

# STRINGS

There is no special string type in C. A string of characters is treated as a one-dimensional array of characters.

## I. DECLARATION AND MEMORY ALLOCATION

### 1.1. Declaration of a string

When declaring a string, we must indicate the size of the memory space to reserve for its storage.

**Syntax**

**char StringName[size];**

The size indicates the number of characters that will be reserved for the variable StringName. If the size is omitted, it will be considered equal to 255 by default (which corresponds to the maximum size of a string).

**Examples**

```
char NAME[20];
char FIRST_NAME[20];
char SENTENCE[];
```

### 1.2. Memory Allocation

The internal representation of a string of characters is terminated by the '\0' symbol (called NULL) as a marker for the end of the string. Thus, for a text of N characters, we must provide for N+1 bytes.

The name of a string is a variable containing the address of the first character of the string.

**Example**

Let's take the string TXT containing the word "HELLO !". The memory representation could be as follows:

TXT :	'H'	'E'	'L'	'L'	'O'	' '	'!'	'\0'
	1E04	1E05	1E06	1E07	1E08	1E09	1E0A	1E0B

For an empty string, it contains only the end-of-string marker '\0'. The size of such a string is zero.

## II. STRING INITIALIZATION

When declaring a string of characters, we can initialize it by indicating the set of characters that will form the string. The string value must be placed between double quotes. The size of the string can be explicitly indicated to set the maximum size that the string can take. If the size is omitted, it will default to the number of characters in the initialization string plus 1 for the end-of-string marker.

**Syntaxes**

**char stringName[maxSize] = "value";**

**or**

**char stringName[ ] = "value";**

- If the fixed maxSize is greater than the length of the initialization string, the rest of the array remains unused.
- If the fixed maxSize is less than or equal to the length of the initialization string, the system generates an error.

### Examples

char TXT[ ] = "Hello";

TXT : 

'H'	'e'	'l'	'l'	'o'	'\0'
-----	-----	-----	-----	-----	------

char TXT[ 6 ] = "Hello";

TXT : 

'H'	'e'	'l'	'l'	'o'	'\0'
-----	-----	-----	-----	-----	------

char TXT[ 8 ] = "Hello";

TXT : 

'H'	'e'	'l'	'l'	'o'	'\0'		
-----	-----	-----	-----	-----	------	--	--

char TXT[ 5 ] = "Hello";      // runtime error

TXT : 

'H'	'e'	'l'	'l'	'o'	*
-----	-----	-----	-----	-----	---

char TXT[ 4 ] = "Hello";      // compilation error

## III. ACCESS TO A STRING OF CHARACTERS

Accessing a string of characters can be done in two ways:

### 3.1. Block access by the name of the string

Reading or writing the string is done by simply specifying the string name.

#### Syntaxes

**printf("%s", stringName);**  
**scanf("%s", stringName);**

#### Example 1

Write a C program that allows entering a character string *ch* with a *maximum* size of 20.

```
#include <stdio.h>
int main() {
    char ch[20];
    printf("Enter a string: ");
    scanf("%s", ch);
    return 0;
}
```

#### Example 2

Write a C program that displays a character string *ch* initialized to "Programming Course".

```
#include <stdio.h>
int main() {
    char ch[] = "Programming Course";
    printf("The value of the string is: ");
    printf("%s", ch);
    return 0;
}
```

### 3.2. Character-by-character access.

Accessing a character is done in the same way as accessing an element of an array. Indeed, to reference a character, you must mention the string name followed by the index of the character between brackets.

#### Syntax

**stringName[index]**

#### Example

Write a C program that displays the contents of a character string *ch* containing the value "Programming Course". The display must be done character by character.

```
#include <stdio.h>
int main() {
    char ch[] = "Programming Course";
    printf("The value of the string is: ");
    for(int i = 0; ch[i] != '\0'; i++)
        printf("%c", ch[i]);
    return 0;
}
```

## IV. STRING MANIPULATION LIBRARIES

### 4.1 The <stdio.h> library

The <stdio.h> library offers functions for input and output operations. In addition to the printf and scanf functions, we find the puts and gets functions, specially designed for writing and reading strings.

#### a. Display of character strings

The puts function does not require a display format like printf.

##### Syntax

**puts(string);**

##### Example

Using the same example from the previous page to display the contents of a character string ch using the puts function.

```
#include <stdio.h>
int main() {
    char ch[] = "Programming Course";
    printf("String display: ");
    puts(ch);
    return 0;
}
```

#### b. Reading character strings

The gets function does not require a display format like scanf.

##### Syntax

**gets(string);**

##### Example

Using the same example from the previous page to enter the contents of a character string ch using the gets function.

```
#include <stdio.h>
int main() {
    char ch[10];
    printf("Enter a string: ");
    gets(ch);
    return 0;
}
```

### 4.2 The <string.h> library

The <string.h> library provides a multitude of practical functions for processing strings as indicated in the following table:

Function	Description
strlen(ch)	Returns the length of the string ch excluding the final '\0'.
strcpy(ch1, ch2)	Copies ch2 to ch1.
strcat(ch1, ch2)	Appends ch2 to the end of ch1.
strcmp(ch1, ch2)	Compares ch1 and ch2 lexicographically, that is, compares the ASCII codes of the characters of the two strings one by one, and as soon as the first difference is found, the function provides a result: <ul style="list-style-type: none"> <li>- &lt;0 if ch1 precedes ch2,</li> <li>- 0 if ch1 follows ch2,</li> <li>- 0 if ch1 is equal to ch2.</li> </ul>
strncpy(ch1, ch2, n)	Copies at most the first n characters of ch2 to ch1.
strncat(ch1, ch2, n)	Appends at most the first n characters of ch2 to the end of ch1.

### 4.3 The <ctype.h> library

The functions of the <ctype.h> library are used to classify and convert characters. National symbols (é, è, ä, ü, ß, ç, ...) are not considered.

#### a. Classification functions

The following classification functions provide a result of type int different from zero if the respective condition is met, otherwise zero.

Function	Returns a non-zero value if
isupper(c)	c is an uppercase letter ('A'...'Z')
islower(c)	c is a lowercase letter ('a'...'z')
isdigit(c)	c is a decimal digit ('0'...'9')
isalpha(c)	islower(c) or isupper(c)
isalnum(c)	isalpha(c) or isdigit(c)
isxdigit(c)	c is a hexadecimal digit ('0'...'9' or 'A'...'F' or 'a'...'f')
isspace(c)	c is a space character (' ', '\t', '\n', '\r', '\f')

#### b. Conversion functions

The conversion functions below provide an int value corresponding to the ASCII code of the character to represent; the original value of the character c remains unchanged:

Function	Description
tolower(c)	Returns c converted to lowercase if c is an uppercase letter.
toupper(c)	Returns c converted to uppercase if c is a lowercase letter.

## V. ARRAYS OF CHARACTER STRINGS

### 5.1. Declaration

An array of character strings corresponds to a two-dimensional array of type char, where each row contains a string of characters.

#### Syntax

**char arrayName[N][C];**

Where N is the number of strings, and C is the number of characters per string.

**Example:** The statement "char DAY[7][10];" reserves memory space for 7 words containing 10 characters each (with 9 significant characters and the tenth corresponding to the '\0' character).

### 5.2. Initialization

When declaring, it is possible to initialize all the components of the array with constant character strings.

#### Syntax

**char arrayName[N][L]={string1, string2, ..., stringN};**

#### Example

char DAY[7][10]= {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};

DAY :

'M'	'o'	'n'	'd'	'a'	'y'	'\0'			
'T'	'u'	'e'	's'	'd'	'a'	'y'	'\0'		
'W'	'e'	'd'	'n'	'e'	's'	'd'	'a'	'y'	'\0'
....	....	....	....	....	....	....	....		....
'S'	'u'	'n'	'd'	'a'	'y'	'\0'			

### 5.3. Access to the different components

Access to the contents of an array of strings can involve an entire string or simply a character in a string.

