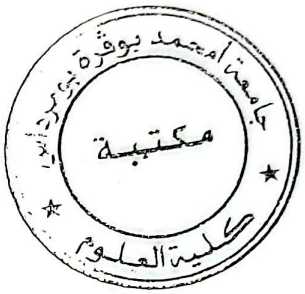
Democratic and People's Republic of Algeria
Ministry of Higher Education and Scientific Research
Université M'Hamed Bougara Boumerdes

Faculty of Sciences

Department of Computer Science



Field: Mathematics and Computer Science
Sector: Computer Science

Course Handout

Algorithms and Data Structures 1

Carried out by:

Dr. Nabila RAHMOUNE (MCB)

This handout is intended for students enrolled in Licence (L1) Field Mathematics and Computer Science.

2024/2025

Democratic and People's Republic of Algeria
Ministry of Higher Education and Scientific Research
Université M'Hamed Bougara Boumerdes

Faculty of Sciences

Department of Computer Science

Field: Mathematics and Computer Science
Sector: Computer Science

Course Handout

Algorithms and Data Structures 1

Carried out by:

Dr. Nabila RAHMOUNE (MCB)

This handout is intended for students enrolled in Licence (L1) Field Mathematics and Computer Science.

2024/2025

Table of contents

General Introduction

This guide has been developed to assist students in the Mathematics and Computer Science (MI) field during their first steps at university, in accordance with the official program following the transition of higher education towards the English language.

Algorithms and data structures are core topics in computer science and information sciences. Algorithmics involves the study of designing and analyzing algorithms sequences of instructions aimed at solving specific problems. Data structures, on the other hand, provide efficient methods for storing and organizing data, enabling fast processing and easy manipulation of information.

Algorithms and data structures are employed in numerous computing applications, ranging from software design and web development to data analysis and data science. They are also utilized in other fields, such as social and natural sciences, engineering, mathematics, finance, and management. A solid understanding of algorithms and data structures is essential for any computer science student or professional. It enables efficient problem-solving, the development of robust software, and the enhancement of application performance.

This handout has been designed as a practical and pedagogical guide to support students throughout the course. It provides a logical progression, starting from fundamental concepts and advancing to more specific topics, while incorporating concrete examples, practical exercises, and systematic translations of concepts into the C programming language to reinforce learning through practice.

The main objective of this course is to provide a solid foundation in algorithms and data structuring. By the end of the course, students will be able to understand and use control structures to create efficient algorithms, manipulate one-dimensional and two-dimensional arrays to represent and process data sets, manage and manipulate strings, translate algorithms into the C language for application to real-world problems, and solve practical problems by applying theoretical concepts.

The document is divided into four chapters, each covering a key aspect of the course. Chapter one introduces fundamental concepts such as algorithms, data structures, and programming basics, along with their translation into the C language. It then transitions to control statements, including conditions and choices, which are essential for directing a program's flow of execution, illustrated with examples in C. Following this, the study of iterative structures, such as for, while, and do-while loops, is presented, emphasizing their role in automating repetitive tasks and their implementation in C. Once the basics of these operations are mastered, the document explores arrays as data structures, highlighting their use in storing and manipulating collections of elements of the same type, with practical programming examples in C. Finally, the document concludes with an analysis of strings as a specific data

type, focusing on their manipulation within algorithms and their translation into the C language. This progression ensures a comprehensive understanding of algorithmic concepts and their practical application.

Each chapter concludes practical application exercises. Depending on the level of difficulty, the exercises are marked with a * and come with their solutions. The document ends with a mini-project that covers all the concepts discussed throughout the document.

This document is designed to be a clear, progressive, and interactive resource. By combining theory, practice, and the translation of concepts into C language, we hope it will effectively serve as both a learning and reference tool.

Chapter 1
Introduction to Algorithm and Some Fundamentals

Introduction

Computer science is the science of automatic information processing. The computer is a machine capable of automatically performing arithmetic andlogical operations from programs defining the sequence of these operations.

Why the computer?

The following table represents a small comparison between the human being and the computer.

Human	Computer
Possesses emotional intelligence, creativity, and the ability to reason abstractly.	Strictly follows instructions (algorithms) without understanding or reasoning.
Relatively slow for complex calculations or processing large amounts of data	Extremely fast in performing calculations and processing vast amounts of data.
Memory is limited, biased, and influenced by emotions.	Accurate and capable of storing large volumes of data, limited only by hardware.
Prone to errors due to inattention, fatigue, or distraction	Highly accurate as long as there are no bugs or programming errors.

We see that the advantage of one constitutes the disadvantage of the other and vice-versa. From there the objectives of the algorithmic course is to bring the intelligence of the human being to be able to solve problems towards the machine to have correct results, in a fast way, whatever the size of the problem

1. Problem-Solving Method

Problem-solving, especially in the context of computerization, requires the preparation of an action plan for your computer to execute. The steps to follow in developing this plan are as follows:

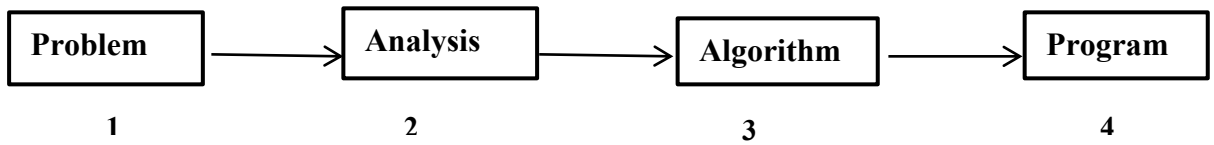


Figure 1. Steps followed for problem-solving.

1.1. Problem Identification: This step focuses on clearly defining the problem by gathering relevant data, identifying assumptions, and understanding the desired outcome.

It involves analyzing the context, setting boundaries, and ensuring clarity on what needs to be solved to avoid misunderstandings during later stages.

1.2. Analysis: This phase involves identifying various possible solutions and actions to achieve the desired outcomes. It includes evaluating the available options, assessing their feasibility and effectiveness, and selecting the approaches best suited to the defined objectives.

1.3. Action Plan: This step involves outlining the elementary actions the computer must execute in a clear, sequential, and organized manner using a conventional language. This structured set of instructions forms the basis of an algorithm.

1.4. Programming: An algorithm is not directly understandable by the computer. This final step involves converting the algorithm into a high-level programming language, such as C. The outcome of this translation is a program that the computer can execute.

2. Algorithm

2.1. Definition: An algorithm is a systematic set of instructions created to solve a specific problem or accomplish a particular task, with the objective of achieving a desired outcome efficiently and effectively. It consists of a finite sequence of clear, unambiguous steps that can be executed in a defined order to produce a result.
Algorithm has the following characteristics :

- **Input**:An algorithm may or may not require input
- **Output**:Each algorithm is expected to produce at least one result
- **Definiteness**:Each instruction must be clear and unambiguous
- **Finiteness**:If the instructions of an algorithm are executed, the algorithm should terminate after finite number of steps

Example: Making a Phone Call

a. Open your phone,
b. Search/Dial the recipient's number,
c. Press the "Call" button.

This set of instructions specifies how to make a phone call. It consists of an ordered sequence of actions (open, search/dial, press) that manipulate data (phone, number, button) to accomplish the task of making a call. It's an *algorithm*. Translate the algorithm into a programming language (such as C, Java, Python, etc.) to make it comprehensible for the computer, defining the *program*.
Since the algorithm is language-independent, we write the steps to demonstrate the logic behind the solution to be used for solving a problem. But before writing an algorithm, keep the following points in mind:

- The algorithm should be clear and unambiguous.

- There should be 0 or more well-defined inputs in an algorithm.
- An algorithm must produce one or more well-defined outputs that are equivalent to the desired output. After a specific number of steps, algorithms must ground to a halt.
- Algorithms must stop or end after a finite number of steps.

2.2. General Principle

Automatic information processing involves executing instructions on input data to generate output data. This process transforms raw data into meaningful results through a series of predefined operations, without manual intervention. It is the foundation of many computing systems, enabling efficient and accurate data processing to produce valuable outcomes.

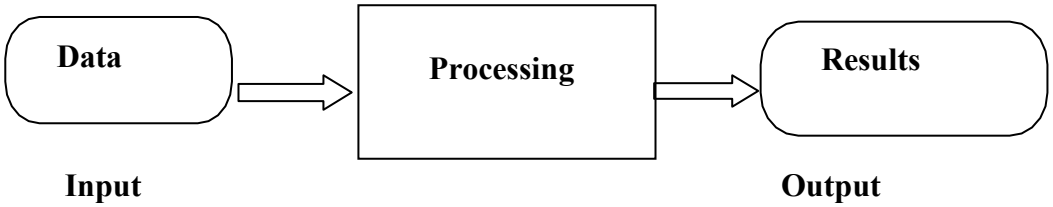


Figure 2. Automated Processing Principle.

Example1: Calculation of a Student's Average **Algorithm**

Algorithm Description:
i. Define the number of subjects along with their grades and coefficients.
ii. Perform the following operations:
- Multiply each grade by its corresponding coefficient.
- Calculate the sum of the multiplication results.
- Divide the sum by the total of all coefficients.
- Display the student's average as the final result.

This set of instructions outlines how to calculate a student's average based on their grades and coefficients. It involves a sequence of steps to process the data and compute the desired outcome. This is an algorithm for calculating the average.

Programming:
To make this algorithm understandable to a computer, it needs to be translated into a programming language such as C, Java, or Python, turning it into a program.

Note

- The algorithm should be clear and unambiguous.
- It should handle well-defined inputs, such as grades and coefficients.
- It should produce a well-defined output: the student's average.
- The algorithm should terminate after completing the steps.

How to write algorithms?

a. **Define your algorithms input** : many algorithms take in data to be processed, e.g. to calculate the area of rectangle input may be the rectangle height and rectangle width.
b. **Define the variables:** algorithm's variables allow you to use it for more than one place. We can define two variables for rectangle height and rectangle width as HEIGHT and WIDTH (or H & W). We should use meaningful variable name e.g. instead of using H & W use HEIGHT and WIDTH as variable name.
c. **Outline:** the algorithm's operations: Use input variable for computation purpose, e.g. to find area of rectangle multiply the HEIGHT and WIDTH variable and store the value in new variable (say) AREA. An algorithm's operations can take the form of multiple steps and even branch, depending on the value of the input variables.
d. **Output the results of your algorithm's operations :** in case of area of rectangle output will be the value stored in variable AREA. if the input variables described a rectangle with a HEIGHT of 2 and a WIDTH of 3, the algorithm would output the value of 6.

Example2:
Write an algorithm to find the largest among three different numbers entered by user.

Step 1: Start
Step 2: Declare variables a, b and c.
Step 3: Read variables a, b and c.
Step 4: If a>b
If a>c
Display a is the largest number.
Else
Display c is the largest number.
Else
If b>c
Display b is the largest number.
Else
Display c is the greatest number.
Step 5: Stop

2.3. Representation of Algorithms: Flowchart

- The flowchart is a diagram which visually presents the flow of data through processing systems. This means by seeing a flow chart one can know the operations performed and the sequence of these operations in a system. Algorithms are nothing but sequence of steps for solving problems. So a flow chart can be used for representing an algorithm. A flowchart, will describe the operations (and in what sequence) are required to solve a given problem. You can see a flow chart as a

blueprint of a design you have made for solving a problem.

- A flowchart is a diagram made up of boxes, diamonds and other shapes, connected by arrows. Each shape represents a step of the solution process and the arrow represents the order or link among the steps.

