Ministry of Higher Education and Scientific Research

Hassiba Benbouali University of Chlef

Faculty of Exact Sciences and Computer Science

Algorithmic and data structures part 2

Algorithmic and data structure, part 2, 1st year MI

Author : Dr BELALIA AICHA

Academic Year :2025-2026

Table of Contents

GENERAL INTRODUCTION

This document represents the **second part of the module "Algorithmic and Data Structures"**, designed for **first-year students** in the *Mathematics and Computer Science* program, *Computer Science stream*. It serves as a **complement to the first part**, which was previously published for the first semester.

The present course has been prepared **after several years of teaching experience** in this module. This accumulated experience has provided valuable insights into the **actual levels, learning pace, and comprehension abilities** of beginner students. As a result, the content of this course has been carefully developed to suit their needs and to ensure a **smooth and progressive learning process**.

In line with this philosophy, the course follows the **same clear and straightforward pedagogical method** as in *Part 1*. Each chapter begins with an **introductory overview** supported by **real-world examples** that illustrate the main concepts. This is followed by the **general syntax and theoretical notions** of the newly introduced algorithmic topics, accompanied by **practical examples and explanations** in the **C programming language**.

The **theoretical materials** and **general syntaxes** are drawn from several **bibliographical references** cited at the end of this document. However, the **examples and exercises** included throughout the course are the result of **personal work and teaching experience**, aiming to make the concepts clearer and more accessible to students.

In this course, we will study the following chapters:

- **Chapter 01:** Sub-Algorithms (Procedures and Functions) and *Recursion*
- **Chapter 02:** *Files*
- **Chapter 03:** *Pointers , Linked Lists*, including the manipulation of different types of linked lists , *Stacks and Queues*

Chapter 01: Sub-Algorithms (Procedures and Functions)

I. Procedures and Functions

Introduction:

As we progress in the design of algorithms, they often become larger and more complex. In such cases, certain sequences of instructions may need to be repeated in different parts of the program. When an algorithm is written as a single continuous block, it becomes difficult to read, understand, and maintain.

To overcome this problem, we divide the algorithm into several smaller and more manageable parts called **sub-algorithms**.

Each sub-algorithm is written separately from the main body of the algorithm and can be **called** whenever necessary.

There are two main types of sub-algorithms: **procedures** and **functions**.

Benefits of Using Procedures and Functions

- Improve the **readability** and **organization** of programs.
- **Reduce redundancy** by avoiding repeated instructions.
- Facilitate **testing**, **debugging**, and **maintenance** of the code.

6. Declaration syntax of procedure and function [01-[02]-[06]-[07]

6.1 Procedure: Is a sub-algorithm designed to perform a specific task **without returning a value.** It allows us to structure an algorithm by dividing it into smaller, reusable blocks that improve clarity and reduce redundancy.

Syntax:

```
Procedure procedure_name (passing_mode parameter1: type, passing_mode parameter2: type,
                          …,  passing_mode parameterN: type)    // Procedure header
Var
   declaration of local variables
Begin
   instructions (body of the procedure)
End.
```

6.2 Function: is similar to a procedure, but it **returns a value** after performing a specific computation. It is mainly used when a result needs to be calculated and reused elsewhere in the algorithm.

Syntax

```
Function function_name (passing_mode parameter1: type, passing_mode parameter2: type,
                        …, passing_mode parameterN: type): return_type    // Function header
Var
   declaration of local variables
```

```
Begin
   instructions (body of the function)
   Return x
End.
```

7. Differences between function and procedure:|[6]

• The function returns one and only one value

• The procedure can return several values or none.

• has a return type, whereas a procedure does not.

• The call of a procedure in the main program is a separate instruction, whereas a function is called within an expression — for example, in an assignment, a condition (if, while), or a display instruction (write).

• Any **function** can be **converted** into a **procedure**, but the **reverse** is not always possible.

8. Examples:

Example 01:

We define a factorial procedure which calculates the factorial of a integer number N:|[4]

Procedure Factorial (Input /N: Integer);

VAR: i, Fact: Integer;

Begin

Fact ← 1 ;

For i from 1 to N

Fact ← Fact ∗ i;

Efor;

write("The factorial =",Fact);

End.

Example 02:

We define a factorial function which calculates the factorial of a natural number N:

Function Factorial (Input/N: Integer): **integer** // ***the type of the function exists in the function***

VAR: i, Fact: Integer;

Begin

Fact ← 1 ;

For i from 1 to N

Fact ← Fact ∗ i;

Endfor;

Return Fact // **the return value statement exists only inside of the function.**

End.

The sections presented above correspond to the declaration of procedures and functions. These components do not generate any output by themselves; they must be explicitly invoked within the main program to perform their defined tasks.

9. Procedure and Function Call

Syntax:

Algorithm name

Procedure1 PName(parameters: type);

Var local variables declaration;

Begin

….

End procedure;

Function name1F(parameters2:type): **type**;

Var: **local variables declaration;**

Begin

….

Return exp;

End function;

//Main algorithm //

Var global variables declaration;

Begin

Instructions

End of main algorithm

9.1 Global Variables [1]-[7]

Global variables are declared before the beginning of the main algorithm (and outside any procedure or function).

They are accessible from any part of the program — including within procedures and functions.

9.2 Local Variables

Local variables are declared inside a procedure, function, or the main algorithm.

They are only accessible within the block where they are declared (they do not exist outside it).

Example 03: Calling a Procedure and a Function in a Main Algorithm[5]- [6]- [7]

Problem:

Write a procedure that calculates the square of a number, and then call it from the main program.

Solution

```
Algorithm square
procedure square (N : integer) ;
begin
N← N*N ;
write(N) ;
end procedure
// principal algorithm //
Var A : integer ;
Begin
write (“Give an integer number ‘’) ;
read(A) ;
square(A) ;
End.
```

```
Algorithm square
Fonction square (N : integer) :integer ;
Begin
N← N*N ;
Return N ;
End
// principal algorithm //
Var A,C : integer ;
begin
write (‘’Give an integer number ‘’) ;
read(A) ;
C←square (A) ;
write(“the square of’’,A,’’is’’,C) ;
End.
```

Note:

- **N**: it is a ***formal parameter***; its value is not known when creating the procedure.
- ***A: is an actual parameter (also called effective parameter).***
- To execute a procedure within a program, it must simply be called.
- The procedure call is written by specifying the procedure name followed by the list of parameters, separated by commas.
- When a procedure is called, the program temporarily interrupts its normal execution, performs the instructions contained in the procedure, then returns to the calling program and continues with the next instruction.
- Program execution always starts from the main part of the algorithm or program.

Process of the previous example:

Execution process:

	A	N
0	#	#
1	5	#
2	5	5
3	5	25
4	5	25

1. The program displays the message: **“Give an integer number.”**
2. The user enters the value **5**.
3. The value of **A** is copied into the variable **N** when the procedure `Square(A)` is called.
4. Inside the procedure, **N** is modified (N ← N × N), so **N = 25**, while **A** in the main program remains unchanged (**A = 5**).
5. The procedure writes: **“The square of A is 25.”**
6. When the control returns to the main program, **A** still equals **5**.

In this case, the type of parameter passing is called *pass by value*.

10. Passing Paramètres[1]- [2]- [5]-

- The exchange of information between a procedure (or function) and the calling algorithm is carried out through **parameters**.
- There are two main types of parameter passing, which allow different forms of interaction:
    1. **Passing by value**
    2. **Passing by address (or by reference)**

10.1 Passing by Value

- In this type of parameter passing, the **formal parameter** receives **only a copy** of the value of the **actual (effective) parameter**.
- Any modification made to the formal parameter inside the procedure does **not affect** the value of the actual parameter in the main program.

```
Algorithm square
procedure square (N: integer, b: integer);
begin
b← N*N ;
end
//principal algorithm //
Var A, C: integer;
begin
c←0 ;
write (‘’Give an integer number ‘’) ;
read(A) ;
square (A, C);
write (“the square of “, A,” is” ,C) ;
End.
```

Sequences:
After read A : A=6, C=0 ;
Calling of procedure
N=A=6 , et b=C=0 ;
then b=6*6=36.
write (A,C)= A=6, C=0.

