Chapter 5: Arrays and Strings

1. Introduction

In programming, data is organized as constants and variables in certain ways to facilitate processing and quick access. There are different types of data, which can be divided into two parts: Simple types, such as integers, floats, characters, and booleans. Composite types: arrays, structures, or records.

Let's say we want to input grades for 1000 students, analyze them, and calculate some statistics. In this case, it would be unreasonable to use 1000 variables to store grades and write 1000 input instructions in the program. It's better to use a single variable that can hold all the grade values and use a loop to input them. A structure that can store multiple values simultaneously is called an array.

In this chapter, we'll cover two types of static arrays: one-dimensional and multi-dimensional arrays. We'll also see that strings are a special case of arrays.

2. The Array Type

2.1. Definition

Array: A complex data structure consisting of a finite set of homogeneous elements (of the same type), accessible by indexes indicating their location.

An array can be seen as a group of variables of the same type with the same name.

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

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

One-Dimensional Array

It's also called a vector: you can access any of its elements using a single index, where each index value selects an element from the array.

2.2. Representation

The array is represented in memory as a sequence of adjacent cells. A new cell cannot be removed or added to the array after its creation (static). The following figure represents an array of 8 real numbers.

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

There's no difference in drawing the array vertically or horizontally.

2.3. Declaration

Algorithm	C
`arrayName [Size] : Array of elementType`	`elementType arrayName [Size] ;`

The phrase "Array of" is a reserved word in the algorithm, used to indicate that the variable is an array.

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

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

Example

`const N=100`
`marks [N] : array of real`
`tab1[50],tab2[20] : array of integer`
	`const int N=100 ;`
	`float marks [N] ;`
	`int tab1[50],tab2[20];`

2.4. Initialization

In C, you can specify initial values for all array elements using curly braces { and } during array declaration. Values are separated by commas ",", and these values must be of the same type.

Example

```
int tab[] = {15, 7, -2, 0, 9, 2, 0, -3};
```

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

Note: You can specify the number of elements between the two square brackets "[ ]", or leave them empty for automatic calculation.

2.5. Usage

An array cannot be treated as a single block like array * 10; each element must be treated separately. To access a single element of the array, we use the array name with an index inside square brackets [and], and the expression inside the brackets evaluates to an integer value. To access the first element of array 'tab', we use tab[0]. To access the fourth element, we use tab[3].

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

le tableau devient

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

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

2.6. Reading an Array

To fill an array of N numbers, we use "read" N times like:

Algorithm	C

`Read (tab[0])`
`read (tab[1])`
…
`read (tab[N-1])`
	`scanf("%d", &tab[0]);`
	`scanf("%d", &tab[1]);`
	…
	`scanf("%d", &tab[N-1]);`

However, we notice that the read instruction is repeated **N** times, iterating from 0 to N-1. Therefore, the "**for**" loop can be used, with the counter playing the role of the index.

Algorithm	C

`for i←0 to N-1 do`
`  write("nb", i, "⇒")`
`  // Just to clarify`
`  read(tab[i])`
`end for`
	`for(i = 0; i < N; i++){`
	`  printf("nb %d ⇒", i);`
	`  // Just to clarify`
	`  scanf("%d", &tab[i]);`
	`}`

The "read" instruction is to fill the table, and the "write" instruction is to show the user what is required.

2.7. Displaying an Array

Like reading, "write" repeats.

Algorithm	C

`for i←0 to N-1 do`
`  write(tab[i])`
`end for`
	`for(i = 0; i < N; i++){`
	`  printf("%d\t", tab[i]);`
	`}`

2.8. Observations

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

2.9. Example

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

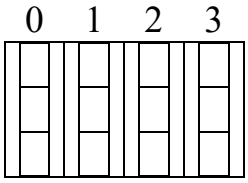
Algorithm	C

Algorithm nb_adjourned
Const MAX=200
var avg[MAX] :**array of** real
i, aj, N : integer
begin
do
write ("enter number of students (<", MAX, ")")
read (N)
while N>MAX
for i←0 **to** N-1 **do**
write ("mark ", i, "⇒")
read (avg[i])
end for
aj ←0
for i←0 **to** N-1 **do**
if avg[i]<10 **then**
aj←aj+1
end if
end for
write("the number of adjourned is ", aj)
end
	#include<stdio.h>
	#define MAX 200
	int main(){
	float note[MAX] ;
	int i, N, aj=0; // aj is nb of adjourned
	// retrieve number of students
	do{
	printf("enter number of students (<%d)",MAX) ;
	scanf("%d",**N**) ;
	}while (N>MAX) ;
	// Fill in the table
	for(i = 0; i < N; i++){
	printf("avg %d ⇒", i);
	scanf("%d", &note[i]);
	}
	// calculate number of adjournments
	for(i = 0; i < N; i++)
	if(avg[i]<10) aj++ ;
	// result display.
	printf("the number of adjourned is %d", aj);
	}

