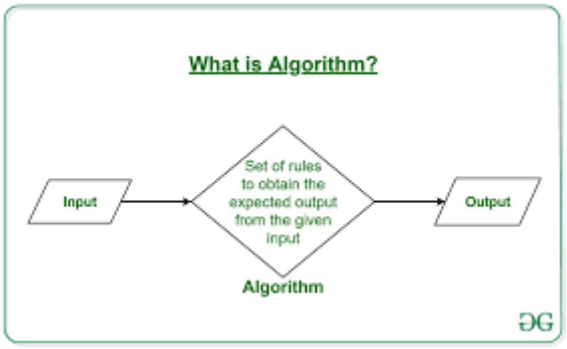
UNIVERSITE BADJI MOKHTAR – ANNABA
Faculty of Technology
Computer Science Department

Text Book

Algorithms and Data Structure 1



Pr. Halima BAHI

Bachelor 1st year

2024 / 2025

Table of Contents

Chapter 2 : Sequential Algorithms

1. Algorithm vs Program

An algorithm is the precise description of the method of solving a problem in the form of simple instructions. The algorithm is characterized by:

- Identification of information involved in a program
- Abstraction: modeling, formalization and representation of this information
- List of operations to be applied to this information.
- Definition of the order of operations to be respected.

Algorithmic Language: Any algorithm is expressed in a natural language called Language Algorithmic or pseudo-language, it describes objects in a structured, clear and complete manner manipulated by the algorithm as well as all the instructions to be executed on these objects to obtain results.

A program is the translation of the algorithm into a formal language that the machine can understand and execute. This language is called a programming language (python, C, Java, etc.) and it is governed by a syntax just as much as the natural language of man is.

2. Algorithm Components

An algorithm consists of three main parts:

- The header: this part is used to give a name to the algorithm. It is preceded by the word Algorithm;
- The declarative part: in this part, we declare the different elements that the algorithm uses (constants, variables, structures, procedures and functions.);
- The body of the algorithm: this part contains the instructions of the algorithm. It is delimited by the words “Begin” and “end”.

3. Data: Variables and constants

Constants represent numbers, characters, strings of characters whose value cannot be modified during the execution of the algorithm. Constants are introduced by the keyword “Const” as:

const name_identifier = value ;

Example:

const pi = 3.14 ;

Variables can store digits, numbers, characters, character strings, etc. whose value can be modified during the execution of the algorithm. Variables are introduced by the keyword “Var” as:

var name_identifier : type ;

Constants as well as variables are designed by identifiers. Herein, an identifier is a name that follows a particular syntax:

The rules for identifiers vary slightly depending on the specific programming language, but here are some general guidelines that apply to most languages:

- Identifiers must consist of letters (both uppercase and lowercase), digits (0-9), and underscores (_).
- The first character must be a letter or an underscore.
- Identifiers cannot contain spaces or other special characters.
- Identifiers can be of varying lengths, but there may be limitations imposed by the specific language or implementation.
- Some languages may have a maximum length for identifiers.
- Identifiers cannot be the same as reserved keywords in the language. Keywords have special meanings and cannot be used as variable or function names.
- Some languages are case-sensitive, meaning that “variable” and “Variable” are considered different identifiers.
- Other languages are case-insensitive, meaning that both variable and Variable would refer to the same identifier.
- It is generally recommended to choose identifiers that are meaningful and descriptive, making your code easier to understand and maintain.

Here are some examples of valid and invalid identifiers in most programming languages:

Valid identifiers:

- number
- my_number
- Number123
- _private_number

Invalid identifiers:

- 123variable (starts with a digit)
- variable with space (contains a space)
- $variable (contains a special character)
- if (a reserved keyword)

By following these rules and guidelines, you can ensure that your identifiers are valid and meaningful.

4. Standard primitive types

The type corresponds to the kind of information used. Standard primitive types are those types that are available on most computers as built-in features.

The standard types in algorithmic languages include the whole numbers, the logical truth values, and a set of printable characters. We denote these types by the identifiers INTEGER, REAL, BOOLEAN, CHAR, and STRING (of characters). Each type has a set of operations.

4.1.The type INTEGER

The type INTEGER comprises a subset of the whole numbers whose size may vary among individual computer systems. If a computer uses N bits to represent an integer in two's complement notation, then the admissible values of x must satisfy $-2N-1 \leq x < 2N-1$. It is assumed that all operations on data of this type are exact and correspond to the ordinary laws of arithmetic, otherwise, the computation will be interrupted. This event is called overflow.

Example :

```
const coefficient = 6 ;
      Temp = -273 ;
var mark : integer ;
```

The standard operators for integers are the four basic arithmetic operations of addition (+), subtraction (-), multiplication (*) and division (/, DIV).

4.2. The type REAL

The type REAL denotes a subset of the real numbers. Whereas arithmetic with operands of the types INTEGER is assumed to yield exact results, arithmetic on values of type REAL is permitted to be inaccurate within the limits of round-off errors caused by computation on a finite number of digits.

Example :

```
const pi = 3.14 ;
var mean : real ;
```

The standard operators are the four basic arithmetic operations of addition (+), subtraction (-), multiplication (*), and division (/).

4.3. The type BOOLEAN

The two values of the standard type BOOLEAN are denoted by the identifiers TRUE and FALSE.

For example, if x has for value 3, and y is 4, the expression “x=y” would be evaluated as FALSE.

4.4.The type CHAR

The character type is a finite and completely ordered set of characters (symbols). It includes:

- The letters of the Latin alphabet: a .. z, A .. Z.
- The numbers: 0 .. 9.
- Symbols used as operators: + - * / < = > …
- Punctuation characters: . , ; ! ? …
- Special characters: @ % & # …
- And others.

The character set defined by the International Standards Organization (ISO), and particularly its American version ASCII (American Standard Code for Information Interchange) is the most widely accepted set. The figure below shows the ASCII table with all 256 characters. They are numbered from 0 to 255. Each character is stored in computer memory on one byte.

Table 1. ASCII Table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
32	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
128	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	ø	£	Ø	×	ƒ
160	á	í	ó	ú	ñ	Ñ	ª	º	¿	®	¬	½	¼	¡	«	»	¦	¦	¦	¦	¦	Á	Â	À	©	¦	¦	+	+	¢	¥	+
192	+	-	-	+	-	+	ã	Ã	+	+	-	-	¦	-	+	¤	ð	Ð	Ê	Ë	È	i	Í	Î	Ï	+	+	¦	_	¦	Ì	-
224	Ó	ß	Ô	Ò	õ	Õ	µ	þ	Þ	Ú	Û	Ù	ý	Ý	¯	´	-	±	=	¾	¶	§	÷	¸	°	¨	·	¹	³	²	¦	

To determine the rank (the order) of a character ('A' for example) in the set of characters represented by this table, one must add the two numbers found on the same line (in the first column on the left) and the same column (in the first row at the top). Example: The rank of the character 'A' is equal to 64 + 1 = 65.

A character type constant is represented by one and only one character framed by two quotes. Examples: ',a', 'b', 'C', '!', '#', …

The operations defined by functions on the Character type are:

ORD(c): This function returns a positive integer corresponding to the rank of the character c, in all the characters.

