flow (loops/decisions).
4. **Write pseudocode/flowchart:** Express the logic in simple steps or diagrams before coding — this clarifies edge cases and flow.
5. **Implement (code):** Convert pseudocode into the chosen programming language carefully, following the design.
6. **Test and debug:** Use sample inputs, boundary conditions and unexpected inputs; correct errors and refine.
7. **Document & maintain:** Add comments and documentation so others (or you later) can understand and modify the solution.

Three characteristics of a good algorithm (with explanation):

- **Correctness:** The algorithm must produce the expected result for all valid inputs. (An incorrect algorithm is useless even if efficient.)
- **Finiteness (Termination):** It must finish after a finite number of steps — it should not loop forever. This ensures it eventually yields an answer.
- **Efficiency (Time & Space):** It should use reasonable time and memory. A correct algorithm that is too slow or memory-heavy may be impractical.

Write a pseudocode and draw a flowchart to determine whether a number is even or odd.

Answer (pseudocode paragraph + step list):
To check parity we use the remainder when dividing by 2. If the remainder is zero the number is even; otherwise it is odd. The logic is simple and fast (O(1) time).

Pseudocode (clear steps):

1. Start
2. Read integer N
3. Compute R = N % 2 (remainder on division by 2)
4. If R == 0 then
    - Print "Even Number"
      Else
    - Print "Odd Number"
5. Stop

Flowchart (described so you can draw it):

- Draw an **Oval** labeled **Start**.
- Draw a **Parallelogram** labeled **Input N**.
- Draw a **Rectangle** labeled **Compute R = N % 2**.
- Draw a **Diamond** labeled **R == 0 ?** with two outgoing arrows:
    - Yes → **Parallelogram**: Print "Even Number" → arrow to **Oval End**.
    - No → **Parallelogram**: Print "Odd Number" → arrow to **Oval End**.
- Draw final **Oval** labeled **End**.

Notes / Edge cases:

- Works for negative integers too (since % gives 0 or 1 for integers in mathematical sense).
- If input may be non-integer, validate before applying modulo.

Explain different flowchart symbols with neat diagrams. Also explain disadvantages of flowchart with examples.

Answer (intro paragraph + symbol list + disadvantages):
Flowcharts are a universal, visual way to represent algorithms. They use a small set of standard symbols so readers can quickly understand the kind of step (input, process, decision) being represented. While flowcharts are very useful for planning and communication, they have limits when programs become large or highly detailed.

Common symbols and meaning (short explanation for each):

1. **Oval (Start/Stop):** Marks the beginning and end points of the flowchart. Always required to show entry/exit.

2. **Parallelogram (Input/Output):** Represents input actions (e.g., read N) or outputs (e.g., print result).
3. **Rectangle (Process):** Represents an instruction or calculation (e.g., sum = a + b).
4. **Diamond (Decision):** Represents a condition with two or more branches (Yes/No, True/False).
5. **Arrow (Flowline):** Shows the direction of control — which step follows which.
6. **Circle (Connector):** Used to join different parts of large flowcharts — acts as a "go to" within the diagram.
7. **Document/File symbol:** (if needed) indicates document or stored data.

Disadvantages (with simple examples):

1. **Scalability:** For complex systems (banking, OS), flowcharts become huge and hard to read. *Example:* A flowchart for full bank transaction processing would span many pages.
2. **Maintenance:** Small code changes may require redrawing large parts of the chart. *Example:* Adding an extra validation step may force rework of several decision paths.
3. **Ambiguity in details:** Flowcharts show structure but not implementation specifics (e.g., exactly how to handle floating-point rounding). They can hide important details.
4. **Not executable:** You cannot run a flowchart; it must be translated into code, which introduces potential transcription errors.
5. **Time consuming to create:** Creating neat, accurate flowcharts takes time — sometimes slower than writing a quick prototype.