3. Multi-Dimensional Arrays

3.1. Definition

A two-dimensional array (also called a matrix) is essentially a simple array (one-dimensional) whose elements are themselves one-dimensional arrays. We see this in the illustration below,



The elements are accessible via two indices, the first specifying the row number and the second specifying the element number in that row (column).

This mechanism can be generalized to create matrices with more than two dimensions. We can create an n-dimensional array, and accessing its elements requires n indices. The arrangement of indices is crucial. The element M[3][2] (36) is different from the element M[2][3] (28).

3.2. Representation

The matrix is represented in memory by a sequence of adjacent cells. A cell cannot be removed or added to the matrix after its creation (static). The following figure represents a matrix with 3 rows and 5 columns of real numbers.

Column Number

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

3.3. Declaration

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

The term "Array of" is a reserved word in the algorithm, used to indicate that the variable is an array.

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

- `Rows`: Number of rows.
- `Columns`: Number of columns.
- `elementType`: The type of elements. It can be any type, such as integer (`int`), float (`float`), ...

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

Example

```
const R=100
const C=100
M[R][C] : Array of real
mat1[50][30],mat2[30][20] : tableau d'entier
```

```
const int R =100 , C=200 ;
float M[R][C] ;

int mat1[50][30],mat2[30][20];
```

3.4. Initialization

In C, the initial values of all matrix elements can be specified by specifying the elements of each row between `{` and `}`, along with the matrix declaration. The values are separated by commas `,`, and each row is separated by a comma `,`. All values must be of the same type.

Example:

int mat[][] = **{{**15, 7, -3 ,0 ,9**},{**6, 12, 4,33,85**},{**2, -8, 17 ,28,-52**},{**14, 42, 36, 49, -12**}}**;

column number

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

Note: You can specify the number of rows and columns between the two brackets or leave them empty to be calculated automatically.

3.5. Usage

To access a single element of the matrix, we use the matrix name with an index inside two brackets `[` and `]` specifying the row number, and another index inside two brackets `[` and `]` specifying the column number. To access the element in the first row and first column of matrix `mat`, we use `mat[0][0]`.

syntax `mat[line][column]`

example `mat[1][3]←mat[1][3]+2`

the matrix becomes

Column number

Row Number	0	1	2	3	4
0	15	7	3-	0	9
1	6	12	4	35	85
2	2	8-	17	28	52-
3	14	42	36	49	12-

3.6. Reading a Matrix

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

Algorithm	C

for j←0 **to** C-1 **do**
read(M[0][j])
end for
for j←0 **to** C-1 **do**
read(M[1][j])
end for
…
for j←0 **to** C-1 **do**
read(M[L-1][j])
	for (j=0 ;j<C ;j++)
	scanf("%d", &M[0][j]);
	for (j=0 ;j<C ;j++)
	scanf("%d", &M[1][j]);
	…
	for (j=0 ;j<C ;j++)
	scanf("%d", &M[L-1][j]);

end for	

However, we notice that the `for` loop is repeated `Rows` times. In other words, it iterates from 0 to `Rows - 1`. Therefore, the outer `for` loop can be used.

Algorithme	C

for i←0 **to** L -1 **do**
 for j←0 **to** C-1 **do**
  read(M[i][j])
 end for
end for
	for(i = 0; i < L; i++)
	 for(j = 0; j < C; j++){
	  printf("M[%d, %d] ⇒", i,j);
	  scanf("%d", &M[i][j]);
	 }

The `read` statement is used to fill the matrix, and the `write` statement is used to explain to the user what is required. The `for(j ...)` loop contains two `printf` statements to explain and a `scanf` to enter values. The `for(i ...)` loop contains only one `for(j ...)` loop statement.