10.2 Passing by Address (or by Reference)

- In this type of parameter passing, the **formal parameter** receives the **memory address** of the actual (effective) parameter.
- Therefore, any modification made to the formal parameter **directly affects** the value of the actual parameter in the main program.

```
Algorithm square
procedure square (N : integer , var b :integer) ;
begin
b← N*N ;
end
//principal algorithm //
Var A,C : integer ;
begin
c←0 ;
write (‘’Give an integer number ‘’) ;
read(A) ;
square(A,C) ;
write(“The square of “,A,” is”,C) ;
End.
```

Sequences

after read A :

calling of procedure

write(A,C)= N=A=6 , et B=&C ;

*b=6*6=36,

.

11. Procedure and Function in C language [1]- [2]- [5]-

11.1 Declaration synatx

```
#include <library.h>                    // Include necessary libraries
type function_name(formal_parameters)
{
   // Instructions
   return variable;                     // Return the computed value
}
void procedure-name (formal parameters)
{
 instructions
}
int main()
{ declaration of  main variables
Type X;
   Instructions
// // Calling a function (used inside an expression or assignment)
 X=function-name (effectives parameters) ;
// Calling a procedure (treated as a standalone instruction)
procedure-name (effectives parameters) ;// calling procedure is an instruction
return 0;
}
```

Example04: Write a procedure that draws a square of the Stars, and a function that calculates the factorial of a number.

6.2 Passing by value

```
#include <stdio.h>
void etoile(int a)
{ int i,j;
 for(i=0;i<=a;i++)
   {for(j=0;j<=a;j++)
    printf("*");
    printf("\n");
} }
int factorielle(int x)
{ int i;
  int fact=1;
  for(i=1;i<=x;i++)
    fact=fact*i;
    return fact; }
int main()
{  int n;
 printf("n ? ");
 scanf("%d", &n);
 printf(" le factorielle de %d est %d  :",n, factorielle(n));
 printf (" \n the  values  of n and  of f are %d et %d ",n,f);
etoile(n);
 return 0; }
```

Memory state :

Main	Factorielle
n = 5	x=5 (copy the value 5 to x so n=x=5)
	fact= 120

calling functiion factorielle(n)
displays 120

6.3 Passing by address:

```
void factoriellead(int x,int *fact)
{ int i;   *fact=1;
  for(i=1;i<=x;i++)
    *fact=*fact*i;
    printf("%d",*fact); }
int main()
{  int n,f=0;// to use it in passing by value
 printf("n ? ");
 scanf("%d", &n);
 printf(" the  factorial  of %d is  :",n);
 factorielle(n,f);
 printf (" \n the  values  of n and f are %d and  %d ",n,f);
```

Memory state :

Main	Factoriellead
n = 5 ,f=0	x=5 fact =memory address of f (copy the adress of f to fact)
	*fact= 120 means that the value of the address in fact =120

calling procedure factoriellead(n,f)
displays f=120

```
printf(" \n passing by address\n");
printf(" the  factorial of %d is  :",n);
factoriellead(n,&f);
printf (" \n the  values  of n and f are %d and  %d ",n,f);
return 0; }
```

N, f are **global** variables they are visible throughout the algorithm.

I is a **local** variable, it is visible only inside the factorial function

Quick Comparaison : Value vs Address

Feature / Case	**Passing by Value**	**Passing by Address**
Original variable in main modified?	No	Yes √
Simple scalar variable (int, float, char)	Yes √	Usually unnecessary
Multiple results returned?	No	Yes √
Arrays, strings, large structures	No (copiy will be large)	Yes (efficient) √

Tableau: **Quick Comparaison, Value vs Address**

Function type	Usually returns the result	Often `void`, result via pointer
Use when	You **don't want to change** main variables	You **want to modify** main variables or **return multiple results**

12. Examples and exercises:

Example 05: Define a function that returns the **greater** of two numbers Different, then write the result , then write a C program that displays the greater and the lesser between two numbers.

Algorithm ex05

```
Variables A, B, M: real;
// Declaration of the Max function
function Max (X: real, Y: real): real
Begin
If (X > Y) then
return X
else
return Y
End if
EndFunction
// Main algorithm
Begin
Write ("Give the value of A")
read(A)
```

```
#include <stdio.h>
/* Sub-program (procedure) */
void max_min(int a, int b, int *max, int *min)
{
   if (a > b)
   {  *max = a;
      *min = b;
   }
   else
   { *max = b;
      *min = a;
   }
}
int main()
{   int x, y, mx, mn;
   printf("Enter two integers:\n");
   scanf("%d %d", &x, &y);
   /* Call the sub-program */
   max_min(x, y, &mx, &mn);
   printf("Maximum = %d\n", mx);
   printf("Minimum = %d\n", mn);
   return 0; }
```

write (“Give the value of B”)

read(B)

// Call the Max function

M ←Max(A,B);

write ("The greater of these two numbers is: ", M)

END.

Example 06 Write an algorithm which first allows you to enter an array of n integers (the size of the array must not not exceed 30 boxes). Subsequently display the number of prime integers in this table.

Reminder: a number is said to be prime if it is only divisible by 1 and by itself.

Solution

Algorithme ex06

Function divid(x: integer): integer ;

Var I, d: integer;

Begin

d←0;

For (i=1 to x step 1) do

If (x mod I = 0) then

d←d+1;

end if

return d;

end function

Function prime (y: integer): booléen;

Begin

If (y= 2 then)

Return true;

Else

Return false;

Endfunction

//Principal algorithm

Var t =array [30] of integer;

I,co : integer;

```
#include <stdio.h>
#include <stdbool.h>
int divid(int x)
{     int i, d=0;
    for(i=1;i<=x;i++)
    {  if(x%i==0)
        d=d+1;
    }
      return d;
}
bool prime(int n)
{ if (n==2)
    return true;
    else return false;
}
main()
{ int t[5],i,div,co=0;
 printf(" give 30 integer numbers \n");
 for (i=0; i<=2; i++)
    {scanf("%d",&t[i]);
      div=divid(t[i]);
      printf(" the number of the dividors is
%d\n",div);
      if (prime(div)==true)
    {   co=co+1;
       printf("%d\n",co);
    }
   }
    printf("there  are %d prime numbers",co);
}
```

Begin

Write (“give 30 integers \n”);

For (i= 0 to 29 step 1) do

Read (t[i]);

div←divid(t[i]);

if (prime(div) = true) then

co←co+1;

end if

endfor;

write (“there are “,co,” prime number \n”);

end.

Home Work: A c program that swap between two numbers.

```
#include <stdio.h>
// Sub-program (procedure) to swap two unsigned integers
void swap(unsigned int *a, unsigned int *b)
{    unsigned int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
int main()
{    unsigned int x, y;
   printf("Enter two unsigned integers:\n");
   scanf("%u %u", &x, &y);
   printf("Before swapping: x = %u, y = %u\n", x, y);
  /* Call the sub-program (passing by address) */
   swap(&x, &y);
   printf("After swapping:  x = %u, y = %u\n", x, y);
   return 0;
}
```

II. Recursion[8]- [9]- [5]-

Introduction:

Imagine two mirrors facing each other — the reflection seems to repeat endlessly. This idea of self-reference perfectly illustrates the concept of recursion. In algorithmics, recursion appears when a function calls itself in order to solve a problem.

1. Problem statement:

How can we solve a complex problem by dividing it into smaller subproblems that are identical to the original one, until we reach a simple case that can be solved directly?

2. **Solution:**

Recursion is a programming technique in which a function calls itself to process smaller parts of a problem. Each call brings the program closer to a **base case**, which stops the recursive process. This approach is widely used in many real-world applications, such as factorial computation, Fibonacci sequences, sorting algorithms (like quick sort and merge sort), tree searching, and even fractal image generation.

Thus, recursion provides an **elegant and intuitive way** to decompose complex problems, while strengthening the connection between mathematical reasoning and programming logic.

3. Declaration Syntax :

Procedure p1(parameters)

or

Function f1(parameters): type

Begin

if (terminal condition) then

Instruction;

else

Call f1 or p1;

End.

Example 07: Write a recursive function to calculate the factorial of a positive integer.

$$n! = n * (n - 1)!$$

```
// Declaration of the Factorial function (Fact)
Function Fact(n: integer): integer
Begin
If (n =1 )then
return 1
else
return (fact(n-1) * n)
End if
EndFunction
```

4. Recursion In C language: [8]- [5]- [6]-

In **C language**, there is **no separate concept of "procedure"** — everything is defined as a **function**. However, when a function does **not return any value**, it is often considered the equivalent of a **procedure** in algorithmic notation.

