Chapter 4 : Loops

Dr. Abderrahmane Kefali

Senior Lecturer Class A,
Department of Computer Science,
University of May 8, 1945 - Guelma

kefali.abderrahmane@univ-guelma.dz

This document is prepared for printing two pages per sheet

1) Introduction

We have already seen that the sequential flow of instructions is not sufficient to solve the problems encountered in everyday life. Fortunately, it is possible to break the sequential flow using what are called control structures.

In the previous chapter, we described the first problem that cannot be solved by simple sequential instructions: the problem where we have multiple cases, and each case requires separate handling. This type of problem was resolved using conditional structures.

Another problem frequently encountered in everyday life is the need to repeat a task multiple times. Indeed, this repetitive processing can be accomplished using labeled branches, but a program that uses labels is difficult to maintain. The ideal solution for implementing repetitive processes is the use of special control structures called loops. These structures allow the repetition of an instruction or a sequence of instructions a certain number of times, which may be known in advance or not.

In this chapter we will introduce the notion of repetition of a sequence of instructions and present the different existing loops in algorithms.

2) What is a loop?

2.1) Definition

Loops, also known as ***repetitive or iterative structures***, are structures that allow the same sequence of instructions to be repeated several times with different values for a finite number of times.

During each repetition, the instructions within the loop are executed, forming what is called a ***loop cycle*** or ***iteration***.

Iteration stops after reaching a termination condition, which is expressed either by a logical expression or by a predefined number of iterations.

However, there are three variants of repetition, and for each variant, algorithmics (and most programming languages) offers a specific type of loop:

- Repeating a block of instructions a given number of times (**`FOR`** loop).
- Repeating a block of instructions as long as a condition is met (**`WHILE`** loop).
- Repeating a block of instructions until a condition is met (**`REPEAT`** loop).

2.2) Components of a loop

A loop consists of four essential elements:

- A ***block of instructions***, which will be executed a certain number of times.
- A ***condition***, similar to conditional instructions. This condition relates to at least one variable, referred to as the loop variable. There can be multiple loop variables for a single loop.
- An ***initialization***, which concerns the loop variable. This initialization can be directly performed by the loop statement or left to the programmer.
- A ***modification***, which also concerns the loop variable. Similar to initialization, it can be integrated into the loop statement or left to the programmer.

3) WHILE loop

The **`WHILE`** loop allows to repeatedly execute an instruction or sequence of instructions as long as a condition is met. When the condition becomes false, the loop terminates. The condition is expressed in the form of a variable or logical expression.

This loop is particularly useful when the number of iterations is not known in advance.

3.1) Algorithmic syntax

The syntax of the **`WHILE`** loop is as follows:

```
WHILE <Condition> DO
    <block of instructions>;
```

The progression of the **`WHILE`** loop involves successively and repeatedly the following steps. First, the entry condition to the loop is evaluated. If it is

satisfied, the body of the loop (the block of instructions) is executed, and we return to evaluate the condition again. This process continues until the condition is no longer satisfied. In this latter case, the instructions within the block are not executed, and the algorithm proceeds to the next instruction just after the block.

Example:

Write an algorithm that allows to enter a person's age via the keyboard and to repeat the entry as long as the value entered by the user is incorrect.

Solution :

```
ALGORITHM input_age;
VAR age : Integer;
BEGIN
Write("Enter the age: ");
Read(age);
WHILE age ≤ 0 DO
     Begin
     Write("Invalid age, re-enter the age:");
     Read(age);
     End;
Write("Valide age ");
END.
```

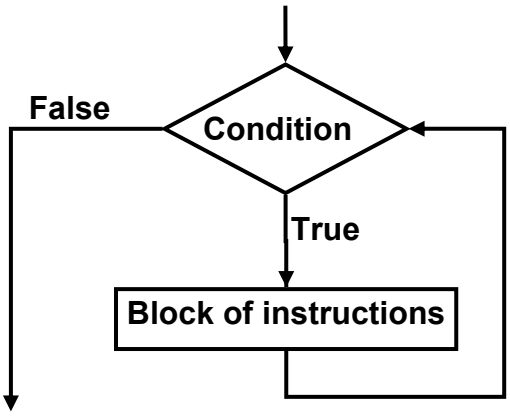
Remarks :

- The condition can be simple or compound.
- Note the absence of a semicolon after the condition and after **DO**.
- In this loop, the condition is tested before entering the loop. Therefore, the block of instructions that forms the body of the loop may never be executed; this happens when the condition is false from the beginning.
- The parameters of the condition must be initialized by reading or assignment before the loop, so that, on the first pass, the condition can be evaluated.
- In the block of instructions, it is imperative to have an action that modifies the condition parameters in such a way that the condition becomes false at some point, otherwise, if the condition remains true, you end up in an infinite loop.

