Chapter 5 : Arrays and Strings

Dr. Abderrahmane Kefali

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

kefali.abderrahmane@univ-guelma.dz

This document is prepared for printing two pages per sheet

1) Introduction

In fact, simple variables are not sufficient to handle complex problems involving a large amount of data and requiring the storage of entered data for future processing.

An alternative is to think about using a single variable that collects all the values. Fortunately, in algorithmics, there are other types, known as ***composite types***, ***structured types***, or ***data structures***, made up of basic types or other declared types, corresponding to a collection of several values.

In this chapter, we will delve into the primary structured type that we consider to be the most essential: **arrays**. Additionally, we will also explore strings, which are regarded as a specific type of array. For each of these types, we will provide details on declaration, manipulation, and offer some practical usage examples.

2) The Array type

2.1) Definitions

2.1.1)Array

An array is a homogeneous data structure that gathers, under a single name, a finite set of elements of the same data type. Therefore, you can have an array of integers, an array of real numbers, an array of characters, and so on.

The elements of the array are arranged in sequentially numbered adjacent **cells**. Each element can be identified by its position in the set.

In memory, an array is essentially a memory space sized to a finite number of contiguous cells or zones, ensuring the storage of multiple elements of the same data type. The elements of the array, therefore, occupy adjacent memory locations. The first element is followed by the second, followed by the third, and so on.

The main advantage of arrays is the ability to represent a set of values using a single identifier. This allows you to work with the array as a whole or access and manipulate individual elements within it.

2.1.2)Array element

An element refers to one of the values contained within the array.

2.1.3)Index

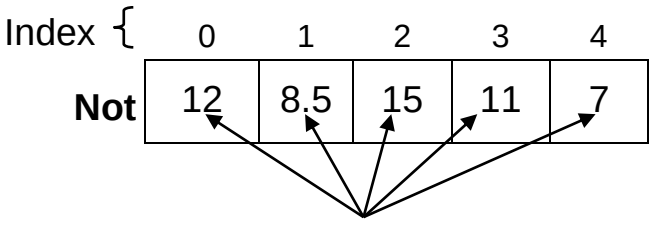
An index indicates the rank or position of the element. Typically, the index is expressed as an integer variable.

2.1.4)Size

The size, length, or cardinality of an array is the number of elements it contains. The size can only be a constant or a constant expression.

Example :

Consider the **Not** array below, which consists of 5 real elements arranged in 5 cells. Each element represents a student's mark.



The elements of the array

2.2) Declaration

The declaration of an array is done by specifying its name, size, and the type of elements stored in the array. We will first present the algorithmic syntax of declaration, followed by the syntax in the C language.

2.2.1)Algorithmic syntax

In algorithmics, the declaration of an array variable can be done in two possible ways: either directly or after defining an array type.

In the first method, you declare the array variable directly, much like you do with simple variables, with the only difference being that you specify it as an array variable by using the keyword **`ARRAY`**, followed by the size in square brackets, and the data type.

The declaration syntax is as follows:

```
Var <name_Array> : ARRAY[<size>] OF <type_elements>;
```

With:

- **`<name_Array>`**: This is the identifier that represents the name of the array.

- **<size>**: The number of elements in the array. This number can be specified as a direct value or as a previously declared constant to increase the flexibility of the algorithm.
- **<type_element >**: The type of elements in the array. It can be any simple or structured data type.

The second way is to define an array type (a new array data type) and then declare variables of the defined type.

The definition of a new array type follows the syntax:

```
Type <name_type> : ARRAY[<size>] OF <type_elements>;
```

Here, **<name_type>** is the name of the new array type.

Afterward, to declare an array variable corresponding to the defined type, you use the following syntax:

```
Var <name_Array> : <name_type>;
```

In this case, **<array_name>** is the name of the array variable, and **<name_type>** refers to the array type that was defined earlier.