Examples: Ord('!') = 33, Ord ('A') = 65, Ord ('a') =97.

CHR(i): is the inverse function of Ord. For a positive integer i, it returns the character of rank i.

Examples: Chr(33) = '!', Chr(65) = 'A', Chr(97) = 'a'.

SUCC(c): provides the character that immediately follows the character c in all the characters. Examples: Succ('a') = 'b', Succ('3') = '4', Succ('%') = '&'.

PRED(c): provides the character immediately preceding the character c in all characters. This is the inverse function of Succ.

Examples: Pred('b') = 'a', Pred ('4') = '3', Pred ('&') = '%'

4.5. The type STRING

A string is a sequence of characters enclosed by two double quotes.

Examples: “Algorithmics”, “Ahmed Ali”, …

The number of characters in a string is called length of the string. "Algorithmic" is a string of length 13.

"": represents the empty string of length 0. It does not contain any characters. We mention two predefined functions on the strings

• length(str): it provides the length of the string str.

• concat(str1, str2): it provides the string obtained by concatenation of the two strings str1 and str2.

Example: concat("Module", "Algorithmic") = "Algorithmic Module”

In summary, each type has a particular size and representation in computer memory. The different forms of constants should not be confused.

Example: 3 (integer type), 3.0 (real type), '3' (character type) and "3" (string type).

5. Basic operations

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

If an expression contains several operators with the same priority, these operators are left associative.

Example:

The expression: X + Y * Z , leads to the evaluation of X + (Y * Z).

In case, one wish to modify the semantic induced by this rule, parenthesis have to be used.

5.1. Arithmetic operations

The standard operators are the four basic arithmetic operations of addition (+), subtraction (-), multiplication (*) and division (/, DIV).

Whereas the slash denotes ordinary division resulting in a value of type REAL, the operator DIV denotes integer division resulting in a value of type INTEGER.

If we define the quotient q = m DIV n and the remainder r = m MOD n, the following relations hold, assuming n > 0: q*n + r = m and 0 ≤ r < n

Examples: 31 DIV 10 = 3 ; 31 MOD 10 = 1 ; -31 DIV 10 = -4 ; -31 MOD 10 = 9

5.2. Boolean operation and comparison

The Boolean operators are the logical conjunction, disjunction, and negation whose values are defined in Table 2.

Table 2. Logical operators

A	B	A and B	A or B	Not(A)
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Note that comparisons are operations yielding a result of type BOOLEAN. Thus, the result of a comparison may be assigned to a variable, or it may be used as an operand of a logical operator in a Boolean expression.

For instance, given Boolean variables p and q and integer variables x = 5, y = 8, z = 10, the two assignments:

p is x = y, and q is (x ≤ y) AND (y < z)

yield p = FALSE and q = TRUE.

Comparison operators are: <, >, =, ≠, ≤, ≥

6. Basic instructions

In algorithmics, there are three elementary instructions: ***read***, ***write*** and the assignment (←). Each instruction is followed by a semicolon (;)to specify its ending.

6.1. The reading instruction (READ)

The READ instruction allows us to give a value to a variable from the keyboard.

Syntax: **read**(variableName);

Example: read(A) ; read(B) ; read(C) ;

Several successive reading instructions can be grouped into a single instruction. For the previous example, we can replace the three read instructions with: read(A, B, C);

On a computer, when the processor receives the ***read(variableName)*** order, it stops execution of the program and waits for a value. The user must then enter a value from the keyboard. As soon as the entry is validated (by pressing the *Enter* key ↵), execution of the program continues. The value passed by the user is assigned to the variable and overwrites its previous value.

The ***read(variableName)*** instruction leads to an error if the value entered does not match the type of the variable, unless it is an integer value and the variable is of real type. In which case the integer value is converted to a real value.

6.2. The writing instruction (WRITE)

The ***write*** instruction allows display on screen. There are two types of display:

- Either we display the value of a variable or an expression:

  Syntax: **write**(variableName);

  Example:

  write(X);

  write(2*X+Y);

  If X = 10 and Y= 5 then, the first instruction displays 10 and the second displays 25

- Either we display a text (a message):

  Syntax: **write** (“a message”);

  Example: write("The result is = ");

Several successive writing instructions can be grouped into a single instruction. For example, the sequence of instructions:

Example :

write (“The result is ="); write (X);

Can be replaced by: write (“The result is =”, X);

This instruction displays the string "Result is =10.”

Note: Message communication is very useful in programming. It offers the possibility:

- To guide the user by telling him what the machine expects of him following a reading order.
- To explain the results of a treatment

6.3. The assignment (←)

Assignment is a statement that stores the value of an expression in a variable.

Syntax: variableName ← expression;

This statement specifies that the *variableName* receives the value of *expression*.

Examples: X ← 7 ; // Assign a constant value to the variable X (X receives 7).

Y ← X ; // Copy the value of variable X into variable Y (Y receives X).

Z ← 2 * X + T / Y // Z receives 2 * X + T / Y.

Remarks:

- If the variable already contained a value, it would be replaced by the value of the expression. The old value of the variable is lost and there is no way to recover it.
- A type check operation is performed before assignment. It is an error if the value of the expression does not belong to the type of the variable, unless it is a value integer and that the variable is of real type. In this case the integer value is converted to real.

Example :

```
Algorithm Permutation ;
var X, Y, Z : integer ;
begin
        read( X, Y ) ;
        Z ← X ;
        X ← Y ;
        Y ← Z ;
        write (X, Y) ;
end.
```

If the read values of X and Y are 17 and 10 respectively, what would be the outputted values.

7. Examples of algorithms

Example 1: The following algorithm displays a hello message.

```
algorithm hello_v1;
begin
        write ("Hello world");
end.
```

Example 2 :

```
Algorithm hello_v2;
var name : string ;
begin
        write ("Please, enter your name ");
        read(name);
        write("hello ", name);
end.
```

Example 3: The following algorithm computes and displays the area of a rectangle

```
Algorithm area_r;
var length, width, area : integer ;
begin
        write ("Enter the rectangle length: ");
        read(length);
        write ("Enter the rectangle width: ");
        read(width);
        area ← length * width ;
        write("The area is : ", surface);
end.
```

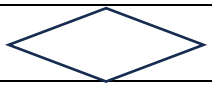
Example 4: The following algorithm computes and displays the perimeter of a circle

```
Algorithm perimeter_c;
const pi = 3.14;
var radius : integer ;
    perimeter : real;
begin
        write ("Enter the circle radius: ");
        read(radius);
        perimeter ← 2 * pi * radius ;
        write("The perimeter is : ", perimeter);
end.
```

8. Flow chart

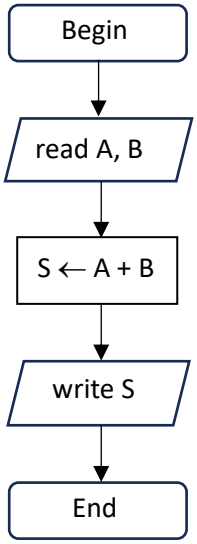
A flowchart is a diagrammatic representation of algorithm to plan the solution to the problem. Constructed by using special geometrical symbols where each symbol represents an activity. The activity would be input/output of data, computation/processing of data etc.

