What is loop

Loops in C are control structures that execute a block of code multiple times until a specific condition is met, helping to reduce code duplication and efficiently handle repetitive tasks. The three main types are for, while, and do-while.

Key Types of Loops

- **For Loop:** Best used when the number of iterations is known; combines initialization, condition, and increment/decrement in one line.

```
for(initialization; condition; increment) { // code }
```

- **While Loop:** Repeatedly executes a block of code as long as a condition is true, checking the condition *before* the loop runs.
- **Do-While Loop:** Similar to while, but checks the condition *after* the loop body runs, ensuring the code executes at least once.

This video explains how to create a for loop in C:

Common Components

- **Initialization:** Sets a counter variable, e.g., `int i = 0`.
- **Condition:** Checked before each iteration; if false, the loop stops.
- **Update:** Changes the counter variable to move toward the condition end.

Example of a For Loop

```
#include <stdio.h>
void main()
{
    for(int i = 0; i < 5; i++) {
        printf("%d ", i); // Prints 0 1 2 3 4
    }
}
```

Loops can be nested, meaning a loop can exist inside another loop. Failing to define a proper exit condition can result in an infinite loop, where the program runs forever.

A loop in C is a control structure that allows a block of code to be executed repeatedly until a specified condition is met. Loops are fundamental for automating repetitive tasks, such as printing a sequence of numbers, processing data arrays, or waiting for user input, thus making programs more efficient and reducing code redundancy.

Key Components of a Loop

1. **Initialization:** Setting a starting value for the counter or control variable (e.g., `int i = 0`).
2. **Condition:** A logical test performed to decide whether to run the loop body. If true, the body executes; if false, the loop terminates.

3. **Update (Increment/Decrement):** Changing the loop control variable after each iteration (e.g., `i++`) to eventually make the condition false.

Types of Loops in C

C provides three primary types of loops:

1. `for` Loop (Entry-Controlled)

The `for` loop is best used when the exact number of iterations is known in advance. It packs the initialization, condition, and update into a single line.

- **Syntax:** `for(initialization; condition; update) { // body }`
- **Example:**

```c
for(int i = 0; i < 5; i++) {
    printf("%d ", i); // Prints 0 1 2 3 4
}
```

2. `while` Loop (Entry-Controlled)

The `while` loop is used when the number of iterations depends on a condition that changes during execution, rather than a fixed count.

- **Syntax:** `while(condition) { // body }`
- **Example:**

C

```c
int i = 0;
while(i < 5) {
    printf("%d ", i);
    i++;
}
```

3. `do-while` Loop (Exit-Controlled)

The `do-while` loop is similar to the `while` loop, but it evaluates the condition **after** executing the body. This guarantees the loop body runs at least once.

- **Syntax:** `do { // body } while(condition);`
- **Example:**

C

```c
int i = 0;
do {
    printf("%d ", i);
    i++;
} while(i < 5);
```

Comparison of Loop Types

Feature	`for` Loop	`while` Loop	`do-while` Loop
Condition Check	Before iteration	Before iteration	After iteration
Guaranteed Exec.	0 times	0 times	At least 1 time
Best Used When	Iterations are known	Iterations are unknown	At least one execution needed

Additional Loop Concepts

- **Nested Loops:** A loop placed inside another loop, useful for working with 2D data (like rows and columns).
- **Infinite Loop:** A loop that runs endlessly because the condition never becomes false (e.g., `for(;;)` or `while(1)`).
- **Loop Control Statements:**
  - **`break`:** Terminates the loop immediately and moves control to the next statement outside.
  - **`continue`:** Skips the rest of the current iteration and jumps directly to the next iteration test.
  - **`goto`:** Transfers control to a labeled part of the program (generally discouraged).

Most programming languages including C support the **for** keyword for constructing a loop. In C, the other loop-related keywords are **while** and **do-while**. Unlike the other two types, the **for** loop is called an **automatic loop**, and is usually the first choice of the programmers.

The **for loop** is an entry-controlled loop that executes the statements till the given condition. All the elements (initialization, test condition, and increment) are placed together to form a **for loop** inside the parenthesis with the **for** keyword.