Examples:

1. The previous ***Not*** array can be declared directly as:

```
Var Not:ARRAY[5] OF Real;
```

   Or it can be declared as:

```
Const card=5;
Var Not:ARRAY[card] OF Real;
```

2. Following the second method of declaration, the ***Not'***array can be declared as follows:

```
Type Nots=ARRAY[5] OF Real;
Var Not:Nots;
```

Remarks:

- Indices are not necessarily integers; they can be of any other scalar type. In this case, the size of the array doesn't need to be explicitly specified.
- Once an array has been declared, its size cannot be changed; it involves static memory allocation.
- Like any other types, you can declare multiple arrays with the same characteristics (same size and element type) by separating them with commas.

Exemple:

```
Const n=5;
Type Nots=ARRAY[n] OF Real;
Var A,B,C:Nots;
```

2.2.2)C language syntax

In the C language as well, you can declare an array variable directly or after defining the type.

When declaring an array directly, you specify the data type of the array's elements, followed by the name of the array and its size in square brackets. The declaration syntax is as follows:

```
<type_elements> <name_Array>[<size>];
```

Where:

- **<type_elements>** is the type of the elements in the array,
- **<name_Array>** is an identifier that represents the name of the array,
- **<size>** is the number of elements in the array. This number can be specified directly or as a pre-declared constant.

In the second way of declaring, the model or type of the array is first defined using the following syntax:

```
typedef <type_elements> <name_type>[<size>];
```

Where **<name_type>** is the name of the new array type being defined.

Subsequently, an array corresponding to the previous model is declared as follows:

```
<name_type> <name_Array>;
```

Examples:

1. The direct declaration in C language for the previous array ***Not*** can be done by:

   ```
   float Not[5];
   ```

   Or by:

   ```
   #define card 5
   float Not[card];
   ```

2. The following declaration allows you to declare an array variable named ***Not*** of type ***Nots***. The ***Nots*** type is a new type describing arrays with 5 elements of real numbers.

   ```
   typedef float Nots[5];
   Nots Not;
   ```

2.3) Manipulation of arrays

In algorithmics, the elements of an array can be manipulated individually, and therefore, they have the characteristics of any other variables. This means they can be read, displayed, used in assignments, expressions, and so on.

2.3.1)Accessing array elements

To access an element of an array, we use the name of the array followed by the index of the element enclosed in square brackets. The syntax for accessing array elements is as follows:

```
<name_Array> [<index>]
```

This syntax is common in both algorithmics and the C language.

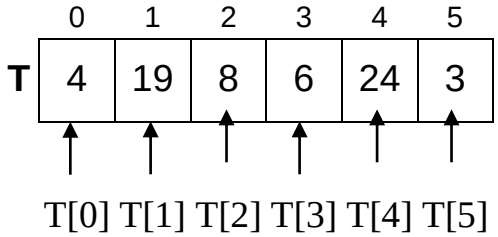
It's important to note that array elements are indexed starting from 0, not 1. Therefore, for an array of size ***N***, the elements are numbered from 0 to ***N*** - 1.

Example:

If ***T*** is an array of 6 integer elements declared as follows:

```
Var T:ARRAY[6] OF Integer;
```

Suppose ***T*** is as follows:



Remarks:

- The value inside the brackets during the array declaration (the maximum size) should not be confused with the value inside the brackets when used in instructions (the index).
- The index used to refer to the elements of an array can either be a direct, explicit value (e.g., `T[2]`), but it can also be a variable (e.g., `T[i]`, where `i` is a variable, typically an integer) or a computed expression (e.g., `T[i+2]`).
- Accessing an element with an index greater than or equal to the size of the array will consistently result in an incorrect result and often a memory error (attempt to access a memory location outside the array). This is referred to as an ***array overflow*** (or ***index overflow***). However, most compilers do not implement index overflow checks.

2.3.2)Filling the array

The array can be filled in by assignment, or by reading from the keyboard.

a) Filling by reading

Since each element is manipulated individually, reading the array involves reading all its elements one by one. This reading can be accomplished using the following loop:

```
For i ← 0 To N-1 Do
      Read(T[i]);
```

In the C language, it can be done as follows (assuming ***T*** is an array of integers):

```
for(i=0;i<N;i++)
      scanf("%d",&T[i]);
```

b) Filling by assignment

When you want to assign a value to a specific element of array ***T***, you can use the following syntax:

```
T[<index>] ← <value>;
```

Where **<index>** is the index of the targeted element, and **<value>** is the value to be placed in the element.