Recursive Functions and Procedures in C Language

A **procedure** (or **function**) is said to be **recursive** if it **calls itself** during execution.

4.1 Recursive Procedure (void function)

A procedure in C is written as a function with `void` as the return type (meaning it returns no value).

Syntax:

```
void procedure_name(parameters)
{
   if (terminal_condition)
   {
      // Base case instructions
   }
   else
   {
      // Recursive call
      procedure_name(arguments);
   }
}
```

4.2 Recursive Function (returns a value)

Syntax:

```
return_type function_name(parameters)
{
   if (terminal_condition)
   {
      // Base case: return a simple value
      return value;
   }
   else
   {
      // Recursive case: return expression including recursive call
      return function_name(arguments);
   }
}
```

Example: Factorial Calculation Using Recursion in C

```
#include <stdio.h>
// Recursive function to calculate factorial
int factorial(int n)
{
```

```
    if (n == 0 || n == 1)
        return 1;  // Base case: factorial of 0 or 1 is 1
    else
        return n * factorial(n - 1);  // Recursive call
}
int main()
{
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    printf("Factorial of %d = %d\n", number, factorial(number));
    return 0;
}
```

Example 08: what is displayed on the screen, if n=5

```
Procedure display1(n: integer);
Begin
If  (n>=1) then
Write (n) ;
Display 1(n-1) ;
Endif ;
End Proc;
```

5,4,3,2,1

```
Procedure display2( n : entier);
Begin
 if  (n>=1) then
    display2(n-1) ;
    write(n) ;
end if;
  EndProc;
```

1,2,3,4,5

Example 09: Write a recursive subalgorithm that calculates the quotient of the division of a by b (where a and b are two natural numbers).

Solution

```
Algorithmexo09
Function div(a: integer ; b: integer): integer;
Begin
If (a<b) then return a ;
Else
Return 1+div(a-b,b);
End
```

15/3= 15-3=12, 12-3=9,9-3=6,6-3=3,

Chapter 02 FILES

1. Introduction

When we run a program, the variables will be saved in memory, if we want to change the information of a student record for example, we have to run the program again, and this is not a reliable solution.

We must therefore save our information in a location separate from RAM: hard disk, flashdisk, etc., to be able to see it, modify it, at any time, we must therefore **create a file.**

2. Definition: [5]- [10]- [11]-[12]

A file is a set of data organized into access units called 'records' or 'articles' (all of the same type). It is always recorded on a medium external to the central memory (hard disk, USB key, floppy disk, etc.). This is why the information it contains is called non-volatile. The most common actions on files are:

- **Creation**: define the file type as well as its physical location.
- **Consultation**: use the articles in the file without modifying a single one.
- **Update**: add, modify or delete records or fields.

3. File types:

3.1 A text file: is made up of ASCII characters, organized into lines, each ending with an end-of-line control character.

3.2 Binary files: these are files made up of a series of bytes.

4. Types of file access

The type of access is the technique that the machine must follow to retrieve the information contained in a file. There are three types of file access

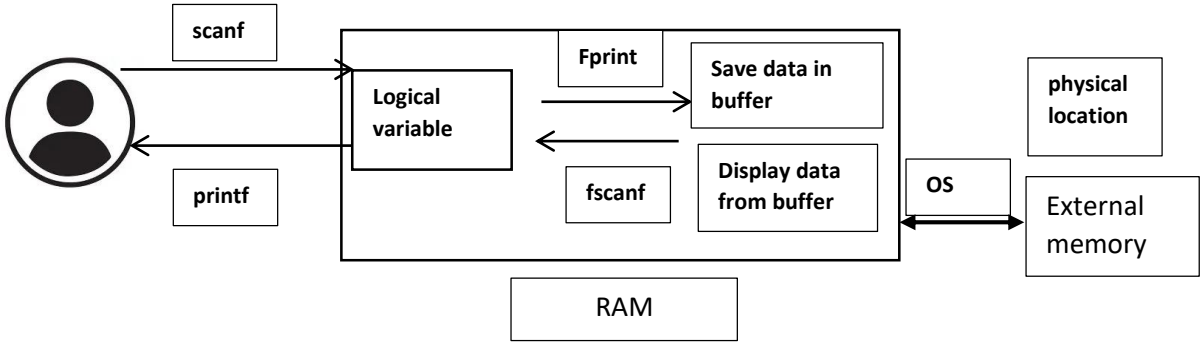
4.1 Sequential access: you cannot access the designated record unless you browse all the records that precede it.

4.2 Direct access: This type of access consists of going directly to the desired information without browsing those that precede it, specifying the position of the element sought.

4.3 Indexed access:

This type of access combines the speed of direct access and the simplicity of sequential access. It is particularly suitable for processing large files, such as databases.

Scheme explains the relation between RAM and External memory using files



5. **Declaration of a file type variable:**

Algorithm:

File file-name;

C language :

FILE *file-name; // file is the name of the variable.

Example:

Type Student = record

id : Integer

nom : String

prenom : String

age : Integer

average : Real

EndType

Variable S : Student;

FS : File

6. **Opening a file: [11]-[12]**

Algo :

file-name ← fopen (“ physical name in external memory.txt”, “ opening mode”);

In C language :

file-name =fopen(**char** *path, **char** *mode);

Mode is one char of:

R	Read mode	Open only for read
r+	Read/write	Open in r/w
W	Write	Open in writing , if the file doesn’t ‘t exist , it will be created, if it is exist , the content contenu will be deleted
w+	Read /write	*Idem a w +read*
A	Write	Open in writing for add in the end of file . if the file doesn’t ‘t exist , it will be created
a+	Read write	*Idem a « a » +r*

7. **Close files :**

Algo : fclose (file-name) ;

In C language: fclose (**file-name**);

Steps to use file: in both the two function to create file or to display data saved in any file we follow this steps:

- struct Declaration
- file daclartion a
- Open file :(fopen) (to read or to write or the two actions)
- Filling information using scanf asking user to enter data to RAM , then using fprintf to save data from RAM to file in un external memory (fprintf (file name, struct-name)

Or

- Reading data from file (in external memory to RAM) using fscanf , then from RAM to user using printf
- Finaly close the file

8. Example write an Algorithm and a C program asking the user for the information of unknown number of student, then save them in to File

Algorithm:

```
Type Student = record
   id : Integer
   name : C.C;
   age : Integer
   average : Real
EndType
Variable  S : Student;
          FS : File
//Open F in write mode ("students.txt")
FS← fopen (“students.txt”,”w”);
write (“ enter the ID of the first student \n”);
repeat
   read (S.id);
   if (S.id = 0 ) Then
  write (“ Opening ERROR \n”);
      Exit;        // return 1; any value different to zero
   EndIf
   read (S.nom, S.prenom, S.age, S.average);
// Now we must save the information of this student in file
   Fwrite(FS,S);
Until (S.ID = 0);
Close (FS);
```

Example :

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
   int id;
   char nom[50];
   char prenom[50];
   int age;

   float average;

} Etudiant;

int main() {

   FILE FS;

   Etudiant S;

   // 1     Open file in write mode

   FS = fopen("students.txt", "w");

   if (f == NULL) {

      printf("opening file Error.\n");

      return 1;

   }

   printf("Enter student information (id = 0 to stop)\n");

   do {

      printf("ID: ");

      scanf("%d", &S.id);

      if (e.id == 0)

         break;

      printf(" enter Name Age and Average ");

      scanf("%s %d %f ", S.nom, &S.age, &S.average);

      // 2  Write into file

      fprintf(FS, "%d %s %s %d %.2f\n", S.id, S.nom,S.age, S.average);

   }while (S.id !=0);

   // 3  Close file

   fclose(f);

   printf("Data saved successfully.\n");

   return 0; }
```

Exercise: Write a procedure allowing you to create and fill in the file (of type **Fi_employee**) which contains information on the employees of a company **(number, surname, first name, grade, salary).**

Algorithm

Decalartion **Type employee =record**

IDNumber:integer;

Surname:c.c;

Firstname:c.c;

Grade : c.c;

Salary : real;

Endrecord

Type fi-employee:file;

Procedure create(var fe:fi-employee),

Var em:employee;

Begin

fe←–fopen(“employeefile.txt”,”w”);//open the file in write mode

write(“give the id number of the first employee \n”);

read(em.idnumber);

while(idnumber <>0) do

write(“give the surname of the first employee \n”);

read(em. surname);

write(“give the firstname of the first employee \n”);

read(em. firstname);

write(“give the grade of the first employee \n”);

read(em.grade);

write(“give the salary of the first employee \n”);

read(em. salary); //at this step all the information are saved in RAM memory

fwrite(fe, em);//save this information in txt file.

write(“give the id number of the next employee \n”);

read(em.idnumber);

endwhile //

fclose(fe);

end.

C program:

#include <stdio.h>