8.1. Flowchart symbols

Symbol	Meaning
	Begin / End
	Operations, instructions, …
	Input / Output
	Relationship
	Decision

8.2. Example

Example of a flowchart that calculate the sum of two numbers.



9. The C language

C is a general-purpose, that has been around since the early 1970s. It's renowned for its efficiency, portability, and flexibility, making it a cornerstone of software development.

The C programming language is considered a **mid-level** language, though it's often described as leaning more towards a **low-level** language due to its close interaction with hardware. Indeed, C provides direct manipulation of memory through pointers, bitwise operations, and access to system-level functions. This allows for writing programs that operate

very close to the machine's hardware, similar to assembly language. C also has features of high-level languages, such as structured programming, functions, and abstractions like data types (e.g., arrays, structs), making it more accessible than true low-level languages like assembly.

9.1. Structure of a C program

The basic structure of a C program is divided into 6 parts which makes it easy to read, modify, document, and understand. The six sections are: Documentation, Preprocessor section, Definition, Global declaration, Main function, and Sub programs. While the main section is compulsory, the rest are optional in the structure of the C program.

9.1.1. Documentation

Documentation consists of the description of the program, programmer's name, and creation date. These are generally written in the form of comments.

In a C program, single-line comments can be written using two forward slashes i.e., //, and we can create multi-line comments using /* */.

Example:

// This is my first program /*

I am 1st year Bachelor student */

Both methods work as the document section in a program. It provides an overview of the program. Anything written inside will be considered a part of the documentation section and will not interfere with the specified code.

9.1.2. Preprocessor Section

All header files are included in this section which contains different functions from the libraries. A copy of these header files is inserted into the code before compilation. The header files are introduced by the C preprocessing directive “#include”.

Example :

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

There are various header files available for different purposes. Here are some examples of header files in C:

- stdio.h: This header file contains declarations for standard input and output functions, such as *printf(), scanf()*, and *fopen()*.
- stdlib.h: This header file contains declarations for general utility functions, such as *malloc(), free()*, and *rand()*.
- math.h: This header file contains declarations for mathematical functions, such as *sin(), cos()*, and *log()*.
- string.h: This header file contains declarations for string manipulation functions, such as *strcpy()*, *strcat()*, and *strlen()*.
- time.h: This header file contains declarations for time and date functions, such as *time()* and *strftime()*.

In addition to these standard header files, there are also many user-defined header files that are available. For example, a header file for a linked list data structure might contain declarations for the struct node type and functions for creating, inserting, and deleting nodes in the linked list.

9.1.3. Definition

A preprocessor directive in C is any statement that begins with the "#" symbol. In particular, the "*#define*" is a preprocessor compiler directive used to create constants, ie, "#define" basically allows the macro definition, which allows the use of constants in our code.

Example :

#define PI 3.14159265358979323846

"*#define*" allows us to use constants in our code. It replaces all the constants with its value in the code.

9.1.4. Global Declaration

This section includes declaration of global variables, function declarations, static global variables, and functions.

Example:

```
1  const double PI = 3.14159265358979323846;
2  int my_global_variable = 10;
3  void my_global_function() {
4  // This function can be called from anywhere in the program. }
5
```

Global variables and functions can be very useful, but they should be used with caution. If you use too many global variables and functions, your program can become difficult to maintain and debug. It is generally better to use local variables and functions whenever possible.

9.1.5. Main() Function

For every C program, the execution starts from the *main()* function. Thus, it is mandatory to include a *main()* function in every C program.

The return type of the *main()* function can be either int or void. *void main()* informs the compiler that the program will not yield any value, while *int main()* indicates that the program will return an integer value.

Example:

```
int main() { printf("Hello, world!\n"); return 0;
}
```

9.1.6. Sub Programs

Includes all user-defined functions (functions the user provides). They can contain the inbuilt functions and the function definitions declared in the Global Declaration section. These are called in the *main()* function. User-defined functions are generally written after the *main()* function irrespective of their order.

Exemple:

```
void my_global_function() {
/* This function can be called from anywhere in the program.
printf("This is a global function.\n"); */
}
```

A little more sophisticated function would be:

```
int add_two_numbers(int a, int b) {
return a + b; }
```

9.1.7. Example of C program

Here is the example of the simplest program, one can write:

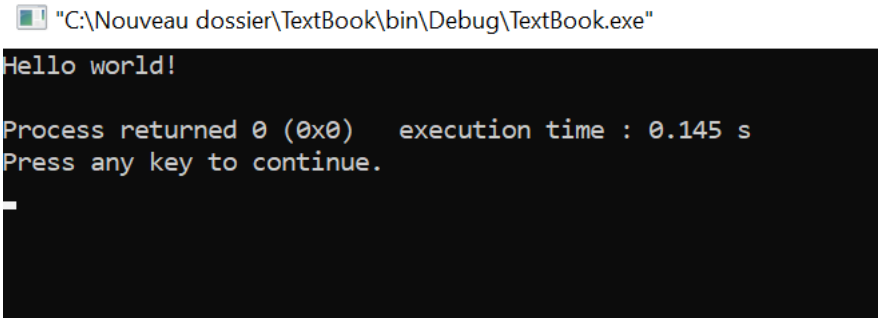
```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      printf("Hello world!\n");
7      return 0;
8  }
9
```

To execute a C program, one should follow these steps:

1. Compile the C program using a C compiler. The C compiler will convert the C source code into machine code that the computer can understand.

2. Link the compiled C program with any necessary libraries. Libraries are collections of pre-compiled code that can be used by C programs.
3. Execute the linked C program.

The execution of this program will produce the following display on the screen:



9.2. Data type in C

C includes the following basic data types:

- int - stores whole numbers
- float - stores real numbers
- double - stores real numbers with higher precision than float
- char - stores single characters
- void - specifies that a function does not return any value

In addition to these basic data types, C also has a number of compiler-specific data types, such as long and short.

Example : // Basic data types

int my_integer_variable = 10;

float my_floating_point_variable = 3.141592653589793;

char my_character_variable = 'a';

void my_void_function() {

// This function does not return any value. }

// Compiler-specific data types

long my_long_integer_variable = 1234567890123456789; // A long integer variable

short my_short_integer_variable = 12345; // A short integer variable

When choosing a data type for a variable, it is important to consider the type of data that the variable will store and the operations performed on it. For example, if you need to store a integer number, you would use an integer data type. If you need to store a real number, you would use a floating-point data type.

Using the correct data type for your variables can help to improve the performance and reliability of your C programs.

Table 3. Integer types in C

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Table 4. Floating-point types in C

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Besides these types, in C, Boolean is a data type that contains two types of values, i.e., 0 and 1. Basically, the bool type value represents two types of behavior, either true or false.

9.3. Assignment and expressions

Assignment is a fundamental operation in C programming. Assignment in C is the process of storing a value in a variable. To assign a value to a variable, you use the assignment operator (=).

Example:

```
int a ;

a = 1 ;
```

We can also assign the value of one variable to another variable. Generally, the value is issued from an expression.