If you want to assign the same value to all elements of the array, you can use a loop like this:

```
For i ← 0 To N-1 Do
      T[i] ← <value>;
```

In C language:

```
for(i=0;i<N;i++)
      T[i] = <value>;
```

Examples:

- The following instruction stores the value 15.75 in the 2nd element of the array ***Not***:

```
Not[1] ← 15.75;
```

- L'initialisation à 0 de tous les éléments d'un tableau ***T*** de 6 entiers se fait par:

```
For i ← 0 To 5 Do
    T[i] ← 0;
```

Note:

In C language, the elements of an array can be initialized during the declaration of the array. To do this, you place the values of the elements within curly braces (`{` and `}`), separated by commas. In this case, it's not necessary to specify the size of the array in square brackets. Furthermore, you can provide values for only the initial elements in the curly braces.

Examples:

- We can initialize an array with the first 5 odd numbers like this:

```
int T[] = {1, 3, 5, 7, 9};
```

- You can also initialize only 3 values out of the 5. The remaining two elements will be uninitialized:

```
int T[5] = {1, 3, 5};
```

2.3.3)Displaying the contents of an array

To write (display) an array, you proceed in the same way. You simply replace the read or assignment instruction with the write instruction:

```
For i ← 0 To N-1 Do
      Write(T[i]);
```

In the C language, displaying the elements of an array ***T*** of size ***N*** can be done as follows (assuming ***T*** is an array of integers):

```
for(i=0;i<N;i++)
      printf("%d",T[i]);
```

3) Multidimensional Arrays

3.1) Definition

Algorithmics and most programming languages, including the C language, provide the possibility of declaring and using arrays with multiple dimensions (2 or more), called ***multidimensional arrays***.

A multidimensional array is an array whose elements can themselves be arrays that can, in turn, contain other arrays, and so on. Each element of such array is identified by several indices, one for each dimension.

For illustrative purposes, the examples will be limited to two dimensions, but the generalization to ***N*** dimensions is straightforward.

When the array is two-dimensional, it is called a ***matrix***. This is considered as a grid consisting of rows and columns. Therefore, two indices are needed to indicate the cells or elements of the matrix: one for rows and one for columns.

The diagram below represents a matrix of real numbers with 4 rows and 5 columns.

	0	1	2	3	4
0	7	4.5	7.25	4	15.75
1	11.5	16	10	9	16
2	13	8	15	14.5	20
3	9.75	19	12.25	14.25	5

3.2) Declaration

The declaration of a multidimensional array is done in the same way as a one-dimensional array, with the exception that here you need to specify the size for each dimension. For example, in the case of a matrix, you need to specify the number of rows and columns.

3.2.1)Algorithmic syntax

Like a simple array, declaring a variable of a multidimensional array type can be done in two possible ways: directly or through type definition.

The syntax for direct declaration is as follows:

```
Var <name_Array> : ARRAY[<S1>,<S2>,...] OF <type_elements>;
```

With:

- **<name_Array>** : being the identifier designating the name of the array.
- **<Si>** : representing the size along dimension **i**.
- **<type_elements>** : indicating the type of the elements in the array.

The second method involves defining a new multidimensional array type and then declaring a variable of that type. The definition of a new multidimensional array type follows this syntax:

```
Type <name_type> = ARRAY[<S1>,<S2>,...] OF <type_elements>;
```

Where **<name_type>** is the name of the newly defined type.

Then, declaring a multidimensional array variable is done with:

```
Var <name_Array> : <name_type>;
```

Examples:

1. The following declaration declares a matrix of real numbers named ***M*** with 4 rows and 5 columns:

```
Const nbR=4;nbC=5;
Var M : ARRAY[nbR,nbC] OF Real;
```

2. The following declaration allows declaring a matrix variable named ***M*** of type ***Mat***. This new type describes matrices of real numbers with 4 rows and 5 columns.

```
Const nbR=4;nbC=5;
Type Mat = ARRAY[nbR,nbC] OF Real;
Var M : Mat;
```

3.2.2)C language syntax

In the C language, a multi-dimensional array can be declared either directly or through a type definition. Unlike in algorithmics, in the C language, each dimension's size must be specified separately within brackets.