```
#include <stdlib.h>
struct employee
    {int idnumber;
     char surname[30];
     char firstname [30];
     char grade [50];
     float salary;
     };
int main(void)
{    FILE *fiemployee;
    struct employee em;
    fiemployee = fopen("employeefile.txt", "w+");// in the next times we put a+ in place of w+
    if (fiemployee == NULL)
   {   printf("the file cannot be opened \n");
       return EXIT_FAILURE;
   }
    printf("the file  employeefile.txt existe\n");
    printf(" give the id number of the first employee \n");
    scanf("%d",&em.idnumber);
    while(em.idnumber!=0)
    { printf(" give the surname of the  employee \n");
      scanf("%s",em.surname);
      printf(" give the first of the  employee \n");
      scanf("%s",em.firstname);
      printf(" give the grade of the  employee \n");
      scanf("%s",em.grade);
      printf(" give the salary of the  employee \n");
      scanf("%f",&em.salary);
 fprintf(fiemployee,"\n  %d %s %s %s  %f
“,em.idnumber,em.surname,em.firstname,em.grade,em.salary);
      printf(" give the id number of the next employee \n");
      scanf("%d",&em.idnumber);
    }
    fclose(fiemployee);
   return 0;
}
```

8.1 Exercice : Write a procedure to display the list of employees from the **Fi_employee** type file.

Procedure display(var fe:fi-employee),

Var em:employee;

Begin

fe←fopen(“employeefile.txt”,”r”);//open the file in read mode

if (fe= NULL) then write (“ the file can’t be open \n”);

exit();

if (fread (fe, em.idnumber, em.surname, em.firstname, em.salary, em.grade)= 5) then;

write (em.idnumber, em.surname, em.firstname, em.salary, em.grade);

else

if (feof(fe)) then

write (“ End of file \n”);

else

write (“ error reading \n”);

endif

endwhile;

fclose(fe); **end.**

In C language

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
   int id;
   char name[50];
    int age;
   float average;
} Etudiant;

int main() {
   FILE *FS;
   Etudiant S;
   // Open file in read mode
   FS= fopen("students.txt", "r");
   if (FS== NULL) {
      printf("Error opening file.\n");
      return 1;
   }
   printf("Students list:\n\n");
   while (fscanf(f, "%d %s %s %d %f",
              &S.id, S.name, &S.age, &S.average) == 5) {

      printf("ID: %d | Name: %s %s | Age: %d | Avg: %.2f\n",
           S.id, S.name,S.prenom, S.age, S.average);
   }
   fclose(f);
   return 0; }
```

Note : *if we want create for each student a file :*

We declare ***char filename [50];*** *// a variable contains the file name*

For example :

char filename [50];

printf (“enter id of student”);

if (S.id !=0)

sprint(filename,”student %d .txt”,S.id); *// to creat file for student ID*

FS=fopen(filename,”w”);

So when the id changed , the file name will be changed

Chapter 03 part 01: Pointer

1. Introduction:

In C programming, **pointers** are one of the most powerful and essential features of the language. A **pointer** is a variable that **stores the memory address** of another variable rather than its actual value.

Pointers allow:

- Direct access to memory
- Efficient data manipulation
- Dynamic memory allocation
- Implementation of complex data structures

They are widely used with:

- Arrays
- Strings
- Functions (pass by address)
- Structures
- Dynamic data structures (like linked lists)

Understanding pointers is fundamental to mastering C programming, as they form the basis of **efficient data handling** and **memory management**.

2. Memory and Addresses

Variables are stored in memory locations, and each location has a unique address. It is the compiler that assigns these memory addresses automatically. For example:

```
int i, j;
i = 3;          // i   is Lvalue ‘’left value’’ for the assignment instruction.
j = i;
```

If the compiler placed the variable i at address 4831836000 in memory, and the variable j at address 4831836004, we have

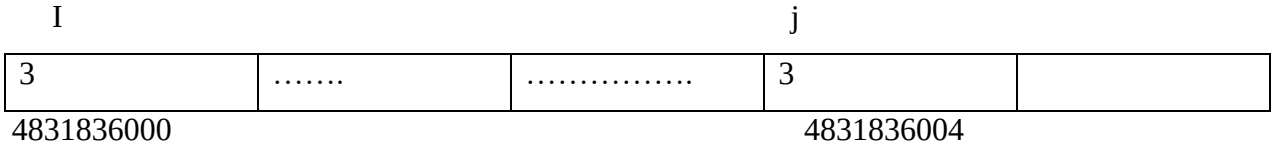
Lvalue	address	Value
I	4831836000	3
J	4831836004	3

A **memory address** is an **integer number** (typically represented on **16, 32, or 64 bits**, depending on the system).

To access the **address** of a variable in C, we use the **unary operator &** (address-of operator).

printf("The address of i = %p\n", &i);

printf("The address of j = %p\n", &j);

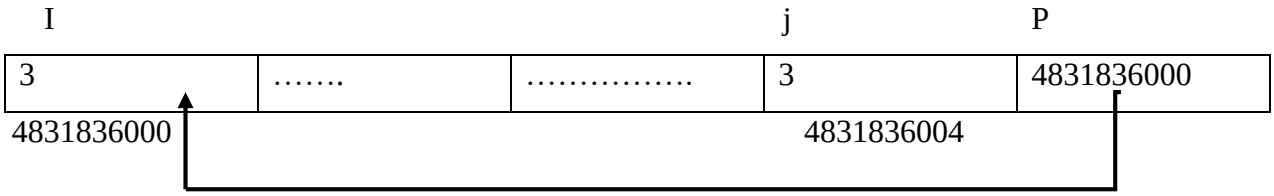


In direct addressing, access to the content of a variable is done via the name of the variable.

Write (I,j); printf("%d%d",I,j);

3. **A pointer** is a variable that holds the address of another variable in memory.

For example: if p is a pointer to a, so p contains the memory address of a:



&a : address of a

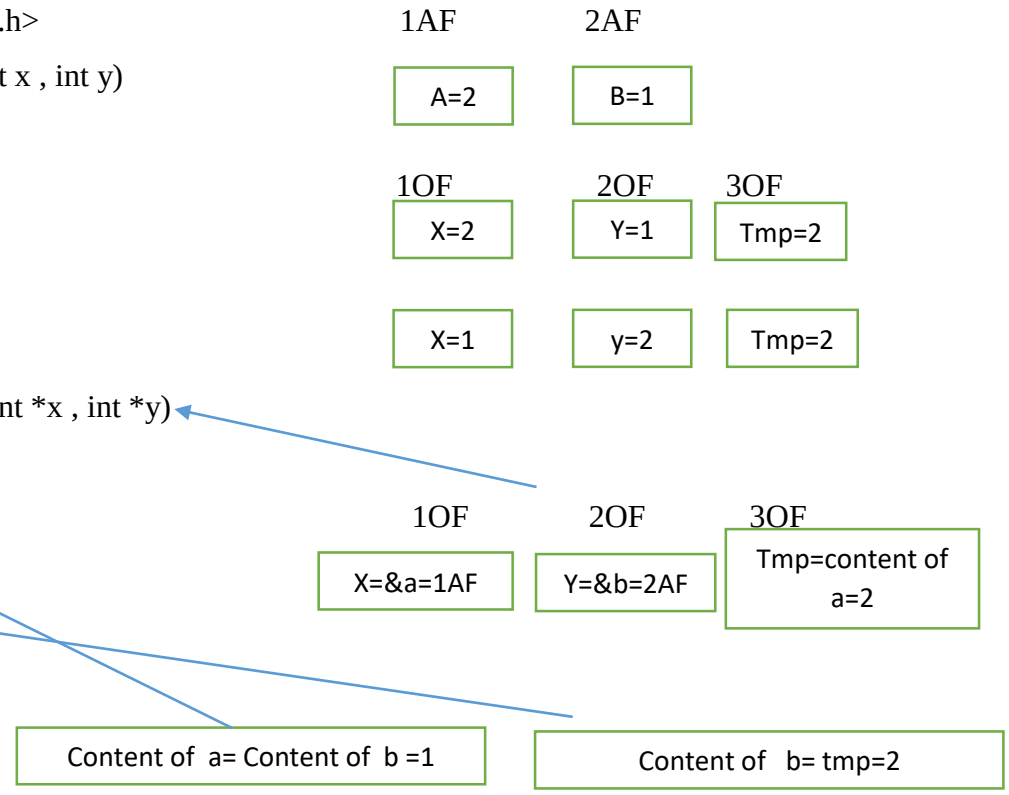
*p : the content of the address in p .

