

Chapter 4 : Loops

1. Introduction

A loop is a control structure aimed at executing a set of instructions repeatedly multiple times. It is executed based on either a known number of iterations in advance (iterative loop) or until a condition allows the loop to exit (conditional loop). There are three types of loops:

- Conditional loop with a pre-condition: The condition is checked before the first iteration.
- Conditional loop with a post-condition: The condition is checked after the first iteration.
- Iterative loop: A counter keeps track of the number of iterations.

A programming error can lead to a situation where the exit condition is never satisfied. This results in the program running indefinitely, which is called an infinite loop.

2. The "While" Loop

The "While" loop is a pre-condition loop where a set of instructions is repeatedly executed based on a Boolean condition. The "while" loop can be seen as a repetition of the "if" statement. It is used when there's a set of instructions that need to be repeated with the possibility that they may not be executed at all (0 or more times), depending on the predefined condition. The loop consists of two parts:

- Condition: This is a logical expression with a value of either true or false.
- Instruction Block: It is executed as long as the condition is true.

2.1.Syntax

Algorithm	C
While Condition Do Instruction Block EndWhile Rest of the program	while (Condition) { Instruction Block } Rest of the program

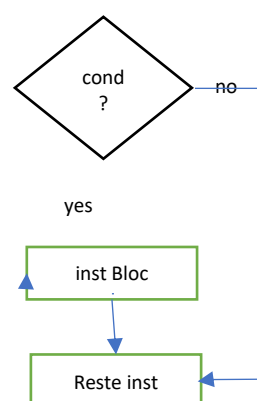
"While" "do" and "EndWhile" are reserved keywords in algorithms. Similarly, in C, the term used is "while." The condition is always placed between the keywords "While" and "do" in algorithms, while in C, it's always enclosed in parentheses. To create the condition, we use comparisons (>, <, =, ≠, ...) and logical operations (and &&, or ||, not !, ...).

In C, the instructions for the "while" loop are enclosed in curly braces {}, and they can be omitted if they contain only a single instruction (optional {}). If we encounter a set of instructions without the curly braces, it means that only the first instruction is repeated.

Notes:

- In C, the Boolean type is expressed using an **int**. False is represented by 0, and true is represented by any non-zero number.
- No ";" is needed after the closing curly brace "}".

2.2. Flowchart:



2.3. Execution

The execution process of the "While" loop begins by evaluating the condition expression, which results in a Boolean value. If the result is true, the instruction block between "Do" and "EndWhile" in the algorithm, or between the curly braces in C, is executed. The condition is re-evaluated, and the process repeats. When the test result becomes false, the loop exits, jumping to the instruction immediately following the loop. The condition is often referred to as the "continuation condition."

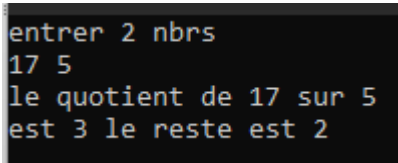
Notes:

Since the **While** loop checks the condition **before** the first iteration, it's possible that the condition isn't checked the first time, and so its instructions aren't executed at all.

2.4. Example

Write a program that reads two integers and then displays the quotient of the first divided by the second, without using the division operator (/ or div).

Note: Division is repeated subtraction.

Algorithm	C	screen
<pre> algorithm quotient var x, y, q, r : integer /* x first nbr, y second, q quotient, r remainder */ begin write ("enter 2 nbrs ") read (x, y) q←0 r←x while r>y Do r←r-y q←q+1 End while write ("the quotient of", x, "on", y, "is", q, "the remainder is", r) end </pre>	<pre> #include <stdio.h> int main() { int x, y, q, r ; printf("enter 2 nbrs \n") ; scanf("%d%d", &x, &y) ; q=0 ; r=x ; while (r>y) { r-=y ; q++ ; } printf("the quotient of %d over %d is %d the remainder is %d\n"), x, y, q, r) ; } </pre>	 <pre> entrer 2 nbrs 17 5 le quotient de 17 sur 5 est 3 le reste est 2 </pre>

The algorithm takes two numbers x and y, and returns the quotient q and remainder r.

At the beginning, let's assume that the remainder is x, and at each iteration we decrease the nominator "y" until it becomes less than the denominator. Each time we decrease "y", we add 1 to the quotient q. The **while** loop can never be executed if x is less than y from the start. In this case q=0 and r=x.

Example 2

Write a program that doesn't stop until the user presses the Enter key.

```

#include <string.h>
int main()
{
  while (getchar() != '\n');
  return 0;
}

```

3. The "Do...While" Loop

The "Do...While" loop is a post-condition loop where a set of instructions is repeatedly executed based on a Boolean condition. It's used when a set of instructions needs to be executed repeatedly at least once, regardless of the condition (1 or more times). The loop consists of two parts:

- **Instruction Block:** It's executed as long as the condition is true, except for the first time when it's executed regardless of the condition.
- **Condition:** A Boolean expression with a true or false value.