To declare a multi-dimensional array directly in the C language, you can use the following syntax:

```
<type_elements> <name_Array>[<S1>][<S2>]...;
```

Here:

- **<type_elements>** is the type of the elements in the array.
- **<name_array >** is an identifier for the array.
- **<Si>** represents the size along the i^{th} dimension.

In the second method, you declare a variable of a multi-dimensional array type in two steps:

The first step is defining the type describing the multi-dimensional array:

```
typedef <type_elements> <name_type>[<S1>][<S2>],...;
```

Here, **<name_type >** is the name of the newly defined type.

The second step is declaring a variable of the already defined type:

```
<name_type> <name_Array>;
```

Examples:

1. The direct declaration of a matrix of real numbers ***M*** with 4 rows and 5 columns is done in the C language by:

```
#define nbR 4
#define nbC 5
float M[nbR][nbC];
```

2. The declaration after defining the type of the matrix ***M*** is done by:

```
#define nbR 4
#define nbC 5
typedef float Mat[nbR][nbC];
Mat M;
```

3.3) Manipulation of Multidimensional Arrays

3.3.1)Accessing elements of a multidimensional array

In a multidimensional array, each element is identified by multiple indices, one for each dimension. Therefore, accessing an element involves specifying all its indices. The syntax for this access is as follows:

<name_Array> [<index1>,<index2>,....]

Where **<name_Array >** is the name of the array, and **<index$_i$>** is the index of the element in dimension ***i***.

In the C language, the access syntax is similar to that in algorithmics, except that each index must be enclosed in separate brackets:

<name_Array> [<index1>][<index2>]....

Example:

To access the element at the intersection of row number 2 and column number 3 of matrix ***M***, you would write:

- In algorithmic: **M[2,3]**
- In C language: **M[2][3]**

3.3.2)Filling a multidimensional array

A multidimensional array can be filled through either input or assignment.

a) Filling by reading

Filling a multi-dimensional array ***A*** of ***N*** dimensions (with respective sizes ***S1***, ***S2***, ..., ***SN***) through input is done using nested loops, as shown below:

```
For i ← 0 To S1-1 Do
   For j ← 0 To S2-1 Do
      ...................
        Read(A[i,j,...]);
```

In the C language (assuming ***A*** is an array of integers):

```
for(i=0;i<S1;i++)
    for(j=0;j<S2;j++)
      ...............
        scanf("%d",&A[i][j]....);
```

Example:

Filling a matrix of real numbers ***M*** with 4 rows and 5 columns through reading is done by:

- In algorithmics:

```
For i ← 0 To 3 Do
  For j ← 0 To 4 Do
        Read(M[i,j]);
```

- In C language:

```
for(i=0;i<4;i++)
  for(j=0;j<5;j++)
      scanf("%f",&M[i][j]);
```

b) Filling by assignment

When you want to assign a value to a single element of the array, you use the syntax:

```
T[<index1>,<index2>,...] ← <value>;
```

Where **<index$_i$>** is the index of the targeted element along dimension ***i***, and **<value>** is the value to be placed in the element.

To assign the value **<value>** to all the elements of the multidimensional array ***A*** with ***N*** dimensions (with respective sizes: ***S1***, ***S2***, ..., ***SN***), we proceed as follows:

```
For i ← 0 To S1-1 Do
   For j ← 0 To S2-1 Do
      ..................
          A[i,j,...] ← <value>;
```

And in the C language (assuming that ***A*** is a multidimensional array of integers):

```
for(i=0;i<S1;i++)
   for(j=0;j<S2;i++)
      ..............
          A[i][j]...← <value>;
```

Examples:

- To store the value 9 in the element at the intersection of the second row and fourth column of the previous matrix ***M***, we write:
  - *In algorithmic :* `M[1,3] ← 9;`
  - *In C language:* `M[1][3] = 9;`
- Let ***T*** be a matrix of integers with 10 rows and 7 columns. The initialization of all elements to 3 is done as follows:
  - *In algorithmic:*

```
For i ← 0 To 9 Do
   For j ← 0 To 6 Do
      T[i,j] ← 3;
```

- *In C language:*

```
for(i=0;i<10;i++)
   for(j=0;j<7;j++)
      T[i][j] = 3;
```

Note:

In the C language, just like with a simple array, it is possible to initialize the elements of a multidimensional array during declaration. To do this, you place the values of each dimension between curly braces, separated by commas, and enclose the entire set within global curly braces. Here, it is not necessary to specify the first dimension.