%p : for display the address.

4. **Examples**

Example 01: permutation between 2 numbers (1AF, 2AF, 1OF, 2OF, 3OF : memory address)

```
#include<stdio.h>
void permut(int x , int y)
{int tmp ;
 tmp=x ;
 x=y ;
 y=tmp ;
}
void permut2(int *x , int *y)
{int tmp ;
 tmp=*x ;
 *x=*y ;
 *y=tmp ;
}
main()
```



```
{ int a,b ;
 a=2 ; b=1 ;
 permut(a,b);
 printf("a =%d  and b= %d \n",a,b);
 printf(" after passage by address \n");
 permut2(&a,&b);// passing the address of a to the pointer x, and the address of b to the pointer y
 printf("a =%d  and b= %d \n",a,b); //a=1  and b=2
}
```

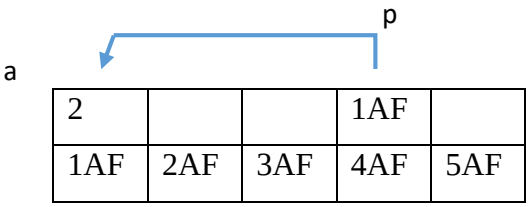
a=2 y=1

5. Declaration Syntax in c

int a ;
int *p;
p=&a;

or

int a ;
int *p=&a;



6. Exercises:

Exercise 01:

```
int a= 2;
int *p;
p=&a;
printf("%d\n",*p); //2
*p=85;   // a=85
printf("%d\n",a);// 85
printf("%d\n",*p);//85
printf("%p\n",&a);//1AF
printf("%p\n",p);//1AF
printf("%p\n",&p);//4AF
```

Exercise 02: complete the following table.

Instructions	A	B	C	P1	P2
Initialization	1	2	3	NULL	NULL
P1=&A	1	2	3	&a	NULL
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**

Exercise 03: complete the following result

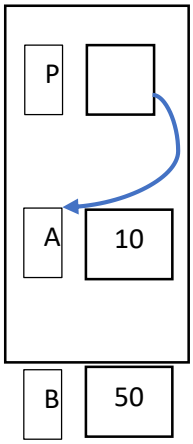
Int A=10, B=50;

Int *p;

P=&A// with color blue

B=*p// *p=10 , so B=10

*p=20;//A=20; and B=10



Chapter 03 part 02: Linked list

1. Introduction:

In C programming, a **linked list** is a **dynamic data structure** used to store a sequence of elements called **nodes**.

Unlike arrays, linked lists **do not require contiguous memory**, and each node contains **data** and a **pointer** to the next node in the sequence.

Linked lists are widely used in applications that require **efficient insertion, deletion, and memory management**, such as **dynamic memory allocation**, **stacks**, **queues**, and **graph representations**.

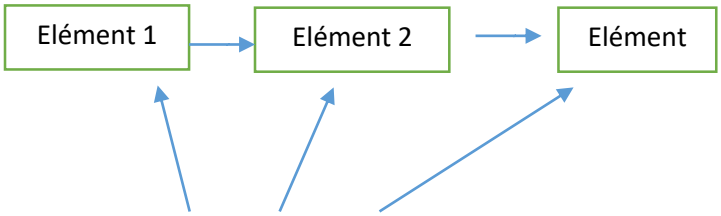
Understanding linked lists is fundamental for mastering **pointer manipulation** and **dynamic data structures** in C programming.

2. Definition: [13]- [14]- [5]-

A **linked list** is a sequence of objects of the same type that can be accessed **sequentially** from the first element to the last.

It is a **linear data structure** with no fixed size when created. Its elements are typically **scattered in memory** and connected to each other through **pointers**. The size of a linked list can **grow or shrink dynamically** depending on the available memory.

A linked list is always accessed through its **head** (i.e., the first element of the list), and traversal proceeds from the head to the last element via the pointers.



3. **Features:**
   - ✓ The elements in a list, called cells, nodes, or links, are not arranged next to each other.
   - ✓ For each element, we need to know the position (address) of the next element.
   - ✓ The first element in the list is called the head and the last element is called the tail.
   - ✓ We have a head pointer which contains the address of the first element of the list
   - ✓ The last node does not point to anything and its pointer needs to be filled with a NULL value.
4. **Type:**

✓ Single linked list

✓ Doubly linked list

✓ Circular linked list

5. **Elemental composition:**

address of elt

Information	**Next**

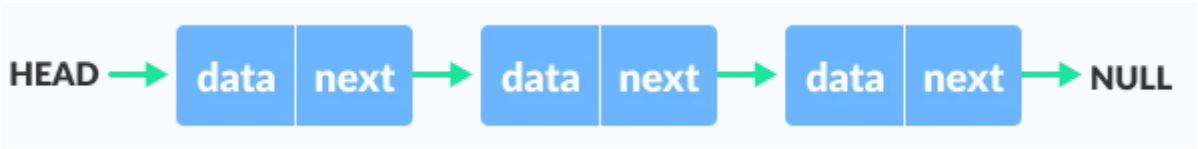
The elements of a list are sets (or structures) formed by:

✓ data or information,

✓ A pointer named **Next** indicating the position of the next element in the list.

Each element is assigned a memory address.

6. **Single linked list (one way) : [13]- [16]- [14]-**



6.1 Declaration Syntax :

```
Types  cells =record
        val: type          //value  to be  saved
        next: ^cells // address of the next  element
End  record    // set type of list items (definer les types des éléments
List= ^ cells  // set pointer type (définir le type pointeur )
Variable
  L: List   // declare a pointer variable
```

Allocate (L) // allocate a memory cell which reserves space in memory and gives P the value of the address of the memory space

6.2 Operations on linked lists: To allow processing to be applied to linked lists, two main functions must be used:

To insert a new element: **allocate (element)**

To delete an element: **free (element)**

Operations:

- ✓ Insert a new element in head
- ✓ Insert a new element after another element
- ✓ Insert new element in last
- ✓ Remove head
- ✓ Delete an item specifically
- ✓ Remove tail

6.3 In C language: [13]- [15]- [5]-

Node is represented as:

```
struct node {
   int data;
   struct node *next;
}
```

Example 01: Create a linked list composed of 2 string elements

6.4 Type declarations for the list:

```
Type List = ^Element ;
        Type Element = record
                Info: character string
                Next: List;
        End  record;
```

6.5 Algorithm and C program to Creat a List of 2Elements

Algorithm CreationList

```
Var Head, P: List
EltNumber: integer;
Begin
Tete ←NULL              /* for the moment the list is empty*/
Allocate(P)             /* reserves memory space for the first element */
```

Read(P^.Info) /* stores the entered value in the Info of the element pointed to by P */

P^.Next← NULL /* there is no next element */

Head← P /* the Head pointer now points to P */

/* We must now add the 2nd element, which amounts to inserting an element at the top of the list */

Allocate(P) /* reserves memory space for the second element */

Read(P^.Info) /* stores the entered value in the Info of the element pointed to by P */

P^.Next← Head /* element inserted at the head of the list */

Head← P

END.

7. Linked List in C language Syntax and exercises

To use linked lists in C, you must use certain functions:

The library:

- **stdlib.h:** declares various utility functions for type conversions, **memory allocation**, algorithms, and other similar use cases.
- The **sizeof(type)** The sizeof() operator in C computes the memory size, in bytes, of variables, pointers, arrays, or expressions.