#### a. Access to strings

It is possible to access the different character strings of an array by indicating the index of the corresponding row.

#### Syntax

**arrayName[index]**

#### Example

Write a program that initializes an array DAY containing the names of the days of the week, then enter an integer and display the corresponding day name.

```
#include<stdio.h>
int main() {
    char DAY[7][10]= {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
    "Saturday", "Sunday"};
    int i;
    do {
        printf("Enter a value between 1 and 7: ");
        scanf("%d", &i);
    } while (i < 1 || i > 7);
    printf("Today is %s!\n", DAY[i-1]);
    return 0;
}
```



**b. Access to characters**

It is possible to directly access the different characters that make up the words of the array by indicating the index of the string followed by the index of the character within the string in square brackets.

**Syntax**

**arrayName[stringIndex][characterIndex]**

**Example**

Write a program that initializes an array DAY containing the names of the days of the week, then display the second character of each day of the week.

```
#include<stdio.h>
int main() {
    char DAY[7][10]= {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday", "Sunday"};
    int i;
    for (i = 0; i < 7; i++)
        printf("%c\t", DAY[i][1]);
    return 0;
}
```

**Note**

Assigning a character string to a component of an array of strings cannot be done by a simple assignment because the component of the array represents an address while the string contains a value. This can be replaced by using the strcpy function from the <string.h> library.

**Example**

```
#include<stdio.h>
#include<string.h>
int main() {
    char DAY[7][10]= {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday", "Sunday"};
    int i;
    char ch[9] = "Vendredi";
    strcpy(DAY[4], ch);
    for (i = 0; i < 7; i++)
        printf("%s\n", DAY[i]);
    return 0;
}
```

# STRUCTURES

## I. DEFINITION OF A STRUCTURE

A structure is a user-defined type composed of a fixed number of elements called fields, each having a name and a type.

### Example

A date is a variable composed of three fields: day, month, and year.

## II. SYNTAX FOR DEFINITION AND DECLARATION

### a. Structure Definition

The definition of a structure involves:

- giving the name of the structure,
- listing the fields along with their types.

This definition can be local to the main function (i.e., placed inside it) or global (placed at the beginning of the program after the include statements).

### Syntax

```
typedef struct StructureName {
    BaseType1 FieldList1 ;
    BaseType2 FieldList2 ;
    ...
};
```

Where FieldList<sub>i</sub> contains a set of fields of the same BaseTyp<sub>e</sub><sub>i</sub> separated by commas.

### b. Structure Declaration

Similar to a function, a structure can be defined after the functions (at the end of the file). In this case, it must be declared before its use. This declaration can be local to the main function (i.e., placed inside it) or global (placed at the beginning of the program after the include statements).

### Syntax

```
typedef struct StructureName { } ;
```

This declaration states that there is a structure named StructureName that will be defined later in the program.

### c. Declaration of Structure Type Variables

Variables of structure type can be declared in one of three ways:

- First declaration method: Using the structure must be done through the declaration of variables of structure type that are already defined.

### Syntax

```
StructureName var1, var2, ..., varN ;
```

### Example

```
typedef struct DATE {
    int day ;
    int month ;
    int year ;
} ;
DATE birthDate, hireDate ;
```

- Second declaration method: Variables can be declared inside the block that forms the structure statement.

### Syntax

```
typedef struct StructureName {
    BaseType1 FieldList1 ;
    BaseType2 FieldList2 ;
    ...
} var1, var2, ..., varN ;
```

### Example

```
typedef struct DATE {
    int day ;
    int month ;
    int year ;
} birthDate, hireDate;
```

- Third declaration method: It is possible to declare and initialize a structure variable at the same time.

### Syntax

```
typedef struct StructureName {
    BaseType1 FieldList1 ;
    BaseType2 FieldList2 ;
    ...
} var1={ valField1, valField2, ...},
var2={ valField1, valField2, ...},
... ;
```

### Example

```
typedef struct DATE {
    int day ;
    int month ;
    int year ;
} birthDate={2, 11, 1980},
hireDate={12,9,2006};
```

## III. NESTING OF STRUCTURES

A structure can contain one or more fields of another structure type. In this case, the structure used for this field must be defined before defining the second structure.

### Example

Consider the PERSON structure defined by the ID, name, surname, date of birth, and hiring date. The two dates are of type DATE, which is defined by the day, month, and year numbers.

```
typedef struct DATE {
    int day ;
    int month ;
    int year ;
} ;
typedef struct PERSON {
    int ID ;
    char name[20] ;
    char surname[20] ;
    DATE birthDate ;
    DATE hireDate ;
} ;
```



## IV. ACCESSING A FIELD OF A STRUCTURE

Accessing a field of a structure is done using a period separating the structure variable name and the field to be accessed.

### Syntax

**structVarName.fieldName**

### Example 1

Suppose a variable p of type PERSON. The declaration statement for p is then as follows: "PERSON p ;". To manipulate the day field of the birth date of person p, you would use the following variable: "p.birthDate.day".

Using "p.birthDate" alone generates a compilation error.

### Example 2

Write a C program that allows using the DATE structure already defined to enter the birth and hiring dates of an employee in a company, then display them in standard date format.

```
#include<stdio.h>
struct DATE {
    int day ;
    int month ;
    int year ;
};
int main() {
    DATE hireDate, birthDate;
    printf ("Enter the birth day: ");
    scanf ("%d", &birthDate.day);
    printf ("Enter the birth month: ");
    scanf ("%d", &birthDate.month);
    printf ("Enter the birth year: ");
    scanf ("%d", &birthDate.year);
    printf ("Enter the hiring day: ");
    scanf ("%d", &hireDate.day);
    printf ("Enter the hiring month: ");
    scanf ("%d", &hireDate.month);
    printf ("Enter the hiring year: ");
    scanf ("%d", &hireDate.year);
    printf("The birth date is: %d/%d/%d\n", birthDate.day, birthDate.month, birthDate.year);
    printf("The hiring date is: %d/%d/%d\n", hireDate.day, hireDate.month, hireDate.year);
    return 0 ;
}
```

### Note

If the structure contains multiple levels of nested structures, then you need to manipulate the structure type variables by specifying the innermost field of the structures.

### Example

With the same example of the PERSON and DATE structures, we can manipulate the day, month, and year of birth of a person p.

```
#include<stdio.h>
typedef struct DATE {
    int day;
    int month;
    int year;
} DATE;
```

```
typedef struct PERSON {
    int mat;
    char nom[20];
    char pnom[20];
    DATE datNais;
    DATE datEmb;
} PERSON;
int main() {
    PERSON p;
    printf("Enter the birth date:\n");
    printf("Enter the day of birth: ");
    scanf("%d", &p.datNais.day);
    printf("Enter the month of birth: ");
    scanf("%d", &p.datNais.month);
    printf("Enter the year of birth: ");
    scanf("%d", &p.datNais.year);
    printf("\nBirth date: %d/%d/%d\n", p.datNais.day, p.datNais.month, p.datNais.year);
    return 0;
}
```

## V. ARRAYS OF STRUCTURES

### 5.1. Definition

For an array of structure type, each element of the array consists of the fields of the structure. Therefore, you need to define the structure first, and then the array.

#### Syntax

```
typedef structName arrayName[numElements];
```

Where:

- structName indicates the name of the structure,
- arrayName indicates the name of the array,
- numElements indicates the number of elements in the array.

#### Example

```
DATE T[20];
PERSON pers[10];
```

### 5.2. Access

Accessing an element of an array of structure type must specify the field to manipulate. This access must specify:

- the name of the array along with the index of the corresponding element,
- the access point,
- followed by the name of the field to be accessed.

#### Syntax

```
arrayName[index].fieldName
```

**Application:** Write a C program that allows entering an array of 20 people, then display the information of the people with the last name "Ben Salah".