When to use vs. not to use: Flowcharts are excellent for teaching, documenting algorithms, and designing small to medium programs. For very large systems, higher-level diagrams (UML, pseudocode, modular design) may be preferable.

OR

Explain development of algorithms using pseudocode. Discuss advantages and disadvantages with examples.

Answer (paragraph + steps + pros/cons):
Pseudocode is a plain-language description of the logic of a program, combining natural language and simple programming-like constructs. It is used to design algorithms before coding: because it is language-independent and readable, it helps validate logic and spot edge cases early.

How to develop an algorithm using pseudocode (practical steps):

1. **Understand problem and inputs/outputs.** Write a brief problem statement.
2. **Break into steps** (high-level tasks), then refine each into smaller steps.
3. **Write pseudocode**: use clear statements like IF, ELSE, FOR, WHILE, READ, PRINT.
4. **Add test cases**: manually simulate pseudocode on sample inputs and edge cases.
5. **Refine and optimize**: reduce unnecessary steps, consider performance.
6. **Translate to code**: convert pseudocode to actual programming language.

Advantages (explain why they matter):

1. **Language-independent:** Works for programmers regardless of the final language; good design tool.
2. **Readable for non-programmers:** Managers or testers can follow logic.
3. **Easy to modify:** Changing pseudocode is faster than rewriting code.
4. **Helps debugging early:** Logical errors can be found before coding.

Disadvantages (caveats):

1. **No strict standard:** Different people write pseudocode differently, which can cause misunderstandings.
2. **Not executable:** You cannot run pseudocode to test it; translation to code may introduce bugs.
3. **May omit implementation details:** Low-level issues (e.g., memory use, exact library calls) are left out and may cause surprises later.

Example: For sorting, writing pseudocode for Bubble Sort or QuickSort helps reason about comparisons and swaps before implementing in C or Python.

Q.2 Explain fgets() and puts() functions with suitable example.

Answer (paragraph + points + example):
In C, fgets() and puts() are safe, simple functions for string input/output. fgets() reads a line (or up to a specified number of characters) from a stream and prevents buffer overflow by taking a size argument. puts() writes a string to stdout and automatically appends a newline.

Key points about fgets() and puts():

1. **fgets(char *s*, *int n*, *FILE stream):* reads at most n-1 characters, stores them in s, and appends a null \0. It stops on newline or EOF.
2. fgets() reads spaces and tabs (unlike scanf("%s",...) that stops at whitespace).

3. *puts(const char s):* prints the string and appends a newline; simpler than printf("%s\n", s).
4. Use fgets() with stdin for safe console input.
5. Remember fgets() keeps the newline character in the buffer if present; you may want to strip it.

Simple example:

```
#include <stdio.h>

int main() {
   char name[20];
   printf("Enter your name: ");
   fgets(name, 20, stdin);  // safe input
   puts(name);              // prints name and newline
   return 0;
}
```

Note: If you need to remove the trailing newline from fgets() result:

```
size_t len = strlen(name);
if (len > 0 && name[len-1] == '\n') name[len-1] = '\0';
```

What is formatted input and formatted output? Give examples using scanf() and printf().

Answer (paragraph + key points + examples):
Formatted I/O means reading and writing data using patterns (format specifiers) so values are converted between text and binary forms in a predictable way. In C, scanf() reads formatted input from stdin; printf() prints formatted output to stdout. They give you control over types, spacing, precision and alignment.

Important points:

1. **printf(format, ...):** format specifiers like %d (int), %f (float), %c (char), %s (string) control how values are printed. You can specify width and precision (e.g., %.2f for two decimal places).
2. **scanf(format, ...):** reads according to specifiers; you must pass addresses for variables (e.g., &x).
3. Always validate return value of scanf() — it returns number of items read successfully.

4. Use field width to prevent buffer overflow for strings: scanf("%19s", name).
5. printf() is versatile for formatting tables, decimals, and debugging.