Explanation of `write`: Let's assume `i = 3` and `j = 5`

write(	"M["	i	","	j	"]⇒"
screen	M[	3	,	5	]⇒

3.7. Displaying a Matrix

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

Algorithm	C

pour i←0 **à** L-1 **faire**
 pour j←0 **à** C-1 **faire**
  write(M[i][j])
 finPour
finPour
	for(i = 0; i < L; i++)**{**
	 for(j = 0; j < C; j++)
	  printf("%d\t", M[i][j]);
	 printf("\n") ;
	}

The `for(j ...)` loop contains the `printf` statement to print element `M[i][j]`. The `for(i ...)` loop contains two `for(j ...)` loops for printing row `i`, and `printf("\n")` to move to the next line at the end of each row `i` of the matrix.

Note: To visit all elements of the matrix, we use two `for` loops.

3.8. Example

Write a program that reads hourly temperatures for 30 days as a matrix (30 by 24), then displays them on the screen. After that, display the highest temperature and when it was recorded.

Algorithm	C

Algorithm temperatures
Const Dy =30 Hr=24
var T[Dy][Hr] :**array of** real
  maxT :real
  i, j,maxDy,maxHr :integer
begin
for i←0 **to** Dy-1 **do**
 for j←0 **to** Hr-1 **do**
  write ("T[", i+1, ",", j, "]⇒")
  read (T[i][j])
 end for
end for
for i←0 **to** Dy-1 **do**
 for j←0 **to** Hr-1 **do**
  write (M[i][j])
 end for
end for
maxT←T[0][0]
maxDy←0
maxHr←0
for i←0 **to** Dy-1 **do**
 for j←0 **to** Hr-1 **do**
  if (T[i][j]>maxT) **then**
	#include<stdio.h>
	#define Dy30 // nb lignes
	#define Hr 24 // nb colonnes
	int main(){
	float T[Dy][Hr] ,maxT; // max température
	int i, j, maxDy,maxHr;
	// Fill in temperatures
	for(i = 0; i < Dy; i++)
	 for(j = 0; j < Hr; j++){
	  printf("T[%d, %d] ⇒", i+1,j);
	  scanf("%d", &T[i][j]);
	 }
	// display all temperatures
	for(i = 0; i < Dy; i++){
	 for(j = 0; j < Hr; j++)
	  printf("%d\t", M[i][j]);
	 printf("\n") ;
	}
	// search for maximum temperature
	maxT=T[0][0];
	maxDy =0 ;
	maxHr=0 ;
	for(i = 0; i < Dy; i++)
	 for(j = 0; j < Hr; j++)
	  if (T[i][j]>maxT){

```
   maxT←T[i][j]
   maxDy←i
   maxHr←j
 end if
end for
end for
write("the maximum temperature is ",
   maxT," and was recorded on ", maxDy+1,
   " at ", maxHr )
end
```

```
    maxT=T[i][j];
    maxDy =i;
    maxHr=j;
   }
// display of results
printf("the maximum temperature is %d and
   was recorded on %d at %d", maxT, maxDy
   +1, maxHr) ;
}
```

The program takes the temperature matrix `Temperatures` and outputs the highest temperature recorded `maxTemp`, the day it was recorded `maxDay`, and the hour it was recorded `maxHour`. After filling the matrix and displaying it, we assume that the highest temperature is in row 0 and hour 0. Then we go through all elements of the matrix, and if we find a temperature higher than the one stored in `maxTemp`, `maxTemp` changes, and so do `maxDay` and `maxHour`. In the end, we display the results on the screen, incrementing `maxDay` by 1 since rows start from 0 and days start from 1.

4. Strings

4.1. Definition

A string is an ordered set of characters, zero or more. They are always enclosed in double quotation marks `"` such as "computer", "Good luck\n", "1", "3.14". In C, character arrays are used to create strings. When you read a string from the keyboard, each character is placed in a location, and when the characters are finished, the character '\0' is added to the end of the text to indicate its end. The character '\0' is called "null," with a code of 0. There is a constant declared in the stdio.h library called **NULL** in uppercase.

```
#define NULL 0
```

Note: Since each character has a code, for example, the code of 'A' is 65, the code of 'a' is 97, similarly, the character '\0' has a code which is 0. NULL? '\0' ? 0.