# RECURSION

## I. Concept of recursion

Recursion is a concept that is frequently encountered in everyday life: arrays within arrays, nested dolls, etc.

In computer science, recursion can involve processing: A recursive process is one that calls itself (function/procedure).

**Examples:**

- **Definition of a mathematical function**  $f(x)=f(x-1)+4$   
 $f(0)=4$
- **Definition of a character string:** character + a string  
**a character:** letter|digit|empty string
- **Recursive definition of the factorial function:**  $N! = N \times (N-1) \times \dots \times 1$ 
  - (i)  $0! = 1$ .
  - (ii) For  $N$  a natural number ( $N > 0$ ),  $N! = N \times (N-1)!$ .

## II. Recursion concept

In any recursive definition, there must be:

- a certain number of cases whose resolution is known, these "simple cases" will form the termination cases of recursion, called the base of recursion;
- a way to reduce from a "complicated" case to a "simpler" case.

**Difficulties:** It must be ensured that one always falls back on a known case, that is, on a termination condition; it must then be ensured that the function is completely defined, that is, it is defined over its entire domain of application.

Example: The factorial function

The factorial function ( $n! = n \times (n-1) \times (n-2) \times \dots \times 1$ ) is usually defined as follows:

- (i) factorial ( $n$ ) = 1 for  $n = 0$ ;
- (ii) factorial ( $n$ ) =  $n \times$  factorial ( $n - 1$ ) for  $n > 0$ .

We obtain the following recursive function for calculating  $N!$  :

```
#include <stdio.h>
int fact(int n) {
    if (n == 0 || n == 1)
        return 1;
    else
        return n * fact(n - 1);
}

int main() {
    int num;
    do{
        printf("Enter a number: ");
        scanf("%d", &num);
    }while(num<0);
    int factorial = fact(num);
    printf("Factorial of %d is: %d\n", num, factorial);
}
```

### III. Types of Recursive Algorithms

#### 3.1. Direct Recursion

In the case of direct recursion, the recursive call of a function P occurs within the function P itself. This call can be made once (simple recursion) or multiple times (multiple recursion).

- **Simple Recursion**

```
FunctionP( )
{
    ...
    P // direct recursive call of P within the body of P
    ...
}
```

**Example:**

The power function  $f(x) = x^n$  can be recursively defined as follows:

$x^n = 1$  if  $n = 0$ ;

$x^n = x \times x^{n-1}$  if  $n \geq 1$ .

The corresponding function is written as follows:

```
int power(int base, int exponent) {
    // Base case: exponent is 0, result is 1
    if (exponent == 0)
        return 1;
    // Recursive case: exponent is positive
    else if (exponent > 0)
        return base * power(base, exponent - 1);
}
```

- **Multiple Recursion**

A recursive definition can contain more than one recursive call.

```
FunctionP( )
{
    ...
    P // direct recursive call of P within the body of P
    ...
    P // direct recursive call of P within the body of P
    ....
}
```

**Example:**

Here we want to compute the combinations  $C_n^p$  using the following relationship:

$$C_n^p = \begin{cases} 1 & \text{if } p = 0 \text{ ou } p = n; \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{else.} \end{cases}$$

```
int combinations(int n, int p) {
    if (p == 0 || p == n)
        return 1;
    else
        return combinations(n - 1, p - 1) + combinations(n - 1, p);
}
```

### 3.2.Indirect Recursion

Sometimes, it is possible for a function P to call a function Q, which in turn calls the function P. This is called indirect recursion.

<pre>FunctionP( ) { ... Q //indirect recursive call of P within the body of Q ... }</pre>	<pre>FunctionQ( ) { ... P //direct recursive call of Q within the body of P ... }</pre>
-------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------

The function P uses the function Q for its definition, which itself uses P to define itself.

**Example:** A simple example of indirect recursion is the recursive definition of even and odd numbers. A positive number n is even if n-1 is odd; a positive number n is odd if n-1 is even.

$$\text{even}(n) = \begin{cases} \text{True} & \text{if } n = 0; \\ \text{odd}(n-1) & \text{else;} \end{cases}$$

$$\text{odd}(n) = \begin{cases} \text{False} & \text{if } n = 0; \\ \text{even}(n-1) & \text{else;} \end{cases}$$

The corresponding algorithms are written as follows:

<pre>#include &lt;stdio.h&gt; bool Even(int n) {     bool L;     if (n == 0)         L = true;     else         L = Odd(n - 1);     return L; }</pre>	<pre>#include &lt;stdio.h&gt; bool Odd(int n) {     bool L;     if (n == 0)         L = false;     else         L = Even(n - 1);     return L; }</pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------

## IV. Functioning of a Recursive Algorithm

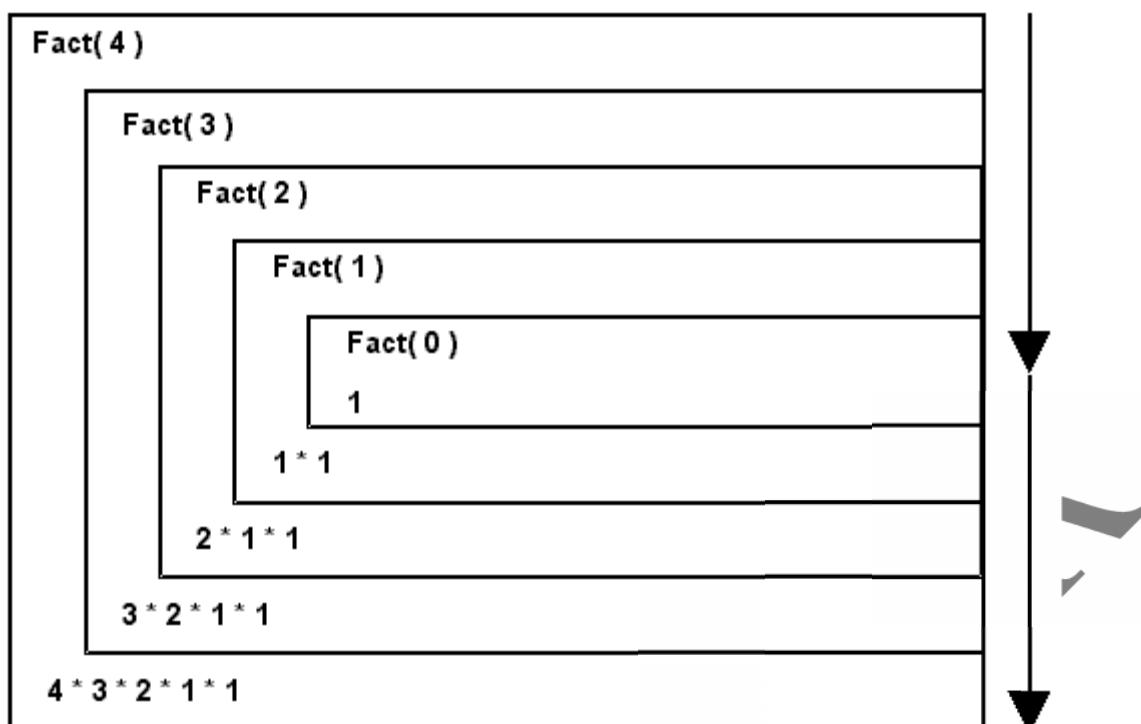
**Example 1:** The factorial function

Let's consider the factorial function Fact as defined in paragraph 2.

Upon calling:

$Y \leftarrow \text{Fact}(4)$

The execution of the Fact function can be described by the following figure:





# POINTERS

## I. DEFINITION

A pointer is a variable data that stores the address of another variable. It references another variable.

A pointer must always have the same type as the pointed variable. The declaration of a pointer uses the \* operator before the type.

### Syntax

```
baseType * ptrName;
```

Where:

- baseType represents the type of the pointed variable,
- \* indicates that it is a pointer,
- ptrName represents the name of the pointer variable.

