

Subscribe to DeepL Pro to edit this document.  
Visit [www.DeepL.com/pro](https://www.deepl.com/pro?cta=edit-document) for more information.

* 1. **Two-dimensional tables**

Consider the following example: a section of 1000 students taking 7 modules, and we want to calculate the average for each student.

What is the data? What are the results? How are they reported?

* Using 1000 \* 7 variables for the marks of 1000 students in 7 modules and 1000 variables for the averages !!!!!! is madness,
* It's also illogical to use 7 vectors (one-dimensional tables) of size 1000 for grades and another for averages.
* In this case, the vector does not correspond to this reality, so look for other structures **⇒**

**Two-dimensional table (matrix), we create a single matrix variable with 1000 rows (students) and 8 columns (grades + average).**

* A table or matrix is a two-dimensional table made up of several rows and several columns.
* To move around a matrix, you need two indices (by convention i and j), the first indicating the row number and the second indicating the column number. The element with index **[i,j] is** the one where row i crosses column j.
* A cell in a matrix is known by the name of the matrix followed in square brackets by its row index and column index: Example **M[i,j].**
* ***As the vectors*** :
  + The index can be a constant, a variable or an arithmetic expression.
* An element of the matrix is handled in exactly the same way as a variable.
* In the Constant section, you can define the size of the matrix (rows, columns). Next, you can declare the number of elements actually to be entered.
* The number of elements to be entered must not exceed the size of the matrix.
* In algorithms, the index of rows and columns starts with 1.

**Lines**

**Index of**

**Columns: j**

**Columns**

**M**

**Line index: i**

**M[2,2]**= 8

**M[3,5]**= 11

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 4 | 23 | 12 | 1 |
| 11 | 8 | 22 | 15 | 2 |
| 7 | 13 | 17 | 5 | 11 |
| 10 | 1 | 4 | 20 | 23 |
| 14 | 76 | 11 | 12 | 33 |

**To declare a matrix :**

|  |  |
| --- | --- |
| **Algorithm** | **Code C** |
| **"Name**": array of "**row\_size,column\_size**" of "**type**".  **"Name**": array of [VminL..VmaxL, VminC..VmaxC] of "**type**".  Exp :  T: array of [1000,8] reals  T: array of [1..1000,1..8] reals | **"type**" "**name**[**size**][ **size**]";  Exp :  float T[1000][8] ; |

* 1. **Operations on matrices**
     1. **Describe any element of a matrix :**

|  |  |
| --- | --- |
| **Algorithm** | **Code C** |
| table name [row index , column index] .  Exp : T[2,3]← 5, T[3,1]← a+1, T[1,1] = 'a', ... | table name [row index][column index]  Exp : T[2][3] =5, T[3][1]=a+1, T[0][0]=='a', ... |

* + 1. **Reading a matrix**

|  |  |
| --- | --- |
| **Algorithm** | **Code C** |
| **For** i from 1 to 1000 **do**  **For** j from 1 to 7 **do**  Read(T[i,j])  **End for**  **End for** | **for** (i=0 ;i<1000 ;i++)  **for** (j=0 ;j<7 ;i++)  **scanf**(''%d'', &T[i][j]) ; |

* + 1. **Displaying a matrix**

|  |  |
| --- | --- |
| **Algorithm** | **Code C** |
| **For** i from 1 to 1000 **do**  **For** j from 1 to 7 **do**  write(T[i,j])  **End for**  **End for** | **for** (i=0 ;i<1000 ;i++)  {  **for** (j=0 ;j<7 ;i++)  **printf**(''%d\t'', T[i][j]) ;  **printf**(''\n'') ;  } |

* 1. **N-dimensional tables**

An N-dimensional array is declared as shown below, specifying the number and type of values it will contain.