Examples:

```
int n;
float x;
char name[20];

printf("Enter an integer and a float: ");
scanf("%d %f", &n, &x);
printf("You entered %d and %.2f\n", n, x);

printf("Enter your name: ");
scanf("%19s", name);  // reads until whitespace
printf("Hello %s\n", name);
```

Caveats: scanf("%s",...) cannot read spaces; use fgets() if spaces are needed.

Write a Menu Driven Program to generate calculator using switch-case statement.

Answer (paragraph + full C program + explanation):
A menu-driven calculator displays choices to the user and uses switch-case to pick the operation. switch is efficient for selecting one among many constant choices.

C Program:

```
#include <stdio.h>

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

    printf("Menu:\n1. Add\n2. Subtract\n3. Multiply\n4. Divide\n");
    printf("Enter choice: ");
    if (scanf("%d", &choice) != 1) return 0;
```

```
    printf("Enter two numbers: ");
    if (scanf("%f %f", &a, &b) != 2) return 0;

    switch(choice) {
        case 1:
            result = a + b;
            printf("Result = %.2f\n", result);
            break;
        case 2:
            result = a - b;
            printf("Result = %.2f\n", result);
            break;
        case 3:
            result = a * b;
            printf("Result = %.2f\n", result);
            break;
        case 4:
            if (b != 0.0f)
                printf("Result = %.2f\n", a / b);
            else
                printf("Error! Division by zero.\n");
            break;
        default:
            printf("Invalid choice\n");
    }
    return 0;
}
```

Explanation & good practices (points):

1. Display menu clearly so user knows options.

2. Validate scanf() return values to avoid undefined behavior.
3. Check for division by zero before dividing.
4. Use break to prevent fall-through between cases.
5. Use default to handle invalid inputs.
6. Use floating types for real numbers; choose double for higher precision if desired.
7. Consider looping the menu (e.g., do...while) if you want repeated operations.

OR

Discuss conditional (ternary) operator with suitable example.

Answer (paragraph + points + example):
The conditional operator ?: in C is a compact form of if-else that evaluates an expression and returns one of two values depending on the condition. It is handy for simple decisions where writing a full if-else is verbose.

Key notes:

1. **Syntax:** condition ? expr_if_true : expr_if_false.
2. **Returns a value:** So it can be used inside expressions or assignments.
3. **Readability:** Good for short expressions; avoid complex nested ternaries.
4. **Use case:** quick selection like assigning max/min.

Example:

```
int a = 5, b = 3;

int max = (a > b) ? a : b;  // max becomes 5
```

Algorithm & flowchart to find largest among three numbers.

Answer (paragraph + algorithm + flowchart description):
To find the largest number among three values, compare them pairwise using decision steps. This is a simple branching algorithm with O(1) time.

Algorithm (steps):

1. Start
2. Read A, B, C
3. If A >= B and A >= C then largest = A
4. Else if B >= A and B >= C then largest = B

5. Else largest = C
6. Print largest
7. Stop

Flowchart description:

- Start → Input A,B,C → Decision: A >= B && A >= C ?
    - Yes → Output A → End
    - No → Decision: B >= C ?
        - Yes → Output B → End
        - No → Output C → End

Notes: Use >= to correctly handle equal values.

Explain the basic structure of a C program and steps of executing a C program.

Answer (paragraph + components + execution steps):
A C program has a predictable structure: preprocessor directives, global declarations, function definitions (with main() as the entry point), and statements inside functions. Execution follows an edit→compile→link→run lifecycle.

Basic components of a C program:

1. **Comments and documentation** (optional): human-readable notes.
2. **Preprocessor directives:** #include <stdio.h> etc. — include headers and macros.
3. **Global declarations:** global variables, constants.
4. **Function definitions:** main() must be present; other functions are helpers.
5. **Local declarations and statements:** inside functions.
6. **Return statement:** return 0; in main indicates success.