### Example

If a pointer pv contains the address of a variable V, then we say that "pv points to V".

```
int v = 10;
```

```
int *pv;
```

### Schematic Representation



### Note :

A pointer without being assigned an address will be initialized by default to the NULL value.

## II. BASIC OPERATIONS ON POINTERS

### 2.1 The & Operator

The & operator is called "address of" and allows obtaining the address of a variable.

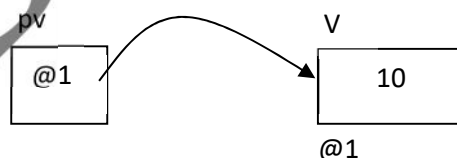
We can assign the address of a simple variable (obtained by the & operator) to a pointer variable. The type of the variable must be the same as that of the pointer variable.

### Example:

```
int v;
```

```
int *pv = &v;
```

the & indicates that the address of the variable v will be assigned to the pointer variable pv.



### 2.2 The \* Operator

The \* operator is called "content of" and allows obtaining the content of a pointer variable.

### Example:

```
main() {
    int v = 10;
    int *pv = &v;
    int w;
```

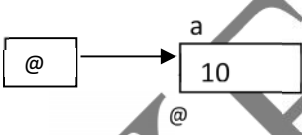
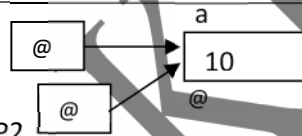
```
w = *pv; // w contains a copy of the value of the variable v
//pointed by pv
printf ("%d", w); // displays the value 10
return 0;
}
```

### 2.3 Assignment

It is possible to assign one pointer to another. This assignment will result in both pointers pointing to the same memory location.

#### Example:

Let p1 and p2 be two pointers of the same type. The statement "p2 = p1" copies the content of p1 (which is an address) to p2. p2 then points to the same variable pointed by p1.

Instructions	Effects on memory
int a=10; int *p1=&a ;	
int *p2 ; p2=p1 ;	

### 2.4 Increment / Decrement

The ++ operator allows incrementing the address contained in the pointer. The pointer will then have a new value, which is the incremented address.

#### Example:

Let's say a variable v exists at address 5C24. The pointer pv points to variable v, so it contains the address 5C24.

The statement "pv++;", allows pv to point to the element following variable v at address 5C28 (since the system advances by 4 bytes, as we work on 4-byte words).

pv will then have the value 5C28. v remains at the same location.

Similarly, the -- operator decrements the address contained in the pointer.

## III. ADDRESSING COMPONENTS OF A ONE-DIMENSIONAL ARRAY

### 3.1 Pointing to an Element of an Array

It is possible to manipulate the elements of an array through pointers. In this case, the address of the array element to be referenced must be placed in a pointer variable.

#### Syntax

```
ptrName = &arrayName[index];
```

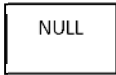
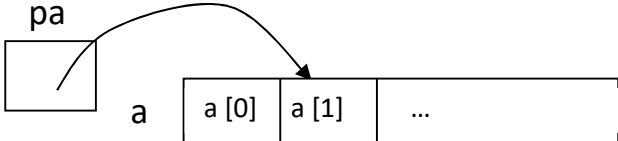
#### Example

The following declaration: "int a[10];" defines an array a of size 10, meaning a block of 10 consecutive objects named a[0], a[1], ..., a[9].

a :	a [0]	a [1]	a [2]	...	a [9]
-----	-------	-------	-------	-----	-------

If pa is a pointer to an integer declared as follows: "int \*pa;", then the assignment "pa = &a[1];" makes pa point to the second element of the array a.

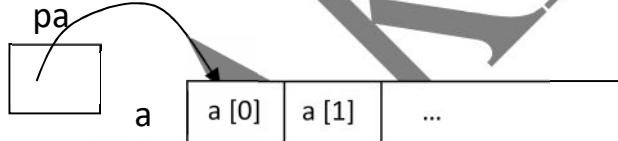
Instruction

Instructions	Memory State
<code>int *pa ;</code>	<p><b>pa</b></p> 
<code>pa = &amp;a[1];</code>	

To point to the beginning of an array, you can assign either the array name or the address of the first element to the pointer.

**Example:** For the same array `a` and the pointer `pa` from the previous example, you can use one of the following equivalent instructions:

- `"pa = a;"`
- `"pa = &a[0];"`

Instructions	Memory State
<code>int *pa ;</code> <code>pa=a;</code>	

### 3.2 Traversing the Elements of an Array

Traversing an array requires:

- pointing to the first element of the array using a pointer,
- then incrementing the pointer to move to the next element.

The incrementation can be repeated as many times as there are elements in the array.

#### Example 1

Write a function `saisie` that allows entering the elements of an array of 10 integers using pointers.

```
void Fill(int a[10]) {
    int *p1;
    for (p1 = a; p1 < a + 10; p1++)
        scanf("%d", p1);
}
```

#### Note

It is worth noting that entering a variable pointed to by a pointer `p` does not require the `&` operator. This is because `p` represents the address of the variable to be entered. Therefore, there is no need for the `&` operator.

#### Example 2

Write a C program that initializes an array of 10 elements with arbitrary values and copies the positive elements into a second array.

1- without using pointers.

```
main() {
    int T[10] = {-3, 4, 0, -7, 3, 8, 0, -1, 4, -9};
    int POS[10];
    int I,J;
```

```

for (J=0,I=0; I<10; I++)
if (T[I]>0) {
POS[J] = T[I];
J++;
}
return 0;
}

```

## 2- using the array name as a pointer.

```

main() {
int T[10] = {-3, 4, 0, -7, 3, 8, 0, -1, 4, -9} ;
int POS[10];
int I,J;
for (J=0,I=0; I<10; I++)
if (*(T+I)>0) {
*(POS+J) = *(T+I);
J++;
}
return 0;
}

```

With this solution, the array name acted as a pointer to the array. The notation  $*(T+I)$  indicates the content of the  $T[I]$  element. Since  $T$  always points to the beginning of the array, to access the element at index  $I$ , you must indicate it by  $T+I$ .

## 3- using a pointer initialized to the beginning of the array.

```

main() {
int T[10] = {-3, 4, 0, -7, 3, 8, 0, -1, 4, -9} ;
int POS[10];
int I,J;
int*p1,*p2 ;
for (p1=T,p2= POS ; p1<T+10 ; p1++)
if (*p1>0) {
*p2 = *p1;
p2++;
}
return 0;
}

```

With this solution, we used a pointer  $p1$  initialized to the beginning of array  $T$ , and used a pointer  $p2$  initialized to the beginning of array  $POS$ .  $p1$  advances with each processing of a case of  $T$ , but  $p2$  is only incremented if a positive value pointed to by  $p1$  is found. This value is then inserted into the  $POS$  array at the location pointed to by  $p2$ , and the latter is then incremented to prepare it for the next case.

**Application:** Write a C program that initializes an array of 10 elements with arbitrary values and counts the number of non-zero elements.

- without using pointers.
- using pointers.

## IV. POINTERS TO STRUCTURES

Pointers to structures are similar to pointers to simple variables.

### Syntax

```
struct structName *structPointer;
```

### Example

```
struct student {
    int id;
    char name[20];
    char surname[20];
    char course;
};
struct student *ptr;
```

ptr is a pointer to the structure type student.

Accessing the fields of a structure pointer is done using the "→" operator instead of ".".

### Syntax

structPointer→fieldName

**Example:** Write a C program that allows entering information about a student using a pointer to the student structure.

```
#include<stdio.h>
struct student {
    int id;
    char name[20];
    char surname[20];
    char course;
};
main() {
    struct student *ptr, s;
    ptr = &s;
    scanf("%d", &ptr →id);
    scanf("%s", ptr →name);
    scanf("%s", ptr →surname);
    scanf("%c", &ptr →course);
    return 0;
}
```

The use of the variable s is mandatory for the pointer to point to a real memory location. The declaration of s serves as memory allocation for the pointer to point to.