Syntax : sizeof(data-type) or
sizeof(var-name) or
sizeof(exp) or
sizeof(ptr)

Example 02 display the information of function sizeof()

```
#include <stdio.h>
int main() {
   // We initialize the variable x
   int x = 56;
 // Memory occupied by x
   printf("Size of x is: %lu\n", sizeof(x));  //Size of x is: 4
 // The value of x
   printf("The value of x is: %d\n", x); // The value of x is: 56
printf("\n %d octets pour variable de type char ",sizeof(char));
      return 0;
}
```

- **malloc(....):** function in the liberray stdlib.h : The **“malloc”** or **“memory allocation”** method in C is used to dynamically allocate a single large block of memory with the specified size:

Syntax:

ptr = (cast-type*) malloc(byte-size)

For Example:

ptr = (int*) malloc(100 * sizeof(int));

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory

Example 03:linked list implementation in C

Implemented A two-member singly linked list can be created as:

Declaration syntax of one node (element)

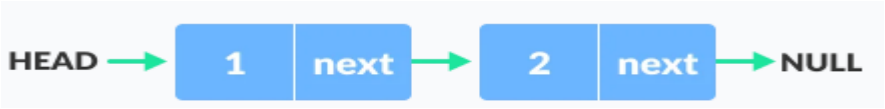
```
struct node
{
  int data;
  struct node *next;
};
```

#include <stdio.h> // library must be used in this case

#include <stdlib.h>

```
struct node
{  int data;
  struct node *next;
};
int main(){
/* Initialize nodes */
struct node *head;
struct node *elt1 = NULL;// in this case I will use only two element
struct node *elt2 = NULL;
struct node *p;      // element used for traversing list.
/* Allocate memory we must use the library    stdlib.h*/
elt1 = malloc(sizeof(struct node));
                if(elt1 == NULL)
                {   printf("Unable to allocate memory.");
                   exit(0);
                }
elt2 = malloc(sizeof(struct node));
                if(elt2 == NULL)
                {   printf("Unable to allocate memory.");
                   exit(0);    }
```

```
/* Assign data values */
elt1->data = 1;
elt2->data = 2;
/* Connect nodes */
elt1->next = elt2;
elt2->next = NULL;
```




```
/* Save address of first node in head */
head = elt1;
```

```
//printing node values
    p=head;
        printf("the values of this list  are: ");
        while(p!= NULL)
        {  printf("%d-> ",p->data);
           p=p->next;
        }
        if(p==NULL)
        printf("NULL");
        return 0;
        }
```

Result :

the value of this node is 1

the value of this node is 2

Example 04 : creating a linked list when the number of element is entered by user

```
 #include <stdio.h>
#include <stdlib.h>
/* Structure of a node */
struct node {
   int data;                // Data
   struct node *next;    // Address
}*head;
 * Functions to create and display list
 void createList(int n);  // prototype
void traverseList();      //
int main()
{    int n;
```

```
   printf("Enter the total number of nodes: ");
   scanf("%d", &n);
   createList(n);              // the procedure is the section 1
   printf("\nData in the list \n");
   traverseList();           //  // the procedure is the section 2
   return 0;
}
```

*** Section 1: Create a list of n nodes**

```
 void createList(int n)
{   struct node *newNode, *temp; // two nodes
   int val, i;
   head = (struct node *)malloc(sizeof(struct node)); // allocate memory area
   // Terminate if memory not allocated
   if(head == NULL)
   {   printf("Unable to allocate memory.");
      exit(0);
   }
   // Input data of node from the user
   printf("Enter the data of node 1: "); //insert data in the first node (head)
   scanf("%d", &val);
   head->data = val;        // Link data field with data
   head->next = NULL;     // Link address field to NULL
   // Create n - 1 nodes and add to list
   temp = head;
   for(i=2; i<=n; i++)
   {       newNode = (struct node *)malloc(sizeof(struct node));

      /* If memory is not allocated for newNode */
      if(newNode == NULL)
      {  printf("Unable to allocate memory.");
         break;
      }
      printf("Enter the data of node %d: ", i);
      scanf("%d", &val);
      newNode->data = val; // Link data field of newNode
```

```
            newNode->next = NULL; // Make sure new node points to NULL
            temp->next = newNode; // Link previous node with newNode
            temp = temp->next;    // Make current node as previous node
        }
    }
```

*** Section 2 Display entire list**

```
void traverseList()
{   struct node *temp;
    // Return if list is empty
    if(head == NULL)
    {   printf("List is empty.");
        return;
    }
    temp = head;
    while(temp != NULL)
    {  printf("Data = %d\n", temp->data); // Print data of current node
        temp = temp->next;              // Move to next node
    }
}
```

8. Operation in linked list:

8.3 Insert a new node:

8.3.1 insert a node at the head of a linked list

Variables we are using here:

head: A pointer of Node type pointing to the first node of the linked list.

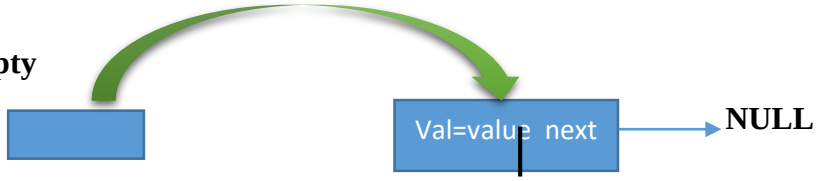
q: The node to be inserted (It will also be of Node type).

P: The node is used to traverse the list node by node, to arrive to the desired destination

- Create a new node with the given data.
- If head == NULL
- head = q
- Else
- q → next = head
- head = q.

Representation:

Case 01: list is empty



Head =NULL… **(1)** **Allocate memory for Q (2),**

Q.value (3)

Q.next = NULL (4)

head=q; (5)

Case 02: list is not empty; allocate memory for new node, filling data for new node

New node. Next = head, head = new node (change the head to be the first node)

Head

Algorithm:

Procedure inserhead(var l:*node , value: integer)

Var p,q:*node;

Begin

Allocate(q);

q->data←value ;

if(l=NULL) then

l←q;

else

q->next←l;

l←q;

end if

end procedure;

Algorithm display the element of the list:

Procedure display(l:*node);

Begin

While (l<> NULL) do

Write(l.data) ;

l←l.next ;

end while

end procedure .

```
void inserthead(int v)
{ struct node *p,*q;
   p = head;
   q = (struct node *)malloc(sizeof(struct node));
      /* If memory is not allocated for newNode */
      if(q == NULL)
      {  printf("Unable to allocate memory.");
         return;
      }
      q->data = v; // Link data field of newNode
      if(head==NULL)
      {head=q;
      q->next=NULL;
      }
      else

{q->next=p;
head=q;
}
}
```

8.3.2 insert a node at the end of a linked list

Algorithm :

Procedure insertail(var l:*node , value: integer)

```
Var p,q:*node;
Begin
Allocate(q);
q->data←value ;
if(l=NULL) then
l←q;
else
p←L;
while (p->next <>NULL) do
p←p->next;
end while
p->next←q;
q->next←NULL;
end if
end procedure;
```

C program to insert a new node

```
void inserttail (int value)
{struct node *p,*q;
   p = head;
   q = (struct node *)malloc(sizeof(struct node));
      /* If memory is not allocated for newNode */
      if(q == NULL)
      {  printf("Unable to allocate memory.");
         return;
      }
      q->data = value; // Link data field of newNode
      q->next = NULL; // Make sure new node points to NULL
      if(head==NULL)
      head=q;
      else
      { while (p->next!=NULL)
         p = p->next;
```

```
            p->next = q; // Link previous node with newNode
            p = p->next;    // Make current node as previous node


        }}
    /////
    int main()
    {    int n,v;
        printf("Enter the total number of nodes: ");
        scanf("%d", &n);
        createList(n);              // the procedure is the section 1
        printf("\nData in the list \n");
        traverseList();           //  // the procedure is the section 2
        printf(" give an other value \n");
        scanf("%d",&v);
        inserttail(v);
        traverseList();
        return 0;
    }
```

8.3.3 insert a node in a given position:

Nodes: head, p, q

Steps to insert a node after a given value

1. allocate a memory for the new node(q)
2. fill in the information in q
3. if the list is empty, ther is no insertion
4. if the list is not empty, p=head;
5. traverse the list until find the position or p.next =NULL)
6. q.next = p.next and p.next =q;

```
procedure insertpos(*l:node, v:intger ;  M: integer )
var *head,*p,*q: node;
Begin
if (head = NULL) then
return;
else
p=head
while(p.data<>m) and  (p.next<>NULL) do
```

p←p.next;

end while

Allocate (q);

q.data←v;

q->next=p->next;

p->next=q;

endif

end.

