Session 4: Mastering Loops in Programming

1. Title + Big Idea (The Setup)

Topic Name: Mastering Loops in Programming

One-line purpose: To deeply understand and effectively utilize different types of loops (`for` , `while` , `do-while`) to automate repetitive tasks and control program flow in C.

2. Why Do We Need It? (The Problem)

Problem it solves: Many programming tasks involve repeating a set of instructions multiple times. Without loops, we would have to write the same code over and over, leading to lengthy, error-prone, and unmaintainable programs. Loops provide an efficient way to handle iteration.

Real-life analogy: Imagine you need to count from 1 to 100 or process every item in a shopping cart. Manually writing `print 1` , `print 2` , etc., or `process item 1` , `process item 2` , etc., is tedious. Loops are like a conveyor belt that automatically handles each item until all are processed.

3. Core Concept / Definition (The Truth)

Simple definition: Loops are control flow statements that allow a block of code to be executed repeatedly based on a given condition. They are essential for tasks requiring iteration, such as processing lists of data, performing calculations until a certain threshold is met, or creating interactive menus.

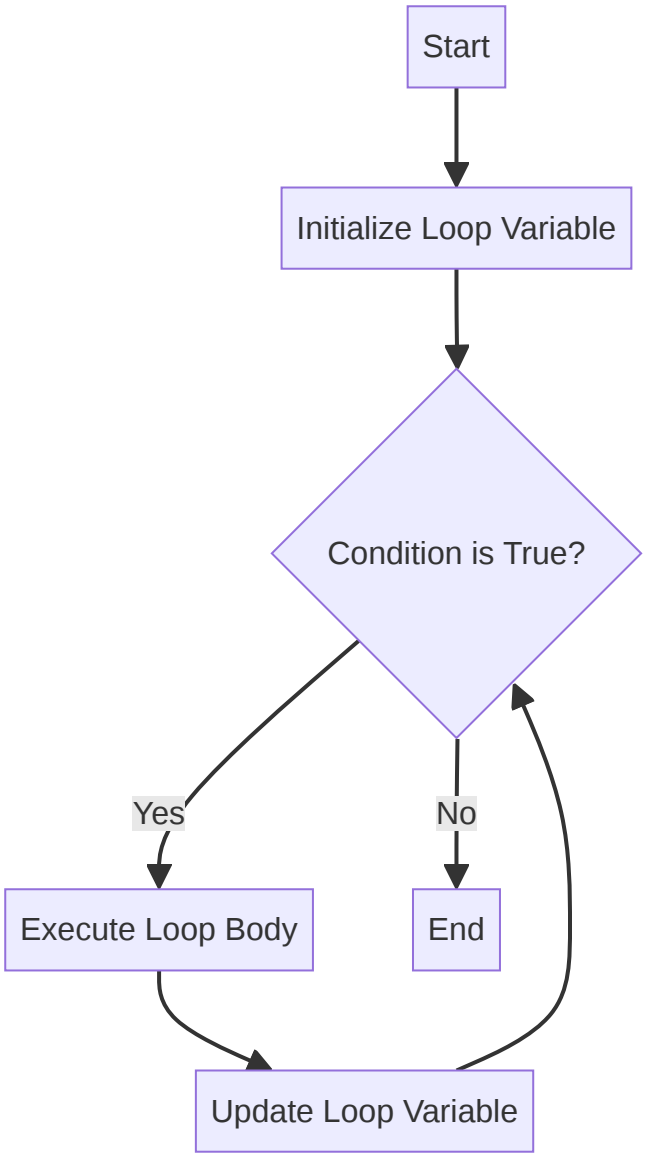
Key rule(s):

- Every loop must have a **termination condition** to prevent infinite loops.