### Note

Entering fields of type integer (or character) requires the "&" operator before the variable name: "&ptr->id" and "&ptr->course".

However, entering fields of type string does not require this operator: ptr->name.

## V. DYNAMIC ALLOCATION

So far, memory allocation has been done automatically using data declarations. In all these cases, the number of bytes to be reserved was already known during compilation. This is called static variable declaration.

In many cases, we may need to work with data whose number and size cannot be predicted during programming. It would then be wasteful to always reserve the maximum foreseeable space. We need a way to manage memory during program execution.

### 5.1 Allocating Memory Space

The malloc function from the <stdlib.h> library helps us allocate or reserve memory during a program.

### Syntax

int malloc(N)

This function provides:

- an integer representing the address of a block of N free bytes in memory,

- or the value zero if there is not enough memory.

This integer contains the address of the allocated memory cell, but its value will not be considered as an address, but rather as a hexadecimal integer.

To make this result usable as an address, it must be converted into a pointer with the desired type using typecasting.

#### Syntax

```
type *pointerName = (type*) malloc(N);
```

The value of N can be a fixed value if the exact number of bytes to be reserved is known, otherwise, the sizeof function applied to the desired type can be used to get the size of the appropriate type from the system.

#### Example 1

```
int *p = (int*)malloc(sizeof(int));
```

#### Example 2

Write a C program that reserves memory for X values of type int; the value of X is read from the keyboard:

```
#include<stdio.h>
#include<stdlib.h>
main() {
    int *PNum;
    int X;
    printf("Enter the number of values:");
    scanf("%d", &X);
    PNum = (int*)malloc(X*sizeof(int));
    return 0;}
```

### 5.2 Freeing Memory Space

If we no longer need a block of memory that we have reserved using malloc, then we can release it using the free function from the <stdlib.h> library.

#### Syntax

```
free(pointerName);
```

This instruction releases the memory block designated by the pointer pointerName. It has no effect if the pointer is null.

#### Note

If a pointer has not been freed, it will be automatically freed at the end of the main execution.

## VI. PASSING BY ADDRESS FOR FUNCTIONS

The formal parameters of a function allow passing a copy of the values it will need during its execution.

The function can modify the values locally, but this will not affect the values of the corresponding actual parameters in the calling function. In this case, the function will only have one result to return. This is the principle of passing by value.

If we need to return multiple values, or if we need to modify the values of the actual parameters of the calling function, then we must use the principle of passing by address.

In this case, the function parameters must be used as addresses rather than values. The calling function then provides the address of its parameter so that the called function can work on the same memory location of the variable.

The function parameters will then be pointers.

**Example:** The swap function, which swaps two values was defined with parameters passed by value. It cannot modify the n and p parameters of the calling program since it only swaps copies of n and p.

The way to achieve the desired result is to make the calling program pass pointers to the values to be modified as parameters. #include< stdio.h>



```
main ( ) {  
    void swap (int* , int *);  
    int n = 10 , p = 20 ;  
    printf ( "Before call: %d %d \n", n, p);  
    swap ( &n, &p);  
    printf ( "After call: %d %d \n", n, p);  
    return 0 ;  
}  
// Definition of the swap function  
void swap (int *a, int *b) {  
    int c ;  
    printf ( "Start swapping: %d %d \n", *a, *b);  
    c = *a ; *a = *b ; *b = c ;  
    printf ( "End swapping: %d %d \n", *a, *b);  
}
```

BOUBAKER

# LINKED LISTS

## 1. Singly Linked Lists

### 1.1. Definition

It is a set of dynamic objects linked together by pointers. The list is defined by the pointer to the first object.

Singly linked lists are used to construct a data structure whose size is not known in advance.

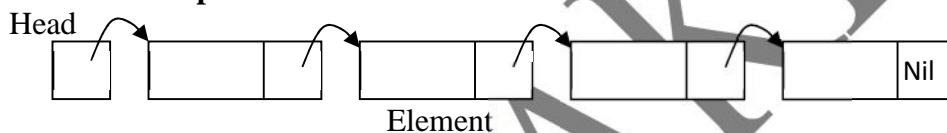
### 1.2. Characteristics

A list is characterized by:

- A head element that denotes the first element of the list.
- Every element of a list consists of two parts:
  - The first part contains one or more fields that form the information of the elements to be processed.
  - The second part contains a field allowing the link to the next element in the list. This field is a 'pointer'.

**Note:** The link field of the last element of the list contains Nil (Nothing In List) which signifies the end of the list.

### 1.3. Schematic Representation



### 1.4. Declaration

Syntax:

**TYPE**

```

ListItem = Structure
Information(s): base_type(s)
Next: ListItem

```

**End Structure**

**VAR**

```

Head: ↑ListItem

```

**where:**

<Information(s)> represents the set of information of an element.

<base\_type(s)> represents the basic types of all information, which can be simple or composite, predefined or custom.

Next is a pointer field that points to the next object.

**Example:** If we want to manage the information of a group of people whose names and ages are known. The necessary structure will then be as follows:

**TYPE**

```

Person = Structure
Name: string[10]
Age: integer
Next: Person
End Structure

```

**VAR**

```

P: Person

```

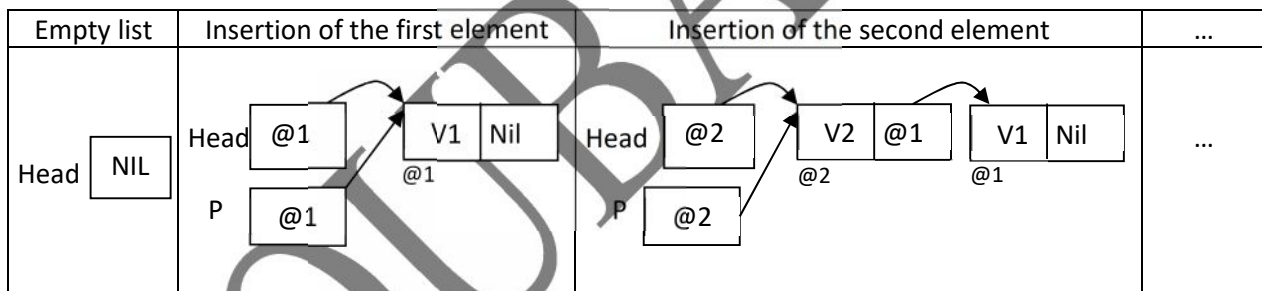
### 1.5. Manipulation Operations

#### ➤ Creating a singly linked list by adding at the head

The following algorithm creates a singly linked list of integers by adding elements at the head as many times as the user desires.

```

ALGORITHM create_head
TYPE
list_item = Structure
    Info: integer
    Next: list_item
End Structure
VAR
    Head, P: list_item
    Resp: char
BEGIN
    Head ← Nil
    Repeat
        Allocate (P)
        Read (P .Info)
        P .Next ← Head
        Head ← P
        Write ("Add another? Y/N")
        Read (Resp)
    Until (Resp = 'N')
END
    
```



