Chapter 4:

The loops

1. Introduction

In everyday problems, we often need to execute a sequence of actions multiple times. This is because the user may not always provide the correct input on the first try.

For example, if we ask a user to enter a yes or no answer, they may accidentally enter a letter instead. In this case, the program should prompt the user to enter a valid answer again.

One way to solve this problem is to implement a data validation check. This check would verify that the data entered from the keyboard is valid. For example, if we are expecting the user to enter a yes or no answer, we could use a conditional statement to check that the input is either "Y" or "N".

```
Algorithm input check ;
Var    Rep : char;
begin
    Write ("Would you like a copy of this course? (Y/N)")
    read (Rep)
    if Rep ≠ 'Y' and Rep ≠'N' then
      begin
       Write ("Input error. Repeat") ;
      read (Rep) ;
    End;
End.
```

This algorithm solves the problem if the user makes only one mistake. The user is prompted to enter a valid value again, and the algorithm continues to the next step. However, if the user makes a second mistake, the algorithm will not be able to continue. In this case, the programmer would need to add another IF statement to the algorithm. This would create a very long and complex algorithm, and it would be difficult to maintain. A better solution is to use ***a repetitive structure***.

A repetitive structure, also known as an ***iterative structure*** or ***loop***, allows the execution of one or more instructions to be repeated multiple times. The number of repetitions can be:

- be known in advance.
- depend on the assessment of a ***condition***.

Looping is a common programming concept that allows us to execute a block of code multiple times. Each repetition is called an ***iteration***.

There are three main types of loops in programming: while loops, do-while loops, and for loops. These loops have the same expressive power, meaning that they can be used to achieve the same results. However, there are conventions for using each type of loop in different contexts.

There are four essential elements to a loop:

- ***The loop body***, which will be executed a certain number of times.
- ***The loop condition***, like for test instructions. This condition is based on at least one variable, which is called the loop variable. There can be multiple loop variables for a single loop.
- ***The loop initialization***, which concerns the loop variable. This initialization can be directly carried out by the loop instruction, or left to the programmer.
- ***The loop modification***, which also concerns the loop variable. Like for initialization, it can be integrated into the loop instruction, or left to the programmer.

If any of these four elements is missing, then the loop is incorrect. This will cause either a compilation error or an execution error, depending on the missing element and the type of loop.

2. The While loop

It is a statement that is executed repeatedly until a certain condition is met. The loop condition is evaluated before each iteration, and if the condition is true, the loop body is executed. When the condition is false, the loop terminates.

The while loop is written as follows:

```
while ‹condition›
        ‹Instructions_Block› ;
```

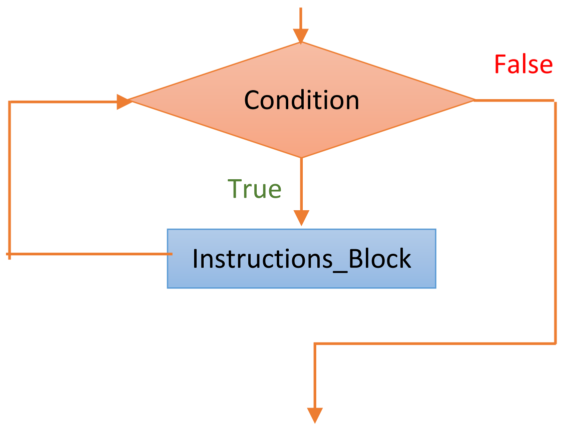
Where:

- *condition* is a Boolean expression that evaluates to true or false.
- *Instructions_Block* is a sequence of one or more instructions.

The actions performed during its execution are as follows:

1. *Evaluate the condition.*
2. *If the evaluation returns an error, the instruction is not executed, and the overall execution terminates (this type of situation should be avoided!). Otherwise, let **E** be the value of this condition.*
3. *If the value of **E** is equal to true, execute the block of instructions, and then go back to **step 1**.*
4. *If the value of **E** is equal to false, the instructions in the block are not executed, and the algorithm continues just after the block.*

Flow chart:



In C language:

```
While (condition)
     {
          /* bloc_d' instructions */
      }
```

Note the absence of a semicolon (;) after the condition.

Remarks :

1. The condition is checked first, so the sequence may not be executed at all. This is possible if the condition is false at the beginning.