An expression in C is a combination of operands and operators. Operands are the values that are operated on, and operators are the symbols that perform the operations. Expressions can be used to evaluate mathematical expressions, compare values, and perform other logical operations. Here are some additional tips for using assignment in C:

- Only assign values to variables of the correct data type.
- Be careful not to overwrite the value of an important variable.
- Use parentheses to make your code more readable and to avoid errors. There are many operators in C, in particular, we mention:
    - Arithmetic Operators
    - Relational Operators
    - Logical Operators

9.4. Input and output instructions

To perform the aforementioned ***read*** and ***write*** instructions, C offers the instructions "***scanf***" and "***printf***" respectively. these functions are inbuilt library functions, defined in *stdio.h* (header file).

9.4.1. printf() function

The *printf()* function is used for output. It prints the given statement to the console.

Syntax: **printf** ("format string", argument_list);

The **format string** can be %d (integer), %c (character), %s (string), %f (float) etc.

9.4.2. scanf() function

The *scanf()* function reads formatted input from the standard input (usually the keyboard). It takes a format string and a list of variables, similar to *printf()*.

9.4.3. Example of C program

```c
#include <stdio.h>
#define PI 3.141592653589793
int main(){
   float radius, area;
   printf("Enter the radius of the circle: ");
   scanf("%f", &radius);
   area=  PI * radius * radius;
   printf("The area of the circke is : %.2f", area);
  return 0;

}
```

Chapter 3: Control Statements

Control statement structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Conditional statements in algorithms allow the algorithm to make decisions based on the value of a Boolean expression. This means that the algorithm can choose different paths of execution depending on whether the condition is true or false.

Conditional statements are one of the most important building blocks of algorithms, and they are used in a wide range of applications, including sorting, searching, and graph algorithms.

1. The if / else statement

1.1. The if statement

Sometimes, the execution of an action requires the fulfillment of a condition. In this structure, a set of instructions is executed if a condition is verified. Its syntax is:

```
if (condition) then
        begin
        //instructions to execute
        end
endIf
```

The condition is a Boolean expression, which the evaluation leads to either True or False. The condition is mainly defined using relational operators: =, ≠, >, <, ≥, and ≤ . The simple expression resulting on the use of these operators may be combined using logic operators: AND, OR, and NOT.

Example:

```
if (A <> 0) then
        D = X / A ;
endIf ;
```

1.2. if / else statement

The most common conditional statement is the if/then/else statement. This statement has the following syntax:

```
if (condition)  then
        begin
        // instructions to execute if the condition is true
        end
```

```
else
        begin
         // instructions to execute if the condition is false
        end
endIf ;
```

Example 1:

```
if (number > = 0) then
   write(“The number is positive”)
else
   write(“The number is negative”)
endIf;
```

Example 2:

```
if (age >= 18 AND hasDriversLicense = TRUE) then
  write("You are eligible to rent a car.")
else
  write("You are not eligible to rent a car.")
endIf ;
```

With age is an *integer* variable, and hasDriversLicense is a *boolean* variable.

1.3. The nested if / else statement

In the first example, if the number is null, it is considered as positive, to avoid this ambiguity, one may use nested if statements. Indeed, nested if statements can be used to create complex decision-making logic, but it is important to use them carefully. If you are not careful, it can be easy to create nested if statements that are difficult to read and understand.

Example:

```
if (number > 0) then
  write(“The number is positive”)
else
   if (number < 0) then
        write(“The number is negative”)
  else
        write(“The number is null”)
  endIf ;
endIf ;
```

2. Multiple choice selection

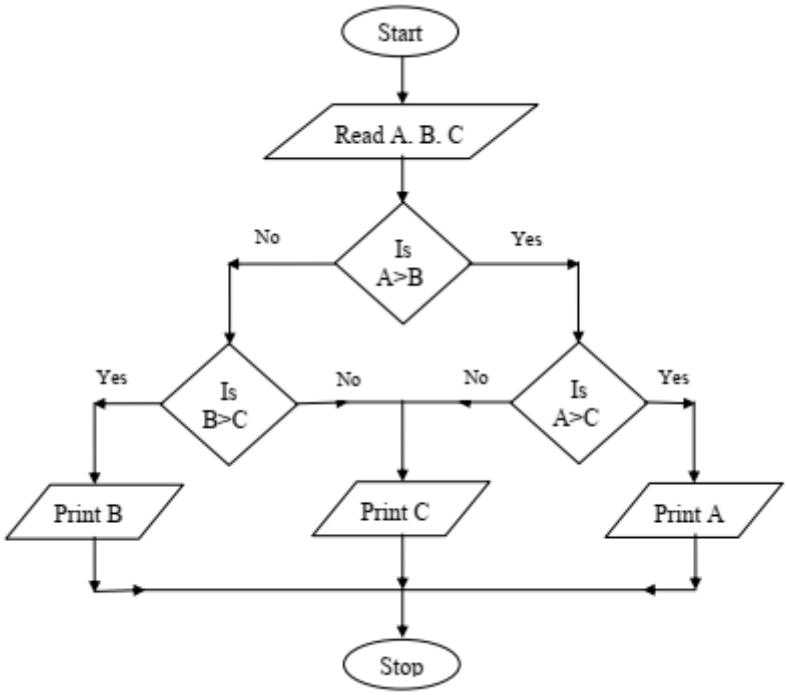
When multiple choices are available, the use of an appropriate structure is suitable. Herein, a conditional control structure is used when the processing depends on the value that a selector will take. This selector is of integer, character or boolean type.

Syntax :

```
case (selector) of
  choice1: instructions1
  choice2: instruction2
…
  Default : instructionsN
endCaseOf ;
```

3. Flowchart with conditional statement

The diamond is used to represent the true/false statement being tested in a decision symbol. In the following is a flowchart that finds the biggest number among the three given ones.



4. Conditional statements in C programs

In C programs the conditional statements are represented by the following structures:

4.1. If expression

```
if (expression){
//code to be executed
}
```

The relational operators in C language are presented in the following table.

Meaning	Operator
Equal to	==
Not equal to	!=
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=

The logical operators in C language are logical AND (**&&**), logical OR (||), and logical NOT (**!**).

4.2. If / else statement

```
if (expression){
    //code to be executed if condition is true
} else {
    //code to be executed if condition is false
}
```

The if-else-if ladder statement is an extension to the if-else statement. It is used in the scenario where there are multiple cases to be performed for different conditions.

```
if(condition1){
//code to be executed if condition1 is true
}else if (condition2){
//code to be executed if condition2 is true
}
else if (condition3){
//code to be executed if condition3 is true
```

```
}
...
else {
//code to be executed if all the conditions are false
}
```

Example 1:

```c
#include <stdio.h>

int main(){
    int number;
    printf("Enter the number: ");
    scanf("%d", &number);

    if (number > 0){
        printf("The number is positive \n");
    } else if (number < 0){
        printf("The number is negative \n");
    } else {
        printf("The number is null \n");
    }
    return 0;
}
```

Example 2: Program that computes the solutions of the equation $ax^2 + bx + c = 0$

4.3. Switch

The switch statement allows to select one of many code blocks to be executed. It is useful when there are multiple cases to consider. The syntax in C is as follows:

```
switch (expression){
  case x:
    //code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

- The switch expression is evaluated once
- The value of the expression is compared with the values of each case
- If there is a match, the associated block of code is executed
- The break statement breaks out of the switch block and stops the execution

- The default statement is optional, and specifies some code to run if there is no case match

Example: The following program translates a grade into an assessment.

