LOOPS IN C

MOHD ADIL
ASSISTANT PROFESSOR

WHAT IS LOOP

- A loop is used to **repeat an action.** When we say something is 'in a loop,' it means it's happening over and over again.
- In programming, we use loops when we want to **execute a block of statements repeatedly.**

Types of loop

While Loop

Do while Loop

For Loop

While-loop

- The while loop allows a code block to be executed **repeatedly as long as a specified condition remains true.**
- The mechanism of a while loop is that it will **keep executing as long as its condition is true.**
- This is where it differs from an if-else statement.
- In an if-else statement, the code is executed **only once** if the condition is true.
- However, with while loop, the block of statements will execute **again and again** as long as the condition remains true.

Syntax:

```
while (condition) {
    // code to be executed
}
```

While is a **keyword.**

The word 'while' literally means 'as long as' or 'until,' which gives you a hint about its purpose.

After while keyword, we specify a **condition.**

If this condition is true, the statements inside the curly braces will execute.

The loop will continue to execute these statements until the condition becomes false.

Once the condition is false, the loop will terminate, and the program will continue with the next line of code.

Example

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

This example demonstrates the three key components of a while loop

- **Initialization**: A starting value for the variable that will be used in the condition.
- **Condition**: The test that determines whether the loop should continue or terminate.
- **Iteration**: The change (in this case, i++) that moves the loop closer to its termination condition.

In this code, we start by declaring a variable I and initializing it with the value **1**. This is called the **initialization step**.

Next, we encounter the while loop.

The first thing the loop does is check its **condition**: is i less than or equal to 5 ? Since 1 is indeed less than or equal to 5 the condition is **true**.

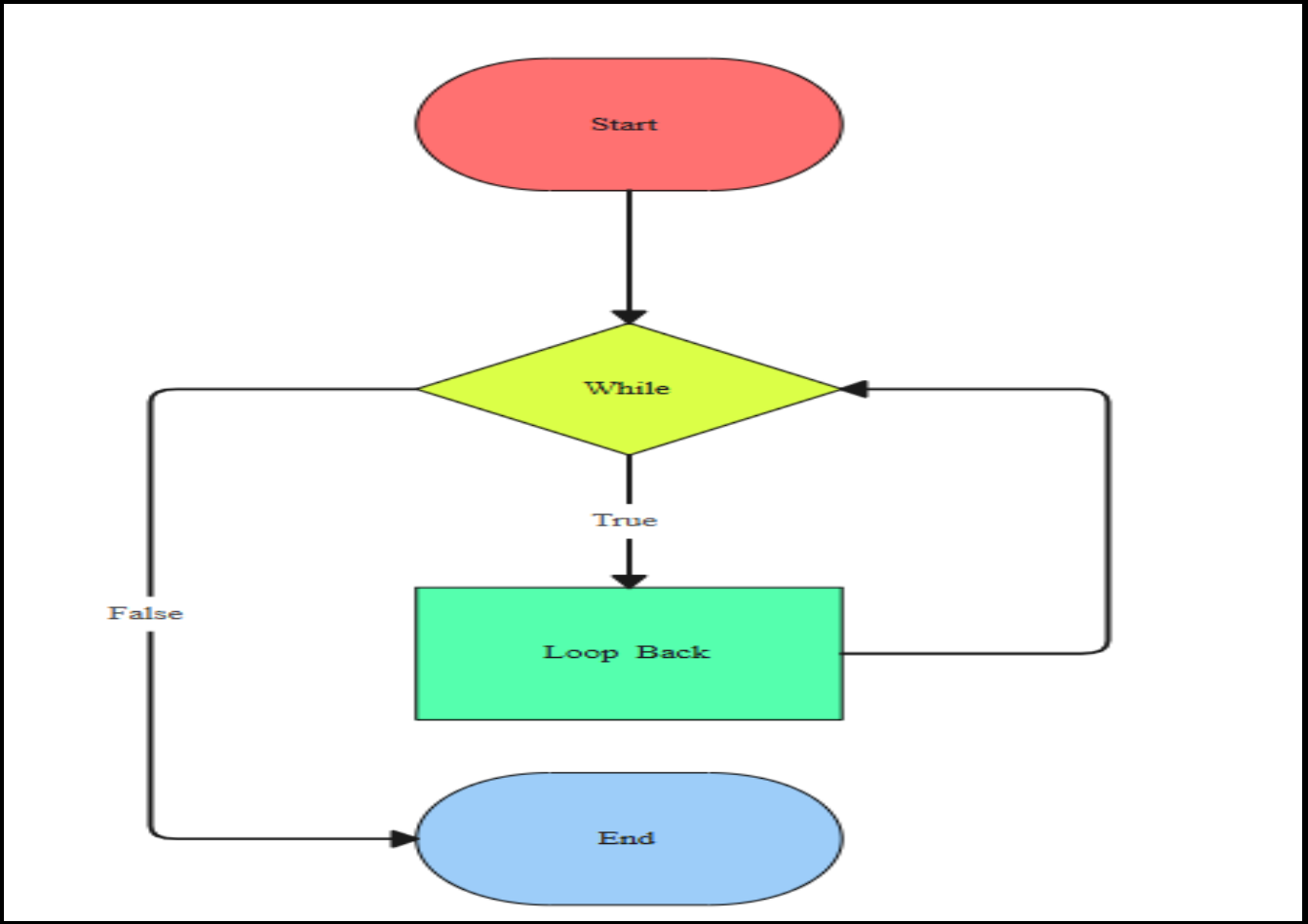
Because the condition is true, the code inside the curly braces {……} will **execute**.

