Chapter 3 : Conditional structures

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

Often, problems require the examination of multiple situations that cannot be addressed through simple sequences of actions. In such cases, it becomes necessary to choose between 2 or more courses of action depending on whether a certain condition is met or not. Since there are multiple situations and, before execution, it is not known which specific scenario will be executed, in the algorithm, we must anticipate all possible cases. Conditional structures (tests) allow for this.

There are several forms of conditional structures, which will be presented in the rest of the chapter.

2) Notion of Condition

In algorithms, a condition is merely a logical expression that can become a true or false statement depending on the values that make up the expression. The condition can be simple or compound.

2.1) Simple conditions

A simple condition consists in the comparison of two expressions of the same type. It comprises three elements: an expression, a comparison operator, and another expression.

Examples **:**

`val=5` `a<b` `x+3 ≥ 5*y-4` `'c'≠ 'a'`

Are simple conditions.

2.2) Compound conditions

These are conditions formed by combining multiple simple conditions using logical **AND** and **OR** operators.

Examples:

nb≥0 and nb≤20 x=0 or y=0 c≠'O' and (c='N' or c='n')

Are compound conditions.

3) Simple conditional structures (`if` statement)

Simple conditional structures allow executing a sequence of instructions only if a condition is satisfied.

3.1) Algorithmic syntax

The syntax of a simple conditional structure is as follows:

IF ***<Condition>*** **THEN** ***<block of instructions>;***

Example:

Write an algorithm that asks for a number and determines if it is negative.

Solution :

```
ALGORITHM test;
VAR a : Integer;
BEGIN
Write("Enter a number : ");
Read(a);
IF a<0 THEN Write("Negative number");
END.
```

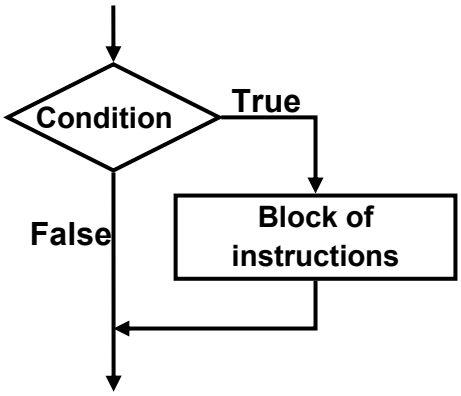
In this example, we test if the value of the variable **a** is less than 0; if yes, we display the message "Negative number," otherwise, we do nothing.

Remarks:

- There are no semicolons after the condition or after **THEN**.
- If the block of instructions contains multiple instructions, they are separated by semicolons and enclosed between **BEGIN** and **END**. If the block contains only one instruction, the words **BEGIN** and **END** are not required.