However, the others are required. As with simple arrays, the final values can be omitted.

Examples:

- The following statement declares an integer matrix with 3 rows and 4 columns and initializes its elements with values from 1 to 12:

```
int T[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}}
```

- In this statement, only a portion of the elements is initialized:

```
int T[3][4]={{1,2},{3,4,5}};
```

3.3.3)Displaying the elements of a multidimensional array

The display of elements in a multidimensional array is done in the same way as reading them, using nested loops. Thus, displaying the array ***A*** of ***N*** dimensions (with respective sizes: ***S1***, ***S2***, ..., ***SN***) is done through the following nested loops:

```
For i ← 0 To S1-1 Do
   For j ← 0 To S2-1 faire
      ..................
          Write(A[i,j,...]);
```

And in C language (assuming that A is an array of integers), it is done as follows:

```
for(i=0;i<S1;i++)
   for(j=0;j<S2;j++)
      ..............
          printf("%d",A[i][j]....);
```

Example:

Displaying the elements of a matrix of reals ***M*** with a size of 4×5 is done as follows:

- *In algorithmic:*

```
For i ← 0 To 3 Do
   For j ← 0 To 4 Do
        Write(M[i,j]);
```

- *In C language:*

```
for(i=0;i<4;i++)
   for(j=0;j<5;j++)
      printf("%f",M[i][j]);
```

4) Strings of characters

4.1) Definition of a string of characters

A string of characters is a homogeneous data structure used to store a finite number of character elements in the same variable.

Thus, a variable of the string data type can contain a sequence of zero, one, or more concatenated characters. The characters can be printable or non-printable.

In fact, the string type can be considered as an array of characters, but it is enriched by other special operations that facilitate its manipulation. These operations depend on the programming language in use.

Remarks:

- String constants must be enclosed in double quotes, in both algorithmics and C language.
- A string that contains no characters is called an empty string.

Examples:

- **`"Algorithmic1"`** is a string containing 12 alphanumeric characters.
- **`"6453"`** is a string of characters consisting of 4 numerical characters. It should not be confused with the numerical value 6453.
- **`"&)+_ $:%"`** is a string of characters composed of 8 special characters.
- **`""`** is the empty string.
- **`" "`** is a string consisting of a single character: the space character.

4.2) Strings in Algorithmics

The string data type is predefined in algorithmics but not in all programming languages. It is designated by the keyword **`STRING`**.

4.2.1)Declaration

The string data type defines "strings of characters" variables with a maximum of 255 characters (in the base case). A string may contain fewer characters if specified during its declaration, where the number of characters (ranging from 1 to 255) is placed within square brackets.

Therefore, similar to arrays, you can declare a string variable either directly or after defining a model. It's possible to declare a string variable directly using the following syntax:

```
Var <name_string> : STRING[<length>];
```

With: **<name_string>** being the name of the declared variable, and **<length>** being its length in characters. It's optional.

Dans la deuxième façon, on défini en premier lieu le type de chaînes de caractères considérées, et ensuite on déclare les variables. La syntaxe est comme suit:

```
Type <name_type> = STRING[<length>];
Var <name_string> : <name_type>;
```

With: **<name_type>** being the name of the type.

Remark:

When the length is not specified, the declared string is of maximum length (255 characters).

Examples:

1. Direct definition

```
Var ch:STRING;        // declare a variable ch of 255 characters
Var name:STRING[20];  // declare a variable name of 20 characters
```

2. The following declaration defines a type named **Information** which is a string of 20 characters, and then declares two variables of this type.

```
Type Information = STRING[20];
Var firstName,lastName : Renseignement;
```

4.2.2)Memory Representation

A string of characters occupies a contiguous space in memory. This space is proportional to the declared character string's length. Therefore, each character, encoded in ASCII, occupies one byte in the memory.