**Examples:**

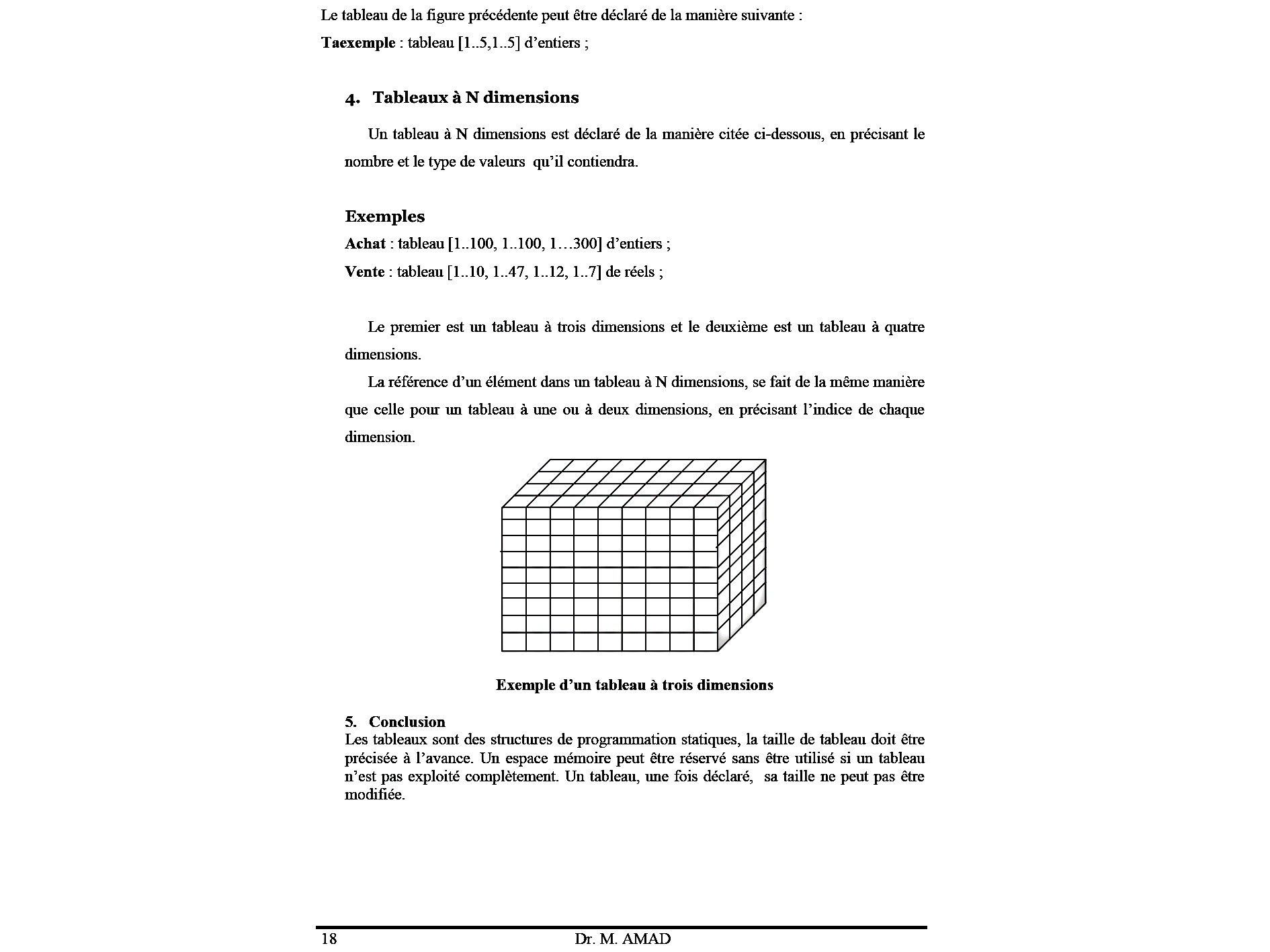
Purchase: array [1..100, 1..100, 1..300] of integers ;

Sale: table [1..10, 1..47, 1..12, 1..7] of real ;

The first is a three-dimensional table and the second is a four-dimensional table.

Referencing an element in an N-dimensional array is done in the same way as for a one- or two-dimensional array, by specifying the index of each dimension.

Example of a three-dimensional table



**Example of a three-dimensional table**

# **Exercises at home**

**Exercise 1**

Write an algorithm for filling and displaying a two-dimensional array

**Exercise 2**

Write an algorithm to find the minimum in a two-dimensional array

**Exercise 3**

Write an algorithm to calculate the average of the anti-diagonal elements of a two-dimensional array

**Exercise 4**

Let A be a table containing the marks for all the modules of a student X, B: the vector of coefficients corresponding to these marks, Write a program to calculate the average for student X.

**Exercise 5**

Write an algorithm to calculate the number of occurrences of a given number x in a two-dimensional array.

* 1. **String (character strings)**

Strings are sequences of characters made up of signs that are part of the computer's representable character set. We have already used strings, usually as parameters to the printf function. For example, in the instruction: printf('' enter an integer''); This character string is a constant, but character strings can also be variables.

In C, ***character strings***, whether constant or variable, are structurally ***one-dimensional arrays with elements of type char***.

For example, the definition char s[17] ; creates an array s with 17 modifiable character elements.

**To declare a character string :**

|  |  |
| --- | --- |
| **Algorithm** | **Code C** |
| **Name**: character string  Exp :  s: character string | **char** "**name**[**size**]";  Exp :  char s[17] ; |

**Remarks** :

1. Unlike data of other types, strings must end with a special character, the null character \0. The function of this character is to indicate where the string ends. In our example, the array s would look like this after the string 'enter integer' had been stored correctly:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **e** | **n** | **t** | **r** | **e** | **R** |  | **U** | **n** |  | **e** | **n** | **t** | **i** | **e** | **r** | **\0** |

1. A char array must therefore always be created with one element (at least) longer than the length of the string to be stored.
2. By using an array to store a string, it can be easy for the programmer to explicitly add the null character to the end of the string (with a specific instruction, for example).
3. By using one of C's string functions to store this string, you don't have to worry about the null character.
4. Also, when using an array to store a string, C's vocabulary does not allow you to manipulate arrays globally. You manipulate them element by element.
5. Using one of C's string functions to store this string, there are a series of specific functions for processing block strings.
   * 1. **Reading and writing a character string**

In algorithms, to enter and display a character string, and among other conventions, we simply use **read(string)** and **write(string),** where string is the name of the character string. In C, several methods are used.