The printf function will run, printing the current value of i which is 1.

Then, the i++ statement will **increment** the value of i by one, i so now becomes 2

when the condition 6<=5 is **false**. At this point, the loop **terminates**. The program skips the code inside the curly braces and continues with any statements that come after the loop.

Flow chart



while loop common mistakes

1. Forgetting to Update Loop Control Variable: If variable in the condition is not updated in the loop body, the loop may become infinite.
2. Incorrect Condition: Incorrect conditions can lead to unexpected behavior.
3. Uninitialized Loop Control Variable: Uninitialized loop control variable cause undefined behavior.

While loop properties

1. Decrementing the Loop Variable: Instead of incrementing the loop variable, it can be decremented.
2. Floating-Point Loop Variable: Loop variable need not be integer.

It can be a float.

3. Loop with Multiple Conditions: Multiple conditions can be applied using the logical operators.
4. Misplaced Semicolon: Adding a semicolon after the while statement ends the loop, resulting in unexpected behavior.

Do -While loop

1. A do…while loop is a type of looping (iteration) statement.
2. Unlike the while loop which checks the condition before executing the loop, the do...while loop checks the condition after executing the code block.
3. ensuring that the code inside the loop is executed at least once, even if the condition is false from the start.
4. Semicolon is required at the end.

Do -While loop

Syntax:

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

- The **statements inside do {...} block** are executed first.
- Then the **condition in while(condition)** is checked.
- If the condition is **true**, the loop runs again.
- If the condition is **false**, the loop stops.
- **Key Point**: Unlike the while loop, the condition is **checked after execution**, so the block executes **at least once**.