2. It is important to avoid infinite loops. An infinite loop is a loop whose condition never changes, which results in an infinite number of repetitions.

Exercices d'application

Exercise 1

Write an algorithm that prompts the user for a student grade between 0 and 20. The algorithm should continue to prompt the user for a new grade until the grade is valid.

Exercise 2

Write an algorithm that prompts the user for a number between 1 and 9. The algorithm should then print the multiplication table for that number.

Exercise 3

Write an algorithm that prompts the user for an integer value N. The algorithm should then calculate and print the sum of the first N integers (1+2+3+…+N).

Exercise 4

Write an algorithm to input N real numbers and calculate and print their sum.

Exercise 5

Write an algorithm to input N real numbers and calculate and print their maximum.

3. Do-while loop

The Do-while loop (also called repeat loop) executes a block of instructions once, unconditionally, and then repeats the loop body as long as a condition is true. The Do-while loop statement is written as follows:

```
Do
‹Instructions_Block›
While ‹condition› ;
```

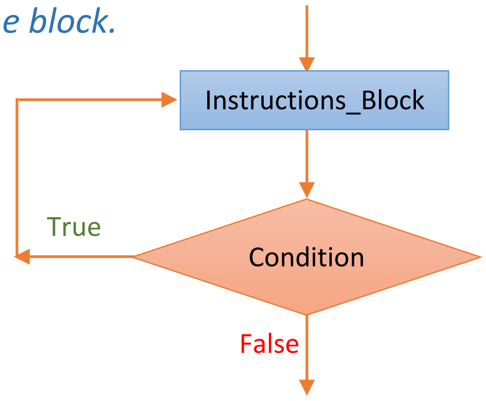
Where:

- *condition* is a Boolean expression that evaluates to true or false.
- *Instructions_Block* is a sequence of one or more instructions.

The following actions are performed during the execution of a Do-while loop:

1. *Execute the block of instructions.*
2. *Evaluate the condition.*
3. *If the evaluation returns an error, the instruction is not executed, and the general execution terminates (this type of situation should be avoided!). Otherwise, let **E** be the value of this condition.*
4. *If the value of **E** is false, go back to **step 1.***
5. *If the value of **E** is true, the instructions in the block are not executed, and the algorithm continues just after the block.*

Flow chart:



In C language :

```
Do
      {
            /* bloc_d' instructions */
       }
While (condition) ;
```

Don't forget the semicolon.

Comparison of “Do-while loop” and "while loop"

The "while loop

- The condition is evaluated before each execution of the processing.
- Therefore, the processing may not be executed.

The Do-while loop

- The condition is checked after each execution of the processing.
- The processing is executed at least once.

Application Exercises

Exercise 6

Write an algorithm that prompts the user for a student grade between 0 and 20. The algorithm should continue to prompt the user for a new grade until the grade is valid.

Exercise 7

Write an algorithm to input an integer and display its mirror image.

Exercise 8

Write an algorithm that prompts the user for an integer value N. The algorithm should then calculate and print the sum of the first N integers (1+2+3+…+N).

4. For Loop

When the number of times a loop should be repeated is known exactly, a for loop is used. A for loop is a defined loop, which means that it will be repeated a specific number of times.

The For loop statement provides three actions in one compact statement: initializing a loop control variable, testing the loop condition, and modifying the loop control variable.

The repetitive statement for is written as follow:

```
For ‹counter› = ‹E1› to ‹E2› [step=‹E3›]
            ‹Instruction_Block› ;
```

The amount by which a For-loop control variable changes is often called a step value. The step value can be any number and can be either positive or negative; that is, it can increment or decrement.

A for loop can express the same logic as a while statement, but in a more compact form. It is never necessary to use a for statement for a loop; a while loop can always be used instead.