```c
#include <stdio.h>

void main(){
char grade = 'B';

switch (grade) {
  case 'A':
    printf("Excellent");
    break;
  case 'B':
    printf("Good");
    break;
  case 'C':
    printf("Average");
    break;
  default :
    printf("invalid grade");
}
}
```

Chapter 4 : Loops

1. Introduction

Sometimes, we repeat a specific code instruction multiple times to solve a problem until a specific condition is met. This is known as iterations, which allows us to write code once and execute it multiple times.

There are mainly three types of loops: For, While, and Repeat. Usually, loops have three main elements which are:

Initialization: we assign an initial value to a variable used when evaluating the loop condition.

The loop condition: is a Boolean expression often evaluated before the execution of the loop body as in the case of *while* and *for* loops, and after the execution of the loop body as in the case of *repeat*. In any case, the body of the loop is running until the condition will be evaluated as false.

Variation of the loop condition: this is usually done at the end of the loop body as an increment or decrement of a counter used to evaluate the loop condition as in the case of *while*, *repeat*. Forgetting or incorrectly performing this step causes the loop body to execute infinitely, or give unexpected results.

2. For loop

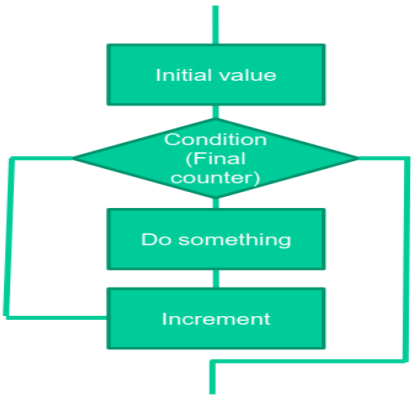
For loops are used when you know how many times you want to repeat a certain block of code. This is known as **definite iteration**. A *for* loop uses a counter to tell it how many times to run the same sequence of activities.

The counter has the following three numeric values:

- Initial counter value
- Increment (the amount to add to the counter each time the loop runs)
- Final counter value

The loop ends when the counter reaches the final counter value.

Syntax	Flowchart

For ctr = initial_value To final_value do
Begin
statements
End
endFor
	

Example: Display the five first number

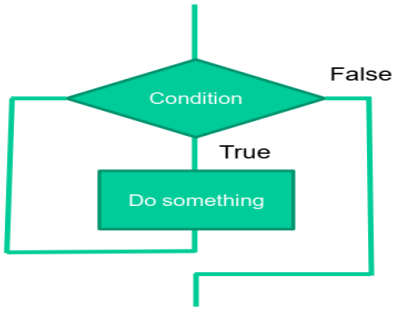
```
Algorithm countV1;
var i:integer;
begin
   for i ← 10 to 20 do
        write(i);
   endFor;
end.
```

3. While loop

While loop is used to execute the body of the loop until a specific condition is false. We apply this loop when we don't know how many times it will run.

The *while* loop consists of a loop condition, a code block as loop body and a loop update expression. First, the loop condition is evaluated, and if true, the code in the body of the loop will be executed. This process repeats until the loop condition becomes false.

Syntax	Flowchart

Initialization of the loop
While (condition of the loop) do
Begin
Loop's body
Condition update
End;
	

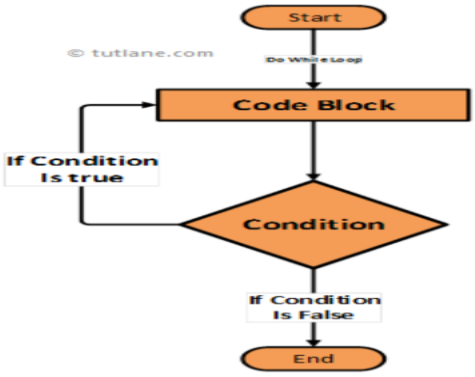
Example:

```
Algorithm countV2;
var i: integer;
Begin
    i ←10; //initialization
    While(i<=20) DO  //the number 20 is included
        begin
          Write(i);
          i ← i+1; //we increment the counter i
        end;
end.
```

4. Repeat loop

Repeat loops are used when it is once again **indefinite iteration**. This means that while we do not know how many times to loop something, we can just use a Repeat loop.

The loop **repeat-until** executes a block of instructions repeatedly, until a given condition becomes true. The condition will be re-evaluated at the end of each iteration of the loop. Because the condition is evaluated at the end of each iteration, a repeat/until loop will always execute at least once.

Syntax	Flowchart
Initialization of the loop repeat Loop's body Condition update Until(condition)	

Example:

```
Algorithm countV3;
var I : integer;
Begin
   i ← 20;//initialization
  repeat
        write(i);
        i ← i-1;     // variation of the condition
   until (i<10)    //the number 10 is included
end.
```

5. Trace of loops execution

To visualize the evolution of the variables manipulated inside the loops as well as to check the condition and the initialization of the value of the loop counter, the loop execution flow table is necessary. This table is composed of n lines for n iterations executed by the loop, plus the initial state of the variables before launching the loop. In columns, we find the iteration number, the variables manipulated inside the loop, and the output if applicable.

Example :

```
Algorithm addition;
var i, sum: integer;
Begin
   sum ← 0;
   i ← 1;
   while(i<=5)do
        Begin
            sum ← sum+i;
            write(sum+ " ");
            i ← i+1;
        End;
End.
```

The following table illustrates the evolution of the involved variables

Iteration	i	sum	Output
initialization	0	0	-
1	1	1	1
2	2	3	1 3
3	3	6	1 3 6
4	4	10	1 3 6 10
5	5	15	1 3 6 10 15

6. Infinite loops

The problem of infinite loops occurs when there is an error in the condition of the loop, or in the variation of the condition, below are the possible causes of an infinite loop:

- The counter is not varied (incremented or decremented), this is not the case of the loop for because the variation of the counter is done automatically.
- Vary the loop counter so as to never check the stopping condition, in this case, it is necessary to clearly specify the range of the counter, its limits, and the movement of the counter (increment or decrement).
- Vary the loop counter inside a conditional statement.
- Wrong condition.

To avoid the infinite loop problem, you must take the following precautions:

- Carefully study the range of variation of the counter.
- Specify the loop condition carefully by taking into account the extreme values of the counter.
- Involve the counter in the loop condition and don't forget to vary it.

7. Nested loops

A nested loop means a loop inside another loop. We can have any number of loops inside another loop.

Example:

```
For i ← 0 to 10 do
        for j ← 0  to 5 do
                statements;
```

8. Loops in C programs

8.1. For loop

Syntax

```
for (i=initial value; condition; variation of i)  {
    Loop's body
 }
```

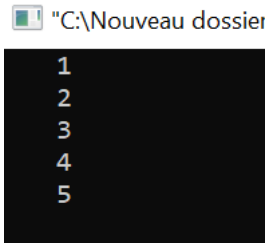
Example :

```c
#include <stdio.h>

void main(){
    int i;
    for (i=1;i<=5;i++)
        printf("%5d\n",i);

}
```

Result



8.2.While loop

Syntax:

```
Initialization of the loop
while(condition of the loop) {
        Loop's body
        Condition update
}
```

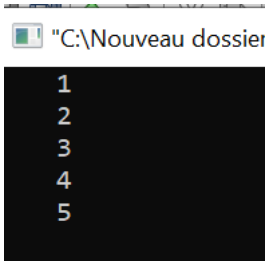
Example:

```c
#include <stdio.h>

void main(){
    int i;
    i = 1;
    while (i<=5){
        printf("%5d\n",i);
        i++;

    }
}
```

Result :

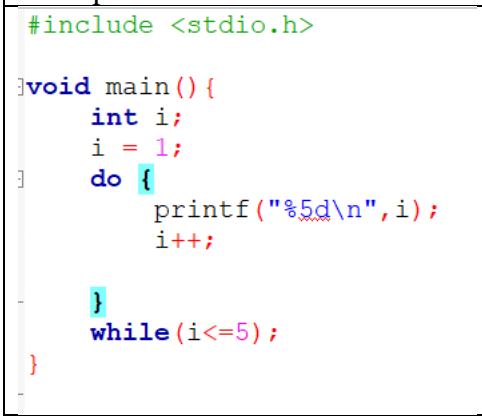
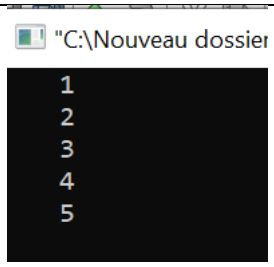


8.3.Do .. while loop

In C language, we use the 'do-while' loop as an implementation of the *repeat* loop.

Syntax

```
Initialization of the loop
Do  {
        Loop's body
        Condition update
}
while(condition loop)
```

Example:	Result:
	

```c
#include <stdio.h>

void main(){
    int i;
    i = 1;
    do {
        printf("%5d\n",i);
        i++;

    }
    while(i<=5);
}
```

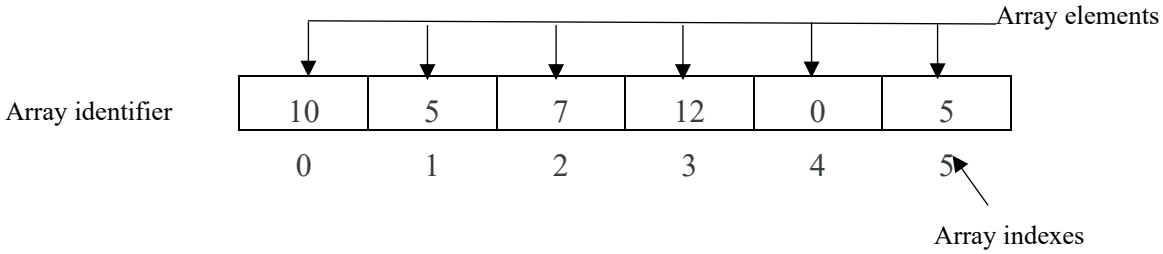
Chapter 5 : Array Structure

1. Introduction

An array is a data structure that consists of a set of values from the same data type with a single identifier name. Elements *are stored at contiguous memory locations.*

Basic terminologies of array

- **Array Index:** In an array, elements are identified by their indexes. Array index starts from 0.
- **Array element:** Elements are items stored in an array and can be accessed by their indexes.
- **Array Length:** The length of an array is determined by the number of elements it can contain.



2. The Arrays

2.1. Creating arrays

Syntax for declaring an array:

Many options are possible to declare an array:

1st option

```
var   array_name : array[0.. number_of_values_to_hold-1] of data_type;
```

2nd option

```
var  array_name : array[number_of_values_to_hold] of data_type;
```

3nd option

```
type name_of_type  : array[0..numberOfValuesToHold-1] of data_type;
var   array_name : name_of_type ;
```

Example :

```
const numDays = 7 ; N = 5 ;
type numArr : array[0..4] of int ;

var daysOfWeek : array[0..numDays-1] of string ;
    vowels : array[0..N-1] of char;
   scores : numArr ;
```

Once declared, arrays have to be initilazed / filled

Example :

```
daysOfWeek = [“sunday”, “monday”,”tuesday”,”wednesday”,”thuesday”,”friday,”saturday”]
vowels = [‘a’, ‘e’, ‘i’, ‘o’, ‘u’] ;
scores = [12, 8, 14, 9, 10] ;
```

2.2. 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. The number inside the square brackets [] is called a **subscript**. It’s used to reference a certain element of the array.

2.2.1. Access to an element

Syntax To refer to a specific element in the array:

```
array_name[subscript number] ;
```

Example:

```
myDay = daysOfWeek[1] ;
```

This instruction assigns the value “monday” to the variable myDay.

2.2.2. Browsing arrays

Navigating through an array can be done in several ways, the most used are: The browsing from the first to the last element, and the browsing from the last to the first element.

2.2.3. Displaying elements of an array T with N elements

In order to go through all the elements of the array, we use a loop with an increment/decrement of the index from the first to the last element of the array or vice versa.

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

```
or   (the reverse order)
        for i ← (N-1) to 0
          write(T[i])
        endFor
```

2.2.4. Reading elements of an array T with N elements

Example1 : Initializing the elements to 0

```
        i ← 0;
        while i < N-1
          Begin
            T[i] ← 0;
            i←i+1;
          End
```

Example2 : Reading the elements

```
        i = 0
        while i < N-1
          Begin
            write('Enter the ", i, " th element of the array : " );
            readln(T[i]);
            i← i+1;
          End
```

2.3. Array algorithms

2.3.1. Find the maximum and minimum element in an array (and their indexes)

```
algorithm max_min_tab;
const N=10;
var tab : array[0..N-1] of integer ;
    i, max, min, indexMin, indexMax : integer;

Begin
   min ← tab[0] ;
   max ← tab[0] ;
  indiceMin ← 0 ;
   indiceMax ← 0 ;
   for  i ←  0 to  (N-1) do
       Begin
          if (min > tab[i])
          then begin
                 min ← tab[i] ;
                 indiceMin ← i;
               end;
          endIf
          if (max < tab[i])
          then begin
```

```
                max← tab[i] ;
                indiceMax ← i;
              end;
          endIf
        end;
      endFor
      writeln("le min =", min,  " at the index : ",  indexMin) ;
      writeln("le max =", max, " at the index : ", indexMax) ;
End.
```

2.3.2. Find the sum and the product of all elements in an array

```
algorithm sum_product_tab;
const N=10 ;
var tab : array [0..N-1] of integer;
    i, sum, product : integer;
begin
  sum ← 0 ;
  product ← 1 ;
  for i ← 0 to (N-1) do
      begin
         sum ← sum + tab[i] ;
         product ← product * tab[i];
     end ;
  endFor
  writeln("The sum is" : , sum) ;
  writeln("The product is" : , product) ;
end.
```

2.3.3. Searching for elements in an array

There are two main algorithms for searching for elements in an array: linear search and binary search.

- Linear search is a simple algorithm that iterates over the array, comparing each element to the target element. If the target element is found, a Boolean variable is set to true, otherwise it is false. Given the array tab having as length N, and the element to search x:

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

- Binary search is a more efficient algorithm that can be used on sorted arrays. It works by repeatedly dividing the array in half and comparing the middle element to the target element. If the target element is equal to the middle element, the algorithm returns its index. Otherwise, the algorithm recursively searches the half of the array that contains the target element.

2.4. Shift array

2.4.1. Left shift with insertion of 0