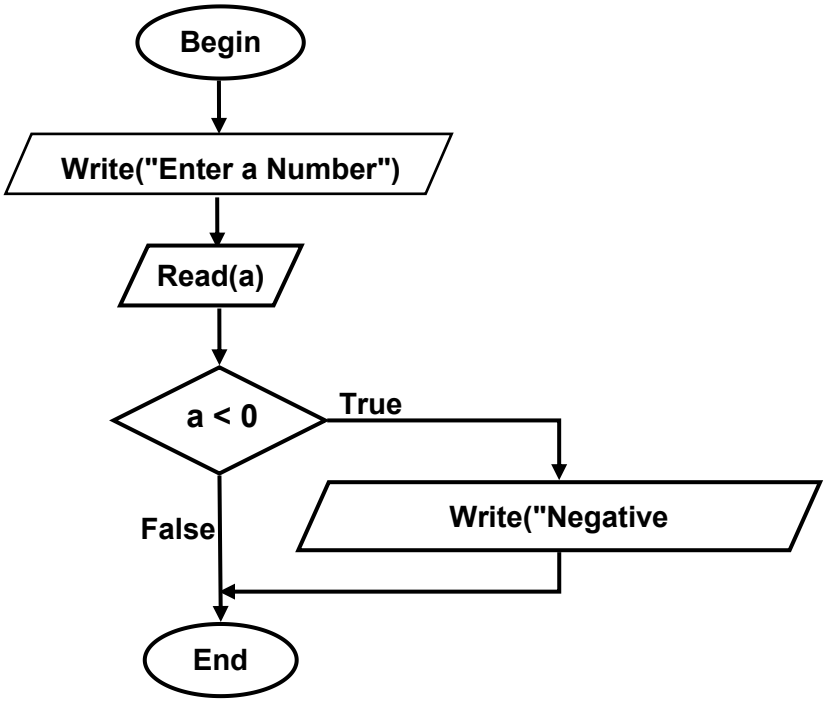
3.2) Flowchart

The simple conditional structure can be represented in a flowchart as follows:



Example:

The flowchart corresponding to the algorithm that checks if an entered number is negative is as follows:



3.3) C language syntax

The syntax of a simple conditional structure in the C language is as follows:

```
if(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 instructions, it should be enclosed in curly braces (**{** and **}**), and if it contains only one instruction, the curly braces are not mandatory.

Example:

The C program that determines if an entered number is negative is as follows:

```
main(){
  int a;
  printf("Enter a number : ");
  scanf("%d",&a);
  if(a<0)printf("Negative number");
}
```

4) Compound Conditional Structure (if-else statement)

The compound conditional structure (also called alternating conditional structure) allows to execute a sequence of instructions if a condition is satisfied and to execute another sequence if the condition is not satisfied.

It combines multiple simple conditional statements to handle different cases.

4.1) Algorithmic syntax

The general form of a compound conditional structure is as follows:

```
IF <Condition> THEN   <block of instructions1>
ELSE <block of instructions2>;
```

Remarks:

- In algorithmics, there is never a semicolon before **ELSE**.
- If a block of instructions consists of two or more instructions, it must be delimited by the keywords **BEGIN** and **END**.

Example:

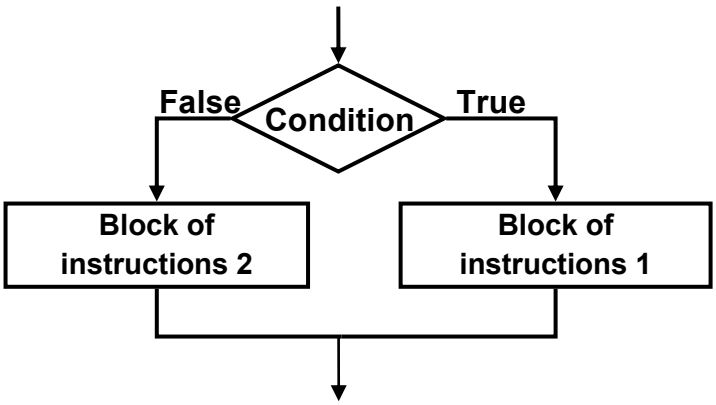
Write an algorithm to input a number and determine whether it is even or odd.

Solution :