3.1.syntax:

Algorithm	C
-----------	---

Do Instruction Block while Condition Rest of the instructions	do { Instruction Block } while (Condition) ; Rest of the instructions
--	---

The words " **do** " and " **while** " are reserved keywords in algorithms and C. The condition always comes after " **while** " in algorithms and is always enclosed in parentheses () in C. To construct the condition, we use comparison operations (>, <, =, ≠, ...) and logical operations (and &&, or ||, not !, ...).

The instructions for "do...while" loops in C are enclosed in two curly braces {} within the "do" and "while" statements. The curly braces {} can be omitted if they contain only a single instruction (optional {}).

Observation :

- The "do...while" loop in C always ends with a semicolon ";".

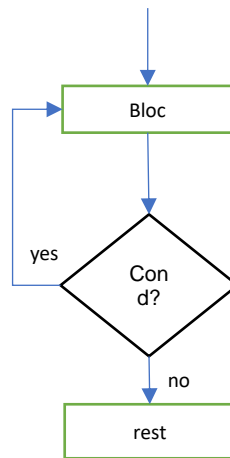
The " do...while " construct can be expressed in algorithmic as "Repeat...Until" (until the condition is satisfied). In this case, the condition becomes a termination condition, not a continuation condition, which is the negation of the " while" loop condition.

Example :

do...while	Repeat...Until
Do Instruction block while x>y The rest of the instructions	Repeat Instruction block Until x≤y The rest of the instructions

Note that the negation of > is ≤.

3.2. Flowchart:



3.3. Execution

The execution process of the conditional loop "Do...While" involves executing the instruction block between "Do" and "While" in the algorithm, or between "do" and "while" in C. After that, the condition expression is calculated, resulting in a Boolean value. If the result is true, the instruction block is executed again, and the process continues until the test result becomes false, at which point the loop exits, jumping to the instruction immediately following the loop.

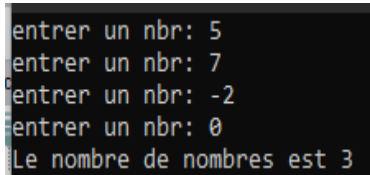
Observation :

Since the "Do... While " loop executes the instructions of the first iteration **before** the condition is verified, the loop executes at least one iteration, even if the condition is not satisfied from the start.

3.4. Example:

Write a program that reads a set of integers using a single variable, stops at the first 0 that reads it, then displays the number of integers entered.

Algorithm	C	Screen
------------------	----------	---------------

<pre> algorithm readNbrs var x, nb : integer /* x to read nbrs, nb to count nbrs */ begin nb←0 Do write ("enter a nbr ") read (x) nb←nb+1 while x≠0 write("The number of numbers is", nb-1) end </pre>	<pre> #include <stdio.h> int main() { int x, nb ; nb=0 ; do { printf("enter a nbr ") ; scanf("%d", &x) ; nb++ ; } while (x!=0) ; printf("The number of numbers is %d"), nb-1) ; } </pre>	 <pre> entrer un nbr: 5 entrer un nbr: 7 entrer un nbr: -2 entrer un nbr: 0 Le nombre de nombres est 3 </pre>
---	--	--

The algorithm needs variable *x* to read the numbers and variable *nb* to count the numbers. We set *nb* 0 as the initial value, then enter a number *x* and add 1 to *nb*. If *x* is 0, we stop, otherwise we repeat the loop until the user enters the number 0. The loop will run at least once. Finally, we show the value of *nb-1* so that the number 0 is not counted.

4. The "for" loop

The "For" loop is an unconditional iterative loop where a set of instructions is executed iteratively a predetermined number of times. The loop consists of two parts:

- Counter: Used to count the number of iterations. It's a variable of integer or character type, with an initial value, a final value, and a method of incrementing or decrementing.
- Instruction Block: Executed in each iteration.

4.1.Syntax (Algorithm)

Algorithm
For Counter ←Initial_Value To Final_Value Step Step_Value Do Instruction Block EndFor Rest of the instructions

The words **For**, **To**, **Step**, **Do** and **EndFor** are reserved words in the algorithm.

- Counter: A name for an integer or character variable.
- Initial_Value: This is the initial value taken by the counter variable.
- Final_Value: This is the final value the counter variable can take.
- Step_Value: This is the value of the counter variable at the end of each iteration. where $Counter \leftarrow counter+step$. Generally equal to 1.