There are standardised symbols to draw flowcharts

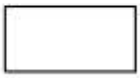
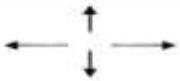
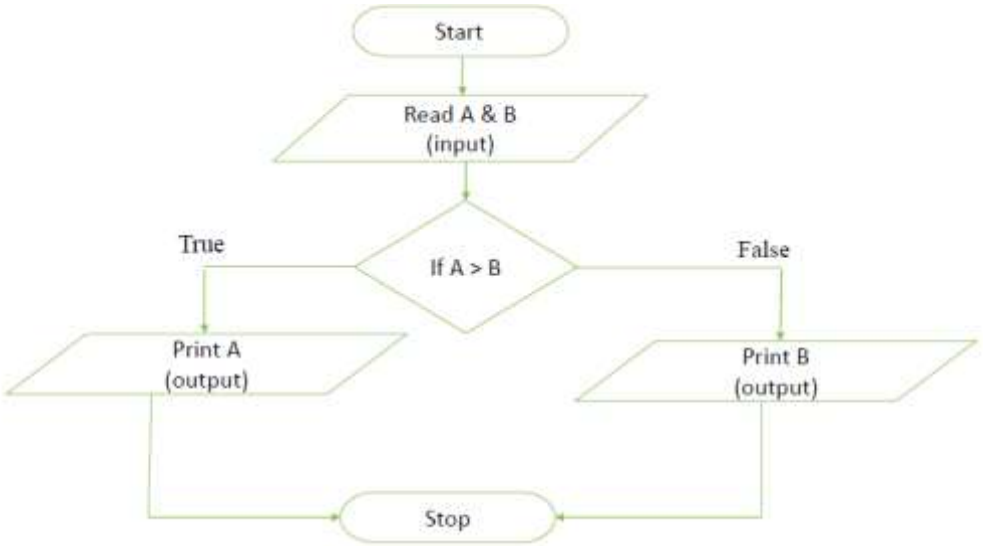
Symbol Name	Symbol	function
Oval		Used to represent start and end of flowchart
Parallelogram		Used for input and output operation
Rectangle		Processing: Used for arithmetic operations and data-manipulations
Diamond		Decision making. Used to represent the operation in which there are two/three alternatives, true and false etc
Arrows		Flow line Used to indicate the flow of logic by connecting symbols

Figure 1.1. Standard symbols to draw flowcharts

Example:



3. Algorithm characteristics

3.1. General structure

An algorithm generally consists of two parts:
Declarative Part: Also known as the algorithm header, it typically contains declarations (constants, variables, etc.).
Body Part: Comprised of one or more sequences of instructions calling for basic operations to be executed by the computer.

Syntax

```
Algorithm name_of_algorithm ;
< List of constants / variables > ; // declaration part
Begin
< Séquence of instructions > ;  // processing part
End.
```

An algorithm always begins with the keyword **'Algorithm'**, followed by its name, which typically reflects its purpose or functionality.

- The **declaration section** is essential, as it specifies the various data, variables, and structures that will be used in the algorithm, along with their respective types.
- The keyword **'Begin'** marks the start of the algorithm's execution or processing phase.
- The **processing section** comprises a sequence of actions required to solve the given problem. Each action corresponds to a specific operation or step.
- The keyword **'End'**, followed by a period (**'.'**), signifies the conclusion of the algorithm's processing and serves as the termination point of the algorithm.

3.2. Declaration

3.2.1. Variables: A variable is a memory location intended to hold values of a pre-defined type (numbers, characters, strings, etc.). It has a **name**, a **type**, and **content** that can be modified during the execution of the algorithm.

Rules to write variable names:

a. A variable name contains maximum of 30 characters/ Variable name must be upto 8 characters.
b. A variable name includes alphabets and numbers, but it must start with an alphabet.
c. It cannot accept any special characters, blank spaces except under score(_).
d. It should not be a reserved word.

Ex : I rank1 MAX min Student_name StudentName class_mark

The keyword is: **Var**.

3.2.2. Constants: Constants **refer** to fixed values that do not change during the execution of a program.

Keyword: Const
Constants and variables and are declared according to the following syntax:
Syntax

Const constant_name = value ;
Var variable_name : type ;

Note:

In the declarative part, variables and constants are primarily characterized by:
An identifier: a name assigned to the variable or constant, which can consist of letters and numbers but no spaces.
A type: defining the nature and size of the variable.

If multiple variables have the same type, they can be declared simultaneously by separating them with a comma:

Example:

Const alpha = 0,5 ;
Var x, y : integer;

In this example, we have declared:

- Two variables (x and y) of integer type, which will be described in the following subsection.
- A constant (alpha) equal to the value 0.5 as an example.

Schematically, the computer's working memory (RAM) resembles a grid containing multiple cells, where each cell (memory) represents a variable.

Example: Adding the variable x=5 and the variable y=7. The result will be stored in the variable z = x + y, which is 12.

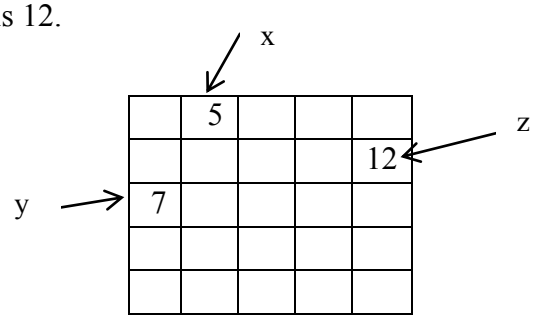


Figure 1.2.Illustrative example representing data in RAM.

3.3. Basic types

The type of a variable determines the range of values it can hold and the operations that can be performed on it. Common predefined simple types include: **integer**, **real**, **character**, and **boolean**.

3.3.1. Integer Type

It is a numeric type representing the set of integers, such as: -9, 0, 31, ... The permitted operations on this type are: +, -, *, div (integer division), and mod (modulo or remainder of integer division).

Keyword: **integer**.
Example: **Var** x : **integer**;

3.3.2. Real Type

It is also a numeric type representing the set of real numbers, such as: 0.25, -1.33, 2.5e+10,... The permitted operations on this type are: +, -, *, and /.
Keyword: **real**.
Example: **Var** y : **real**;

3.3.3. Character Type

This type represents all alphanumeric characters, including symbols and spaces, such as: 'a', 'A', '3', '%', ' ', etc. The supported operations for this type are: =, ≠, <, <=, >, and >=.
Keyword: `character;`
Example:
Var a : **character**;

3.3.4. Boolean Type

This type is used in logic to represent the two values: true and false. The supported operations are: NOT, AND, OR.
Keyword: **boolean**.

Example:
Var b : **boolean**;
They are defined by the truth tables below. Let A and B be two Boolean variables.

A	NON A
False	True
True	False

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

A	B	A OR B
False	False	False
True	True	Vrai
True	Faux	True
True	True	True

3.3.5. String Type

This type represents words and phrases such as "Algorithm", "Course", etc.
The keyword used is: **string**.
Example: Var c : **string**;
In summary, the declarative part of an algorithm can be represented as follows.

Example :

```
Var
x, y : integer ;
z, w : real ;
lettre : character ;
Last_name : string ;
Etat : boolean ;
Const n = 100;
arobase = '@' ;
 word = "Hello" ;
```

4. Examples of algorithms

Example1: algorithm to multiply 2 numbers and print the result:

Step 1: Start
Step 2: Get the knowledge of input. Here we need 3 variables; a and b will be the user input and c will hold the result.
Step 3: Declare a, b, c variables.
Step 4: Take input for a and b variable from the user.
Step 5: Know the problem and find the solution using operators, data structures and logic
We need to multiply a and b variables so we use * operator and assign the result to c.
That is c <- a * b.
Step 6: Check how to give output, Here we need to print the output. So write print c
Step 7: End of algorithm

Example 2: Write an algorithm to find the maximum of all the average of 3 subjects.
Follow the algorithm approach as below:

Step 1: Start
Step 2: Declare and read 3 variables, let's say **S1, S2, S3**
Step 3: Calculate the sum of all the 3 subject values and store result in Sum variable **(Sum = S1+S2+S3)**

Step 4: Divide sum by 3 and assign it to average variable. **(Average = Sum/3)**
Step 5: Print the value of **Average of 3 Subjects**
Step 6: End of algorithm

5. Expressions and Basic Instructions

5.1. The expressions

An expression is a sequence of operations applied to a set of factors (arguments or parameters). Each expression has a value and a type.

Example: a + 5 is an expression representing an addition between the two factors a and 5.

5.2. The operators

Arithmetic or logical expressions are composed of at least two terms connected by one or more operators, which can be distinguished as follows:

a. Arithmetic operators in order of priority

^ or **: Exponentiation
*, /, mod: Multiplication, Division, and Modulo (the remainder of the division)
+, -: Addition and Subtraction

b. Logical or Boolean Operators

NOT: Logical NOT (negation)
AND: Logical AND (conjunction)
OR: Logical OR (disjunction)
NOT AND: Negation of conjunction
NOT OR: Negation of disjunction

c. Comparison or relational operators

>= : Greater than and greater than or equal to
< , <= : Less than and less than or equal to
= , ≠ (or < >) : Equal to and not equal to

Note: Logical expressions can be composed of logical and/or relational operators. For example, (x < 10) AND (y >= 5) is True if x is less than 10 and y is equal to or greater than 5, and False otherwise.

5.3. Rules for evaluating an expression

The computation of the value of an expression containing more than one operator depends on the interpretation given to that expression.

Example: X + Y * Z is an ambiguous expression.

- Parentheses can resolve ambiguity: (X + Y) * Z or X + (Y * Z).
- In the absence of parentheses, to avoid ambiguity, evaluation rules have been established.

A decreasing order of priority among operators is introduced as follows:

- Unary operators: + (example: +X), - (example: -X), NOT (NOT X).
- Multiplicative operators: *, /, Div, Mod, AND.
- Additive (binary) operators: +, -, OR.
- Relational operators: <, ≤, =, ≠, ≥, >.

If an expression contains multiple operators of the same priority, these operators are left-associative.

Therefore, the interpretation assigned to X + Y * Z is indeed X + (Y * Z).

- In cases where one wishes to modify the semantics defined by these rules, parentheses must be introduced.

6. Basic instructions

In algorithms, there are three basic instructions: Read, Write, and Assignment.

6.1.The assignment statement

This statement is fundamental in algorithmic; a value can be stored in a variable with the use of assignment operator. The assignment operator (←) is used in assignment statement and assignment expression. Operand on the left hand side should be variable and the operand on the right hand side should be variable or constant or any expression. When variable on the left hand side is occur on the right hand side then we can avoid by writing the compound statement. For **Example**:

int x ← y;
int Sum ← x + y + z;

It allows assigning a value to a variable according to the following syntax:
variable ← expression;

An assignment statement is executed as follows:

Evaluation of the expression located to the right of the statement; and assignment of the result to the variable located to the left of the statement.

The expression can be:

- a constant (c ← 10)
- a variable (v ← x)
- an arithmetic expression (e ← x + y)
- a logical expression (d ← a or b)

Note: - A constant never appears on the left side of an assignment statement. Example of an incorrect statement:

Const z = 1;
z ← 2; "**Incorrect"**

- After an assignment, the old content of a variable is replaced (overwritten) by the new content.

Example

Var x: integer;
x ← 1;
x ← 2;

- After the second assignment, the value of x has become 2 (the value 1 is overwritten).
- An assignment statement must be between two compatible types.

Note

When a variable appears on both sides of the assignment operator, it typically denotes an **accumulation** operation. It means that the values are gradually aggregated or modified within a variable as the algorithm progresses.
For example: x ← 10; y ← 3;

x ← x+5 implies adding 5 to the current value of x and storing the result back into x.
y ←y*6 he current value of y was multiplied with 6 and storing the result back into y.
As result x contains 15 and y contains 18.