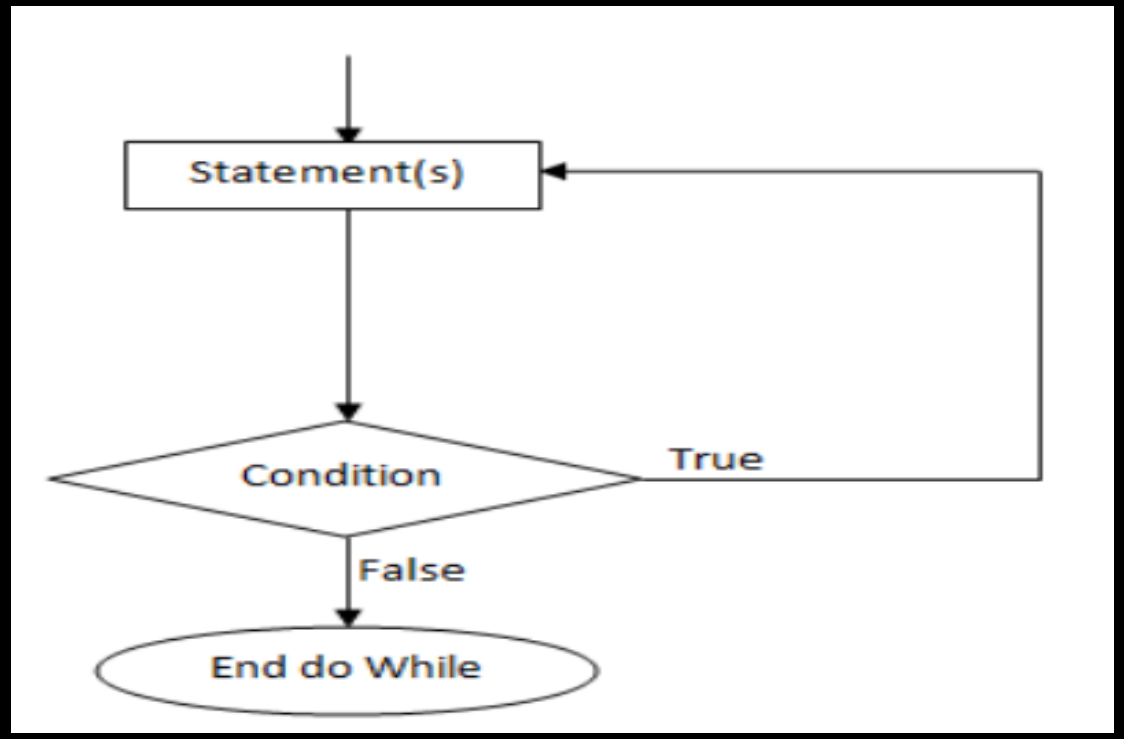
Do -While loop

Example

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

- int i=1; variable i is initialized with value **1**
- do {…} starts the **do…while loop block**.
- **printf("%d\n", i);** prints the current value of I
- i++ increases the value of i by **1** after printing.
- **while(i <= 5);** condition is checked **after execution** of the block.
- **If i** is **less than or equal to 5,** the loop repeats.
- **If i** becomes **greater than 5,** the loop stops.

Do -While loop flow chart



for loop

- Allows executing a code repeatedly for a specific number of times.
- We've already learned that a while loop is used to execute a block of code repeatedly.
- The for loop does the same thing, but it's typically used when you know the **exact number of times** you want to repeat the code.
- This distinction is a crucial difference between for and while loops. While both can perform the same tasks, the for loop is designed for situations where the number of iterations is predetermined.
- A for loop executes a code repeatedly for a specific number of times. It allows us to combine the three essential components of a loop—initialization, condition, and update—into a single, compact statement.
- This highlights the main advantage of a for loop: its structured and concise syntax.

for loop

- Syntax:
- for (initialization; condition; update) {
- // code to be executed
- }

Here for is a **keyword**. Inside the parentheses, we have three distinct statements separated by semicolons.

- **Initialization:** This statement is executed only **once** at the very beginning of the loop. It is used to declare and initialize a loop control variable.
- **Condition:** This is a **conditional expression** that is checked before each iteration. If the condition is True the loop continues. If it's false the loop terminates.
- **Update:** This statement is executed **after** each iteration of the loop body. It's typically used to increment or decrement the loop variable.

The structure of a for loop is very different from a while loop.