```
ALGORITHM test;
VAR nb : Integer;
BEGIN
Write("Enter a number : ");
Read(nb);
IF nb mod 2 = 0 THEN Write("The number is even")
ELSE  Write("The number is odd");
End.
```

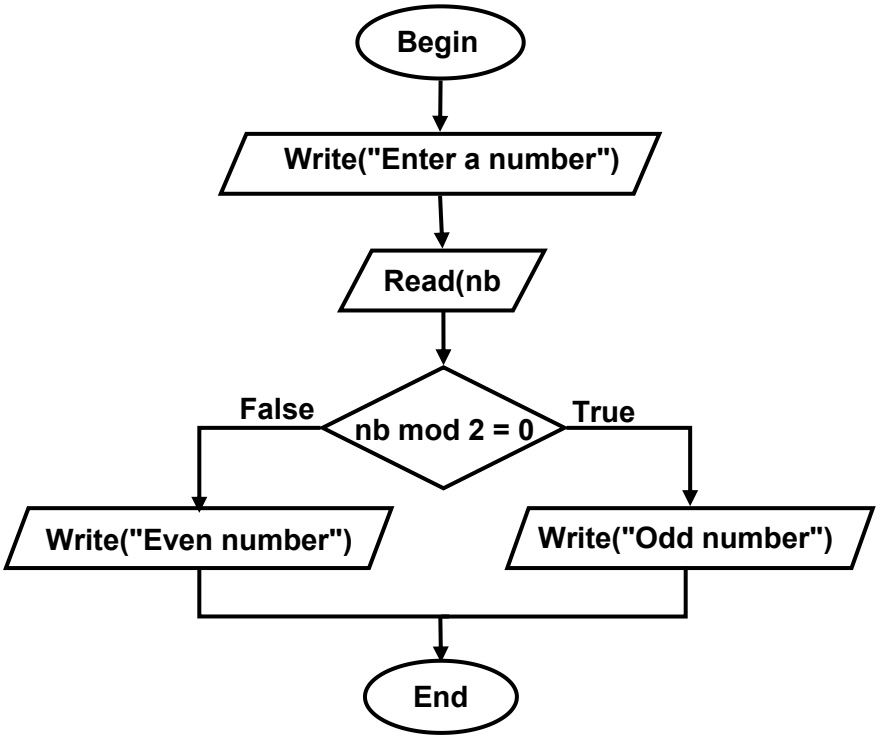
4.2) Flowchart

The compound conditional structure can be represented in a flowchart as follows:



Example:

The flowchart corresponding to the algorithm that tests the parity of an integer entered by the user is as follows:



4.3) C language syntax

The syntax of a compound conditional structure in the C language is:

```
if(condition) <block of instructions1>;
else <block of instructions2>;
```

Remarks:

- The condition must be enclosed in parentheses.
- In the C language, **`else`** can be preceded by a semicolon.
- If a block of instructions consists of multiple instructions, it must be enclosed in braces (**`{`** and **`}`**).

Example:

The C program that allows to input a number and determine whether the number is even or odd is as follows:

```
main(){
   int nb;
   printf("Enter a number : ");
   scanf("%d",&a);
   if(a%2==0)printf("The number is even");
   else printf("The number is odd");
}
```

5) Nested Conditional Structures

The instruction blocks of **`IF`** and **`ELSE`** are sequences of instructions. These blocks can contain reading, writing, assignment instructions, as well as conditional structures. This is referred to as having nested structures.

5.1) Algorithmic syntax

The general form of nested conditional structures is as follows:

```
IF <Condition1>  THEN  <block of instructions1>
ELSE IF <Condition2>  THEN <block of instructions2>
     ELSE IF <Condition3>  THEN <block of instructions3>
                  ...
                  ...
               ELSE <block of instructions_n>;
```

Condition1 is evaluated first. If it is true, block of instruction 1 is executed. If it's not true, we evaluate condition2. If condition2 is true, we execute block of instruction 2; otherwise, we move on to evaluating condition3, and so on.

Example:

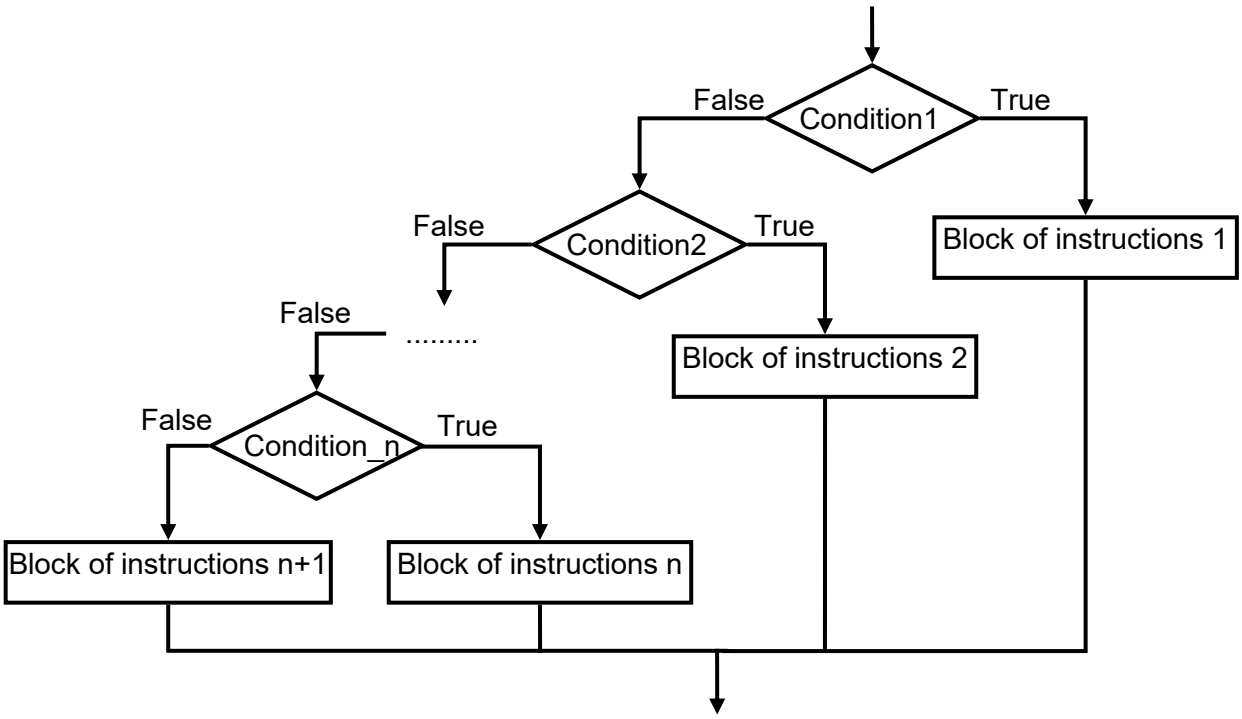
Write an algorithm that reads a real number and displays whether this number is positive, negative, or zero.

Solution:

```
ALGORITHM test;
VAR x : Real;
BEGIN
Write("Enter a number : ");
Read(x);
IF x > 0 THEN Write("The number is Positive ")
ELSE  IF x < 0 THEN Write("The number is Negative")
      ELSE  Write("The number is Zero");
END.
```

5.2) Flowchart

The flowchart corresponding to the nested conditional structures:



5.3) C language syntax

The nesting of multiple tests is done in the C language as follows:

```
if(condition1) <block of instructions1>;
else if(condition2) < block of instructions 2>;
     else if(condition3) < block of instructions 3>;
          .....
          else < block of instructions 4>;
```

Example:

The C program that determines whether an entered number is positive, negative, or zero:

```
main(){
  float x;
  printf("Enter a number : ");
  scanf("%f",&x);
  if(a>0)printf("The number is Positive");
  else if(a<0)printf("The number id negative");
  else printf("The number is Zero");
}
```

6) Multiple-choice structure (CASE statement)

The nesting of a large number of tests tends to make the algorithm heavier and more challenging to read and manage. Fortunately, there is a structure that makes the task somewhat easier, called the *multiple-choice statement* or *the selective structure*.

The multiple-choice structure allows to select or distinguish several cases based on the values of an expression. This expression is called a ***selector***, and it must be a scalar-type variable or expression.

6.1) Algorithmic syntax

The syntax of the multiple-choice statement is as follows:

```
CASE  <expression>  OF
   <Value1> : <block of instructions 1>
   <Value2> : <block of instructions 2>
   ……
   <Value_n> : <block of instructions n>
   OTHERWISE:  <block of instructions (n+1);
END;
```

The value of the expression is successively compared to each of the selection values. As soon as a match is found, the comparisons are stopped, and the associated block is executed. If no value matches, then the block associated with **`OTHERWISE`**, if it exists, is executed.

Remarks:

- The selector and the values to choose must be of the same type.

- If a block of instructions consists of more than one instruction, they must be surrounded by the **BEGIN** and **END** keywords
- The default case (**OTHERWISE**) is optional. It is used to perform a task when none of the cases is true.

Example:

Write an algorithm that allows entering an integer between 1 and 5 and displays it in words.