Example

Var
x, y: integer;
z: real;
a, b: character;
ans : bool;

Correct instructions	Incorrect Instructions
x ← y ; y ← x ; z ← x ; a ← b ; b ← a ;	x ← z ; x ← a ; b ← y ; a ← z ; ans ← b ;

Example : Unroll this portion of the algorithm.

```
y  ← 2 ;
x  ← y ;
x  ← x*2 ;
x1 ← y + x ;
1  ← x1+2 ;
y1 ← y1*2 ;
x  ← y1+3 ;
```

Instruction	y	x	x1	y1
y ←2 ;	2	?	?	?
x←y ;	2	2	?	?
x ←x*2 ;	2	4	?	?
x1←y+x ;	2	4	6	?
y1 ←x1+2 ;	2	4	6	8
y1 ←y1*2 ;	2	4	6	16
x ←y1+3 ;	2	19	6	16

Remark : you can assign an integer type value to a real type variable, but not vice versa..

6.2. The input instruction

This instruction puts the computer in a reading state. To read the data, the user must enter or input the data, for example, via the keyboard. The following algorithm will display on the screen the sum of two values entered via the keyboard:

Syntax :

read (var1, var2,…) ;
Example:
read(x): reads and stores a given value in the memory location associated with x.
read(x, y): reads and stores two values, the first one in x and the second one in y.

Illustration

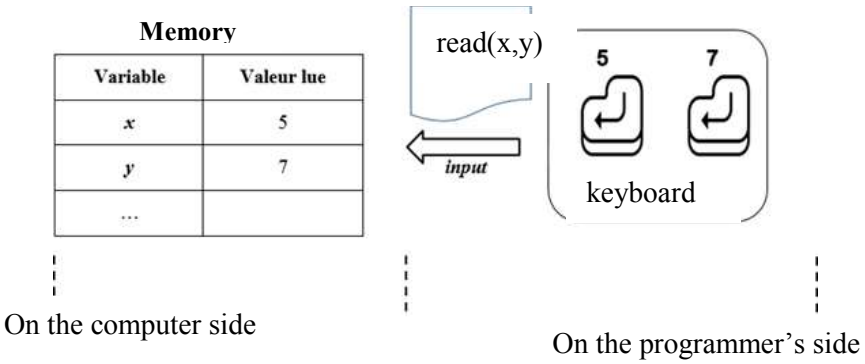


Figure 1.3. Reading operation.

6.3. The output instruction

This instruction simply performs the display of the values of the objects mentioned in the list. Objects are separated by ',‘

• An object can be :

A value : like 5, “hello”,…. **A value is displayed as it is.**
A variable : like x, y…. **It is the value of the variable that is displayed.**
An expression : like (x+y)/2, A or B,… **The expression is evaluated and the result is displayed.**

Syntax

print (var1, var2, expr1, expr2, ...) ;
write (var1, var2, expr1, expr2, ...) ;

Remark :

- It is important to precede any display of a value of a certain variable with a message so that the user understands the values displayed (as in the last display).
- In the case of writing an expression, it is the result of evaluating that expression that is displayed, not the expression itself.

Example

Given two variables x and y such that x=5 and y=7, the instruction:
print (x+y); or write (x+y);
Outputs the result of adding x and y (which is 12).

Illustration

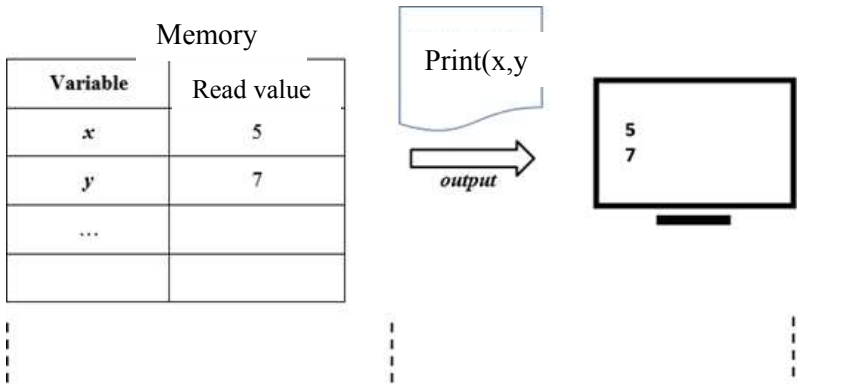


Figure 5. Writing **o**pération

Example

Algorithm Average_of_two_real_numbers
Var x, y, z: real;
Begin
print ("Enter the first value: ");

```
read (x);
print ("Enter the second value: ");
 read (y);
z ← (x + y)/2;
print ("The average is: ", (x + y)/2 ); // in this case we don't need z
End
```

If the user enters 10 for x and 20 for y, then the output will be : The average is 15.

7. The essentials of the C language

After finalizing the algorithm and solving the problem, it must be translated into a programming language to be usable on the machine. The program is then compiled and executed using a compiler. C is a highly popular and versatile programming language known for its speed and efficiency. Mastering C not only opens doors to various programming opportunities but also facilitates learning other popular languages like Java, Python, C++, and C#. Due to its similar syntax, transitioning between these languages becomes seamless. Furthermore, C's exceptional speed makes it ideal for performance-critical applications.

7.1. Creating C Programs

There are four fundamental stages, or processes, in the creation of any C program:

1- Editing
2- Compiling
3- Linking
4- Executing

Editing: Editing is the process of creating and modifying C source code. The name given to the program instructions you write. Some C compilers come with a specific editor program that provides a lot of assistance in managing your programs. In fact, an editor often provides a complete environment for writing, managing, developing, and testing your programs. This is sometimes called an integrated development environment (IDE). We can also use a general-purpose text editor to create your source files, but the editor must store the code as plain text without any extra formatting data embedded in it.
Working with Linux, the most common text editor is the Vim editor. Alternately you might prefer to use the GNU Emacs editor. With Microsoft Windows, you could use one of the many freeware and shareware programming editors.

Compiling: Compilation is a two-stage process. The first stage is called the preprocessing phase, during which the code may be modified or added to, and the second stage is the actual compilation that generates the object code. The source file can include preprocessing macros, which you use to add to or modify the C program statements.

The compiler converts the source code into machine language and detects and reports errors in the compilation process. The input to this stage is the file you produce during your editing, which is usually referred to as a source file.
The compiler can detect a wide range of errors that are due to invalid or unrecognized program code, as well as structural errors where, for example, part of a program can never be executed. The output from the compiler is known as object code and it is stored in files called object files, which usually have names with the extension .obj in the Microsoft Windows environment, or .o in the Linux/UNIX environment.
The compiler can detect several different kinds of errors during the translation process, and most of these will prevent the object file from being created.
The result of a successful compilation is a file with the same name as that used for the source file, but with the .o or .obj extension.

Linking : The linker combines the object modules generated by the compiler from source code files, adds required code modules from the standard library supplied as part of C, and welds everything into an executable whole. The linker also detects and reports errors; for example, if part of the program is missing or a nonexistent library component is referenced.
In practice, a program of any significant size will consist of several source code files, from which the compiler generates object files that need to be linked. A large program may be difficult to write in one working session, and it may be impossible to work with as a single file. By breaking it up into a number of smaller source files that each provides a coherent part of what the complete program does, you can make the development of the program a lot easier. The source files can be compiled separately, which makes eliminating simple typographical errors a bit easier.

Executing: The execution stage is where the program is run, having completed all the previous processes successfully. Unfortunately, this stage can also generate a wide variety of error conditions that can include producing the wrong output, just sitting there and doing nothing, or perhaps crashing your computer for good measure. In all cases, it's back to the editing process to check your source code.
The processes of editing, compiling, linking, and executing are essentially the same for developing programs in any environment and with any compiled language. Figure 1-6 summarizes the process the code goes through during the creation of a C program.

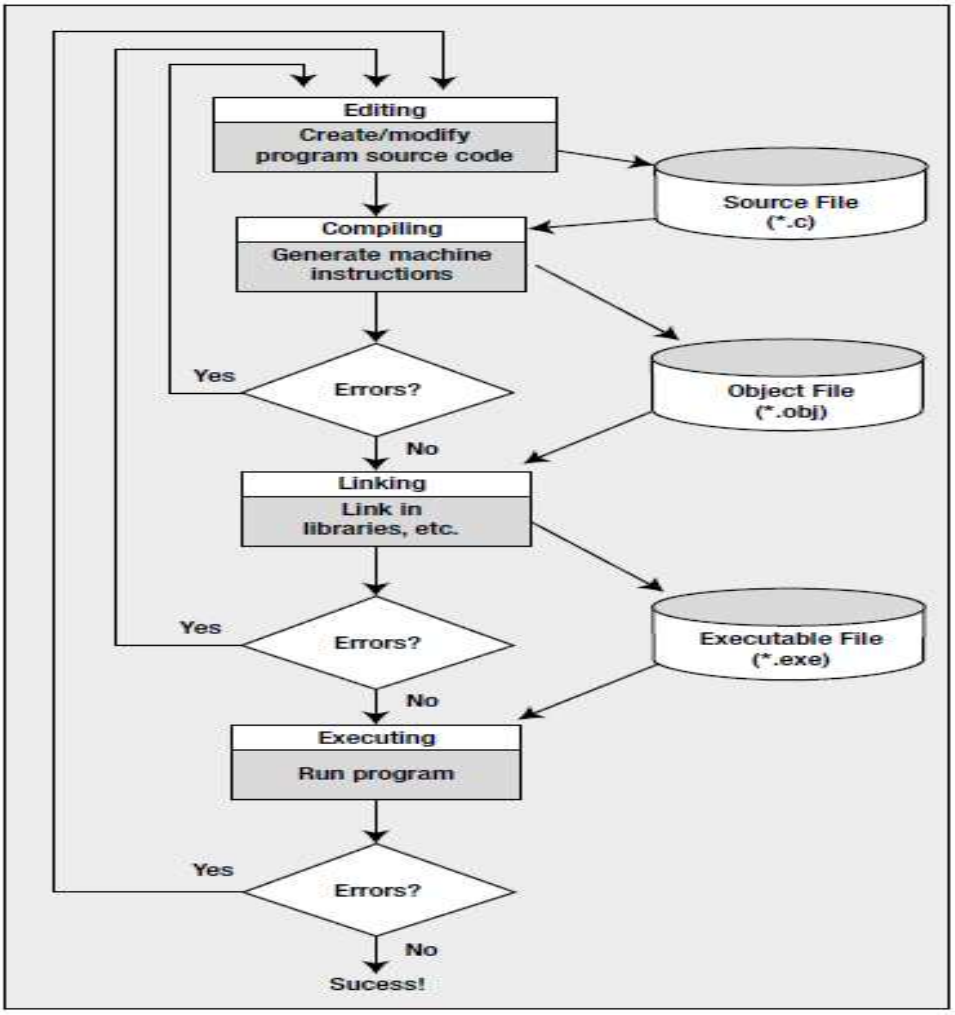


Figure 1.6. Creating and execution a program

Its versatility allows for its use in a wide range of applications and technologies, solidifying its importance in the programming world.

A summary of the basic concepts previously covered in algorithmic is presented in the following table.

In algorithm	**In C langauge**
Begin	{
End	}
read()	scanf()
print()	printf()
←	=
=	= =
integer	int
real	float
character	char
boolean	bool

Example

```
#include <stdio.h>
int main() {
   int number1, number2, sum;
   printf("Enter two integers: ");
   scanf("%d %d", &number1, &number2);

   // calculate the sum
   sum = number1 + number2;
   printf("%d + %d = %d", number1, number2, sum);
   return 0;
}
```

Execution:
Enter two integers:
3 9
3+9=12

8. Practice Exercises

1. Indicating the possible type of each variable.

. z ← (x > 2) and (y < 5) ;
z ← ((x + y) > 2) and y ;
z ← (x > 2) and y ;
z ← ((x > 2) and y) < 5 ;
if (y <>0) then z ← x/y ;
else z ← false ; endif ;

2. *a. Given a time in hours h, minutes m, and seconds s. Write an algorithm that transforms this time into seconds.Given a time in seconds.
b. Write an algorithm that transforms this time into h, m, s.