- If the block of instructions to be repeated contains multiple statements, it must be enclosed by **`BEGIN`** and **`END`**.
- The **`WHILE`** loop is the most generic loop. It can be used whether the number of repetitions is known in advance or not.

3.2) Flowchart

The formalism of the **`WHILE`** loop in a flowchart is as follows.



3.3) C language syntax

The syntax of the **`WHILE`** loop in the C language is as follows:

```
while(condition)
    <block of instructions>;
```

Remarks:

- The condition must be enclosed in parentheses.
- There is no semicolon after the condition.
- If the block of instructions consists of multiple statements, it must be enclosed in curly braces (**`{`** and **`}`**).

Example:

Here is the C program that repeats reading the age until a valid age is entered:

```
#include<stdio.h>
main(){
    int age;
    printf("Enter the age: ");
    scanf("%d",&age);
    while(age <= 0){
        printf("Invalid age, re-enter the age:");
        scanf("%d",&age);
    }
    printf("Valid age");
}
```

4) REPEAT loop

The **REPEAT** loop allows to repeat the execution of a block of instructions until a condition is met. As with the **WHILE** loop, **REPEAT** is a generic loop that doesn't require knowing the number of iterations in advance. However, unlike **WHILE**, the **REPEAT** loop executes the block of instructions unconditionally first and then repeats its execution as long as the condition is false. The loop execution stops as soon as the condition becomes true.

4.1) Algorithmic syntax

The syntax of the **REPEAT** loop in algorithmic is as follows:

```
REPEAT
 <block of instructions>;
UNTIL <Condition>;
```

The progression of the **REPEAT** loop can be described as follows. First, the block of instructions that makes up the body of the loop is executed for the first time. Then, the condition is evaluated. If it's true, the block of instructions is executed again, and the condition is re-evaluated. This process repeats until the condition is satisfied. In this case, the loop is exited, and the normal execution of the algorithm continues.

Exemple:

Re-implement the algorithm that repeats the input of a person's age until a valid age is provided, but this time using the **REPEAT** loop.

Solution :

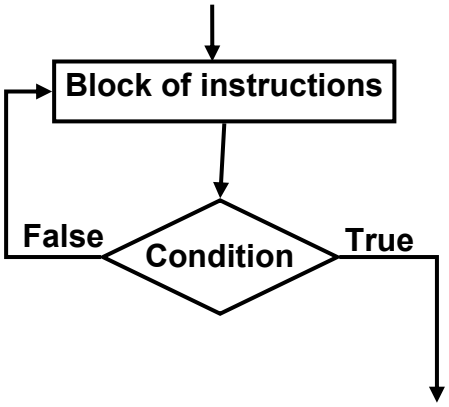
```
ALGORITHM input_age;
VAR age : Integer;
BEGIN
REPEAT
  Write("Enter the person's age:");
  Read(age);
UNTIL age > 0;
Write("Valid age");
END.
```

Remarks :

- The condition can be simple or compound.
- Note the absence of a semicolon after **REPEAT** and **UNTIL**.
- In this loop, the condition is only evaluated at the end of the loop. Therefore, the block of instructions that forms the body of the loop is executed at least once, even if the condition is satisfied from the beginning. Thus, the first execution is not subject to any condition.
- The condition in the **REPEAT** loop is the exit or termination condition of the loop, not the repetition condition, as is the case with the **WHILE** loop.
- The variables on which the condition is based must be initialized by reading or assignment before the condition is evaluated (and not before the loop).
- The body of the loop must contain an instruction that modifies the condition parameters to reach the exit condition at some point; otherwise, you would end up in an infinite loop.
- The block of instructions does not need to be enclosed in **BEGIN** and **END**, even if it consists of multiple instructions.