NULL ⇔ '\0' ⇔0

4.2. Declaration

In algorithms, we use the string, while in C, we use a character array. Suppose we have the string `str`, which can contain a maximum of 30 characters, including '\0'. It is declared as follows:

```
var str : string
var str[30] :array of characters
```

```
char str[30] ;
```

4.3. Initialization

In the following example, we create a character array and initialize it with the word "Welcome."

```
char greeting[] = {'W', 'e', 'l', 'c', 'o', 'm', 'e', '\0'};
```

This statement creates an 8-character array (7 slots for the word "Welcome" and one slot containing the character '\0'). However, there is a simpler and faster way to create and initialize a string:

```
char greeting[] = "Welcome";
```

This leads to the same result, which is creating an 8-character array, ending with the character '\0'.

	0	1	2	3	4	5	6	7
greeting	W	e	l	c	o	m	e	\0

The size of the array can also be specified:

```
char greeting[30] = "Welcome";
```

	0	1	2	3	4	5	6	7	8	…	28	29
greeting	W	e	l	c	o	m	e	\0		…		

4.4. Assignment

Since strings are arrays, a string cannot be assigned to a variable directly after its declaration. The following operation is incorrect:

```
char slt[30];
slt = "Welcome";// error : assignment to an array.
```

To assign a string to a variable or copy one variable to another, we use the `strcpy()` function.

```
strcpy(slt , "Welcome" );
```

4.5. Displaying Strings

The `%c` format can be used to display the string character by character until we reach the `\0` symbol.

```
i←0
while str[i] ≠ '\0' do
   write (str[i])
   i←i+1
end while
```

```
for (i=0 ;str[i] != '\0' ;i++)
   printf("%c",str[i]) ;
```

The string can also be displayed directly using the %s format.

```
write(str)
```

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

4.6. Reading Strings

The string can be directly entered using the `%s` format and without using `&` before the string's name.

```
read(str)
```

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

To enter text containing spaces, in C, we use the `gets` instruction, defined in the `string.h` library, because `scanf` stops at the first space.

```
#include<string.h>
…
gets(slt) ;
```

4.7. Some String-Specific Functions

C supports a wide range of functions that deal with strings, which are defined in the `string.h` library. Some of them are:

`strcpy(s1, s2);`	Copies string `s2` into string `s1`.
`strcat(s1, s2);`	Appends string `s2` at the end of string `s1`.
`strlen(s1);`	Returns the length of string `s1`.
`strcmp(s1, s2);`	Returns 0 if `s1` and `s2` are identical; less than 0 if `s1` < `s2`; greater than 0 if `s1` > `s2`.

4.8. Examples

Example 1 :

An empty string str = "", which has a length of 0.

	0	1	2	3	4	5	6	…	28	29
str	\0							…		

The contents of the slots don't matter after `\0`, the string ends at the first `\0`. Thus, any string can be converted to an empty string by placing `str[0] = '\0'`.

Example 2 :

A string containing a single character, it is different from the character type. So, `"w"` ≠ `'w'` because `"w"` is an array.

	0	1	2	3	4	5	6	…	28	29
str	w	\0						…		

Example 3 :

Write a program that takes text, then converts uppercase letters to lowercase and lowercase letters to uppercase.

Algorithm	C

`algorithm inverse`
`var txt :string[200]`
`i : integer`
	`#include<stdio.h>`
	`#include<string.h>`
	`int main(){`

```
begin
 write(("enter text")
 read(txt)
 i←0
 while txt[i]≠'\0' do
   if txt[i]>='A' and txt[i]<='Z' then
    txt[i]=txt[i]+'a'-'A'
   else
    if (txt[i]>='a' and txt[i]<='z') then
      txt[i]=txt[i]-('a'-'A');
    end if
   end if
 end while
 write(txt)
end.
```

```
 char txt[200] ;
 int i ;
 printf("enter a text") ;
 gets(txt)
 for(i=0 ;txt[i] !='\0' ;i++)
  if (txt[i]>='A'&&txt[i]<='Z')
   txt[i]+='a'-'A' ;
  else
   if (txt[i]>='a'&&txt[i]<='z')
     txt[i]-='a'-'A' ;

 printf("%s",txt) ;
 return 0 ;
}
```