In for loop, all the loop control components are grouped together at the top, which makes it easier to read and understand a loop's behavior

for loop

- We have seen and learned about the for **loop**. Now, let's understand it with an example:

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

Here, for is a keyword. Inside the parentheses, we write three things separated by semicolons:

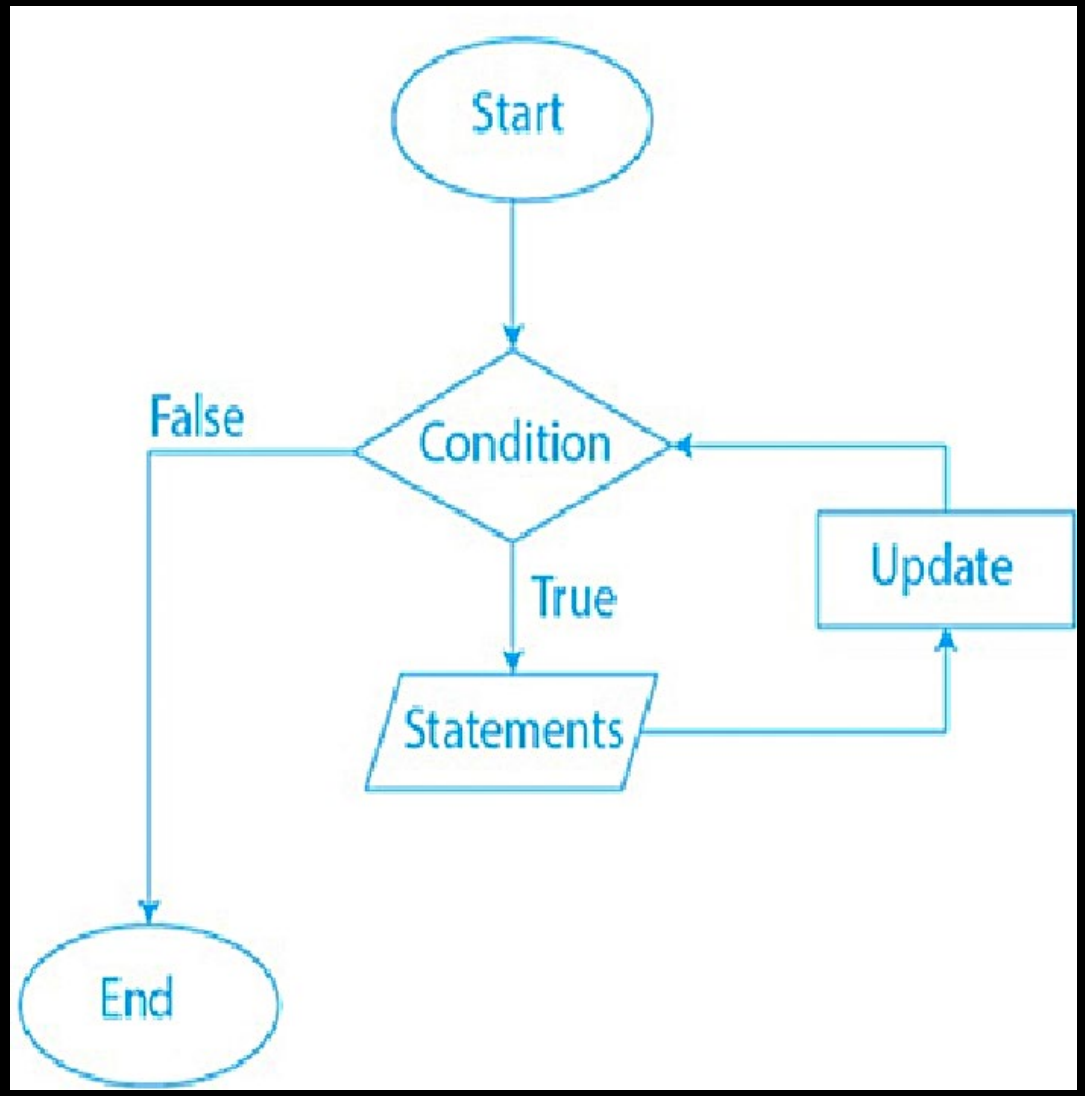
- **Initialization statement: int i=1**
- **Conditional expression: i<=5**
- **Update statement: i++**
- The code inside the curly braces **{……}** is called the **loop body.** In this case, it contains a single printf

Function.