In practice, one byte is added to this reserved space, used to store the actual length of the string. This length is the number of characters actually contained in the string and is not the number specified during declaration. If this length is not specified, a default length of 255 characters is applied.

4.2.3)Manipulating Strings

Strings can be manipulated in a global or local manner (each character individually).

a) Accessing a character in the string

Accessing a specific character in a string is done in the same way as with arrays. The access syntax is as follows:

```
<name_string>[<rank>];
```

Where **<name_string>** is the name of the string, and **<rank>** is the position of the character you want to access.

Remarks:

- In algorithmics, character indexing in a string starts from 1, unlike arrays where it typically starts from 0. The cell at position 0 is used to store the actual length of the string.
- If the effective length is less than the declared length, the remaining reserved space in the string will be filled with ***null*** characters.

Example:

Let the character string **S** declared by:

```
Var S:STRING[5];
```

Suppose this string contains the value "**Box**". This string can be represented as follows:

S

0	1	2	3	4	5
3	**B**	**o**	**x**	Null	Null

b) Reading

Unlike arrays, a string of characters in algorithmics can be read in its entirety with a single read instruction. It is done as follows:

```
Read(<name_string>);
```

Example:

```
Read(firstName);
```
//reads the string **firstName** from the keyboard

b) Writing

Similarly for writing, a string in algorithmics can be displayed using a single writing instruction as follows:

```
Write(<name_string>);
```

It is also possible to display a single character from the string by specifying its rank (position) in brackets in the writing instruction. The syntax is:

```
Write(<name_string[rank]>);
```

Where **<rank>** is the position of the character to display.

Examples:

Write(firstName); //displays the content of the string **firstName**

Write(firstName[1]); //displays only the first character of **firstName**

c) Assignment

We can assign a string expression to any string variable using the assignment symbol ←. The assignment syntax is as follows:

```
<name_string> ← <expression>;
```

Where **<name_string>** is the name of the string of characters, and **<expression>** is the value being assigned.

We can also assign a value (which must be of character type) to a specific character within a string. The syntax is as follows:

```
<name_string>[<rank>] ← <value>;
```

Where **<rank>** is the position of the character to be modified, and **<value>** is the value to be stored, which must be of character type.

Examples:

firstName ← "Ali"; Assign the constant **"Ali"** to the variable **firstName**

firstName[2] ← 'a'; Assign the letter **'a'** to the 2nd character of **firstName**

ch ← firstName; Copy the content of **firstName** into the variable **ch**.

d) Operations specific to strings of characters

Algorithmics provides a set of operators and predefined functions specifically designed for manipulating strings of characters. Here are a few of them:

d.1) Concatenation: Concatenation is the operation used to create a new string by joining together two or more strings. In algorithmics, this is done using the "+" operator.

d.2) Comparison: It's possible to compare two strings using the standard comparison operators (=, >, <, ≥, ≤, ≠). Just like with individual characters, the comparison of character strings is based on the order of ASCII character codes.

d.3) Calculating effective length: Algorithmics offers a predefined function to determine the number of characters in a string. This function is called **Length**. The syntax for using this function is as follows:

```
Length(<name_string>);
```

Example:

```
Var ch1,ch2:String[20];len1:Integer;comp:Boolean;
.................
ch1 ← "Hello";
ch2 ← ch1 + " World"; // ch2 will contain "Hello world".
len1 ← Length(ch1); //len1 will contain the value 7.
comp ← "Annaba" < "Guelma"; // comp will contain the value True.
```

4.3) Strings in C language

In the C language, there is no real data type specifically for strings of characters (like the algorithmic **STRING** type). However, there is a convention for representing character strings. This convention involves treating a string as an array of characters and providing specific handling for it.

4.3.1)Declaration

A string variable can be declared directly in C using the following syntax:

```
char <name_string>[<size>];
```

Here, **<name_string>** is the name of the string, and **<size>** is the maximum number of characters that the string can hold.