Steps to execute a C program:

1. **Edit:** Write source code in a text editor, save as .c.
2. **Preprocessing:** #include and macros handled by preprocessor.
3. **Compilation:** Compiler (gcc, cl) translates source to object code and reports syntax errors.
4. **Assembling:** Compiler output assembled into machine code (often combined with linking step).
5. **Linking:** Linker combines object code with libraries (libc) to produce executable.

6. **Execution:** Run the produced executable; OS loads it and starts at main().

Practical tips: Check compiler warnings, use -Wall flags, and test with varied inputs.

Q.3

Write difference between break and continue statement.

Answer (paragraph + comparison points):
break and continue control loop execution. break exits the innermost loop immediately and resumes execution after it. continue skips the remainder of the current loop iteration and jumps to the loop's next iteration (checking the loop condition). Both are useful for controlling flow, but must be used carefully to keep code readable.

Comparison (concise):

1. **Effect:** break exits loop; continue skips to next iteration.
2. **Use case:** break when goal is achieved and no further looping required; continue to ignore certain iterations but keep looping.
3. **Control flow:** break transfers control outside loop; continue transfers control to loop update/check step.
4. **Applies to:** for, while, do-while, switch (break in switch exits the switch).
5. **Readability caution:** Overuse can make loops hard to follow.
6. **Example:**

```
for(i=0;i<10;i++){
  if(i==5) break;   // stop entirely when i==5
  if(i%2==0) continue; // skip even numbers
  printf("%d", i);
}
```

Describe in detail how the control flow works in a nested for loop. Include step-by-step execution sequence.

Answer (paragraph + step sequence + example):
Nested loops are loops placed inside other loops. Execution proceeds by executing the outer loop one iteration, and for each outer iteration the inner loop runs fully. This pattern is useful for multi-dimensional data (tables, matrices) and pattern generation.

Step-by-step:

1. **Initialize outer loop** (set outer index).
2. **Check outer condition**; if false, exit both loops.
3. **Enter outer iteration body**.
4. **Initialize inner loop** (set inner index).
5. **Check inner condition**; if false, inner ends and control goes back to outer increment.
6. **Execute inner loop body** for current inner index.
7. **Increment inner index** and repeat inner condition check.
8. **When inner finishes**, store result if needed, then **increment outer index**.
9. **Repeat outer** until its condition fails.

Example (C):

```
for (i = 1; i <= 3; i++) {
   for (j = 1; j <= 2; j++) {
      printf("%d %d\n", i, j);
   }
}
```

Execution order: (1,1) → (1,2) → inner ends → outer increments → (2,1) → (2,2) → (3,1) → (3,2) → end.

Complexity note: Total iterations = outer_count × inner_count. Use carefully when counts are large.

Explain one-dimensional array declaration and initialization with example.

Answer (paragraph + points + code):
A one-dimensional array is a collection of elements of the same type stored in contiguous memory, accessible by an index starting at 0. Arrays allow grouping related data (like marks, readings) under one name and efficient indexed access.

Key points:

1. **Declaration syntax (C):** type name[size]; e.g., int arr[5]; reserves space for 5 integers.
2. **Indexing:** First element arr[0], last arr[size-1].
3. **Initialization at declaration:** int arr[5] = {10, 20, 30, 40, 50};

4. **Partial initialization:** int arr[5] = {1,2}; remaining elements set to 0.
5. **Dynamic filling (runtime):** Use loops and scanf to read values into array elements.
6. **Memory layout:** Elements in contiguous addresses → arr[i+1] follows arr[i] in memory.
7. **Use cases:** Storing sequences, implementing other data structures.