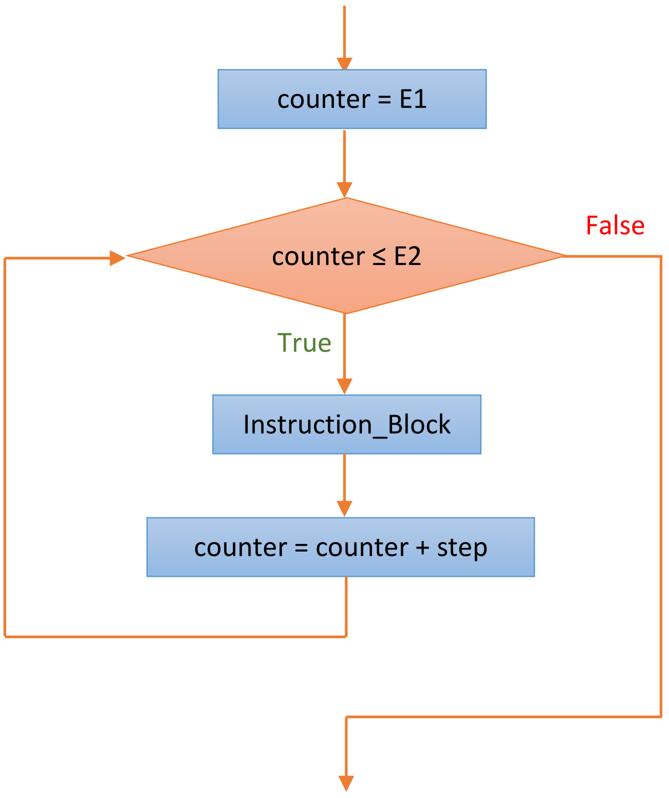
The following actions are performed during the execution of a For loop:

> *Evaluate expressions E1, E2, E3.*
> 1. *Let V1, V2, V3 be their respective values.*
> 2. *Set counter = V1.*
> 3. *Assume V3 is positive.*
> 4. *Repeat the following steps:*
>   * Evaluate counter. Let Vcounter be its value.*
>   * If Vcounter > V2, terminate the iteration.*
>   * Otherwise, execute the block of instructions.*
>   * Set counter = Vcounter + V3.*
>   * Go back to step 4*

Remarks

- The part [step=‹E3›] is optional. If omitted, E3 is eqqual to 1.
- E1, E2, E3 are evaluated once before the iteration, and their values cannot be changed by the body of the iteration.
- The body of the iteration cannot change the value of the variable ‹counter›.

Flow chart :



In C language:

```
For (initialization ; condition ; modification)
{
     /* Instruction_Block */
}
```

Comparison "For" and "While" loops

- If the number of iterations is known in advance, choose the "For" loop.
- If the loop needs to stop when an event occurs, choose the "while" loop.

Application Exercises

Exercise 9

Write an algorithm that prompts the user for an integer value N. The algorithm should then calculate and print the sum of the first N integers (1+2+3+…+N).

Exercise 10

Write an algorithm that prompts the user for a non-zero positive integer, and to calculate and display its divisors.

Exercise 11

Write an algorithm that allows the user to enter N integer values, and calculate and display their sum.

Infinite loops problem...

while ‹condition› ‹Instructions_Block› ;

The **‹Instructions_Block›** must modify at least one variable in the logical expression **‹condition›** in a way that causes condition to evaluate to false. This is the only way to terminate the loop.

```
val1 ← 2 ;
val2 ← 3 ;
while (val1 <100)
     val2 ← val2 * val1 ;
```

Be careful when creating loops!

Always make sure your loop has a way to end. If the condition is always true, your program will run forever.

The following loop is an example of an infinite loop. It will continue running forever, unless it is interrupted by a user or a system error:

```
i ← 3 ;
while (i >0)
    begin
    write (i);
    i← i+1;
    end;
```

Although this loop contains all the essential elements, it is infinite. It displays all integers greater than 3, indefinitely.

5. Nested loops

The body of a loop can contain another loop. This is called nested loops. **Nested loops** are two or more loops that appear inside each other. The loop that contains the other loop is called the **outer loop**, and the loop that is contained is called the **inner loop**.

Example : consider the following algorithm

```
Algorithm nesting ;
Var i, j : integer ;
begin
i←1 ;
while  (i≤3)
    begin
    j←1 ;
    while (j≤4)
        begin
        write ('i=', i, 'j=', j) ;
        j←j+1;
        end;
    i←i+1;
    end ;
End.
```

Inst	i	j
1	1	
2		1
3		2
4		3
5		4
6		5
7	2	
8		1
9		2
10		3
11		4
12		5
13	3	
14		1
15		2
16		3
17		4
18		5
19	4	

Exercice 10

Write an algorithm that takes as input a positive integer N and determines if it is prime.

Modify this algorithm to display the first 10 prime numbers.