➤ **Creating a singly linked list by adding at the tail**

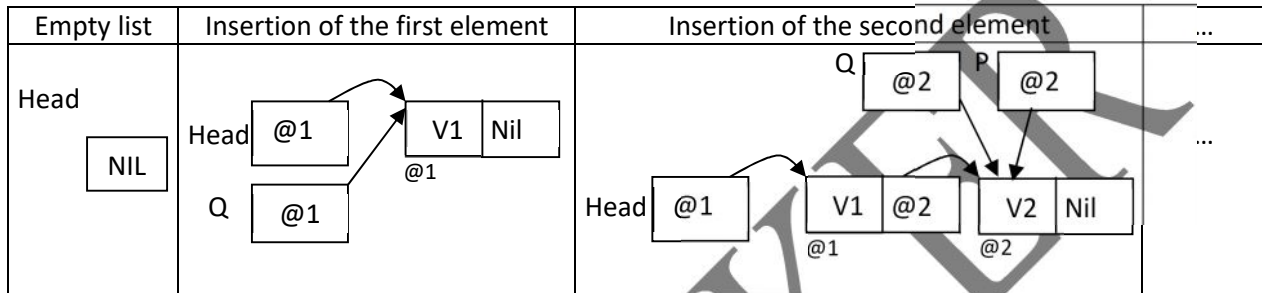
```

ALGORITHM create_tail
TYPE
list_item = Structure
    Info: integer
    Next: ↑list_item
End Structure
VAR
    Head, P, Q: list_item
    Resp: char
BEGIN
    Head ← Nil
    Q ← Head
    Repeat
        Allocate (P)
        Read (P .Info)
        P .Next ← Nil
    
```

```

    If (Head = Nil) Then
        Head  $\leftarrow$  P
    Else
        Q .Next  $\leftarrow$  P
    End If
    Q  $\leftarrow$  P // Q contains the address of the last element in the list
    Write ("Add another? Y/N")
    Read (Resp)
    Until (Resp = 'N')
END

```



### ➤ Traversing a List

If we have a singly linked list and we want to display all the information it contains, we must traverse the list element by element.

To move a pointer P (which points to a given element) to the next element, we must use the action  $P \leftarrow P .\text{Next}$ .

#### Application:

Write a procedure to display the elements of a singly linked list pointed to by a pointer Head.

**Procedure** Display\_List(**INPUT** Head:  $\uparrow$ list\_item)

**Var**

Ptr : list\_item

**Begin**

Ptr  $\leftarrow$  Head

**While** (Ptr  $\neq$  Nil) **do**

Write( Ptr $\uparrow$ .Info)

Ptr  $\leftarrow$  Ptr $\uparrow$ .Next

**End While**

**End**

### ➤ Adding to a List

#### 1. Adding to the Head of a List

If we want to add an element to the head of an already created list, its Next field must point to the first element of the list. The head must then change to point to the new element to be inserted.

**Application:** Write a procedure to add an element to the head of a list.

**Procedure** Add\_To\_Head(**IN** NewItem: integer ; **INOUT** Head: list\_item)

**Var**

P : list\_item

**Begin**

Allocate (P)

P .Info  $\leftarrow$  NewItem

```
P .Next ← Head
Head ← P
```

**End**

## 2. Adding to the Tail of a List

If we want to add an element to the tail of the already created list, we must proceed as follows:

- Create the element to add and set its Next field to Nil.
- Traverse the list and position ourselves on the last element.
- Perform the necessary chaining.

**Application:** Write a procedure to add an element to the tail of the list.

**Procedure** Add\_To\_Tail(IN NewItem: integer ; INOUT Head: list\_item)

**Var**

Ptr,P : list\_item

**Begin**

Allocate (P)

P .Info ← NewItem

P .Next ← Nil

If (Head = Nil) Then

Head ← P

Else

Ptr ← Head

While (Ptr .Next <> Nil) do

Ptr ← Ptr .Next

End While

Ptr .Next ← P

End If

**End**

## ➤ Deleting an element from a list

### 1. Deleting the head element

To delete the Head element of the list, we must use an intermediary pointer to store the address of the head, move the head pointer to the next element, and free the intermediary pointer.

**Application 1:** Write a procedure to delete the first element of a list

**Procedure** Delete\_From\_Head(INOUT Head: list\_item)

**Var**

P : list\_item

**Begin**

If (Head <> Nil) Then

P ← Head

Head ← Head↑.Next

Free(P)

End If

**End**

### 2. Deleting the last element

To delete the last element of the list, we must traverse the list until we locate the last element and its predecessor.

**Application 2:** Write a procedure to delete the last element of a list

**Procedure** Delete\_From\_Tail(INOUT Head: list\_item)

**Var**

BP, P: list\_item

**Begin**

```

If (Head < > Nil) Then
  If (Head .Next = Nil) Then
    P ← Head
    Head ← Nil
  Else
    BP ← Head
    P ← Head .Next
    While (P .Next < > Nil) do
      BP ← P
      P ← P .Next
    End While
    BP .Next ← Nil
  End If
  Free(P)
End If
End

```

## 2. Doubly Linked Lists

### 2.1. Definition

Sometimes, there is a need to traverse a list backwards or to provide the previous and next elements of a given element in a list. To achieve this, we can add a second pointer in each element of the list that will point to the element preceding a given element.

### 2.2. Declaration

**Syntax:**

**TYPE**

```

ListItem = Structure
  Previous: ↑ListItem
  Information(s): base_type(s)
  Next: ↑ListItem
End Structure

```

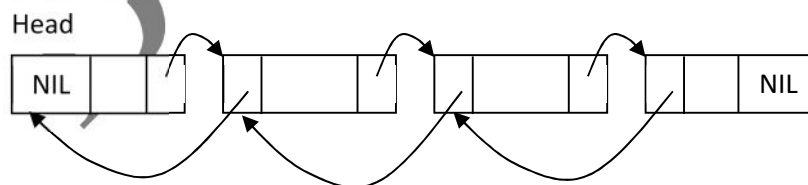
**VAR**

```

Head: ↑ListItem

```

### 2.3. Schematic Representation



**Application 1:** Write a procedure to add an element at the head of a doubly linked list.

**Procedure** Add\_To\_Head(IN NewItem: integer ; INOUT Head: list\_item)

**Var**

P : list\_item

**Begin**

Allocate (P)

P .Info ← NewItem



```

P .previous ← NIL
If(Head=NIL) then
    P .Next ← NIL
Else
    P .Next ← Head
    Head .previous ← P
End if
Head ← P
End
    
```

**Application 2:** Write a procedure to add an element at the tail of a doubly linked list.

**Application 3:** Write a procedure to add an element to a doubly linked list after an element pointed to by a pointer Q.

**Application 4:** Write a procedure to delete any element from a doubly linked list assumed to be created. The element to be deleted is pointed to by a pointer P.



### 3. Stacks

#### 3.1.Definition

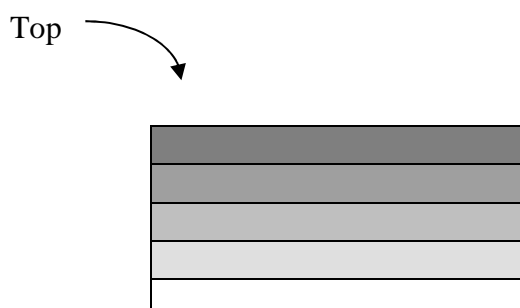
A stack is a collection formed of a variable, possibly zero, number of dynamic data elements. A stack is based on the principle of "last in, first out" (or LIFO for Last In, First Out), which means that the most recently added elements to the stack will be the first to be retrieved. The operation is similar to that of a stack of plates: plates are added to the stack, and they are retrieved in reverse order, starting with the last one added.

#### 3.2.Examples of stack usage

The concept of a stack is involved in many computer problems. For example:

- In a web browser, a stack is used to memorize the visited web pages. The address of each new visited page is stacked, and the user pops the address of the previous page by clicking the "Back" button. 
- The "Undo" function  in a word processor memorizes the changes made to the text in a stack.
- Particularly for recursive problems, an implicit call stack is used.