Example program (C):

```
#include <stdio.h>


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

Notes: Arrays in C do not carry their size; you must track length separately.

OR

Write difference between Entry Control Loop and Exit Control Loop.

Answer
Entry control loops check the loop condition **before** executing the body (for, while), so they may execute zero times if the condition is false initially. Exit control loops (do-while) check the condition **after** executing the body, so they always execute at least once.

Comparison points:

1. **Entry checks first:** while(condition) → may skip loop entirely.
2. **Exit checks after:** do { ... } while(condition); → runs body at least once.
3. **Typical usage:** Use entry loop when iterations might be zero; use exit loop when you need the body to run once (e.g., menu that must show once).

Explain for loop with syntax and suitable example.

Answer (paragraph + syntax + example):
A for loop is compact and convenient when the number of iterations is known before entering the loop. It includes initialization, condition testing, and update expression in one header.

Syntax (C):

```
for (initialization; condition; update) {
    statements;
}
```

Example:

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

Explanation: i is initialized to 1, loop continues while i <= 5, and i increments by 1 after each iteration. The loop prints 1 2 3 4 5.

Explain nested loops and why required in C programming.

Answer (paragraph + points + examples):
Nested loops are used when you have to perform repeated operations inside another repetition — common in processing multi-dimensional data like matrices, grid traversal, or printing nested patterns (like triangles). They let you iterate over multiple dimensions naturally.

Why nested loops are required:

1. **2D data processing:** To access each row and column in a matrix you use for (i) outer and for (j) inner.
2. **Pattern printing:** Many console patterns need nested iteration (rows & columns).
3. **Complex algorithms:** Some algorithms (like certain brute-force checks) naturally require nested iteration.
4. **Clarity:** They map well to nested structures — simpler to read and implement than flattening indices.

Example (multiplication table):

```
for (i = 1; i <= 3; i++) {
    for (j = 1; j <= 5; j++) {
```

```
        printf("%d x %d = %d\n", i, j, i*j);
    }
}
```

Caution: Be mindful of complexity — nested loops multiply iteration counts.

Q.4

Features of C Language.

Answer (paragraph + features):
C is a general-purpose, structured programming language widely used for system and application software. It balances low-level access to memory with high-level structured constructs.

Key features with brief explanations:

1. **Structured programming:** Supports functions, blocks and modular design so programs are logically divided.
2. **Efficiency and speed:** Compiled C programs are fast and suitable for performance-critical code.
3. **Low-level access:** Pointers and direct memory manipulation make system programming possible.
4. **Portability:** Well-written C code can be compiled on many platforms with little change.
5. **Rich library (standard C library):** Functions for I/O, math, string, memory, etc.
6. **Extensibility:** Easy to interface with assembly language or embed as needed.

Explain main function in C.

Answer (paragraph + important points):
The main() function is the entry point of every C program — execution begins at the first statement inside main. The return value of main is returned to the operating system; return 0; usually indicates successful execution.

Important points:

1. **Prototype forms:** int main(void) or int main(int argc, char *argv[]) (for command line args).
2. **Return type:** Conventionally int; returning 0 signals success.

3. **Body:** Local variables and program logic go inside { }.
4. **Execution start:** When program runs, OS calls main.
5. **Command line args:** argc is argument count, argv is array of C-strings for each argument.
6. **Without main:** Program has no valid entry point and won't execute.

Example:

```
int main() {
   printf("Hello\n");
   return 0;
}
```

Explain two-dimensional arrays with declaration, initialization and example program.

Answer (paragraph + declaration + example + explanation):
Two-dimensional arrays represent data in rows and columns, like a matrix. In C, they are declared as type name[rows][cols]. They are useful for representing grids, images, and matrices.

Key points:

1. **Declaration:** int arr[3][4]; declares 3 rows, 4 columns.
2. **Static initialization:** int a[2][2] = {{1,2}, {3,4}}; rows provided as sub-lists.
3. **Access:** a[i][j] where 0 <= i < rows, 0 <= j < cols.
4. **Memory layout:** Stored row-wise (row-major order) in contiguous memory.
5. **Looping:** Use nested loops to traverse rows and columns.

Example program (C):