Observations :

- The Final_Value termination value is calculated once before the loop is executed.
- The (**Step** Step_Value) part is optional and, in its absence, means that Step_Value is 1.
- If Step_Value is positive, it is added to *counter* until $counter \geq Final_Value$. In the case of a negative Step_Value, it is decremented to $counter \leq Final_Value$.
- $counter = Final_Value$ is executed.
- If Initial_Value is greater than Final_Value and Step_Value is positive, the **for** loop is not executed.
- If Initial_Value is less than Final_Value and Step_Value is negative, the **for** loop is not executed.
- The *counter* value cannot be modified inside the loop.

4.2. Syntax « for » in C

The "for" loop in C is more general than the "for" loop in the algorithm. It's closer to the "while" conditional loop than to the " for " loop.

The general form	Its algorithm equivalent
------------------	--------------------------

<pre> for (initialization; test ; iteration) { Instruction block } The rest of the instructions </pre>	<pre> for (Counter = Initial_Value; Counter <= Final_Value; Counter += Step_Value) { Instruction block; } The rest of the instructions </pre>
---	---

for is a reserved word in C.

The first line of **for** consists of three parts enclosed in parentheses (), all optional, separated by a semicolon ";".

- **Initialization** : This part is executed once before the loop is executed. It is generally used to assign an initial value to the counter. Ex : `i=0`
- **condition**: a Boolean expression. Its value must be true to execute the loop. If the condition is false, the loop is exited. It is evaluated at the start of each loop iteration. Usually, the counter is tested. Like: `i<10`.
- **Iteration**: It is executed at the end of each iteration. It is generally used to increment or decrement the counter. Like : `i++` or `i--`.

C "for" statements are enclosed in {} . They can be omitted if they contain only one instruction ({} optional). If we find a set of instructions after **for** and we don't find the two braces, only the first instruction is repeated.

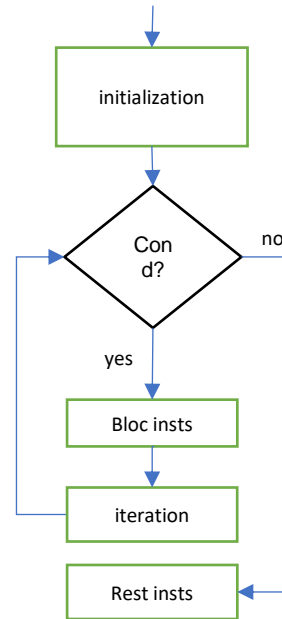
Notes:

- The variable (the counter) can be declared in the initialization part, in which case the scope of its definition is only inside the **for** loop, not outside it.
- The counter value in the iteration section can be incremented or decremented, or modified in any other way.
- All "for" parts (initialization, test, iteration) are optional, and can be omitted and left empty. but ";" is mandatory and cannot be omitted. The following script is valid **for** (; ;)
- The initialization part and the iteration part can contain several instructions separated by commas ','.
- The instruction ";" is the empty instruction.

The following example codes are equivalents

<pre> int i=0; j=10; for (; ;){ if (!(i<j)) break; i++; j--; } </pre>	<pre> for (int i=0,j=10 ;i<j ; i++,j--); </pre>
--	---

4.3. Flowchart:



4.4. Execution:

The execution process of the "For" loop involves assigning the initial value to the counter variable. If the counter's value is less than or equal to the final value (for positive steps) or greater than or equal to the final value (for negative steps), the instruction block between "Do" and "EndFor" (algorithm) or between curly braces in C is executed. After each iteration, the counter is incremented or decremented by the step value. The process continues until the counter value no longer satisfies the condition, at which point the loop exits, jumping to the instruction immediately following the loop.

In C, the initialization expression is only executed once before the loop is executed. The command then passes to the condition. It is tested before each iteration. If the result is true, it executes the block of instructions between {} in C, then executes the iteration part, then re-evaluates the test and starts again. The iteration part is executed at the end of each iteration. When the test result becomes false, we exit the loop by jumping to the instructions immediately following it.

4.5. Example :

Write a program that reads two integers and then displays all the integers in between.

Algorithm	C	ecran
<pre> algorithm numbers var x, y, i : integer /* i is the counter */ begin write ("enter 2 nbrs ") read (x, y) for i←x to y Do write (i) End for end </pre>	<pre> #include <stdio.h> int main() { int x, y, i ; printf("enter 2 nbrs \n") ; scanf("%d%d", &x, &y) ; for (i=x ; i<=y ; i++) printf("%d\t", i) ; return 0 ; } </pre>	

The algorithm takes two numbers x and y, and needs a variable i, which acts as a counter. Where it takes successive values from the interval x to y. At the end of each iteration, 1 is added to counter i. Since the step is implicitly 1. If x is greater than y, no number is displayed. In C, it must be written.

{ } has been omitted from the **for** loop because it contains only one instruction.