```
ALGORITHM example;
TYPE Digit = 1..5;
VAR n : Digit;
END
Write("Enter a number between 1 and 5 : ");
Read(n);
CASE n OF
  1: Write("One");
  2: Write("Two");
  3: Write("Three");
  4: Write("Four");
  5: Write("Five");
  OTHERWISE: Write("Input error");
  End;
END.
```

6.2) C language syntax

The multiple-choice conditional structure is expressed in the C language as follows:

```
switch(<expression>){
    case <value>:    <block of instructions 1>
                     break;
    case <value2>:   <block of instructions 2>
                     break;
             ..............
    case <value_n>: <block of instructions n>
                     break;
    default:         <block of instructions n+1>
}
```

Remarks:

- The **`default`** case is optional.
- The **`break`** keyword is mandatory after each block of instructions to indicate the end of a case.
- The instruction blocks should not be enclosed in braces.

Example:

The C program that allows to input an integer between 1 and 5 and display it in words is as follows:

```
main(){
   int n;
   printf("Enter a number between 1 and 5: ");
    scanf("%d",&n);
    switch(n){
        case 1: printf("One");
                break;
        case 1: printf("Two");
                break;
        case 1: printf("Three");
                break;
        case 1: printf("Four");
                break;
        case 1: printf("Five");
                break;
        default: printf("Input error");
    }
}
```

7) Branching statement

The branching instruction allows for the interruption of the normal flow of an algorithm by jumping from one point in the algorithm to another and continuing execution from that point. This is why they are also referred to as ***jump instructions***.

To perform a branch, you must first identify the instruction in the algorithm to which you want to branch using a label. Then, it is possible to jump to that instruction to execute it (along with the subsequent instructions) by knowing its label.

A label is an identifier assigned to an instruction in the algorithm in order to identify it. This makes it possible to go directly to the instruction by knowing its label.

7.1) Algorithmic syntax

To assign a label to an instruction, simply write the label (which is an identifier) followed by a colon "**:**" before the instruction.

Then, branching or jumping to that label is done using the "**GOTO**" statement, specifying the label.

The syntax is as follows:

```
<Label_name>: instruction i ;
      ..........
GOTO <Label_name>;
      ..........
```

Example:

The following algorithm allows entering a student's exam mark and, if applicable, the makeup exam mark, to determine whether the student is admitted or deferred using branching.

```
ALGORITHM example;
VAR exam, makeup : real;
BEGIN
WRITE("Enter the exam mark: ");
READ(exam);
IF exam ≥ 10 THEN GOTO label;
WRITE("Enter the makeup exam mark: ");
READ(makeup);
IF makeup ≥ 10 THEN GOTO label;
WRITE("You are deferred");
GOTO last;
label: WRITE("You are admitted");
last: WRITE("End of the algorithm");
END.
```

Remarkes:

- It is possible to jump to:

- An instruction that precedes the branching instruction, creating a loop effect.
- An instruction that follows the branching instruction to advance more quickly in the algorithm.

- It is discouraged to use branching instructions to reduce the complexity of algorithms in terms of time.

7.2) C language syntaxe

The definition of a label in the C language is done in the same way as in algorithmics, by writing the label followed by a colon "**:**" before the instruction to be marked.

For branching, it is done using the **goto** statement.

The syntax is as follows:

```
<Label_name>: instruction i ;
              ..........
goto <Label_name>;
              ..........
```

Example:

The program that allows to enter a student's exam mark, and if applicable, the makeup exam mark, and determine whether he is admitted or deferred is as follows:

```
main(){
    float exam,makeup;
    printf("Enter the exam mark : ");
    scanf("%f",&exam);
    if(exam >= 10)goto label;
    printf("Enter the makeup exam mark : ");
    scanf("%f",&makeup);
    if(makeup >= 10)goto label;
    printf("You are deferred");
    goto last;
    label: printf("You are admitted ");
    last: printf("End of the algorithm");
}
```