If we look closely, the structure of a for loop is very different from a while loop but they work in a very similar way.

- **Initialization i=1 the** statement is executed **only once** at the beginning. This sets the initial value of i.
- **Condition Check:** The conditional expression i<=5 is checked. Since 1 is less than or equal to 5, the condition is **true**
- **Loop Body Execution:** Because the condition is true, the code inside the loop body is executed.
- The printf("%d",i) function is called the value of I (which) is 1) is printed to the screen.
- **Update:** After the loop body finishes, the update statement i++ is executed. This increments the value of I to 2.
- This process (steps 2, 3, and 4) repeats as long as the condition i<=5 remains true.
- When i becomes 6, the condition 6<=5 is false and the loop terminates.
- The program then moves on the code after the loop .

Flow chart



Multiple Variables in for loop

multiple variables can be used in the initialization and update sections, separated by commas.

Example

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

Break and Continue keyword

- The Break keyword is used to **terminate** a loop or switch statement immediately.
- When break is executed, the control comes **out of the loop.**
- Mostly used when you want to **stop the loop early** if a certain condition is met.

Example:

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

Output:
1 2

- Similar to break but instead of terminating from the loop, it forces to execute the next iteration of the loop.
- The continue keyword is used to **skip the current iteration** of the loop.
- The loop does not terminate; it moves to the **next iteration.**
- Useful when you want to **ignore some values** in the loop.

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue;  // skip printing when i = 3
    }
    printf("%d\n", i);
}
```

Output:
1 2 4 5

DIFFERENCE BETWEEN WHILE , DO WHILE , FOR

Aspect	while loop	do-while loop	for loop
Condition Checking	At the **beginning**	At the **end**	At the **beginning**
Minimum Execution	May execute **0 times**	Executes **at least once**	May execute **0 times**
Syntax	while(condition) { … }	do { … } while(condition);	for(init; condition; update) { … }
Initialization	Done **before** loop	Done **before** loop	Done **inside** loop header
Update/Increment	Usually inside loop body	Usually inside loop body	Usually inside loop header
Best Use Case	When number of iterations is **unknown**	When loop must run **at least once**	When number of iterations is **known/fixed**

Nested loops

- A **nested loop** means placing one loop **inside another loop**.
- The **outer loop** controls how many times the **inner loop** will run.
- For each single execution (iteration) of the outer loop, the entire inner loop runs completely.
- They are used when we need to perform **repeated tasks in multiple levels** (like rows and columns in a table).
- Commonly used in **patterns printing, 2D arrays, and matrix problems.**

- syntax:

```
for(initialization; condition; increment/decrement) {      // Outer loop
    for(initialization; condition; increment/decrement) {  // Inner loop
        // Statements to be executed
    }
}
```

Nested loops

- Syntax while loop

```
while(condition1) {              // Outer loop
    while(condition2) {          // Inner loop
        // Statements to be executed
// increment/decrement
    }
// increment/decrement
}
```

- Syntax do while

```
do {                             // Outer loop
    do {                         // Inner loop
        // Statements to be executed
    } while(condition2);
} while(condition1);
```

Outer loop controls the number of times the inner loop will run.

Inner loop executes completely for each outer loop iteration.

Goto statement

- The goto **statement** in C is a **jump statement** used to transfer control from one part of the program to another.
- It jumps directly to a **label** defined in the program.
- The label is a name followed by a colon **:**
- Useful for **breaking out of deeply nested loops** or skipping certain code.
- **overuse of goto is not recommended** because it makes code hard to read and maintain.

- Syntax
- goto label; // jump statement
- ...
- label: // target location
- statement;

Goto statement

- **Example:**

```
#include <stdio.h>
int main() {
    int num = 3;
     if (num == 3) {
        goto jump;   // Jump to label "jump"
    }
    printf("This will be skipped.\n");
    jump:   // Label
    printf("Jumped using goto statement.\n");

    return 0;
}
```

- Syntax
- goto label; // jump statement
- ...
- label: // target location
- statement;