4.2) Flowchart

The **REPEAT** loop can be represented in a flowchart as follows:



4.3) C language syntax

The C language loop corresponding to **`REPEAT`** is the **`do-while`** loop. It is introduced by the do statement, followed by the block of instructions, and finally, the condition enclosed in parentheses, placed after a **`while`**, just like the **`while`** loop.

The syntax is the follows:

```
do{
   <block of instructions>;
}while(condition);
```

Indeed, the meaning of the **`REPEAT`** loop in algorithmics is a bit different from that of the **`do-while`** loop in the C language. The difference lies in the loop's condition. In algorithmics, it's "repeat until the condition is satisfied," while in C, it's "repeat as long as a condition is satisfied". Therefore, the condition in **`REPEAT`** is an exit condition, whereas in **`do-while`**, it's an entry condition for the loop. It's similar to a **`while`** loop, except the condition is at the end of the loop.

Remarks:

- The condition must be enclosed in parentheses.
- Note the presence of a semicolon after the condition.
- If the block of instructions contains only one instruction, the curly braces (**`{`** and **`}`**) are not obligatory.
- The condition in **`do-while`** is the inverse of the condition in **`REPEAT`**.

Example:

The C program that repeats reading the age using the **`do-while`** loop until a valid age is entered is as follows:

```
#include<stdio.h>
main(){
    int age;
    do{
        printf("Enter the person's age: ");
        scanf("%d",&age);
    }while(age <= 0);
    printf("Valid age");
}
```

5) FOR loop

The **FOR** loop allows you to repeat the execution of a block of instructions a certain number of times known in advance. This loop automates the phases of initializing and modifying the loop variable.

5.1) Algorithmic syntax

In this loop, a control variable of integer type, called the ***counter***, is used to control the number of iterations of the loop.

The counter takes its values in an interval whose bounds are known. Thus, in the header of the **FOR** statement, you must specify the initial value, the final value, and optionally the step (when it's different from 1).

The syntax of the **FOR** loop in algorithmic is as follows:

```
FOR <counter> ← <initial value> TO <final value>
STEP=<step value> DO
                <Block of instructions>;
```

Such as:

- **<counter>** is the control variable (of integer type) that counts the number of loop iterations.
- **<initial value>** is the initial value to which the counter is initialized. It can be a constant or an integer-type variable.
- **<final value>** is the final value at which the counter ends. It can also be a constant or an integer-type variable.
- **<Step Value>** is the increment or decrement value for the counter. The step can be omitted if its value is 1.

The block of instructions is executed each time the counter's value is between the initial value and the final value. The progression of the **FOR** loop can be described as follows:

First, the **counter** is initialized to the **initial value** at the moment of entering the loop. If the **counter**'s value does not exceed the **final value**, the **block of instructions** is executed, and the **counter** is automatically increased (incremented) by the increment value (**step value**). When the increment is not specified, the default increment is 1. This process repeats until reaching the **final value**. In this case, the loop terminates, and execution continues normally after the loop.

Example:

Using the **FOR** loop, write an algorithm to display natural numbers from 1 to 5.

Solution :

```
ALGORITHM display_numbers;
VAR i : Integer;
BEGIN
FOR i ← 1 TO 5 STEP=1 DO
     Write(i);
END.
```

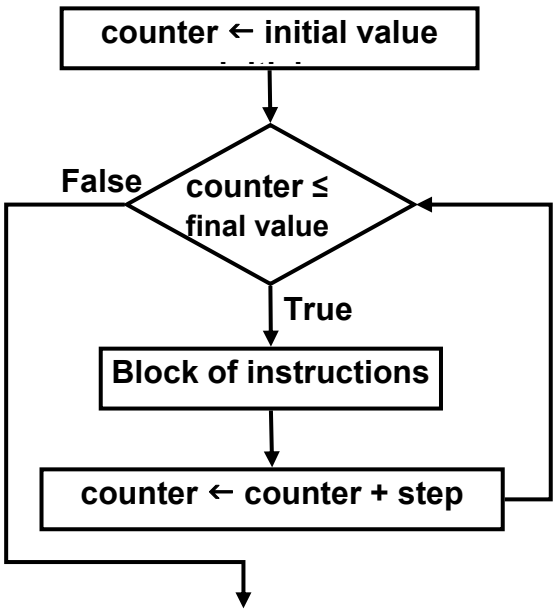
Remarks :