```
#include <stdio.h>


int main() {
   int a[2][2] = { {5,10}, {15,20} };
   int i, j;
   for (i = 0; i < 2; i++) {
      for (j = 0; j < 2; j++) {
         printf("%d ", a[i][j]);
```

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

Output:

5 10

15 20

OR

Explain getchar() and putchar() with suitable example.

Answer (paragraph + points + example):
getchar() reads a single character from standard input and returns it as an int. putchar() writes a single character to standard output. They are simple and fast for character-level I/O.

Points & example:

1. int c = getchar(); reads a char (or EOF).
2. putchar(c); writes character c.
3. Useful for processing input char by char (parsers, simple editors).
4. Example:

```
#include <stdio.h>
int main() {
    int c;
    c = getchar();    // read one char
    putchar(c);       // print same char
    return 0;
}
```

Algorithm & flowchart to find if number is positive or negative.

Answer (paragraph + algorithm + flowchart desc):
To determine the sign of a number, compare it to zero and handle three cases: positive (>0), negative (<0), or zero.

Algorithm steps:

1. Start
2. Read N
3. If N > 0 → Print "Positive"
4. Else if N < 0 → Print "Negative"
5. Else → Print "Zero"
6. Stop

Flowchart description: Start → Input N → Decision N > 0? → Yes: Print Positive → End. No → Decision N < 0? → Yes: Print Negative → End; No → Print Zero → End.

Discuss Ladder else-if Statement with example.

Answer (paragraph + structure + example + explanation):
The else-if ladder is a sequence of if and else if conditions that allow multi-way branching. The program tests conditions from top to bottom; the first true condition executes and the rest are skipped. This structure is clearer than deeply nested if statements and good for mutually exclusive ranges.

Structure and example:

```
if (condition1) {
   // block1
} else if (condition2) {
   // block2
} else if (condition3) {
   // block3
} else {
   // default block
}
```

Example (grading):

```
int marks = 72;
if (marks >= 75)
```

```
    printf("Distinction");
else if (marks >= 65)
    printf("First Class");
else if (marks >= 50)
    printf("Second Class");
else if (marks >= 40)
    printf("Pass");
else
    printf("Fail");
```

Notes:

- Conditions are evaluated in order; correct ordering is crucial.
- Avoid overlapping ranges or incorrect order; put highest ranges first.
- Else-if ladder improves readability compared with nested ifs.

Q.5

Write a program to add two integer numbers.

Answer (paragraph + code + explanation):
A simple C program reads two integers, adds them, and prints the result. Use int variables and scanf() for input.

Program (C):

```
#include <stdio.h>


int main() {
    int a, b, sum;
    printf("Enter two integers: ");
    scanf("%d %d", &a, &b);
    sum = a + b;
    printf("Sum = %d\n", sum);
    return 0;
```

```
}
```

Explanation (points):

1. Read two integers using scanf.
2. Compute sum = a + b.
3. Print result using printf.
4. Handle white space and check scanf return in robust programs.

Discuss if-else statement with suitable example.

Answer (paragraph + parts + example):
if-else is the basic decision statement: if the condition is true, execute one block; else execute another. Use if-else when there are two alternative paths.

Example:

```
int num;
scanf("%d", &num);
if (num % 2 == 0)
   printf("Even\n");
else
   printf("Odd\n");
```

Key points:

1. Condition evaluated to true/false.
2. if block executed if true; else block otherwise.
3. Parentheses around condition required in C.
4. Good for binary decisions; use else if for multi-way decisions.

Program to check whether a student is pass or fail and give grade.

Answer (paragraph + program + explanation):
This program reads student marks and assigns grades according to ranges. Use ladder if-else to ensure only one grade applies.

Program (C):

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