IN C LANGUAGE:

```
    void insertpos (int value, int m)
    {struct node *p,*q;
       p = head;
       q = (struct node *)malloc(sizeof(struct node));
          /* If memory is not allocated for newNode */
          if(q == NULL)
          {  printf("Unable to allocate memory.");
             return;
          }
          q->data = value; // Link data field of newNode
           if(head==NULL)
          printf(" the list is empty \n");
             else
          { while ((p->next!=NULL)&&( p->data!=m))do
            { p = p->next;
             }
          q->next = p->next; // Link the new node with the next node
          p->next = q;    // link the previous node with the new node
       }}/////
int main()
{    int n,x,m;
   printf("Enter the total number of nodes: ");
   scanf("%d", &n);
   createList(n);              // the procedure is the section 1
   printf("\nData in the list \n");
   traverseList();           //  // the procedure is the section 2
    printf(" give an other value \n");
```

```
    scanf("%d",&x);
printf(" give a value to insert after \n");
    scanf("%d",&m);
    insertpos(x,m);
    traverseList();
    return 0; }
```

8.4 Deletion of node from the linked ist:

8.2.1 Deletion of the Head

```
Procedure delethead(var head:*node);
var P: *node
P ← head
if head=NULL then
write (“deletion impossible );
else
head ← head->next
free(P)
endif
find
```

8.2.2 Deletion of the tail

```
Procedure delettail(var head:*node);
var P, q: *node
début
q ← head
P ← q-> next
while ( P->next <> null) do
q ← q->next
P ← P->next
end while
q->next ← Null
free(P)
end
```

8.2.2 Delete a node P from the list

```
procedure deletpos( var head, P: *node)
var q: *node
begin
if head <>null then
q ← head
while ( q->next  <>P )do
q ← q^.next
endwhile
q->next ← P->next
free(P)
end if
END.
```

```
Procedure delet(var l:*node, x:integer;)
Var p,q:*node;
Begin
p←l;
if(l=null) then
write(“deletion impossible”);
else
while (p->next<>null) and (p->data<>x) do
q←p;
p←p<>next;
end while
if(p->next= null) then write(“not found”);
else
q->next←p->next;
free(p);
endif
endif
end
```

Solution of exercise

9. Circular linked list :

9.1 Syntax:

```
type element : record
      data : type
      next : * element
end record
var head :*element
begin
head ←null;
```

9.2 Insertion a new node :

Algorithm:

9.2.1 Procedure inserhead(var l:*element , value: integer)

```
Var p,q:*node;
Begin
Allocate(q);
q->data←value ;
if(l=NULL) then
l←q;
q->next←l;
else
p←l ;
while (p.next<>l) do
p←p.next ;
endwhile
q->next←l;
l←q;
p->next←l;
end if
end procedure;
```

```
void inserthead(l:* struct element, int v)
{ struct element  *p,*q;
   p = l;
   q = (struct element *)malloc(sizeof(struct
lement));
      /* If memory is not allocated for
newNode */
      if(q == NULL)
      {  printf("Unable to allocate memory.");
         return;
      }
      q->data = v; // Link data field of
newNode
      if(l==NULL)
      {l=q;
      q->next=l;
      }
      else
      {   while (p.next!=l)    p=p->next;
          q->next=l;
          l=q;
          p->next=q;
      }
      }
```

9.2.2 Procedure addtail(Var L:element , X:integer) //

```
var q, p :*element ;
Begin
Allocate (q)
q->data←X
q->next←NULL ;
p←L
repeat
p←p->next
 until  (p->next = L)
p->next←q ;
q->next←L
end
```

9.3 Deletion a node from circular linked list (The head)

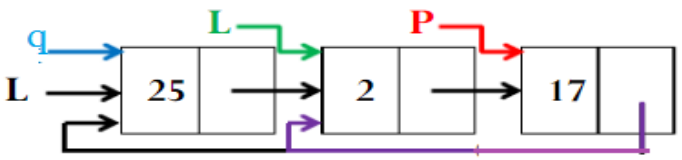
Delete a head from a list:

```
Procedure (var L: *element );
Var p,q: *element;
Begin
q←L;
p←L;
 repeat
P ←P->next ;
Until  (P->next = L)
L← q^.next
p^.next← L
free (q) ;
end.
```

10. Doubly linked list:

10.1 Declaration Syntax:

```
Type node = record
        Data : type
        Previous : *node;
        Next :*node
End record
```




```
Var head :
*node
Begin
head←NULL
```

10.2 Operation in linked list:

10.3 Insert a new node:

A. In head : steps

1. allocate memory for newnode
2. fill in the information
3. newnode->next = head;
4. newnode->previous= NULL
5. head->previous=newnode;
6. head = new node;

B. In tail: steps:

1. allocate memory for newnode
2. fill in the information
3. use a node p , to traverse the list until p.next = NULL
4. p->next=newnode
5. new node->previous= p:
6. newnode->next=NULL

C. In a given position :

Steps insert after a value

1. allocate memory for newnode
2. fill in the information
3. use a node p , to traverse the list until p.val=value
4. newnode->next=p->next
5. p->next->previous=newnode
6. newnode->previous=p;
7. p->next=newnode

10.4 Exercise:

Write a sub-algorithm, which takes as argument a doubly linked list of integers and displays the elements from the list in both directions.

```
Type element = record
        val: integer ;
        next:*element;
        previous: *element;
end
var head
begin head←NULL;
procedure display( var L:*element);
var p:*element;
begin
p←L;
```

```
while (p->next <> NULL ) do
write (p->val, ”->”);
p←p->next;
endwhile
while(p->previous<> NULL ) do
write (p->val,”<-“);
p←p->previous;
endwhile.
```

Exercise

Write a subroutine that takes as argument a doubly linked list of integers and an integer X, removes from the list all occurrences of elements that contain X.

```
Type element = record
            val: integer ;
             next:*element;
             previous: *element;
end
var head
begin head←NULL;
procedure delete( var L:*element,);
var p:*element;
begin
p←L;
if (p= NULL ) then
return;
else
while (p <> NULL ) do
if(p->val <>x) then
p←p->next;
else
q←p;
p←p->next;
p->previous←q->previous;
q->previous->next←q->next;
free(q );
endif
```

end while

end.

11. Special types of linked lists: Stacks and Queues

Introduction: Stacks and queues are abstract data types (ADT), not strictly linked lists. They define specific rules for data access (LIFO for stacks and FIFO for queues), but they can be implemented using different underlying data structures such as arrays or linked lists.
In this chapter, we present their implementation using arrays for simplicity, although linked list implementations are also possible.

1. **Real-world**

example:

Consider a stack of plates in a cafeteria. You **add plates on top** and **remove plates from the top** — the last plate added is the first one removed. This is a **stack** (LIFO: Last In, First Out). Now imagine a line of people waiting at a ticket counter: the **first person to arrive** is the **first to be served**. This is a **queue** (FIFO: First In, First Out).

2. **Problematic:**

How can we efficiently manage and organize data in such a way that **insertion and removal** follow specific rules (LIFO or FIFO), depending on the situation?

3. **Solution:**

Algorithmically, we use **stacks** and **queues** to model these scenarios.

- **Stack:** allows operations at one end only (push and pop), suitable for **function calls, undo operations, and expression evaluation**.
- **Queue:** allows insertion at the rear and removal at the front, ideal for **task scheduling, printer queues, and customer service systems**.

These structures make it easier to design **efficient algorithms** that mimic real-world processes and constraints.

4. Queues (Files) in Algorithmics [13]- [14]- [16]-

4.1 Definition

A **queue** is a **linear data structure** where elements are inserted at the **rear (end)** and removed from the **front (beginning)**.

It follows the **FIFO** principle: **First In, First Out**.

4.2 Declaration / Syntax in Algorithmic Pseudocode:

// Declaration of a queue

```
Queue Q;
Q.front ← 0
Q.rear ←1
// Operations
Enqueue(Q, element)  // add element at the rear
Dequeue(Q)           // remove element from the front
```

Explanation:

- **front:**

This variable **points to the first element** of the queue — the element that will be **removed next** when we perform a dequeue operation.

- Initially, when the queue is empty, front is usually set to 0.
- After removing elements, front moves forward to the next element.

 rear:

This variable **points to the last element** of the queue — the element that was most recently **added**.

- Initially, when the queue is empty, rear is usually set to -1.
- Each time we insert (enqueue) a new element, rear increases by 1 to indicate the new last element.

4.3 Queue Visualization

4.3.1 Empty queue:

Queue: [ ]
front → 0
rear → -1
There are no elements. rear = -1 indicates that the queue is empty.