#### 3.3.Schematic representation of stacks



### 3.4.Stack Declaration

**TYPE**

**Stack\_Element = Structure**

**Info : Element\_Type**

**Next : Stack\_Element**

**End Structure**

**VAR**

**TOP : Stack\_Element**

### 3.5.Memory Representation of Stacks



### 3.6.Stack Manipulation Operations

A stack of objects is characterized by the following operations:

- **CREATE\_STACK**: creation of an empty stack,
- **STACK\_EMPTY**: test if the stack is empty,
- **PUSH**: adding an element to the stack,
- **POP**: removing an element from the stack,

**Example: Adding an element to a stack.**

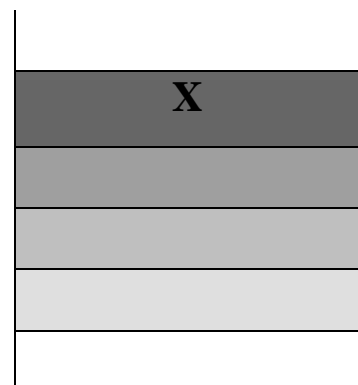
**Initial State**



**Action**

Push (X)

**Final State**



**Application:** Write the procedures and functions to perform:

- Creating a stack,
- Adding an element to the stack,
- Removing an element from the stack,
- Testing if the stack is empty or not,
- Calculating the number of elements in the stack.

## 4. Queues

### 4.1. Definition

A queue is a collection formed of a variable, possibly zero, number of dynamic data elements. A queue is based on the FIFO (First In, First Out) principle, which means that the first elements added to the queue will be the first to be retrieved. The operation is similar to a waiting line: the first people to arrive at a counter are the first to leave the queue.

## 4.2.Example of queue usage

Queues are applied in various computer science domains, including:

- The domain of operating systems and process control, where events can occur at any time and must be processed in their arrival order.
- Print servers, which must process requests in the order they arrive and insert them into a queue.

### 4.3.Queue Declaration

```

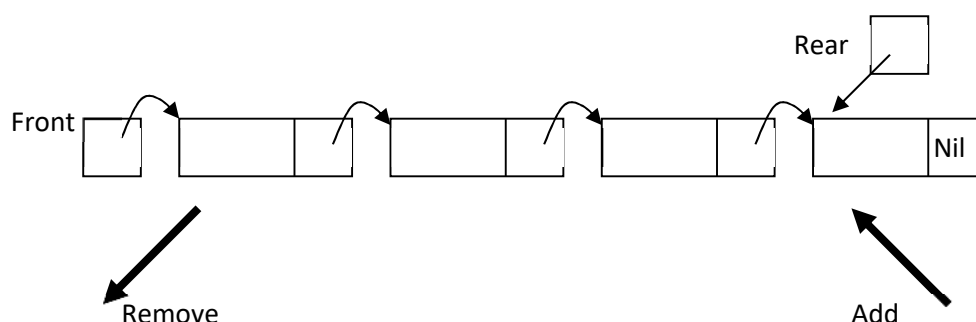
TYPE
    Queue_Element = Structure
        Info : Element_Type
        Next : ↑Queue_Element
    End Structure
    Queue = Structure
        Front, Rear : Queue_Element
    End Structure
VAR
    Q : Queue

```

#### 4.4.Schematic representation of a queue



#### 4.5.Memory representation of a queue



## 4.6.Queue Manipulation Operations

A queue of objects is characterized by the following operations:

- CREATE\_QUEUE: creation of an empty queue,
- QUEUE\_EMPTY: test if the queue is empty,
- ENQUEUE: adding an element to the queue,
- DEQUEUE: removing an element from the queue,

**Application: Write the procedures and functions to perform:**

- Creating a queue,
- Adding an element to the queue,
- Removing an element from the queue,
- Testing if the queue is empty or not,
- Calculating the number of elements in the queue.

## 5. Trees

### 5.1.Definition

A tree is a recursive structure organized into levels, where each element at a given level can be connected to several elements at the next level.

More precisely, a tree is a structure comprised of a root (called Root) and a finite (possibly empty) set of trees (called Subtrees).

For example, files and folders in a file system are typically organized in a tree-like fashion.

In a tree, two categories of elements are distinguished:

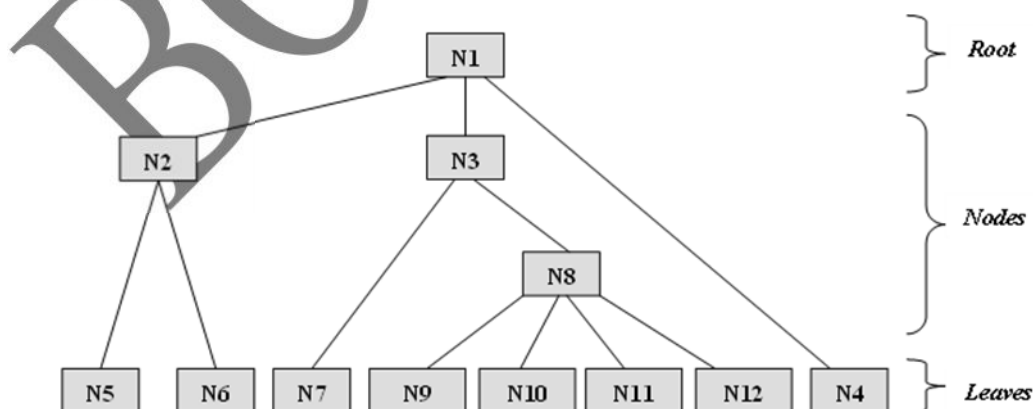
- Leaves (or external nodes), elements that do not have children in the tree;
- Internal nodes, elements that have children (sub-branches).

The root of the tree is the unique node that does not have a parent. Nodes (parents with their children) are connected to each other by an edge.

Two types of trees are distinguished:

- Binary trees (also known as search trees) in which each node has at most two children.
- N-ary trees, which are a generalization of binary trees: each node has at most n children.

### 5.2.Schematic representation of a tree



### 5.3.Binary Tree

#### Definition

A Binary Tree is a tree in which the maximum number of children a node can have is two.

#### Declaration

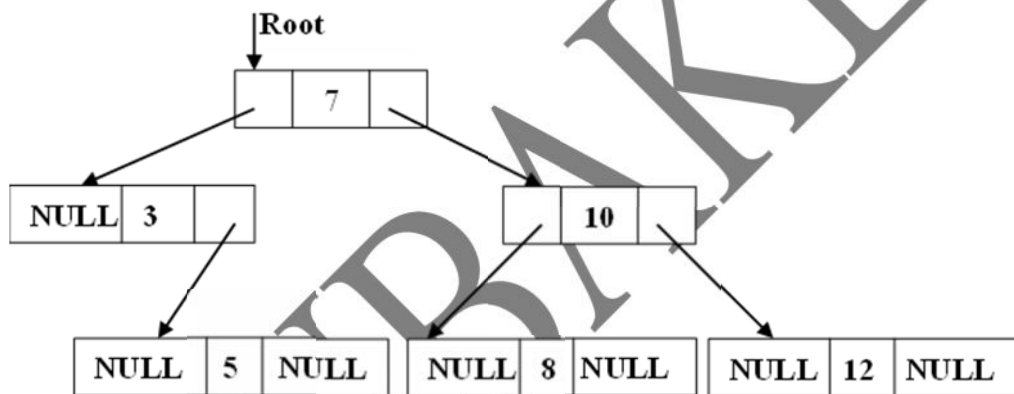
```
typedef struct noeud
{
    struct noeud *RightChild;
    int info;
    struct noeud *LeftChild;
};

struct noeud *R;
```

Where: RightChild is the pointer to the Right Subtree of a tree element.

LeftChild is the pointer to the Left Subtree of a tree element.

#### Schematic representation



RightChild and LeftChild can contain:

- NULL,
- or the address of another node.

#### Manipulation Operations : Traversal

This paragraph deals with traversing the vertices of a tree in the case where the only constraint is to be able to access the set (or a subset) of the tree's vertices.

Traversal can be done according to one of the following methods:

**Pre-order Traversal:** Traversing a tree according to this principle starts by processing the root, then moving on to process the elements of the left subtree (LeftChild), and finally those of the right subtree (RightChild). The processing of the elements of the left subtree (or right subtree) follows the same principle.