```
int main() {
    int marks;
    printf("Enter marks (0-100): ");
    if (scanf("%d", &marks) != 1) return 0;

    if (marks > 75)
        printf("Distinction\n");
    else if (marks >= 65)
        printf("First Class\n");
    else if (marks >= 50)
        printf("Second Class\n");
    else if (marks >= 40)
        printf("Pass Class\n");
    else
        printf("Fail\n");

    return 0;
}
```

Explanation & notes:

1. Check marks range and handle invalid inputs in robust systems.
2. Use > vs >= carefully to avoid gaps/overlaps.
3. This ladder ensures the highest applicable grade is selected.
4. You may convert this to percentages if marks are out of different total.

OR

Define strings. How are character arrays different from strings?

Answer
In C, a string is a sequence of characters terminated by a null character '\0'. A character array is the memory container used to store characters; when we add a '\0' at the end it becomes a C-string.

Differences (concise):

1. **Character array:** raw storage like char s[10]; — just memory for chars.
2. **String:** character array with a terminating '\0', e.g., char s[] = "Hello";.
3. **Operations:** Many string library functions expect the '\0' terminator; without it, functions like strlen misbehave.
4. **Example:** char arr[5] = {'a','b','c','d','\0'}; is a string; char arr[5] = {'a','b','c','d','e'}; is not a proper C string (no null terminator).

Explain any four string-handling functions with examples.

Answer (paragraph + functions + examples):
C standard library (<string.h>) provides many functions for handling strings. Here are four commonly used ones:

1. **strlen(s)** — returns length of string (number of characters before \0).
2. int len = strlen("Hello"); // len = 5
3. **strcpy(dest, src)** — copies src into dest (dest must have enough space).
4. char dest[10];
5. strcpy(dest, "Hi"); // dest now "Hi"
6. **strcat(dest, src)** — appends src to end of dest.
7. char s[20] = "Hello ";
8. strcat(s, "World"); // s now "Hello World"
9. **strcmp(s1, s2)** — compares two strings; returns 0 if equal, <0 if s1<s2, >0 if s1>s2.
10. if (strcmp("abc","abd") < 0) { /* abc < abd */ }

Notes: Always ensure destination arrays are large enough to avoid buffer overflow; prefer strncpy/strncat with size limits when necessary.

Detailed note on file handling: opening modes, reading, writing, and closing with examples.

Answer (paragraph + modes + operations + examples):
File handling allows a program to store and retrieve data persistently. In C, file handling is performed using FILE* pointers and functions such as fopen, fclose, fscanf, fprintf, fgetc, fputc, fgets, and fputs.

Opening modes (common):

1. **"r"** — open for reading (file must exist).

2. **"w"** — open for writing (creates file or truncates existing).
3. **"a"** — open for appending (write at end, creates file if not exists).
4. **"r+"** — open for read/write (file must exist).
5. **"w+"** — open for read/write, truncates file.
6. **"a+"** — open for read/append.

Basic operations:

- **Open:** FILE *fp = fopen("file.txt", "r");
- **Read:** fscanf(fp, "%d", &x);, fgets(buf, size, fp);, c = fgetc(fp);
- **Write:** fprintf(fp, "Hello %d\n", n);, fputs("text", fp);, fputc('A', fp);
- **Close:** fclose(fp); — always close files to flush buffers and free resources.

Examples:

- **Write to file:**

```
FILE *f = fopen("data.txt", "w");
if (f != NULL) {
   fprintf(f, "Hello File\n");
   fclose(f);
}
```

- **Read from file:**

```
FILE *f = fopen("data.txt", "r");
char buf[100];
if (f != NULL) {
   while (fgets(buf, sizeof(buf), f) != NULL) {
      printf("%s", buf);
   }
   fclose(f);
}
```

Good practices:

1. Always check that fopen returned non-NULL before using the file pointer.
2. Use fclose even on error paths.
3. Handle file I/O errors using feof() and ferror() if necessary.

4. Be mindful of text vs binary modes ("rb", "wb" on some systems).
5. For portability, avoid system-dependent features.

If you'd like any of these in **Gujarati**, **as a PDF**, **exam-formatted**, or **with shorter student-version summaries (bullet points only)**, tell me which format and I'll produce it immediately.