**Loops**

In programming, a loop is used to repeat a block of code until the specified condition is met.

You may encounter situations, when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages −



C programming language provides the following types of loops to handle looping requirements.

|  |  |
| --- | --- |
| **Sr.No.** | **Loop Type & Description** |
| **1** | [while loop](https://www.tutorialspoint.com/cprogramming/c_while_loop.htm)  Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| **2** | [for loop](https://www.tutorialspoint.com/cprogramming/c_for_loop.htm)  Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| **3** | [do...while loop](https://www.tutorialspoint.com/cprogramming/c_do_while_loop.htm)  It is more like a while statement, except that it tests the condition at the end of the loop body. |
| **4** | [nested loops](https://www.tutorialspoint.com/cprogramming/c_nested_loops.htm)  You can use one or more loops inside any other while, for, or do..while loop. |

**for Loop:**

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

The syntax of a **for** loop in C programming language is −

for ( init; condition; increment )

{

statement(s);

}

Here is the flow of control 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 flow of control jumps to the next statement just after the 'for' loop.
* After the body of the 'for' loop executes, the flow of control 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 condition). After the condition becomes false, the 'for' loop terminates.

**Flow Diagram**



**Example**

#include <stdio.h>

int main () {

int a;

/\* for loop execution \*/

for( a = 10; a < 20; a = a + 1 ){

printf("value of a: %d\n", a);

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

**Example:**

// Print numbers from 1 to 10

#include <stdio.h>

int main() {

int i;

for (i = 1; i < 11; ++i)

{

printf("%d ", i);

}

return 0;

}

**Output**

1 2 3 4 5 6 7 8 9 10

1. *i* is initialized to 1.
2. The test expression i < 11 is evaluated. Since 1 less than 11 is true, the body of for loop is executed. This will print the **1** (value of *i*) on the screen.
3. The update statement ++i is executed. Now, the value of *i* will be 2. Again, the test expression is evaluated to true, and the body of for loop is executed. This will print **2** (value of *i*) on the screen.
4. Again, the update statement ++i is executed and the test expression i < 11 is evaluated. This process goes on until *i* becomes 11.
5. When *i* becomes 11, *i < 11* will be false, and the for loop terminates.

**Example 2: for loop**

// Program to calculate the sum of first n natural numbers

// Positive integers 1,2,3...n are known as natural numbers

#include <stdio.h>

int main()

{

int num, count, sum = 0;

printf("Enter a positive integer: ");

scanf("%d", &num);

// for loop terminates when num is less than count

for(count = 1; count <= num; ++count)

{

sum += count;

}

printf("Sum = %d", sum);

return 0;

}

**Output**

Enter a positive integer: 10

Sum = 55

The value entered by the user is stored in the variable *num*. Suppose, the user entered 10.

The *count* is initialized to 1 and the test expression is evaluated. Since the test expression count<=num (1 less than or equal to 10) is true, the body of for loop is executed and the value of *sum* will equal to 1.

Then, the update statement ++count is executed and the count will equal to 2. Again, the test expression is evaluated. Since 2 is also less than 10, the test expression is evaluated to true and the body of for loop is executed. Now, the *sum* will equal 3.

This process goes on and the sum is calculated until the *count* reaches 11.

When the *count* is 11, the test expression is evaluated to 0 (false), and the loop terminates.

Then, the value of sum is printed on the screen.

**while loop:**

A **while** loop in C programming repeatedly executes a target statement as long as a given condition is true.

The syntax of a **while** loop in C programming language is −

while(condition)

{

statement(s);

}

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true.

When the condition becomes false, the program control passes to the line immediately following the loop.

## Flow Diagram



Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

## Example

#include <stdio.h>

int main () {

/\* local variable definition \*/

int a = 10;

/\* while loop execution \*/

while( a < 20 ) {

printf("value of a: %d\n", a);

a++;

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

**Example:**

// Print numbers from 1 to 5

#include <stdio.h>

int main()

{

int i = 1;

while (i <= 5)

{

printf("%d\n", i);

++i;

}

return 0;

}

**Output**

1

2

3

4

5

Here, we have initialized *i* to 1.

1. When *i* is 1, the test expression i <= 5 is true. Hence, the body of the while loop is executed. This prints 1 on the screen and the value of *i* is increased to 2.
2. Now, *i* is 2, the test expression i <= 5 is again true. The body of the while loop is executed again. This prints 2 on the screen and the value of i is increased to 3.
3. This process goes on until *i* becomes 6. When *i* is 6, the test expression i <= 5 will be false and the loop terminates.

**do...while loop:**

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

The syntax of a **do...while** loop in C programming language is −

do

{

statement(s);

} while( condition );

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

## Flow Diagram



## Example

[Live Demo](http://tpcg.io/pVAh6H)

#include <stdio.h>

int main () {

/\* local variable definition \*/

int a = 10;

/\* do loop execution \*/

do {

printf("value of a: %d\n", a);

a = a + 1;

}while( a < 20 );

return 0;

}

When the above code is compiled and executed, it produces the following result −

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

**How do...while loop works?**

* The body of do...while loop is executed once. Only then, the test expression is evaluated.
* If the test expression is true, the body of the loop is executed again and the test expression is evaluated.
* This process goes on until the test expression becomes false.
* If the test expression is false, the loop ends.

**Example 2: do...while loop**

// Program to add numbers until the user enters zero

#include <stdio.h>

int main()

{

double number, sum = 0;

// the body of the loop is executed at least once

do

{

printf("Enter a number: ");

scanf("%lf", &number);

sum += number;

}

while(number != 0.0);

printf("Sum = %.2lf",sum);

return 0;

}

**Output**

Enter a number: 1.5

Enter a number: 2.4

Enter a number: -3.4

Enter a number: 4.2

Enter a number: 0

Sum = 4.70

## Nested Loop:

C programming allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept.

The syntax for a **nested for loop** statement in C is as follows −

for ( init; condition; increment ) {

for ( init; condition; increment ) {

statement(s);

}

statement(s);

}

The syntax for a **nested while loop** statement in C programming language is as follows −

while(condition) {

while(condition) {

statement(s);

}

statement(s);

}

The syntax for a **nested do...while loop** statement in C programming language is as follows −

do {

statement(s);

do {

statement(s);

}while( condition );

}while( condition );

A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example, a 'for' loop can be inside a 'while' loop or vice versa.

## Example

The following program uses a nested for loop to find the prime numbers from 2 to 100 −

[Live Demo](http://tpcg.io/GQGXBh)

#include <stdio.h>

int main () {

/\* local variable definition \*/

int i, j;

for(i = 2; i<100; i++) {

for(j = 2; j <= (i/j); j++)

if(!(i%j)) break; // if factor found, not prime

if(j > (i/j)) printf("%d is prime\n", i);

}

return 0;

}

When the above code is compiled and executed, it produces the following result −

2 is prime

3 is prime

5 is prime

7 is prime

11 is prime

13 is prime

17 is prime

19 is prime

23 is prime

29 is prime

31 is prime

37 is prime

41 is prime

43 is prime

47 is prime

53 is prime

59 is prime

61 is prime

67 is prime

71 is prime

73 is prime

79 is prime

83 is prime

89 is prime

97 is prime