Example: The elements of the above tree can be displayed according to pre-order traversal. The result will be: 7 3 5 10 8 12

```
void prefixe(struct noeud *R)
{
    if(R)
    {
        printf("%d\t",R->info);
        prefixe(R->sag);
    }
}
```

```

        prefixe(R->sad);
    }
}

```

**Post-order Traversal:** Traversing a tree according to this principle starts by processing the left subtree (LeftChild), then the right subtree (RightChild), and finally the root.

Example: With the previous example, we will have the following display result: 5 3 8 12 10 7

```

void postfixe(struct noeud *R)
{
    if(R)
    {
        postfixe(R->sag);
        postfixe(R->sad);
        printf("%d\t",R->info);
    }
}

```

**In-order Traversal:** This principle processes the left subtree (LeftChild), then the root, and finally the right subtree (RightChild).

Example: With the same example, we will have: 3 5 7 8 10 12

```

void infixe(struct noeud *R)
{
    if(R)
    {
        infixe(R->sag);
        printf("%d\t",R->info);
        infixe(R->sad);
    }
}

```



## Lab : Strings

### Exercise 1

Write a program that reads a text TXT (less than 200 characters) and removes all occurrences of the character 'e' by compacting the remaining elements. The modifications will be made in the same variable TXT.

Example:

This line contains some letters e.

This lin contains som ltt rs .

### Exercise 2

Write a program that reads 5 words, separated by spaces, and then displays them in a line, but in reverse order. The words are stored in an array of strings.

Example:

Here is a small sentence

sentence small a is Here

### Exercise 3

Write a program that reads a string CH and converts all uppercase letters to lowercase and vice versa. The result will be stored in the same variable CH and displayed after conversion.

## Lab : Structures

### Exercise 1

Consider the following structure:

```
struct point {
    char c;
    int x, y;
};
```

Write a program that reads an array of 10 elements of type point. Display the coordinates of the points whose abscissa is 5 and those whose ordinate is 9.

### Exercise 2

Write a program that defines the DATE structure, inputs two variables of this type, and displays the smallest date.

### Exercise 3

Consider the exam structure:

```
struct exam {
    int MAT;
    char NAME[20];
    int COEF;
    char DEPARTMENT[20];
    int LEVEL;
    DATE DATE_EXAM;
    DATE DATE_EFF;
};
```

Write a C program that allows you to:

- Read an array of 10 exams.
- Determine the number of exams for the "COMPUTER SCIENCE" department.
- Display the exam schedule for level "2" of the "COMPUTER SCIENCE" department.

## Lab : Pointers

### Exercise 1

```
#include<stdio.h>
void main( ) {
    int A = 1, B = 2, C = 3, *P1, *P2;
    P1=&A;
    P2=&C;
    *P1=(*P2)++;
    P1=P2;
    P2=&B;
    *P1-=*P2;
    ++*P2;
    *P1*=*P2;
    A=++*P2**P1;
    P1=&A;
    *P2=*P1/=*P2;
}
```

Copy the following table and complete it for each instruction of the above program.

	A	B	C	P1	P2
Init.	1	2	3	/	/
P1=&A	1	2	3	&A	/
P2=&C	1	2	3	&A	&C
*P1=(*P2)++	3	2	4	&A	&C
P1=P2	3	2	4	&C	&C
P2=&B	3	2	4	&C	&B
*P1-=*P2	3	2	2	&C	&B
++*P2	3	3	2	&C	&B
*P1*=*P2	3	3	6	&C	&B
A=++*P2**P1	24	4	6	&C	&B
P1=&A	24	4	6	&A	&B
*P2=*P1/=*P2	6	6	6	&A	&B

### Exercise 2

Write a program that arranges the elements of an array A of type int in reverse order. The program will use pointers P1 and P2 and a numeric variable AUX for the permutation of elements.

### Exercise 3

Write a program that reads a string CH and determines the length of the string using a pointer P. The program will not use numeric variables.

## Lab : Recursion

### Exercise 1

Write a recursive function NBDigits that calculates the number of digits in a positive integer.

Example: for N = 4586: the result is 4

This function is called by the main function.

### Exercise 2

Write a recursive function SomDigits that calculates the sum of the digits of a positive integer.

Example: for N = 4586: the result is 23

This function is called by the main function.

### Exercise 3

Write a recursive function to reverse a given string of characters.

This function is called by the main function.

### Exercise 4

Write a recursive function to check if a given string of characters is a palindrome or not.

This function is called by the main function.

### Exercise 5

Write a recursive function that displays the elements of an integer array T of size N.

This function is called by the main function.

## Lab : Stack and Queue

### Exercise 1: Manipulation of a queue

```
#include "stdlib.h"
#include "stdio.h"
typedef struct Node {
    int value;
    struct Node* next;
} Node;
typedef struct Queue {
    Node* front;
    Node* rear;
} Queue;
Queue* create(Queue* queue) {
    queue = malloc(sizeof(Queue));
    queue->front = NULL;
    queue->rear = NULL;
    return queue;
}
int is_empty(Queue* queue) {
    return queue == NULL || queue->front == NULL && queue->rear == NULL;
}
Queue* enqueue(Queue* queue, int val) {
    if (queue == NULL) {
        return NULL;
    }
```

```

Node* new = malloc(sizeof(Node));
new->value = val;
new->next = NULL;

if (queue->front == NULL) {
    queue->front = new;
} else {
    queue->rear->next = new;
}
queue->rear = new;
return queue;
}
Queue* dequeue(Queue* queue, int* out) {
    if (is_empty(queue)) {
        return NULL;
    }
    Node* tmp = queue->front;
    if (queue->front == queue->rear) {
        queue->front = NULL;
        queue->rear = NULL;
    } else {
        queue->front = queue->front->next;
    }

    *out = tmp->value;
    free(tmp);
    return queue;
}
void display(Queue * queue) {
    if (is_empty(queue)) {
        printf("Queue is empty!\n");
    } else {
        Node * current = queue->front;
        while (current) {
            printf("Value: %d\n", current->value);
            current = current->next;
        }
    }
}

int count(Node* el) {
    if (el == NULL) {
        return 0;
    }

    return 1 + count(el->next);
}
int main() {
    Queue* queue = NULL;
    int x;

```

```

queue = create(queue);
queue = enqueue(queue, 1);
queue = enqueue(queue, 2);
queue = enqueue(queue, 3);
printf("Count: %d\n", count(queue->front));
display(queue);
printf("-----\n");
queue = dequeue(queue, &x);
printf("x: %d\n", x);
queue = dequeue(queue, &x);
printf("x: %d\n", x);
queue = dequeue(queue, &x);
printf("x: %d\n", x);
display(queue);
printf("Count: %d\n", count(queue->front));
return 0;
}

```

### Exercise 2: Manipulation of a stack

```

#include "stdlib.h"
#include "stdio.h"
typedef struct Element {
    int value;
    struct Element* next;
} Element;
Element * create() {
    return NULL;
}
int is_empty(Element* top) {
    return top == NULL;
}
Element * push(Element * top, int value) {
    Element* new = malloc(sizeof(Element));
    new->value = value;
    new->next = top;
    top = new;
    return top;
}
Element * pop(Element* top, int* out) {
    if (!is_empty(top)) {
        Element* p = top;
        *out = p->value;
        top = top->next;
        free(p);
        return top;
    }

    return NULL;
}
void display(Element * top) {
    Element * current = top;

```

```
    while (current) {
        printf("Value: %d\n", current->value);
        current = current->next;
    }
}

int count(Element * top) {
    if (top == NULL) {
        return 0;
    }

    return 1 + count(top->next);
}

int main() {
    Element* top = create();
    top = push(top, 1);
    top = push(top, 2);
    top = push(top, 3);
    display(top);
    printf("-----\n");
    int x;
    top = pop(top, &x);
    display(top);
    printf("Out: %d\n", x);
    printf("Stack is empty: %s\n", is_empty(top) ? "true" : "false");
    printf("Count: %d", count(top));
    return 0;
}
```