Also, a string variable can be declared after defining a type or template as follows:

```
typedef char <name_type>[<size>];
<name_type> <name_string>;
```

Where **<name_type>** is the name of the declared type.

Examples:

- The string **name** of 20 characters can be declared as follows:

```
char name[20];
```

  Or as:

```
typedef StrType[20]
StrType name;
```

4.3.2)Memory Representation

In the C language, a string of characters is represented in memory as a sequence of bytes corresponding to each of its characters, specifically their ASCII codes. This sequence is terminated by an additional byte used to store a special character known as the ***null character*** or ***end of string character***, represented as **'\0'**. This null character indicates that the string ends at the preceding character. This

means that, in general, a string of **n** characters occupies a memory location of **n+1** bytes.

Example:

For example, the string "Guelma" is represented in an array of size 10 as follows:

0	1	2	3	4	5	6	7	8	9
G	**u**	**e**	**l**	**M**	**a**	**\0**			

Note:

To be valid, a string must necessarily end with the null character **'\0'**. Otherwise, a runtime error occurs.

4.3.3)Manipulating Strings

a) Accessing a character in the string

Accessing a specific character in a string is done in the same way as accessing an element in an array. The access syntax is as follows:

```
<name_string>[<rank>];
```

Where **<name_string>** is the name of the string, and **<rank>** is the position of the character we want to access.

Note:

Unlike in algorithmics, in the C language, the first character starts at index 0, not 1.

b) Reading

In the C language, reading a string can be done, like any other variable, using the **scanf** function with the format code **"%s"**.

```
scanf("%s",<name_string>);
```

Here, **<name_string >** is the name of the string variable to be read.

Note the absence of the **'&'** symbol in **scanf** when reading a string.

However, the C language has another function specifically designed for reading a string: the **gets** function. Its syntax is as follows:

```
gets(<name_string>);
```

Example:

Given the following string of characters:

```
char ch[20];
```

Reading this string from the keyboard can be done using:

```
scanf("%s",ch);
```

Or by:

```
gets(ch);
```

b) Writing

For displaying a string in the C language, you can use the **printf** function with the format code **'%s'**.

```
printf("%s",<name_string>);
```

Here, **<name_string >** is the name of the string variable to be displayed.

Just like reading, C also has another function specifically designed for writing a string: the **puts** function. Its syntax is as follows:

```
puts(<name_string>);
```

Exemple:

Displaying the string of characters **ch** is done as follows:

```
printf("%s",ch);
```

Or by:

```
puts(ch);
```

c) Assignment

In the C language, it's not possible to directly assign a value to a string variable, except during its declaration. The syntax for such an assignment is as follows:

```
char <name_string>[<length>] = <value>;
```

Here, **<value>** is the initial value assigned to the **<name_string >** variable. It's a string of characters constant enclosed in double quotation marks.

Note that you should not include a final **'\0'** character; it is added automatically.

However, we can assign a value to a specific character within a character string. This assignment follows the following syntax:

```
<name_string>[<rank>] ← <value>;
```

With **<rank>** representing the position of the character to be modified, and **<value>** representing the value to be stored, which must be of character type.

Example:

- We can declare the 30-character string **course** and initialize it with the constant **"Algebra"** in a single line as follows:

```
char course[30] = "Algebra";
```

- To assign the letter **'v'** to the 5th character of the **course** variable, we write:

```
course[4] ← 'v';
```

d) Operations specific to strings of characters

The C language provides a variety of functions for manipulating strings of characters. These functions are defined in the **<string.h>** library (header file). In the following table, we list some of these functions:

Fonctions	Description
strcat()	Concatenate two strings
strcmp()	Compare two strings
strcpy()	Copy one string into another
strlen()	Return the length of a string

Example:

```
char s[20]="Hello ";
char r[20]="World";
chat t[20];
strcpy(t,s);        //copy the contents of the string s into the string t.
int nb=strlen(s);//Store the length of the string s, which is 6, in nb.
strcat(s,r);//Concatenate the two strings s and r and store the result in s.
int v=strcmp(t,r); //The value contained in v will be a negative value
indicating that the string "Hello " is less than the string "World" in
alphabetical order.
```