Syntax of for Loop

The syntax of the **for** loop in C programming language is –

```
for (init; condition; increment){
   statement(s);
}
```

Control Flow of a For Loop

Here is how the control flows in a "for" loop −

The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control **variables**. You are not required to put a statement here, as long as a semicolon appears.

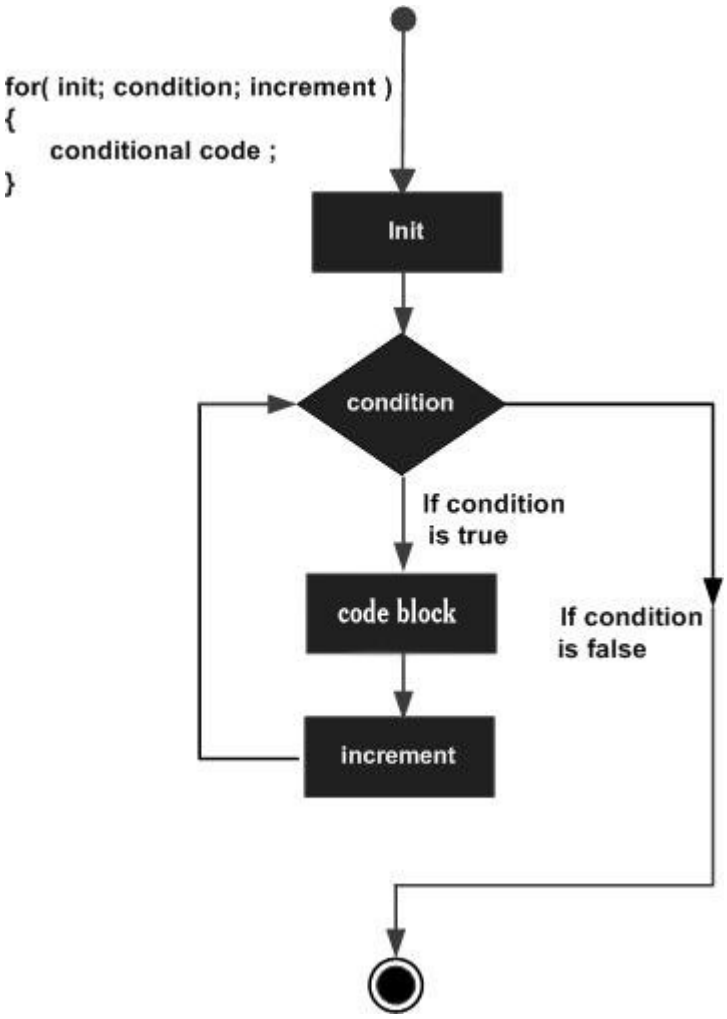
Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the control jumps to the next statement just after the "for" loop.

After the body of the "for" loop executes, the control flow jumps back up to the increment statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again the condition). After the condition becomes false, the "for" loop terminates.

Flowchart of for Loop

The following flowchart represents how the **for** loop works −



Developers prefer to use **for** loops when they know in advance how many number of iterations are to be performed. It can be thought of as a shorthand for **while** and **do-while** loops that increment and test a loop variable.

The **for** loop may be employed with different variations. Let us understand how the **for** loop works in different situations.

Example: Basic for Loop

This is the most basic form of the **for** loop. Note that all the three clauses inside the parenthesis (in front of the **for** keyword) are optional.

```
#include <stdio.h>

int main(){
   int a;

   // for loop execution
```

```
   for(a = 1; a <= 5; a++){
      printf("a: %d\n", a);
   }

   return 0;
}
```

Output

Run the code and check its output –

```
a: 1
a: 2
a: 3
a: 4
a: 5
```

Initializing for Loop Counter Before Loop Statement

The initialization step can be placed above the header of the **for** loop. In that case, the **init** part must be left empty by putting a semicolon.

Example

```
#include <stdio.h>

int main(){
   int a = 1;

   // for loop execution
   for( ; a <= 5; a++){
      printf("a: %d\n", a);
   }
   return 0;
}
```

Output

You still get the same output –

```
a: 1
a: 2
a: 3
```

```
a: 4
a: 5
```

Updating Loop Counter Inside for Loop Body

You can also put an empty statement in place of the increment clause. However, you need to put the increment statement inside the body of the loop, otherwise it becomes an **infinite loop**.