5. Nested Loops

A loop can contain any type and number of instructions, including another loop. When a loop is inside another, it's called a nested loop. In nested loops, the execution proceeds as follows:

- Enter the outer loop

- Enter the inner loop
- Execute the inner loop until it's finished
- Return to the outer loop to execute the remaining instructions
- Repeat the process of executing the outer loop until it's finished.

Example

Write the program that reads the number of lines n, then displays on the screen in the first line *, in the second **, in the third ***, and so on until it displays in the last line n *.

Algorithm	C	الشاشة
<pre> algorithm asterisk var n, i, j : integer /* i, j counters */ début write("enter no. of lines") read(n) for i←1 to n Do for j←1 to i Do write("**") EndFor EndFor end </pre>	<pre> #include <stdio.h> int main() { int n, i, j ; printf("enter no. of lines") ; scanf("%d", &n) ; for (i=1 ;i<=n ;i++) { for (j=1 ;j<=i ;j++) printf("**") ; printf("\n") ; } return 0 ; } </pre>	

The external for(i) contains two instructions: for(j) and printf("\n"). The internal for(j) contains a single instruction, printf("**"). printf("\n") is repeated n times. printf("**") is repeated 1+2+...+n times.

6. Loop Equivalence

- The "While" loop is used when the number of iterations is unknown in advance and when there's a possibility of not executing the instruction block at all.
- The "Do...While" loop is used when the number of iterations is unknown in advance, and the instruction block must be executed at least once.
- The "For" loop is used when the number of iterations is known in advance, or when the starting and ending values of the counter range are known.
- In general, any "While" loop can be expressed using "Do...While" by adding a condition before "Do," and any "Do...While" loop can be expressed using "While" by adding the instruction block before "While." Any "For" loop can be expressed using "While" by initializing the counter before the loop, using the final value as the exit condition, and adding the instruction that modifies the counter's value at the end of the loop. However, it's not always possible to express "While" or "Do...While" loops using "For," unless there's a counter involved.
- In C, "While" or "Do...While" loops can be expressed with "For," and all loops can be expressed using "Goto" and "If."

Examples :

while

while	do...while	for	goto +if
<pre> ... r=x ; while (r>y) { r-=y ; q++ ; } printf(...) ; } </pre>	<pre> ... r=x ; if (r>y) do { r-=y ; q++ ; } while (r>y) printf(...) ; } </pre>	<pre> ... r=x ; for (;r>y;) { r-=y ; q++ ; } printf(...) ; } </pre>	<pre> ... r=x ; again : if (r>y) { r-=y ; q++ ; goto again ; } printf(...) ; } </pre>

do...while

do...while	while	for	goto +if
------------	-------	-----	----------

<pre> ... nb=0 ; do { printf("entrer un nbr") ; scanf("%d", &x) ; nb++ ; } while (x!=0) ; printf(...) ; } </pre>	<pre> ... nb=0 ; printf("entrer un nbr") ; scanf("%d", &x) ; nb++ ; while (x!=0) { printf("entrer un nbr") ; scanf("%d", &x) ; nb++ ; } printf(...) ; } </pre>	<pre> ... nb=0 ; printf("entrer un nbr") ; scanf("%d", &x) ; nb++ ; for (;x!=0 ;) { printf("entrer un nbr") ; scanf("%d", &x) ; nb++ ; } printf(...) ; } </pre>	<pre> ... r=x ; again : printf("entrer un nbr") ; scanf("%d", &x) ; nb++ ; if (r>y) goto again ; printf(...) ; } </pre>
--	--	---	--

for

for	while	do...while	goto +if
<pre> ... for (i=x;i<=y;i++) printf("%d\t",i); ... </pre>	<pre> ... i=x ; while (i<=y){ printf("%d\t",i); i++ ; } ... </pre>	<pre> ... i=x ; if (i<=y) do { printf("%d\t",i); i++ ; } while (i<=y) ... </pre>	<pre> ... i=x ; again : if (i<=y) { printf("%d\t",i); i++ ; goto again ; } ... </pre>

7. Loop Termination Commands

These commands are used within loops to perform an early exit from the loop based on a condition. They are generally used when checking a condition. Any "for," "while," or "do...while" loop can be terminated by executing any jump instruction like "break," "return," or "goto" (to a label outside the loop). The "continue" instruction only ends the current iteration and proceeds to the end of the loop, starting the next iteration. These instructions are often used within an "if" statement. In the case of nested loops, "break" and "continue" only exit the inner loop.

Example :

<pre> for (int i=1 ;i<10 ;i++){ if(i%3==0) continue ; printf("%d\t", i) ; } </pre>	<pre> for (int i=1 ;i<10 ;i++){ if(i%3==0) break ; printf("%d\t", i) ; } </pre>
<p>All numbers will appear except for multiples of 3: 1 2 4 5 7 8</p>	<p>The loop stops at the first multiple of 3: 1 2</p>