3. *Write an algorithm named **Disp_Double** that performs the following steps:
a. Assign the value 4 to an integer variable named val.
b. Assign the double of val to a double variable named double.
c. Display the values of val and double.

4. *Write a C program that performs the following steps:
a. Reads the price excluding tax of an item.
b. Reads the number of items purchased.
c. Reads the VAT rate (tax rate).
d. Calculates the total amount including tax.
e. Displays the total amount including tax.

Solution2*:

a.

```
 Algorithm Time_in_secondes;
Var h, m, s, T : integer;
Begin
/*Data reading*/
Print (“Enter the time in hours, minutes, and seconds :“);
Read (h, m, s);
/*Result calculation*/
T←h * 3600 + m * 60 + s ;
/*Resulat display*/
Print (“The time in secondes is : “, T);
End
```

b.

```
Algorithm Time_in_HMS;
Var h, m, s, T : integer;
Begin
/*Data reading*/
Print (“Enter the time in secondes : “);
Read (T);

/*Result calculation*/
hT←  Div 3600;                    s← T Mod 60;
T← T Mod 3600;                    h← T Div 3600;
m← T Div 60;                      m← (T mod 3600) Div 60;
                                  s← (T Mod 3600) mod 60;
 /*Result display*/
Print ("h=", h, " m=", m, " s=", s);
End
```

Solution3*:

```
Algorithm Disp_Double
Begin
val, double: integer ;
val ← 4
double ← val * 2
write (" the double of : ", val, "is: ", double);
End
```

Solution 4*:

```
#include <stdio.h>
int main() {
     double priceExclTax, vatRate, totalInclTax;
    int quantity;
    printf("Enter the price excluding tax: ");
    scanf("%lf", &priceExclTax);
    printf("Enter the number of items: ");
    scanf("%d", &quantity);

    printf("Enter the VAT rate (in percentage): ");
    scanf("%lf", &vatRate);

    // Calculate total amount including tax
    double totalExclTax = priceExclTax * quantity;
    totalInclTax = totalExclTax * (1 + vatRate / 100);

    // Display the result
    printf("Total amount including tax: %.2f\n", totalInclTax);
}
    Total_With_Taxes();
return 0;}
```

Chapter 2
Control structures

In computer science, control structures are like signposts that guide the flow of a program. By using self-contained modules called logic or control structures, we can make algorithms or programs easier to understand. These structures help decide which path the program takes based on certain conditions. There are three basic types of control structures in computer science:

a. Sequence logic, or sequential flow
b. Selection logic, or conditional flow

1. Sequential Logic (Sequential Flow)

Sequential logic, as the name implies, follows a step-by-step or sequential flow, where the program's execution depends on the sequence of instructions provided to the computer. Unless new instructions are provided, the modules are executed in the order they appear. This sequence can be explicitly specified with numbered steps or implicitly follows the order in which modules are written. Most processing tasks, even some complex problems, typically follow this straightforward flow pattern.

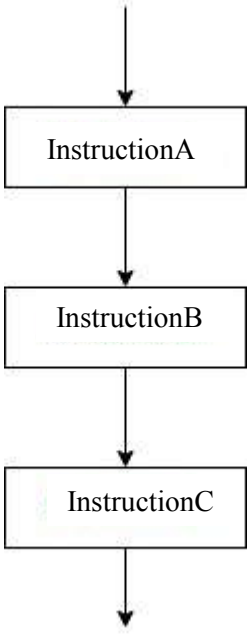


Figure 2.1. Sequential

2. Selection Logic (Conditional Structures)

Selection Logic simply involves a number of conditions or parameters which decides one out of several written modules.

In conditional control , the execution of statements depends upon the condition-test. If the condition evaluates to true, then a set of statements is executed otherwise another set of statements is followed. This control is also called Decision Control because it helps in making decision about which set of statements is to be executed. These structures can be:

2.1. Single Alternative: In this form, an action that corresponds to one or more instructions is executed if a condition is satisfied. Otherwise, the algorithm proceeds directly to the block of instructions that immediately follows the conditional block

Syntax : In Algorithmic vs in C language

```
If (condition(s)) then                    if(condition(s)) {
     (Instruction(s));                    (Instruction(s));
Endif                                      }
```

Flowchart

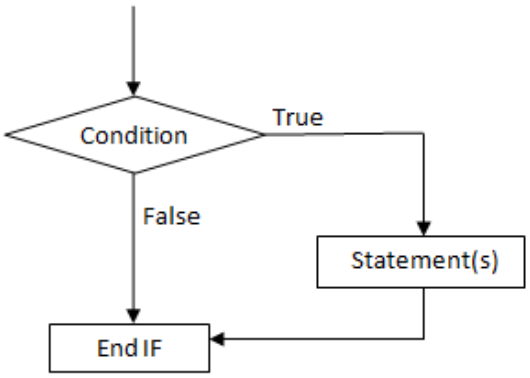


Figure 2.2. Simple Alternative Control

Example : An algorithm to display a number if it is negative

```
Algorithm number;
var
 number : integer;
Begin
 print("Enter an integer: ");
 read(number);
 // true if number is less than 0
    if (number < 0) then
       print("You entered", number);
    endif
 print("The if statement is easy.");
End.
```

Program in C

```
// Program to display a number if it is
negative

#include <stdio.h>
int main() {
   int number;
   printf("Enter an integer: ");
   scanf("%d", &number);

   // true if number is less than 0
   if (number < 0) {
      printf("You entered %d.\n", number);
   }
   printf("The if statement is easy.");
   return 0;
```

Execution

Output 1

```
Enter an integer: 5
The if statement is easy.
```

Output 2

```
Enter an integer: -2
You entered -2.
The if statement is easy.
```

When the user enters 5, the test expression number<0 is evaluated to false and the statement inside the body of if is not executed
And when he enters -2, the test expression number<0 is evaluated to true. Hence, You entered -2 is displayed on the screen.

2.2. Double Alternative: This structure form allows choosing between two actions depending on whether a condition is satisfied or not.

Syntax

```
If (Condition(s)) then              if (Condition(s))
      InstructionA(s);                    { InstructionA(s);}
Else                                      else
      InstructionB(s);            { InstructionB(s);}
Endif
```

Flowchart

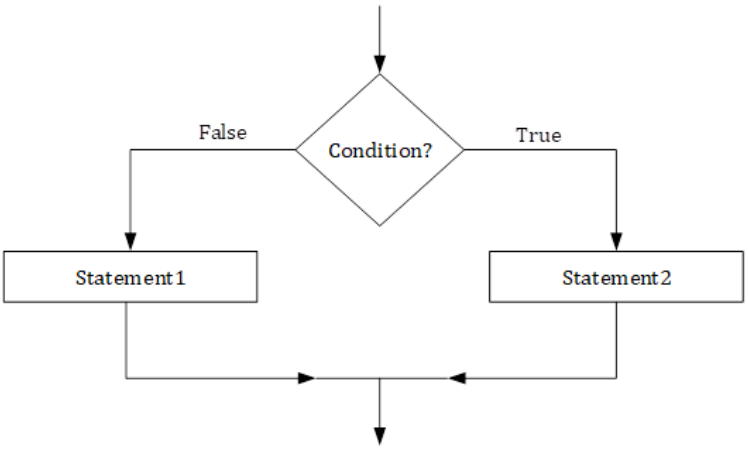


Figure 2.3. Double Alternative Control

Example : Check whether an integer is odd or even.

```
Algorithm number;
var
   number: integer;
Begin
 print("Enter an integer: ");
   read(number);
   // True if the remainder is 0
   if  (number%2 == 0) then
      print(number,"is an even integer.");
   else
      printf number,  is an odd integer.");
   endif
End.
```

// Check whether an integer is odd or even

```
#include <stdio.h>
int main() {
   int number;
   printf("Enter an integer: ");
   scanf("%d", &number);
   // True if the remainder is 0
   if  (number%2 == 0) {
      printf("%d is an even integer.",number);
   }
   else {
      printf("%d is an odd integer.",number);
   }
   return 0;
}
```

Execution

```
Enter an integer: 7
7 is an odd integer.
```

When the user enters 7, the test expression number% 2= =0 is evaluated to false. Hence, the statement inside the body of else is executed.

Note

- The condition evaluated after the "If" statement is a variable or a boolean expression that, at a given moment, is either True or False.
  For example: x=y; x <= y;
  Example: x ← 5 ; y ← 9 ; If (x = y) Then print (″x is equal to y″) ; In this example, the message "x is equal to y" will not be displayed since the condition (x = y) is not satisfied.
- Some problems sometimes require formulating conditions that cannot be expressed in the form of a simple comparison.

For example, the condition x ∈ [0, 1[is expressed by the combination of two conditions x >= 0 and x < 1 that must be satisfied simultaneously.

To combine these two conditions, logical operators are used. Thus, the condition x ∈ [0, 1[can be written in the form: (x >= 0) AND (x < 1). The latter is called a compound or complex condition.

Example

x ← 5 ; y ← 9 ;

if (x = y) **then print** (″x est égale à y ″) ;

else print (″x est différente de y ″) ;

With this form, we can handle both possible cases. If the condition (x=y) is satisfied, the first message is displayed; if it is not satisfied, the second message is displayed.

2.3. if...else Ladder

The if...else statement executes two different codes depending upon whether the test expression is true or false. Sometimes, a choice has to be made from more than 2 possibilities.It is used when there are more than two possible action based on different conditions.

Syntax

```
if (test expression1) {
   // statement(s)
}
else if(test expression2) {
   // statement(s)
}
else if (test expression3) {
   // statement(s)
}
.
.
else {
   // statement(s)
}
```

Flowchart

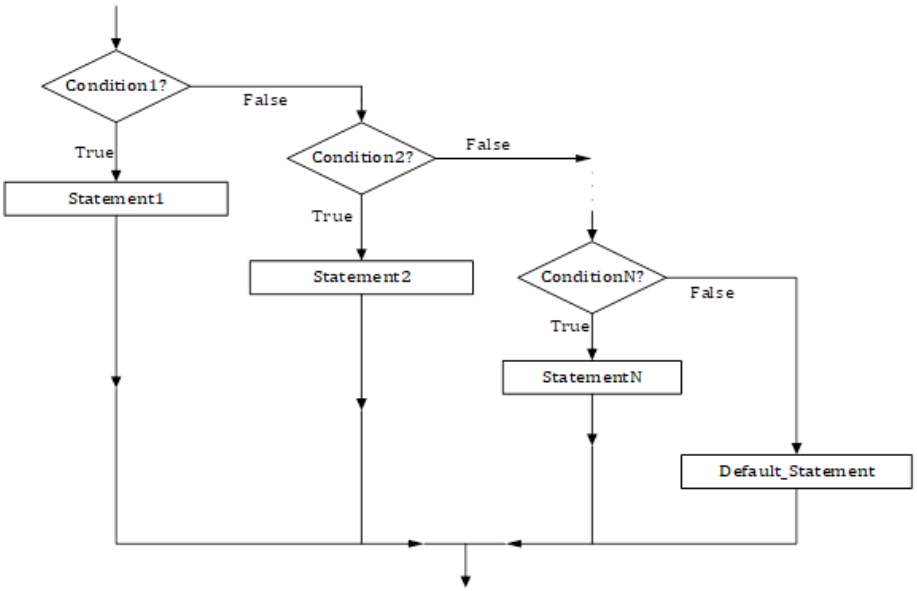


Figure 2.4. Working of if-else-if Ladder

Example: An algorithm that find largest from three numbers given by user to explain working of if-else-if statement or ladder.

```
Algorithm largest3numbers;
var
 a,b,c : integer;

Begin
 print("Enter three numbers: ");
 read(a, b,c);
        if(a>b and  a>c) then
                print("Largest = ", a);
        endif
        else if(b>a and b>c) then

                print("Largest =", b);
        endif
        else

                printf("Largest = ", c);

End.
```

```c
#include<stdio.h>
int main()
{       int a,b,c;
        printf("Enter three numbers: \n");
        scanf("%d%d%d", &a, &b, &c);
        if(a>b && a>c)
        {
              printf("Largest = %d", a);
        }
        else if(b>a && b>c)
        {
              printf("Largest = %d", b);
        }
        else
        {
              printf("Largest = %d", c);
        }
        return(0);
}
```

2.4.Nested Tests (If ….else ….If statement)

The **"If...Then...Else"** form allows for two choices corresponding to two different treatments. In other situations, there may be more than two cases, which make this alternative insufficient to handle all possible cases (see example below).