Example

```
#include <stdio.h>

int main(){

   int a;

   // for loop execution
   for(a = 1; a <= 5; ){
      printf("a: %d\n", a);
      a++;
   }
   return 0;
}
```

Output

Here too, you will get the same output as in the previous example –

```
a: 1
a: 2
a: 3
a: 4
a: 5
```

Using Test Condition Inside for Loop Body

You can also omit the second clause of the test condition in the parenthesis. In that case, you will need to terminate the loop with a **break statement**, otherwise the loop runs infinitely.

Example

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

```c
int main(){
   int a;

   // for loop execution
   for(a = 1; ; a++){
      printf("a: %d\n", a);
      if(a == 5)
      break;
   }
   return 0;
}
```

Output

On executing this code, you will get the following output –

```
a: 1
a: 2
a: 3
a: 4
a: 5
```

Using for Loops with Multiple Counters

There may be initialization of more than one variables and/or multiple increment statements in a **for** statement. However, there can be only one test condition.

Example

```c
#include <stdio.h>

int main(){
   int a, b;

   // for loop execution
   for(a = 1, b = 1; a <= 5; a++, b++){
      printf("a: %d b: %d a*b: %d\n", a, b, a*b);
   }

   return 0;
}
```

Output

When you run this code, it will produce the following output −

```
a: 1 b: 1 a*b: 1
a: 2 b: 2 a*b: 4
a: 3 b: 3 a*b: 9
a: 4 b: 4 a*b: 16
a: 5 b: 5 a*b: 25
```

Decrement in for Loop

You can also form a decrementing **for** loop. In this case, the initial value of the looping variable is more than its value in the test condition. The last clause in the for statement uses decrement operator.

Example

The following program prints the numbers 5 to 1, in decreasing order −

```
#include <stdio.h>
int main(){
   int a;

   // for loop execution
   for(a = 5; a >= 1; a--){
      printf("a: %d\n", a);
   }

   return 0;
}
```

Output

Run the code and check its output −

```
a: 5
a: 4
a: 3
a: 2
a: 1
```

Traversing Arrays with for Loops

For loop is well suited for traversal of one element of an array at a time. Note that each element in the array has an incrementing index starting from "0".

Example

```
#include <stdio.h>

int main(){
   int i;
   int arr[] = {10, 20, 30, 40, 50};

   // for loop execution
   for(i = 0; i < 5; i++){
      printf("a[%d]: %d\n", i, arr[i]);
   }

   return 0;
}
```

Output

When you run this code, it will produce the following output –

```
a[0]: 10
a[1]: 20
a[2]: 30
a[3]: 40
a[4]: 50
```

Example: Sum of Array Elements Using for Loop

The following program computes the average of all the integers in a given array.

```
#include <stdio.h>
int main(){
   int i;
   int arr[] = {10, 20, 30, 40, 50};
   int sum = 0;
   float avg;

   // for loop execution
   for(i=0; i<5; i++){
      sum += arr[i];
   }
   avg = (float)sum / 5;
   printf ("Average = %f", avg);
```

```
   return 0;
}
```

Output

Run the code and check its output –

```
Average = 30.000000
```

Example: Factorial Using for Loop

The following code uses a **for** loop to calculate the factorial value of a number. Note that the factorial of a number is the product of all integers between 1 and the given number. The factorial is mathematically represented by the following formula –

```
x! = 1 * 2 * . . . * x
```

Here is the code for computing the factorial –

```
#include <stdio.h>

int main(){

   int i, x = 5;
   int fact = 1;

   // for loop execution
   for(i=1; i<= x; i++){
      fact *= i;
   }
   printf("%d != %d", x, fact);

   return 0;
}
```

Output

When you run this code, it will produce the following output –

```
5! = 120
```

The for loop is ideally suited when the number of repetitions is known. However, the looping behaviour can be controlled by the **break** and **continue keywords** inside the body of

the **for** loop. Nested **for** loops are also routinely used in the processing of two dimensional **arrays**.