- The **FOR** loop can only be used when the number of iterations is known in advance.
- A **FOR** loop can be executed 0 times (when the final value is less than the initial value), 1 time (when the initial value and the final value are the same), or multiple times (the normal case).
- The initial value, the final value, and the increment step can be numeric expressions.
- In the **FOR** loop, the initialization of the loop variable (the counter), its modification (the increment of the counter), and the evaluation of the stopping condition are performed automatically.
- The increment step is optional. If omitted, its default value is 1.
- The increment step can also be negative, and in that case, the counter is decremented by the increment step at each iteration.

- In the body of the loop, the counter can be used for calculations, but it must not be modified either by reading or by assignment.
- The number of iterations in the **`FOR`** loop is equal to: **`final value - initial value + 1`** (when the increment step is equal to 1).
- The initial and final values, and the increment step are evaluated once and for all before the iteration; the body of the loop cannot modify their value.
- If the block of instructions consists of 2 or more instructions, it must be delimited by the keywords: **`BEGIN`** and **`END`**.

5.2) Flowchart

The **`FOR`** loop can be graphically represented in a flowchart as follows:



5.3) C language syntax

The loop corresponding to **`FOR`** in the C language is also called the **`for`** loop. However, the syntax of the latter in C language is a bit different from that of **`FOR`** in algorithmics.

The syntax in C of the **`for`** loop is as follows:

La syntaxe de la boucle **`for`** est la suivante:

```
for (<initialization>;<condition>;<modification>)
    <block of instructions>;
```

Thus, the header of the **`for`** loop is composed of three expressions separated by semicolons within parentheses:

- The first expression (**<initialization>**) is an initialization expression. It is executed only once at the beginning of the loop and is typically in the form: ***counter = initial value***.
- The second (**<condition>**) is a comparison expression. It is evaluated at the beginning of each iteration, including the first one.
- The last (**<modification>**) is a progression expression. This expression is used to increment (or decrement) the loop counter and is executed at the end of each iteration.

The execution of the **for** loop proceeds as follows. At the beginning of the for loop, the **initialization** statement is executed. Then, the **condition** is tested. If the condition is true, the instructions within the **for** loop's body are executed, followed by the **modification** statement. The **condition** is re-evaluated with the new **counter** value before the next iteration, and so on, as long as the **condition** remains true. Once the **condition** becomes false, the loop terminates.

Remarks:

- It is possible to initialize/modify multiple loop variables simultaneously by using commas in the expressions.
- When the block of instructions consists of more than one statement, it must be enclosed in curly braces (**{** and **}**).

Example:

The C program that displays natural numbers from 1 to 5 is as follows:

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

6) Choice of the appropriate repetitive structure

The choice of the appropriate repetitive structure depends on the problem to be solved.

If the number of repetitions is known in advance, it is advisable to use the **FOR** loop. On the other hand, if the number of iterations is not known in advance, either the **WHILE** loop or the **REPEAT** loop should be used.

However, the choice between these two loops is possible and depends on the minimum number of repetitions desired. If you want to execute the instructions in the block at least once, it is recommended to use the **REPEAT** loop. When the number of iterations can be zero, the **WHILE** loop must be used.

7) Nested loops

As we have seen before, loops execute one or more instructions (instruction block) a certain number of times. These instruction blocks can, in turn, contain loops. In this case, we refer to them as ***nested loops***.

Hence, a **WHILE** loop can contain another **WHILE** loop, another **REPEAT** loop, or another **FOR** loop, and vice versa.

Example:

Consider the following algorithm:

```
Algorithm nested_loops;
Var i,j:integer;
Begin
i ← 1;
WHILE i ≤ 3 DO
    Begin
    j ← 1;
    WHILE j ≤ 2 DO
        Begin
        Write(i+j);
        j ← j + 1;
        End;
    i ← i + 1;
    End;
End.
```

The step-by-step execution of the algorithm is summarized in the table below:

Iteration	i	j	Displayed value
1	1	1	2
2		2	3
3	2	1	3
4		2	4
5	3	1	4
6		2	5