The complete form allows choosing between several actions by nesting simple forms according to the syntax below.

Flowchart for nested statement

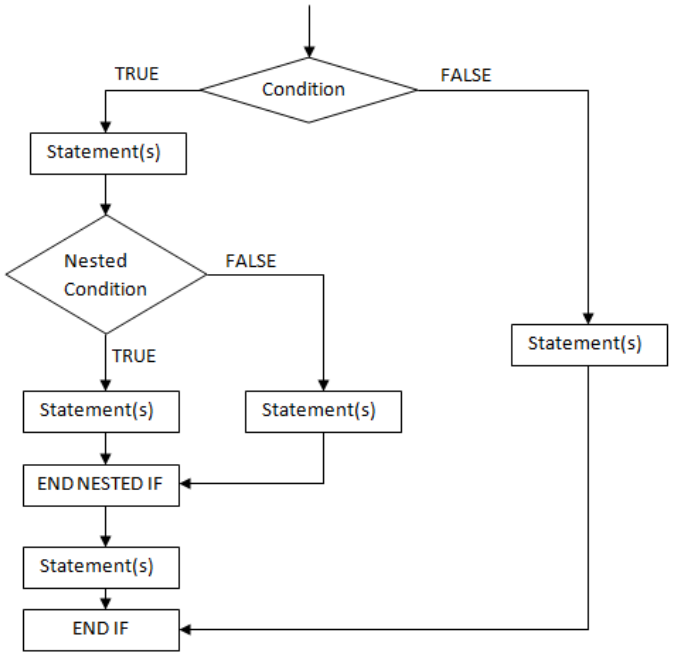


Figure 2.5. Flowchart for nested statement

Example: Write an algorithm that solves a second-degree equation.

```
Algorithm Equation 2;
 Var
  a , b , c , delta , x , x1 , x2 : real
; Begin
   print ( ' Give a , b et c ' ) ;
   read ( a , b , c ) ;
   delta ← b * b – 4 * a * c ;
  If ( delta < 0 ) then
   print ( ' No solution ' )
   Else
    If ( delta = 0 ) then
      x   ←    - b / ( 2 * a ) ;
      print ( ' Double solution  x = ' , x ) ;
   Else
     x1 ← ( - b – sqrt ( delta ) ) / ( 2 * a ) ;
     x2   ←  ( - b + sqrt ( delta ) ) / ( 2 * a ) ; ;
     print ( ' There are two solutions: x1 = ' , x1 , ‘ x2 = ‘  x2 ) ;
   Endif ;
Endif
End.
```

2.5. Multiple Selection: The switch Statement

There is another variant of conditional instructions that allows performing different actions depending on the different values a variable can have. This structure is described as follows:

Syntax

```
According to choice Do
Case choice1: Block1;
Case choice2: Block2;
...
Case choiceN: BlockN;
Default case: BlockM; //optional EndSelect
```

choice: Variable or Expression.

If the choice does not have any value among (choice1, choice2,...,choiceN), the default block will be executed.

In C

Syntax

```
Switch  (variable)
value1: instruction(s) 1;
value2: instruction(s) 2;
…
valueN: instruction(s) N;
default: default instruction(s);
End;
```

Flowchart switch statement

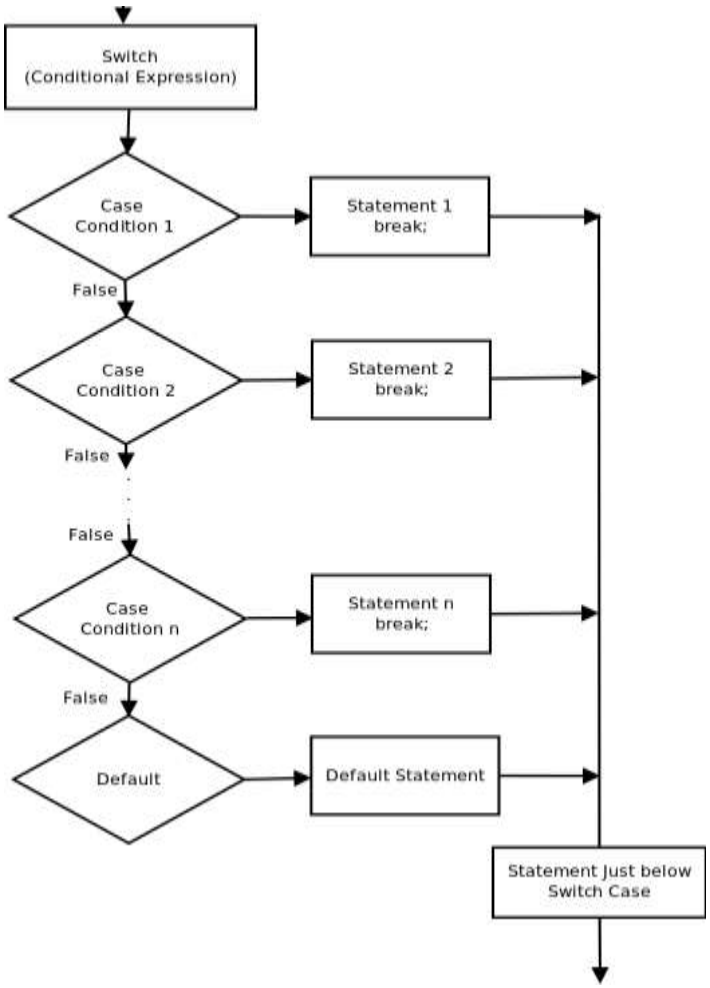


Figure 2.6. Flowchart switch statement

Note

.In the multiple-choice test structure:

- The action can be a sequence of instructions;
- The value is a constant of the same type as the variable;
- The "default" part is executed if none of the other cases is satisfied;
- The execution of the different cases (including the default case) is exclusive, meaning the execution of only one case causes the exit of this structure.

Example

In the following, the name of the corresponding weekday is displayed according to the value of the variable "day".

```
day ← 5 ;
switch day
1 : print ("Sunday") ;
2: print ("Monday") ;
3: print ("Mardi") ;
4: print ("Wednesday") ;
5: print ("Thursday ") ;
6: print ("Friday ") ;
 7: print ("Saturday") ;
 Défaut : print ("Invalid day number Haut du formulaire .") ;
End
```

So, the expression "**Thursday**" is displayed in this case.

9. Practice Exercises

1. Write an algorithm that reads two numerical variables a and b and displays them before and after their permutation. For example, before: a=5 and b=7, after: a=7 and b=5.
2. Write a simple algorithm to input an integer from user. Check if the given integer is negative, zero or positive?
3. *Write an algorithm that asks the user to type in two integer values at the terminal. Test these two numbers to determine if the first is evenly divisible by the second, and then display an appropriate message at the terminal.
4. *Write an algorithm to create menu driven calculator that performs basic arithmetic operations (add, subtract, multiply and divide) using switch case and functions. The calculator should input two numbers and an operator from user. It should perform operation according to the operator entered and must take input in given format.

Solution 3*

```
Algorithm numbers;
 num1, num2 : integer;

   // Prompt user for input
   write("Enter the first integer: ");
   read(num1);

   write("Enter the second integer: ");
   read(num2);

   // Check for division by zero
   if (num2 == 0) {
      write("Division by zero is not allowed.\n");
   } else {
      // Check divisibility
      if (num1 % num2 == 0) {
         write(num1, " is evenly divisible by ", num2);
      } else {
         write(num1, "is not evenly divisible by ", num2);
      }
   }
}
```

Solution 4*

```
#include <stdio.h>

int main() {
   double num1, num2, result;
   char operator;

   // Display instructions
   printf("Menu-Driven Calculator\n");
   printf("Enter your calculation in the format: number1 operator number2\n");
   printf("Available operators: + for addition, - for subtraction, * for multiplication, /
for division\n");

   // Input numbers and operator
```

```
    printf("Enter your calculation: ");
    scanf("%lf %c %lf", &num1, &operator, &num2);

    // Perform operation based on the operator
    switch (operator) {
        case '+':
            result = num1 + num2;
            printf("Result: %.2lf\n", result);
            break;
        case '-':
            result = num1 - num2;
            printf("Result: %.2lf\n", result);
            break;
        case '*':
            result = num1 * num2;
            printf("Result: %.2lf\n", result);
            break;
        case '/':
            if (num2 != 0) {
                result = num1 / num2;
                printf("Result: %.2lf\n", result);
            } else {
                printf("Error: Division by zero is not allowed.\n");
            }
            break;
        default:
            printf("Invalid operator! Please use +, -, *, or /.\n");
    }

    return 0;
}
```

Chapter 3
The repetitive structures (Loops)

In real life we come across situations when we need to perform a set of task repeatedly till some condition is met. Such as sending email to all employees, deleting all files, printing 500 pages of a document. All of these tasks are performed in loop.

Loop instructions are used for repetitive processing. That is, they allow executing a process multiple times. Like any process, a loop must run a certain number of times. Therefore, after each execution, a decision to end the loop must be made. This is done based on an exit condition for the loop.

1. The For loop instruction

The "For" loop is an unconditional iterative loop where a set of instructions is executed iteratively apredetermined 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 aninitial value, a final value, and a method of incrementing or decrementing.
- Instruction Block: Executed in each iteration.

Syntax

For (Counter ← Initial_Value To Final_Value Step Step_Value) Do

Instruction Block

EndFor

`Rest of the instructions`

- **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 ← counter+step. Generally equal to 1.

Note

- 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 ≥ Final_Value. In the case of a negative Step_Value, it is decremented to counter ≤ 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.