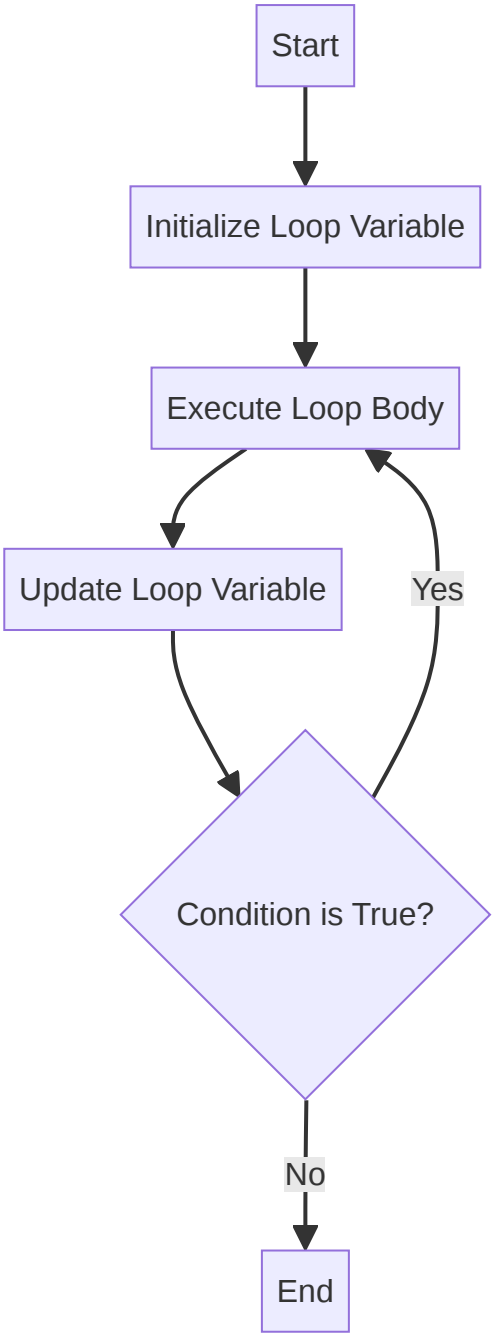
- The loop control variable must be **initialized** before the loop starts and **updated** within the loop.
- Choose the appropriate loop type (`for` , `while` , `do-while`) based on whether the number of iterations is known beforehand or if the loop must execute at least once.

4. Visual / Flow / Diagram (Show, Don't Tell)

`while` Loop Flowchart:



`do-while` Loop Flowchart:



5. Syntax / Structure (The Tool)

`for` loop: Best for when the number of iterations is known.

```
for (initialization; condition; update) {
    // code to be executed repeatedly
}
```

- **Initialization:** Executed once at the beginning (e.g., `int i = 0;`).
- **Condition:** Evaluated before each iteration; if true, the loop continues (e.g., `i < 10;`).
- **Update:** Executed after each iteration (e.g., `i++;`).

`while` loop: Best for when the number of iterations is not known beforehand, and the loop might not execute at all.

```
while (condition) {
    // code to be executed as long as condition is true
    // Loop variable update must be inside the loop body
}
```

`do-while` loop: Similar to `while`, but guarantees at least one execution of the loop body.

```
do {
    // code to be executed at least once
    // Loop variable update must be inside the loop body
} while (condition);
```

6. Example (The Action)

```c
#include <stdio.h>

int main() {
    // Example of a for loop: Print numbers from 1 to 5
    printf("For loop (1 to 5):\n");
    for (int i = 1; i <= 5; i++) {
        printf("%d ", i);
    }
    printf("\n\n");

    // Example of a while loop: Sum numbers until sum > 10
    printf("While loop (sum until > 10):\n");
    int num = 1;
    int sum = 0;
    while (sum <= 10) {
        sum += num;
        printf("Adding %d, current sum: %d\n", num, sum);
        num++;
    }
    printf("Final sum: %d\n\n", sum);

    // Example of a do-while loop: Prompt user until valid input (e.g.,
positive number)
    printf("Do-while loop (valid input):\n");
    int input;
    do {
        printf("Enter a positive number: ");
        scanf("%d", &input);
    } while (input <= 0);
    printf("You entered a valid positive number: %d\n", input);

    return 0;
}
```

7. Common Mistakes (The Conflict)

1. **Infinite Loops:** Failing to update the loop control variable or setting a condition that always remains true. This can crash the program or make it unresponsive.

2. **Off-by-One Errors:** Incorrectly setting the loop's starting or ending condition, causing the loop to run one too many or one too few times (e.g., `i < 5` instead of `i <= 5`).
3. **Misplaced Semicolon:** Placing a semicolon immediately after the `for` or `while` statement, which terminates the loop prematurely and causes the loop body to execute only once or not at all.
4. **Incorrect Loop Choice:** Using a `while` loop when a `for` loop is more appropriate (known iterations) or vice-versa, leading to less readable or less efficient code.

8. Key Points / Rules (The Takeaway)

- Loops (`for` , `while` , `do-while`) are fundamental for automating repetitive tasks.
- `for` loops are ideal when the number of iterations is known.
- `while` loops are suitable when the number of iterations is unknown, and the condition is checked before each iteration.
- `do-while` loops guarantee at least one execution of the loop body, with the condition checked after the first iteration.
- Always ensure proper initialization, condition, and update of loop control variables to avoid infinite loops.
- `break` and `continue` statements can alter loop behavior (exit loop, skip iteration).

9. One-Line Summary (The Ending)

Loops are powerful programming constructs that enable efficient repetition of code blocks, making programs dynamic and capable of handling iterative tasks with precision and control.

10. (Optional) Practice Trigger

Question: Write a C program using a `for` loop to calculate the factorial of a number entered by the user.

Variation Idea: Modify the factorial program to use a `while` loop instead of a `for` loop, and add input validation to ensure the user enters a non-negative number using a `do-while` loop.