```
Algorithm left_shift_zeroInsert_tab;
const N=10;
var tab : array[0.. N-1]  of integer ;
     i : integer;
begin
   // read the array
    ....
    for i ← 0 to (N-1-1) do
            tab[i]  ← tab[i+1];
     endFor;
     tab[N-1] ← 0;
end.
```

2.4.2. Right Circle shift without insertion

```
Algorithm right_shift_zeroInsert_tab;
const N=10;
var    tab : array[0.. N-1] of integer ;
       i , tmp : integer;
begin
   // read the array
   ...
    tmp ← tab[N-1]
    for i ← (N-1) to 1 do
        tab[i]  ← tab[i-1];
    endFor;
    tab[0] ← tmp;
end.
```

3. Sorting Arrays

Sorting is one of the basic operations on the arrays. Indeed; it's usually helpful when we have an 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 fast while others are slow.

3.1. 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 is repeated until the entire array is sorted.

Illustration : Imagine you have a list of number: 8 4 2 5 3 7

Look through the list and find the smallest. Exchange the smallest with the first item in the list.

2 | 4 8 5 3 7

Now look through everything to the right of the | for the smallest. Exchange the smallest with the first item in the list to the right of the |

2 3 | 8 5 4 7

Repeating the steps results in:

2 3 4 | 5 8 7

2 3 4 5 | 8 7

2 3 4 5 7 | 8

2 3 4 5 7 8 |

The associated algorithm would be :

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

3.2. 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-1-1 do
   begin
     for j ← 0 to n-i-1 do
         if  (tab[j]>tab[j+1])
         then  begin
```

```
                tmp ←tab[j];
                tab[j] ← tab[j+1];
                tab[j+1] ←tmp;
            end;
        endIf;
    endFor ;
end ;
endFor.
```

Illustration

Array to sort : 8 4 2 5 3 7

Pass 1

Compare 8 and 4: Swap

Array: 4 8 2 5 3 7

Compare 8 and 2: Swap

Array: 4 2 8 5 3 7

Compare 8 and 5: Swap

Array: 4 2 5 8 3 7

Compare 8 and 3: Swap

Array: 4 2 5 3 8 7

Compare 8 and 7: swap

Array: 4 2 5 3 7 8

Pass 2

Compare 4 and 2: Swap

Array: 2 4 5 3 7 8

Compare 4 and 5: No swap required

Array: 2 4 5 3 7 8

Compare 4 and 3: Swap

Array: 2 3 4 5 7 8

Compare 4 and 7: No swap required

Array: 2 3 4 5 7 8

Compare 4 and 8: No swap required

Array: 2 3 4 5 7 8

Pass 3

Compare 2 and 3: No swap required

Array: 2 3 4 5 7 8

Compare 2 and 4: No swap required

Array: 2 3 4 5 7 8

Compare 2 and 5: No swap required

Array: 2 3 4 5 7 8

Compare 2 and 7: No swap required

Array: 2 3 4 5 7 8

Compare 2 and 8: No swap required

Array: 2 3 4 5 7 8

3.3. Insertion Sort

Insertion sort is a simple sorting algorithm that works by inserting each element in an unsorted array into its correct position in a sorted array. In other words, as an element is added to the list, it is **inserted** into the correct position.

```
for i ← 1 to N-1 do
   temp ← array[i] ;
   j ← i – 1 ;
   while j >= 0 and array[j] > temp do
    begin
       array[j + 1] ← array[j] ;
       j ← j – 1 ;
    end ;
   array[j + 1] ← temp ;
```

4. 2-Dimensional arrays (matrices)

The elements of an array may be of any type. They could even be an array. This type of array is known as a **multi-dimensional array**.

One-dimension arrays are called vectors. When an array has 2 or more dimensions, it is called a matrix.

	1st column	2nd column	…	..	Mth column
1st row					
2nd row					
…					
Nth row					

Syntax of declaration:

array_name : array[0 ..numberOfRows-1, 0 ..numberOfColumns-1] of data_type;

4.1. Accessing elements of a matrix

To refer to a specific element in the array, must specify two subscripts.

Syntax:

```
array_name [i, j]
```

Example : Initilization of elements to 0

```
algorithm init_mat ;
const  n=4 , m=5;
var      mat : array [0 ..n-1, 0 .. m-1] of integer ;
          i, j : integer;

Begin
  for  i ← 0 to (n -1) do
     for j ← 0 to (m-1) do mat[i, j] ← 0;
     endFor;
endFor;
End.
```

4.2. Operations on matrixes

4.2.1. Sum of elements of a matrix

```
Algorithm sum_mat ;
const n=4, m=5;
var     mat : array[0 ..n-1, 0 ..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;
          writeln(sum);
        end;
    endFor;
End.
```

4.2.2. Sum of two matrices

To sum two matrices, matrices should have the same dimensions.

```
Algorithm sum_2_mat ;
const n=4, m=5 ;
```

```
type  matrice : array [0 ..n-1][0 ..m-1] of integer ;
var    mat1, mat2, mat3 : matrice ;
       i,  j : integer;

Begin
    for i ← 0 to (n-1) do
        for j ← 0 to (m-1) do
            mat3[i, j] ← mat1[i, j] + mat2[i, j] ;
        endFor;
    endFor;
End.
```

4.2.3. Product of two matrices

To perform the product of two matrices, if the first matrix is (N,M) dimension, the second should be (M, P).

```
algorithm product_mat;
const n=4,m=5,p=6;
var mat1 : array[0.. n-1, 0 ..m-1] of integer ;
    mat2 : array[0 ..m-1, 0 ..p-1] of integer ;
    mat3 : array[0.. n-1, 0 ..p-1] of integer ;
    i, j, k, sum : integer ;  // the variable sum is used to compute the product between vectors

Begin
   for k ← 0 to (n-1) do
       for i ← 0 to (p-1) do
           begin
              sum ← 0;
              for j ← 0 to (m-1) do sum ← sum+(mat1[k, j]*mat2[j, i]);
              endFor ;
             mat3[k, i] ← sum;
           end ;
       endFor;
   endFor ;
End.
```

4.3. Arrays in C programs

In C programs, subscripts start from 0 and run to N-1 (where N is the value within the square brackets). The syntax to create an array is as follows :

```
data_type    array_name[array_size];
```

Example : int tab[10] ;

4.3.1. Searching for an element in an array

```c
#include <stdio.h>
#include <stdbool.h>

int main() {
  int arr[] = {1, 3, 5, 7, 9};
  int target = 5;
  int i;
  bool found = false;


  for (i = 0; i < sizeof(arr)/ sizeof(arr[0]); i++) {

    if (arr[i] == target) {
      found = true;
      break;
    }
  }

  if (found) {
    printf("The element %d is found at index %d\n", target, i);
  } else {
    printf("The element %d is not found in the array\n", target);
  }
    return 0;
}
```