For loop Flowchart

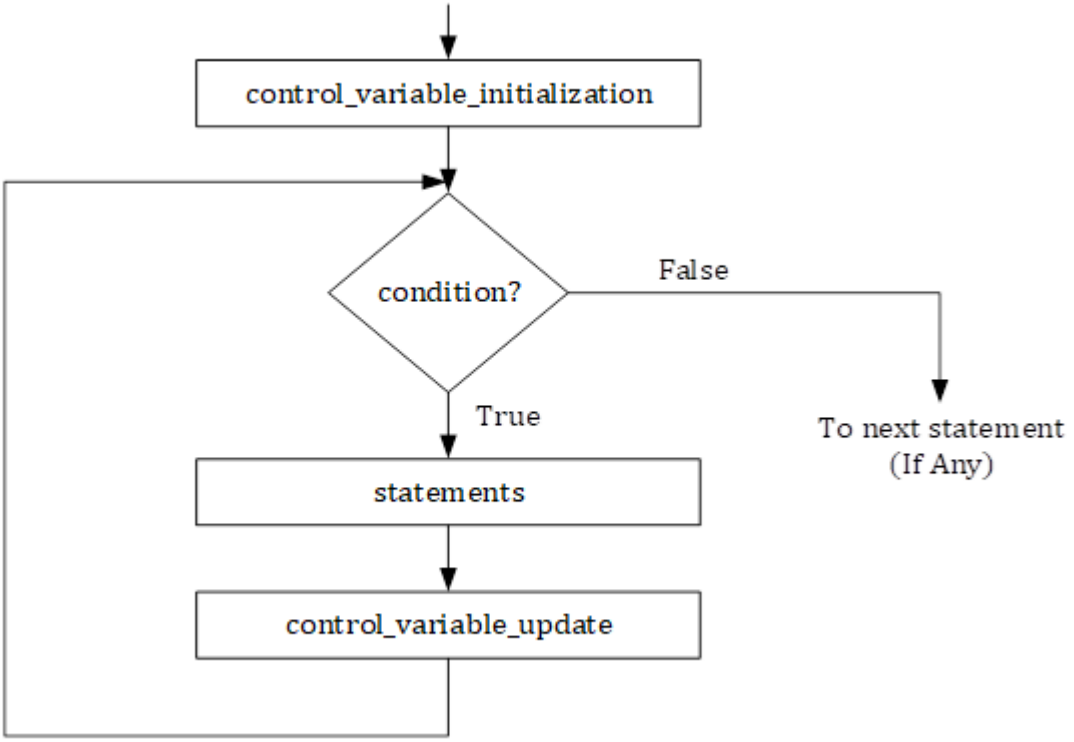


Figure 3.1. Flowchart For loop

Syntax in C

```
for (control_variable_initialization;condition;control_variable_update)
{
    statement(s);       //body of loop
}
```

Example: Increasing/Decreasing Counter

The following two algorithms count from 1 to N and from N to 1, respectively.

Algorithm Increasing_Counter ;

Var i : integer;
Const N=100 ;
Begin
For(i ← 1 to N) /* By default, the step size = *1* */
print (i);
EndFor
End.

Execution Result**:** 1,2, 3, … , 99, 100

Example : Program to print "Programming is useful." 5 times to explain working of for loop

```c
#include<stdio.h>
int main()
{
 int i;
 for(i=1;i<=5; i++)
 {
  printf("Programming is useful.\n");
 }
 return(0);
}
```

Output

```
Programming is useful.
Programming is useful.
Programming is useful.
Programming is useful.
Programming is useful.
```

2. The While Loop

The "While" loop is a pre-condition loop where a set of instructions is repeatedly executed based on a booleancondition. 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.

Syntax In algorithmic vr C language

```
control_variable_initialization;
While (condition (s) ) do
        statement(s);
        control_variable_update;
  EndWhile
```

```
control_variable_initialization;
while (condition(s))
{   statement(s);
   control_variable_update;
}
```

The condition is always placed between the keywords " **While** " and "**do**" in algorithms. To create the condition, we use comparisons (>, <, =, ≠, ...) and logical operations (and &&, or ||, not !, ...).

While loop Flowchart

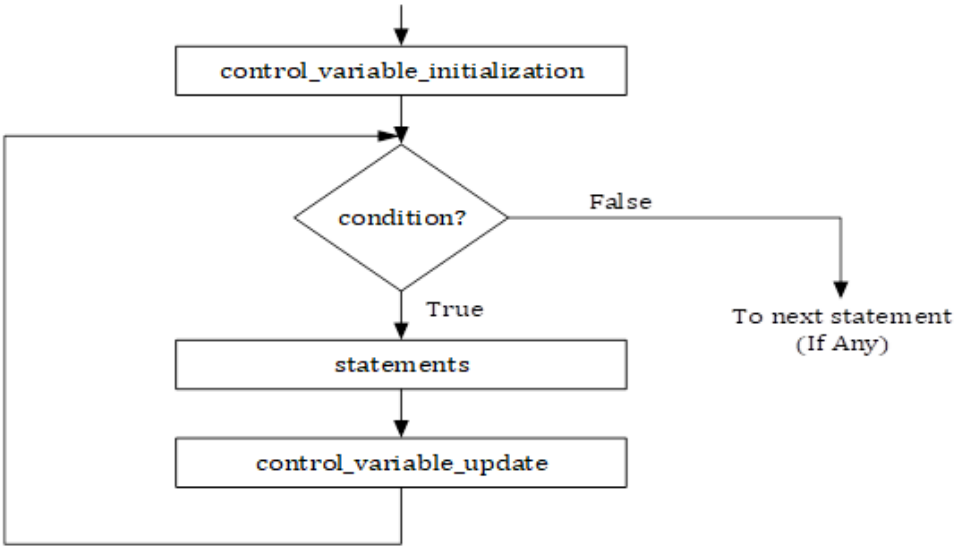


Figure 3.2. Flowchart While loop

Example: Let's rewrite the previous algorithm using this instruction.

```
var
i : integer ;
Begin
i ← 1 ;
Whike (i<=100) do
print (i) ;
i ← i+1 ;
EndWhile.
```

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:

a. 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.
b. Condition: A Boolean expression with a true or false value.

Syntax

Do

Instruction Block

while (Condition(s));

Rest of the instructions

The execution process of the "Do...While" conditional loop involves executing the block of instructions between "Do" and "While" in the algorithm. Following this, the condition expression is evaluated, yielding 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 that point, the loop exits, proceeding to the instruction immediately following the loop.

Do While loop Flowchart

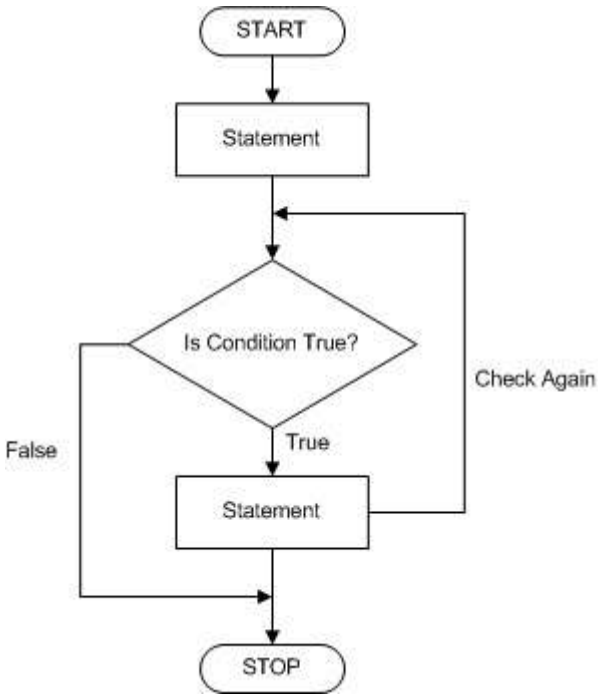


Figure 3.3. Flowchart Do While loop

Example

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

```
Algorithm readNbrs
var x, nb : integer;
/* x to read nbrs, nb to count nbrs */
Begin
Nb ← 0 ;
Do
      print ("enter a nbr ") read (x)
      nb ←  nb+1;
 While x≠0;
print("The number of numbers is", nb-1);
EndWhile ;
End.
```

```
entrer un nbr: 5
entrer un nbr: 7
entrer un nbr: -2
entrer un nbr: 0
Le nombre de nombres est 3
```

4. Nested Loops

Loops can be nested within each other. Two or more nested loops can be the same or different

Example

```
For ( i ← 1  to 2)
print ("i = ", i) ;
For( j ← 1 to 3 )              // loop 1
print ("j = ", j) ;                 // loop 2
EndFor ;
EndFor;
```

In the above example, each iteration of the outer loop (loop 1) executes the inner loop (loop 2) until its completion before moving on to the next iteration, and so on until both loops are finished. Therefore, the execution result can be represented as follows:

5. Practice Exercises

1. Write an algorithm that takes an integer keyed in from the terminal and extracts and displays each digit of the integer in English. So, if the user types in 932, the algorithm should display : nine three two

2. Write a while loop that prints out the numbers 0, 4, 8, 12, ... , 96, 100.

3. Write an algorithm that will find the GCD (greatest common divisor) and LCM (least common multiple) of two positive integers.

4. Enter a number from the keyboard and then calculate the number of digits and the sum of digits of that number using a while loop.

5. Write an algorithm that prompts the user to input a series of integers until the user stops by entering 0 using a do-while loop. Calculate and print the sum of all positive integers entered.

6. *Factorial of an integer n, written n!, is the product of the consecutive integers 1 through n. For example, 5 factorial is calculated as 5! = 5 x 4 x 3 x 2 x 1 = 120

   Write an algorithm to generate and print a table of the first 10 factorials.

7. *Write an algorithm that calculates the sum of the digits of an integer. For example, the sum of the digits of the number 2155 is 2 + 1 + 5 + 5 or 13.

Solution 6*

```
#include <stdio.h>

int main() {
   int i;
   unsigned long long factorial = 1; // Using unsigned long long to handle large
factorial values

   printf("Table of the first 10 factorials:\n");

   // Loop through numbers 1 to 10
   for (i = 1; i <= 10; i++) {
      factorial *= i; // Calculate factorial
      printf("%2d! = %llu\n", i, factorial); // Print the factorial
   }

   return 0;
}
```

Solution 7*

```
#include <stdio.h>

int main() {
   int n, digit, sum = 0;

   // Input the integer
   printf("Enter an integer: ");
   scanf("%d", &n);

   // Calculate the sum of digits
   while (n != 0) {
      digit = n % 10; // Extract the last digit
      sum += digit;   // Add the digit to the sum
      n /= 10;        // Remove the last digit
   }

   // Output the result
   printf("The sum of the digits is: %d\n", sum);
   return 0;
}
```

Chapter 4
The Arrays and Character Strings

A series of variables ordered by index and treated as a single block is called indexed variables. These variables can be of the same type to take the form of an array. An array, that contains character data-type variables, is known as a string.

The Array type

A variable of array type is no longer a simple variable, but rather a complex data structure that groups information of the same type, accessible via indexes indicating their location. An array can be seen as a group of variables of the same type with the same name.

We distinguish two types of arrays: one-dimensional arrays (vectors) and multi-dimensional arrays (matrices).

Dimension: The dimension of an array is the number of indices needed to identify a single element.

Index: When data is stored in an array, the element is identified by an index which is a non-negative integer (≥ 0). The index ranges from 0 to N – 1 (for C language) or from 1 to N (where N is the array size).

Array element: Elements are items stored in an array and can be accessed by theirindex.

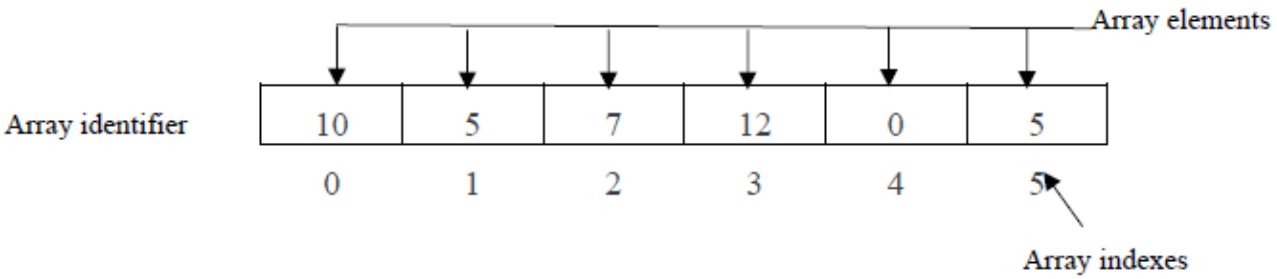
Array Length: The length of an array is determined by the number of elements it can contain.