1. **Entering and displaying strings in character mode :**

Exp: to enter and display the previous string using the keyboard in character mode :

#include<stdio.h>

Main()

{

Char s[20] ; int i ;

i=0 ;

while((s[i]=getchar()) !='\n')

i++ ;

s[i]='\0' ;

i=0 ;

while(s[i] !='\0')

{

putchar(s[i]) ; i++ ;

}

}

1. **Entering and displaying strings with scanf and printf :**

The scanf and printf functions have a %s format specification which can be used to manipulate strings.

Exp: to enter and display the previous string using scanf and printf :

char s[20] ;

scanf(''%s'', s); // no addressing operator & in front of s

printf(''%s'', s) ;

**Rq:**

In both scanf and printf, the %s format can be supplemented by an integer indicating the maximum number of characters to be read or displayed.

Exp: scanf(''%10s'', a); reads at most 10 characters and arranges them in array a. Similarly, printf(''%10s'', a); displays at most 10 characters of the string arranged in array a.

1. **Entering and displaying strings with gets and puts :**

A simple alternative to scanf is the gets function. This reads a character string en bloc from the keyboard, and its simple syntax is as follows: **gets**(s); s is the name of the previous string.

The display routine corresponding to gets is given by the **puts** function.

Exp : puts(s) ; displays the character string s on the screen.

**Rq:** The puts argument can also be a constant string, as in this instruction: puts(''character string''); which displays the string 'character string.

* + 1. **Operation on character strings**

1. **Assignment operation (initialisation) :**

String variables can be initialised as soon as they are defined by one of the following methods (like all other arrays):

* char s[30]={'c','h','a','î','n','e',' ','d','e',' ','c','a','r','a','c','t','è','r','e','s','\0'} ;
* char s[] = {'c','h','a','î','n','e',' ','d','e',' ','c','a','r','a','c','t','è','r','e','s','\0'} ;
* char s[30]='''character strings'' ;
* char s[]='''character strings'' ;

1. **Other assignment techniques :**

s[0]='c' ; s[1]='a' ; s[2]='î' ; s[3]='n' ; s[4]='e' ; s[5]=' ' ; s[6]='d' ; s[7]='e' ; s[8]=' ' ; s[9]='c' ; s[10]='a'; s[11]='r'; s[12]='a'; s[13]='c'; s[14]='t'; s[15]='è'; s[16]='r'; s[17]='e'; s[18]='s'; s[19]='\0' ;

1. **The strcpy function:** copies a string into an array.

**Ex:** strcpy(s, '' string of characters''); copies the string '' string of characters'' into s (which is a char array or a string). The old contents of s are overwritten.

**Rq:** to use **destrcpy and** some other string processing functions, you need to include the library**: #include<string.h>.**

1. **The strcat function:** is used to reconcile two strings by tuning one of them to the end of the other. Its syntax is: strcat(c1,c2), where c1 is the first string and c2 is the second.

**Ex:** c1[]=''concatenation ''; c1[]=''character string''; strcat(c1,c2); gives the result: c1[]=''concatenation character string''.

**Rq:** when concatenating, the null character in the first string is overwritten. The new string logically ends with a null character.

1. **The strcmp function:** compares two strings character by character, until a difference is detected or the null character is reached. The comparison is performed in lexicographic (alphanumeric) order. In other words, we check each time whether or not the two characters to be compared occupy the same place in the character table used (the ASCII table, for example). For example, in the ASCII table, character Z is larger than character A, and smaller than character a.
2. **The strlen function:** calculates the length of a character string in bytes (i.e. the number of characters), the null character not being included here. Its syntax is: strlen(s); s is the character string.

Exp : **a=strlen(s)** ; gives as result an integer a which is the number of characters in s without the null character, a=20 in the example where s[]=''character string''.

# **Exercises at home**

**Exercise 1**

Write an algorithm to calculate the number of vowels, consonants and white units in a string of characters. We assume that our string contains only vowels, consonants and white units, and not numbers.

**Exercise 2**

Write a C program that displays the number of words in a sentence. We assume that the words are separated by a single blank and that the sentence ends with a full stop.

**Exercise 3**

Write an algorithm that deletes one blank unit from two units. i.e. if two words are separated by two blanks, then one is deleted.

**Exercise 4**

Write a C program to replace the letter a with the letter A in a character string.

**Exercise 5**

Write a program to check whether a word is a palindrome or not.

**Exercise 6**

Write an algorithm and its C program that calculates the number of lowercase letters in a string.