4.3.2. Sorting

The following program performs the sorting according to the selection sort algorithm. This program works by iterating over the array and finding the smallest element in the unsorted part of the array. It then swaps the smallest element with the current element. The program repeats this process until the entire array is sorted.

```c
#include <stdio.h>
int main() {
  int arr[] = {5, 2, 8, 7, 1};
  int n = 5;
  int i, j, temp, smallest ;

  for (i = 0; i < n - 1; i++) {
    smallest=i;
    for (j = i + 1; j < n; j++) {
      if (arr[j] < arr[smallest]) {
       smallest = j;
      }
    }
    temp = arr[i];
    arr[i] = arr[smallest];
    arr[smallest] = temp;
  }

  printf("The sorted array is:\n");
  for (i = 0; i < n; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");

  return 0;
}
```

4.3.3. Operations on Matrices

The sum of the elements of the diagonal

```c
#include <stdio.h>

int main() {
  int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
  int i, sum = 0;

  for (i = 0; i < 3; i++) {
    sum += matrix[i][i];
  }

  printf("The sum of the diagonal elements is %d\n", sum);

  return 0;
}
```

Chapter 5 : Enumerations and Structures

1. Enumerations

1.1. Enumerations

The set type is created by defining the domain of values it contains, i.e., the list of constant values that variables of this type can take. Variables of this type take a value among a set.

Syntax for declaration

type enum_type = **set**(value1, value2, …);

Example :

type color= **set** (blue, green, red);

var c1 : color;

c1 ← green;

Enumerations are especially useful for enhancing code readability making the code more understandable and maintainable. The sets can be used to iterate in loops like

type Color= set (blue, green, red, white);

var c1:color;

for c1← blue to white do write (c1);

1.2. Enumerations in C

In C language, an enumeration is declared using the 'enum' keyword.

Syntax:

enum enum_type {value1, value2, …};

Example:

enum color {white, blue, yellow, green, black};

We declare a variable of type 'color' by specifying the name of the *enum* followed by the name of the variable to be declared, for example: enum color c1;.

Additionally, we can assign numerical constants to each color as follows:

enum color {white=10, blue=11, yellow=12, green=13, black=14};

If numerical constants are not specified, these values start from 0 and increment by 1 for each subsequent enumerator.

2. Structures

Unlike arrays, which are data structures where all elements are of the same type, records/structures are data structures where the elements can be of different types and relate to the same semantic entity.

- The elements that compose a record are called fields or attributes.
- A record is a user-defined data type that allows grouping a finite number of elements of different types.
- A record is a complex variable that allows designating, under a single name, a set of values that can be of different types (simple or complex).
- Before declaring a record variable, it's necessary to have previously defined the name and type of the fields that compose it.
- It's possible to create custom types and then declare variables or arrays of elements of that type

2.1. Declaration of Records/Structures

The declaration of structure types occurs within a specific section of algorithms called 'Type,' which precedes the section for variables and follows the section for constants.

Algorithmic language	C language

type recordName = structure
id1 : type1;
id2 : type2;
...
idN : typeN;
end;
	struct recordName {
	type1 id1;
	type2 id2;
	...
	typeN idN;
	};

Where <id1>, <id2>, ..., <idN> are the identifiers of the fields, and <type1>, <type2>, ..., <typeN> are their types respectively.

Example :

```
type car = structure
        brand: string;
        c: color; /* color is an enumeration type, as shown previously */
        price: float;
end;
var v1 : car;
```

2.2. Accessing the fields of a structure

To access a field of a structure, the variable ID of the structure type is used, followed by a dot and then the name of the field you want to access.

Syntax:

<structVariable>.<fieldName>

For example, to access the 'price' field of the variable 'v1' of type 'car',

v1.price

Assignment: v1.price ← 2000.00;

Or:

floatVariable ← v1.price;

Reading: read(v1.price);

2.3. Array of structures

In order to manage multiple entities of a given type (e.g., cars), we use a one-dimensional array structure to store these elements. In a rent car agency, the information of the available cars is stored in an array of structure from the type car.

Similarly, in a school context, information about the student is stored in an array of records from the type "student" that can be defined as follows.

type **student** = structure;

firstName : string;

lastName : string;

… end;

var **studentList** = array [0..99] of **student**

studentList is an array with 100 records that describe students.

2.4. Operations on structures

In the following is an example for reading data to fill the array of car structure.

```
Algorithm fill_array_car; const n=10;
type car = structure
        brand: string;
```

```
        c: string;
        price: float;
end;

var listCar = array[0..n-1] of car;
    i:integer;
Begin
        for i ← 0 to (n-1) do begin
                read(List_car[i]. brand);
                read (List_car[i].c);
                read (List_car[i]. price);
        end;
        endFor;
End.
```

In the following are instructions to display the content of the aforementioned array.

```
for i ← 0 to (n-1) do begin
        write(listCar[i]. brand);
        write (listCar[i].c);
        write(listCar[i]. price);
end;
endFor;
```

In the following, instructions to sort elements of *listCar* array according to the price (bubble sort)

```
var  tmp : car;
     i, j : integer;
     ...
        for i ← (n-2) to 0 do begin
                for j ← 0 to i do
                        if (listCar[j].price > listCar[j+1].price) then begin
                        tmp ← listCar[j];
                        listCar[j] ← listCar[j+1];
                        listCar[j+1] ← tmp;
                        end;
                        endIF;
                endFor;
        endFor;
```

2.5. Structures in C

To use structure in our program, we have to define its instance. We can do that by creating variables of the structure type. We can define structure variables using two methods:

- Variable declaration with structure declaration:

```
struct structureName {
  type1 id1;
```

```
    type2 id2;
    ....
    ....
}var1, var2, ...;
```

- Variable declaration after structure declaration:

```
// structure declared beforehand

struct structureName var1, var2, .....;
```

Members of a structure are accessed with a dot after the structure identifier.

Example in C

In the following is a C program that creates a list of three cars, and display them according to their year.

```c
#include<stdio.h>
void main (){
 enum color {red, blue,green};
 struct Car {
    int id;
    char model[20];
    char brand[20];
    enum color c;
    int year;
    } ;

 char listOfColor[3][20]={"Red","Blue","Green"} ;
 int n=3;

 struct Car listCar[n];

 for (int i=0;i<n; i++){
    listCar[i].id=i+1;
    printf("enter the model: ");   scanf("%s",&listCar[i].model);
    printf("enter the brand: ");   scanf("%s",&listCar[i].brand);
    printf("enter the c: ");       scanf("%d",&listCar[i].c);
    printf("enter the year: ");    scanf("%d",&listCar[i].year);
    }
```

```
24
25      struct Car temp;
26      for (int i=0; i<n-1;i++){
27          for (int j=0; j<n-i-1;j++){
28              if(listCar[j].year>listCar[j+1].year){
29                  temp=listCar[j];
30                  listCar[j]=listCar[j+1];
31                  listCar[j+1]=temp;
32              }
33          }
34      }
35
36      for (int i=0; i<n;i++){
37          printf("car id %d is %s %s with a %s color, year =%d\n",
38          listCar[i].id, listCar[i].model, listCar[i].brand,
39          listOfColor[listCar[i].c] ,listCar[i].year);
40      }
41  }
```