1. One-Dimensional Array

It is also referred to as a vector, allowing you to access any of its elements using a single index. Each index value selects a specific element from the array.

1.1. Representation

The array is stored in memory as a sequence of contiguous cells. Once created, new cells cannot be added to or removed from the array (static). The figure below illustrates an array consisting of 8 real numbers.



1.2. Declaration

Var array_name : array [size] of elementType;

- arrayName: The identifier name given to the array (the variable name).
- Size: The number of elements in the array.
- elementType: The type of elements in the array, which can be of any type like integer (int), float, ...

To declare multiple arrays of the same type, use a comma "," while specifying the size of each array between square brackets [] or ().

Example

const N=100
Var
marks [N] : **array of** real
tab1[50],tab2[20] : **array of** integer;

1.3. Accessing Array Elements

An element from an array is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array and the expression inside the brackets evaluates to an integer value.

Syntax to refer to a specific element in the array:

array_name[index] ;

Example

Var Tab : array [8] of integer;

	index	0	1	2	3	4	5	6	7
Tab=	value	15	7	2-	0	9	2	0	3-

To access the first element of array 'tab', we use tab[0].

To access the fourth element, we use tab[3].

```
tab[5-3]←tab[tab[3]+1]   ⇔   tab[2]←tab[0+1]   ⇔   tab[2]←7
```

The array Tab becomes

Indice	0	1	2	3	4	5	6	7
valeur	15	7	7	0	9	2	0	-3

Note: Accessing an element that doesn't exist (if the index is greater than the array size or negative) will cause the program to terminate.

1.4. Assignment

Assigning a value x to an element i of a numeric array T is done by:
T[i] ← x
For example, the statement: T[0] ← 5; assigns the value 5 to the first element of the array T.

1.5. Reading an Array

To fill an array of N numbers, we use "read" N times like:
read (tab[0])
read (tab[1])
…
read (tab[N-1])
However, it's observed that the read instruction is repeated N times, iterating from 0 to N-1. Hence, the "for" loop can be employed with the counter serving as the index.

```
for (I ← 0 to N-1) do
read(tab[i])
endFor
```

or the reverse order

```
for (i← N-1) to 0
read(tab[i])
endFor
```

Example : Initializing the elements to 0

```
i = 0
while (i < N-1 Do)
        T[i] ← 0; i ← i+1 ; endWhile
```

The "read" instruction is used for populating the table, while the "write" instruction is employed to display to the user what is needed.

1.6. Displaying an Array

We substitute the read operation with the write operation.

```
write (tab[0])
write (tab[1])
…
write (tab[N-1])
```

```
for (i ← 0 to N-1) do
print(tab[i])
endFor
```

or the reverse order

```
for (i← N-1) to 0
print(T[i])
endFor
```

Note

- To visit all elements of an array, we generally use the "**for**" loop.
- The array size must be specified during programming (declaration), but we can give the user the impression that the array size can be changed by declaring a large array and using only part of it. We ask the user for the desired size, it must not exceed the actual array size.

Example

Write an algorithm that receives the averages of N students, where N is determined by the user, then calculates the number of students who failed the subject (average less than 10).

```
Algorithm nb_adjourned
var avg :array(100) of real;
 i, aj, N : integer
Begin
print ("enter number of students (<", MAX, ")")
read (N)
for (i ← 0 to N-1) do
read (avg[i]);
endfor
aj ← 0
for i ← 0 to N-1 do
  if  (avg[i]<10)  then
aj ← aj+1;
end if
  end for
print("the number of adjourned is ", aj)
End.
```

1.7. Somme Array Manipulation

1.7.1. Searching for elements in an array

Linear search is a straightforward algorithm that traverses the array, comparing each element to the target element. If the target element is found, a Boolean variable is set to true; otherwise, it remains false. Let's consider an array named tab with a length of N, and an element x that we aim to search for:

```
found ← false ; i ← 0 ;
while ((found = false) and (i<N)) do
if (tab[i] = x) then
found ← true ;
endIf ;
i ← i+1 ;
endWhile
```

1.7.2. Insertion of an element in an array

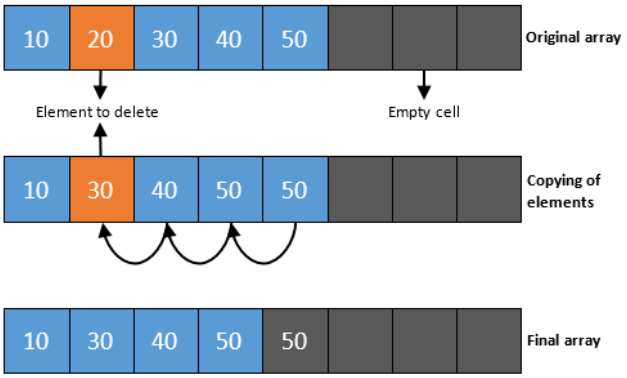
The algorithm witch inserts a given value in unsorted array.

```
  // Input the value to be inserted and its position
print("Input the value to be inserted : ");
read(x);
printf("Input the Position, where the value to be inserted :");
  read(p);
// Move all data to the right side of the array to make space
  for (i ← n to p)
     tab[i] = tab[i - 1];
   endFor
// Insert the new value at the given position
  tab[p - 1] = x;
  // Display the array after insertion
  print("After Insert the element the new array is :");
  for (i ← 0 to n)
     print(tab[i]);
  endFor
```

1.7.3. Deleting an element in an array

Logic to remove element from array

1. Move to the specified location which you want to remove in given array.
2. Copy the next element to the current element of array. Which is you need to perform array[i] = array[i + 1] .
3. Repeat above steps till last element of array.
4. Finally decrement the size of array by one.



// Algorithm to delete an element from array at specified position

```
Algorithm deleting;
Const  MAX_SIZE 100;
var
    arr: array [MAX_SIZE]of integer;
   i, size, pos : integer;
Begin
   /* Input size and element in array */
   print("Enter size of the array : ");
   read(size);
   print("Enter elements in array : ");
   for(i=0 to size) do
    read("%d", &arr[i]);
   endFor
   /* Input element position to delete */
   print("Enter the element position to delete : ");
   read(pos);
```

```
   /* Invalid delete position */
   if(pos < 0 or  pos > size)  then
     print("Invalid position! Please enter position between 1 to size);
```

```
else
/* Copy next element value to current element */
   for(i=pos-1 to i<size-1) do
         arr[i] = arr[i + 1];
   endFor
   /* Decrement array size by 1 */
   size= size-1-;
   /* Print array after deletion */
   print("Elements of array after delete are : ");
   for(i=0 to size) do
   printf("%d\t", arr[i]);
   endFor
End.
```

1.7.4. Sorting Arrays

Sorting is one of the basic operations on the arrays. Indeed; it's usually helpful when we havean array to be able to put it in some sort of order (be it numerical or alphabetical). There are many algorithms to sort a list. Some are simple and some are complex. Some are fastwhile others are slow. We'll look at a few of the most common ones.

a. Selection sort

Selection sort is a simple sorting algorithm that works by repeatedly finding the smallest element in an unsorted array and swapping it with the first element in the array. This process isrepeated until the entire array is sorted.

```
for (i ← 0 to(N – 2) do
smallest = i ;
for j ← (i + 1) to N-1 do
if (tab[j] < tab[smallest] )then
smallest = j;
endif
endFor ;
temp = tab[i]; tab[i] = tab[j]; tab[j] = temp;
endFor
```

b. Bubble Sort

Bubble sort is a simple sorting algorithm that works by repeatedly comparing adjacent elements in an array and swapping them if they are in the wrong order. Thus, the smallest element "bubbles" to the top of the array. This process is repeated until the entire array is sorted.

```
for (i ← 0 to n-2 ) do
 for (j ← 0 to n-i-1) do
    if (tab[j]>tab[j+1]) then
          tmp ← tab[j];
                    tab[j] ← tab[j+1];
                    tab[j+1] ←  tmp;
    endif;
 endFor ;
endFor.
```

2. Bidimensional array -2D array-

A two-dimensional array (2D-array) in computer science organizes data into rows and columns, resembling a grid or matrix. It differs from a one-dimensional array by allowing elements to be accessed using two indices: one for the row and one for the column.

2.1. Representation

The matrix is represented in memory by a sequence of adjacent cells. A cell cannot be removed or added to the matrix after its creation (static). The following figure represents a matrix with 4 rows and 5 columns of real numbers.

	0	1	2	3	4
0	15	7	-3	0	9
1	6	12	4	33	85
2	2	-8	17	28	-52
3	14	42	36	49	-12

2.2. Declaration

matrixName: **Array**[Rows][Columns] **of** elementType;

matrixName`: The identifier given to the matrix (variable name).

Rows`: Number of rows.

Columns`: Number of columns.

elementType`: The type of elements. It can be any type, such as integer, float, ...

The number of elements is the product of the number of rows and the number of columns.

Example :

const R=100 ;

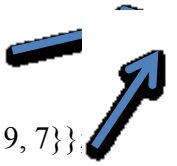
const C=100;

M : **Array** [R][C] **of** real ;
mat1,mat2 : **array**[30][20] of integer;

2.3. Initializing 2D array

There are many ways to initialize a 2D array the simple one is the initialization during declaration as:

float X[3][3]={1.2, 5, 10, 2, 4, 3, 6, 9, 7};
Or in equivalent way as
float X[3][3]={{1.2, 5, 10}, {2, 4, 3}, {6, 9, 7}};



	col 0	col 1	col 2
row 0	X[0][0]=1.2	X[0][1]=5	X[0][2]=10
row 1	X[1][0]=2	X[1][1]=4	X[1][2]=3
row 2	X[2][0]=6	X[2][1]=9	X[2][2]=7

To assign values to elements in 2D array the scanf() can be used as follows :
int M[2][2];
scanf("%d", &M[0][0]);/* to enter the value of the first element in array M*/

2.4. Accessing elements of a matrix

To access a single element of the matrix, we use the matrix name with an index inside two brackets `[` and `]` specifying the row number, and another index inside two brackets `[` and `]` specifying the column number.

To access the element in the first row and first column of matrix `mat`, we use `mat[0][0]`.

Syntax:

array_name [i][j];

2.5. Reading a matrix

To fill the matrix `M[Rows][Columns]`, we fill in `Rows` rows. Since each row is a one-dimensional array with `Columns` elements.

For (j ← 0 to C-1) do
read(M[0][j])
end for
	For (j← 0 to C-1) do
	read(M[1][j])
	endfor
		for (j← 0 to C-1) do
		read(M[L-1][j]) ;
		endFor

So, we can fill all the matrix using double loop:

```
For ( i← 0 to L -1) do
 For ( j← 0 to C-1) do
  read(M[i][j]);
 endfor;
enfor;
```

2.6. Displaying a Matrix

Similar to reading, the `write` statement is repeated.

```
for (i← 0 to L-1) do
    for (j← 0 to C-1 ) faire
 print (M[i][j])
  endFor
endFor
```

Example: Sum of elements of a matrix

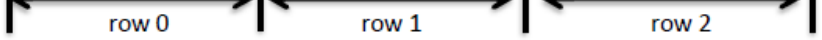
```
Algorithm sum_mat ;
const n=4, m=5;
var              mat : array[n-1][m-1] of integer ;
 i, j, sum : integer ; begin
for (i ← 0 to n -1) do begin
sum ← 0 ;
for (j ← 0 to m-1) do
sum ← sum + mat[i][j] ;
endFor;
print(sum);
endfor;
end.
```

2.7. Storage of 2D array

The elements of 2D array are stored in memory in a linear way in row-major order i.e. the rows are placed one by one in memory as one dimensional array. The element of the 2D array **int x[3][2]={1, 2, 2, 1, 1, 4};** it can be represented in this way.

Example

Element	x[0][0]	x[0][1]	x[1][0]	x[1][1]	x[2][0]	x[2][1]
Value	1	2	2	1	1	4
Address	2000	2004	2008	2012	2016	2020



Program	Output

#include <stdio.h>
int main() {
int mat[3][2]={{1,2},{5,9},{7,9}};
int i,j;
int* pt;
pt=&mat[0][0];
for(i=0;i<6;i++)
printf("%p\n", pt+i);
return 0; }
	0x7ffe872ca670
	0x7ffe872ca674
	0x7ffe872ca678
	0x7ffe872ca67c
	0x7ffe872ca680
	0x7ffe872ca684

2.8. Length of 2D array

The total number of elements in 2D array=number of rows × number of columns. For example the 2D array int A[10][7] can store 10 × 7=70 elements. Each of element occupy a space of 4 bytes in memory, and then the array has a storage space given with size of A[10][7] =70 × 4=280 bytes.

Example

Program	Output

#include <stdio.h>
int main() {
int mat[3][2]={{1,2},{5,9},{7,9}};
int length;
length=sizeof(mat)/sizeof(mat[0][0]);
printf("length=%d", length);
return 0;
}
	length=6

3. Strings in C

A character string, or simply a string, is a sequence of characters, like letters, numbers, or symbols that are grouped together. It's commonly used in programming to represent text data. For example, "hello" and "123abc" are both strings.

- A string in C is an array of characters terminated by a null character "\0". Simply, a string contents are the array characters plus an extra character that is called null character. The null character has 0 for ASCII code and is written '\0'. The significant values of the character string are all those placed before the null character.
- We therefore notice that if the null character is in the first position, we have an empty character string .

3.1. Declaration of a string

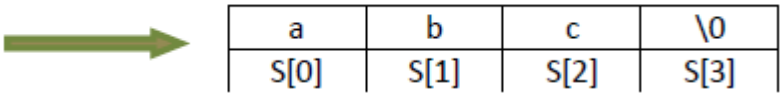
<NameStringe> [<sizeMaxString>] : string ;

In C :

char < NameStringe > [<sizeMaxString >] ;

Example

Char S[4]= "abc";



Note

We can only place a maximum of 199 letters in 'C', not 200, because we need to include a null character after the last letter of the string.

3.2. Initialization of a string

The initialization of a string can be achieved either through the conventional array syntax, where the last character indicates the end of the string, or by using quotation marks. There are two ways to initialize a string:

3.2.1. Direct initialization:

Program	Output

#include <stdio.h>
int main() {
char letter[]="Good to see you";
int i,
length=sizeof(letter)/sizeof(letter[0]);
printf("String length=%d\n",length);
printf("String=%s\n", letter);
printf("String elements:\n");
for (i = 0; i < length; ++i)
{
printf("%c\n", letter[i]);
}
return 0;
}
	String length=16
	String=Good to see you
	String elements:
	G
	o
	o
	d
	t
	o
	s
	e
	e
	y
	o
	u
	.

It is worth note that the “space” is counted and the null character is printed as a point (“.”).

3.2.2. Initiaialization during declaration as an aarray:

Example

Program	Output

#include <stdio.h>
int main() {
char letter[]={'G','o','o','d',' ','t','o',' ','s','e','e',' ','y','o','u','\0'};
printf("String=%s\n", letter);
return 0;
}
	String=Good to see you

Example

```
#include <stdio.h>
int main() {
   // Initializing a string using array syntax
   char string1[] = {'H', 'e', 'l', 'l', 'o', '\0'}; // '\0' indicates
the end of the string
   // Initializing a string using quotation marks
   char string2[] = "World";
   printf("String 1: %s\n", string1);
   printf("String 2: %s\n", string2);
   return 0;
}
```

In this example:

- We declare a character array **str** to store the string. It has a size of 100 characters, which means it can store strings up to 99 characters long (plus the null terminator **'\0'**).

- We use **scanf("%s", str)** to read a string from the user and store it in the **str** array. **%s** specifier reads a string until it encounters whitespace (space, tab, newline).

- We use **printf()** to write the string stored in **str** to the console.

3.3. Reading and writing a string

During the reading operation, reading of the string stops as soon as it encounters a separator character: newline, space, tabulation, etc. In C, writing a string is done as follows:

```
#include <stdio.h>

int main() {
   char str[100]; // Declare a character array to store the string

   // Reading a string from the user
   printf("Enter a string: ");
   scanf("%s", str); // Read a string from the user and store it in 'str'

   // Writing the string
   printf("You entered: %s\n", str); // Print the string

   return 0;
}
```

4. String Library Functions

#include <string.h>

Name	Description
strlen	return the length of string not counting \0
strcopy	copies string from source to dest
strncopy	copies n chars from source to dest
strcat	appends string from source to end of dest
strncat	appends n chars from source to end of dest
strcmp	compares two strings alphabetically
strncmp	compares the first n chars of two strings
strstr	finds a string inside another
strtok	breaks string into tokens using delimiters

4.1. Some String manipulation

Strings can be manipulated like arrays. In this case, we can apply our knowledge of manipulating simple arrays as discussed earlier. Another approach is to use the ***string.h*** library provided by the compiler, which offers a set of predefined functions to assist in string manipulation.

Operations on strings in C involve various manipulations such as concate

nation, comparison, copying, and more. Here's an overview of some common operations:

4.1.1. String Concatenation: The process of joining two or more strings together to form a single, continuous string.

```
#include <stdio.h>
#include <string.h>

int main() {
   char str1[50] = "Hello";
char str2[50] = " World";
   strcat(str1, str2); // Concatenate str2 to str1
   printf("Concatenated String: %s\n", str1);
   return 0;
}
```

4.1.2. String Comparison: Comparing two strings**.**

```
#include <stdio.h>
#include <string.h>

int main() {
   char str1[] = "apple";
   char str2[] = "banana";
   int result = strcmp(str1, str2); // Compare str1 and str2
   if (result < 0)
      printf("str1 is less than str2\n");
   else if (result > 0)
      printf("str1 is greater than str2\n");
   else
      printf("str1 is equal to str2\n");
   return 0;
}
```

4.1.3. String Copying: Copying the contents of one string to another.

```
#include <stdio.h>
#include <string.h>
int main() {
   char src[] = "Copy me!";
   char dest[50];
   strcpy(dest, src);    // Copy src to dest
   printf("Copied String: %s\n", dest);
   return 0;}
```

4.1.4. String Length: Finding the length of a string.

```
#include <stdio.h>
#include <string.h>

int main() {
   char str[] = "Hello, World!";
   int length = strlen(str); // Get the length of str
   printf("Length of String: %d\n", length);
   return 0;
}
```

5. Practice Exercises

1. Write an algorithm that receives the averages of N students, where N is determined by the user, then calculates the number of students who failed the subject (average less than 10).
2. Write an algorithm that finds the maximum and minimum element in an array of N elements and their indexes.
3. Write an algorithm that finds the sum and the product of all elements in an array
4. Write an algorithm that reads hourly temperatures for 30 days as a matrix (30 by 24), then displays them on the screen. After that, display the highest temperature and when it was recorded.
5. Write an algorithm to add and subtract two matrices. Make sure to handle cases where the matrices have different dimensions.
6. Write an algorithm to find the maximum and minimum values in each row of a given matrix.
7. *Write an algorithm or C programme to find the string with the maximum and minimum length from a given list of strings. Your program should prompt the user to enter the number of strings in the list and then the strings themselves. After input, the program should determine and display the longest and shortest strings from the list along with their lengths.
8. *Write an algorithm or C programme to check if a given string is a palindrome. A palindrome is a word, phrase, number, or other sequence of characters that reads the same forward and backward.

   radar is palindrome - hello isn't palaindrome

Solution 7*

```
#include <stdio.h>
#include <string.h>

int main() {
   int n, i;
   char strings[100][100]; // Limitation to 100 strings of a maximum of 99 characters
each
   char longest[100], shortest[100];
   int max_length = 0, min_length = 100;

   // Input the number of strings
   printf("Enter the number of strings: ");
   scanf("%d", &n);

   // Input the strings
   for (i = 0; i < n; i++) {
      printf("Enter string %d: ", i + 1);
      scanf("%s", strings[i]);
      int length = strlen(strings[i]);

      // Check for the longest string
      if (length > max_length) {
         max_length = length;
         strcpy(longest, strings[i]);
      }
      // Check for the shortest string
      if (length < min_length) {
         min_length = length;
         strcpy(shortest, strings[i]);
      }
   }
   // Display the results
   printf("\nLongest string: %s (Length: %d)\n", longest, max_length);
   printf("Shortest string: %s (Length: %d)\n", shortest, min_length);

   return 0;
}
```

Solution 8*

```
Algorithme Palindrome;
Var
ch : char[20];
i, L : Entier;
Pal : Boolean;
Begin
write(”Entrer une chaîne non vide : ”);
read(ch);
L←long(ch);
Pal ←Vrai;
i ←1:
While ((i <= L/2) ET (Pal)) do
   if (ch[i] = ch[L-i+1]) then
     i ←i + 1;
  else
Pal← Faux;
  endif
endwhile
if (Pal) then
write(ch, ” est un palindrome”);
else
write(ch, ” n’est pas un palindrome”);
endif
End
```

Application: Monthly Precipitation Analysis

Problem Statement

The meteorological station has tasked you with developing a C program that will allow the recording and analysis of rainfall amounts measured over 12 months. The program should be able to:

1. Allow the user to input the rainfall amounts for each of the 12 months of the year.
2. Calculate and display the average precipitation for the year.
3. Ask the user to provide a threshold value and identify which months had precipitation that exceeded this threshold.
4. Calculate and display the average difference in precipitation between each month and the previous month.
5. Determine the month with the highest precipitation and the month with the lowest precipitation.
6. Extend your program to handle precipitation data for multiple years. Store the data for three years in a 3x12 matrix.
7. For each year, calculate the difference in precipitation for each month compared to the previous year.
8. Compare the precipitation for January in Year 1 to January in Year 2. Which year had more precipitation in January?
9. Find the month with the greatest change in precipitation between Year 1 and Year 2.
10. Determine the month with the highest overall precipitation across all three years.

Bibliography

1. Stephen G. Kochan, Programming in C, 4th Edition, Sams Publishing, USA, 2014.
2. Thomas H. Cormen, Algorithms Unlocked, MIT Press, USA, 2013.
3. Robert Sedgewick, Kevin Wayne, Algorithms, 4th Edition, Addison-Wesley, USA, 2011.
4. Steven S. Skiena, The Algorithm Design Manual, 2nd Edition, Springer, USA, 2008.
5. K. N. King, C Programming: A Modern Approach, 2nd Edition, W. W. Norton & Company, USA, 2008.
6. Deepali Srivastava, C in Depth, 1st Edition, BPB Publications, India, 2008.
7. Mark A. Weiss, Data Structures and Algorithm Analysis in C, 2nd Edition, Pearson Education, USA, 2006.
8. Robert Sedgewick, Algorithms in C, 3rd Edition, Addison-Wesley, USA, 2002.
9. Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language, 2nd Edition, Prentice Hall, USA, 1988.
10. Dennis M. Ritchie, The C Programming Language, 2nd Edition, Prentice Hall, USA, 1988.
11. D. E. Knuth, The Art of Computer Programming. Volume 1: Fundamental Algorithms, Addison-Wesley, Reading, MA, 1968.

.