4.3.2 After 1 insertion (A) :

Queue: [A]
front → 0
rear → 0
The first element is added at index 0. Both front and rear point to A.

4.3.3 After 3 insertions (A, B, C) :

Queue: [A | B | C]
front → 0
rear → 2
front stays at the first element (A), and rear points to the last added element (C).

4.3.4 After 1 removal (dequeue):

Queue: [B | C]
front → 1
rear → 2
A is removed. front moves to index 1 (B), rear remains at C.

4.3.5 After inserting a new element (D):

Queue: [B | C | D]
front → 1
rear → 3
D is added at the end. rear moves to point to the new element.

4.4 Example Exercise (Algorithmic Pseudocode) [15]-

Problem:

A printer queue can store up to 5 print jobs. Insert jobs A, B, C, then remove one job, and finally insert job D. Show the content of the queue after each operation.

Solution :

Queue Q
Q.front ← 0
Q.rear ← 1

// Insert jobs A, B, C
Enqueue(Q, 'A') // Queue: A
Enqueue(Q, 'B') // Queue: A B
Enqueue(Q, 'C') // Queue: A B C

// Remove one job
Dequeue(Q) // Queue: B C

// Insert job D
Enqueue(Q, 'D') // Queue: B C D
Resulting Queue: `B -> C -> D`

5 Queue Implementation in C Language

```
// Declaration
#include <stdio.h>
#define MAX 5
int queue[MAX];
int front = 0, rear = -1;
// Function to insert an element
void enqueue(int element) {
    if (rear == MAX - 1) {
        printf("Queue is full!\n");
    } else {
        rear++;
        queue[rear] = element;
    }
}
// Function to remove an element
int dequeue() {
    if (front > rear) {
        printf("Queue is empty!\n");
        return -1;
    } else {
        int element = queue[front];
        front++;
```

```
        return element;
    }
}

// Function to display the queue
void display() {
    if (front > rear) {
        printf("Queue is empty!\n");
    } else {
        for (int i = front; i <= rear; i++)
            printf("%c ", queue[i]);
        printf("\n");
    }
}

int main() {
    enqueue('A'); display(); // A
    enqueue('B'); display(); // A B
    enqueue('C'); display(); // A B C
    dequeue();    display(); // B C
    enqueue('D'); display(); // B C D
    return 0;
}
```

6 Stacks in Algorithmics

6.1 Definition

A stack is a linear data structure in which elements are inserted and removed from the same end, called the top. It follows the LIFO principle: Last In, First Out.

6.2 Declaration / Syntax in Algorithmic Pseudocode

```
// Declaration of a stack
Stack S;
S.top ← -1

// Operations
Push(S, element)   // add element on top
Pop(S)             // remove element from top
```

Exercise :

Solution 01 : « « « « « with queue » » » » »

```
#include <stdio.h>
#define MAX 5
int queue[MAX];
int front = 0;
int rear = -1;
// Function to add an element
void enqueue(int element) {
    if (rear == MAX - 1) {
        printf("Queue is full!\n");
    } else {
        rear++;
        queue[rear] = element;
    }
}
// Function to remove an element
int dequeue() {
```

```
    if (front > rear) {
        printf("Queue is empty!\n");
        return -1;
    } else {
        int removed = queue[front];
        front++;
        return removed;
    }
}
// Function to display the queue
void display() {
    if (front > rear) {
        printf("Queue is empty!\n");
    } else {
        printf("Queue content: ");
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}
int main() {
    // Step 1: Insert 10, 20, 30
    enqueue(10);
    enqueue(20);
    enqueue(30);
    // Step 2: Remove one element
    dequeue();
    // Step 3: Insert 40
    enqueue(40);
    // Step 4: Display final queue
    display();
    return 0;
}
```

Solution 02 : ***** With Stacks ******[5]- [7]- [15]-**

```
#include <stdio.h>
#define MAX 5
int stack[MAX];
int top = -1;
// Function to push an element
void push(int element) {
    if (top == MAX - 1) {
        printf("Stack is full!\n");
    } else {
        top++;
        stack[top] = element;
    }
}
// Function to pop an element
int pop() {
    if (top == -1) {
        printf("Stack is empty!\n");
        return -1;
```

```
    } else {
        int removed = stack[top];
        top--;
        return removed;
    }
}
// Function to display the stack
void display() {
    if (top == -1) {
        printf("Stack is empty!\n");
    } else {
        printf("Stack content: ");
        for (int i = 0; i <= top; i++) {
            printf("%d ", stack[i]);
        }
        printf("\n");
    }
}

int main() {
    // Step 1: Push 10, 20, 30
    push(10);
    push(20);
    push(30);
    // Step 2: Pop one element
    pop();  // removes 30
    // Step 3: Push 40
    push(40);
    // Step 4: Display final stack
    display();
    return 0;
}
```

GENERAL CONCLUSION

This document is the result of **eleven years of work and teaching experience** at Hasiba Ben Bouali University in Chlef, where I have taught both lectures and practical sessions. It has been designed to be **well-structured and accessible**, covering the fundamental concepts of programming and data management.

The polycopié includes **four main chapters**:

1. **Procedures and Functions** – to understand modularity and code reuse.
2. **Recursion** – to solve problems using successive function calls.
3. **Files, Pointers, and Linked Lists** – to handle dynamic data and efficiently store information.
4. **Queues and Stacks** – to manage insertion and deletion of elements according to specific rules (FIFO and LIFO).

Each chapter begins with a **real-life example**, showing how these concepts can be applied in everyday situations or practical computing problems. Then, it presents the **syntax in algorithmic pseudocode and in C language**, followed by **worked examples and exercises** to reinforce understanding and encourage self-learning.

This document is intended as a **complete pedagogical tool**. Every student can **learn the basic concepts effectively**, **but only if they follow the course step by step**, as each chapter builds on the previous one. It provides a structured and progressive approach to mastering essential programming concepts and data structures.

BIBLIOGRAPHY

[1] Berthet, D., & Labatut, V. (2014). *Algorithmique & programmation en langage C – vol. 1* (pp. 1–232). Istanbul, Turquie : Algorithmique et Programmation. Licence. https://hal.science/cel-01176119v1

[2] Coupey, P. (n.d.). *Travaux pratiques n°5 — Procédures et fonctions* [PDF]. Université Paris 13. http://www-info.iutv.univ-paris13.fr/~coupey/cours/C/cours5/tp5.pdf

[3] El OUKKAL, S. (2019). *Cours Algorithmique : Procédures & Fonctions.* Université Internationale de Casablanca.

[4] [02] Mezhoud, N. (n.d.). *Chapitre 4 : Les sous-programmes (Procédures et Fonctions).* Université Constantine 1, Module « Informatique 2

[5] Chaine youtube ,sefiane Info

[6] Chaine youtube, Hassan EL BAHI

[7] Chaine youtube Mohamed Chiny

[8] Lucidarme, P. (2022, 23 novembre). *Cours 13.1. Fonctions récursives en C*. Le blog de Lulu. https://lucidar.me/fr/c-class/lesson-13-01-recursive-functions-in-c/

[9] Hivert, F. (n.d.). *Algorithmique : Récursivité* [Cours en ligne]. Laboratoire de Recherche en Informatique (LRI). http://www.lri.fr/~hiver

[10] Chaine youtube : Algomius ,Programmation C : Accéder aux fichiers.

[11] Chaine youtube ; Ikode, Les Fichiers en Algorithmes

[12] Lucas-84, Taurre & informaticienzero. (2024, 25 août). Les fichiers (1) – Le langage C. Zeste de Savoir. https://zestedesavoir.com/tutoriels/755/le-langage-c-1/1043_aggregats-memoire-et-fichiers/4911_les-fichiers/

[13] Albuquerque, P., & Malaspinas, O. (2024, 9 décembre). *Applications des piles, listes chaînées et files d'attente* [Cours, Algorithmique et structures de données, ISC, HEPIA]. HEPIA.

[14] Pixees. (n.d.). *NSI Terminale – Structures de données : listes*. https://pixees.fr/informatiquelycee/n_site/nsi_term_structDo_liste.html

[15] OpenClassrooms. (n.d.). *Apprenez à programmer en C : Contrôlez l'ajout d'éléments avec les piles et les files*. https://openclassrooms.com/en/courses/19980-apprenez-a-programmer-en-c/19868-controlez-l-ajout-d-elements-avec-les-piles-et-les-files

[16] El-Kalam. (n.d.). *Algorithms and data structures : Piles et files*. https://www.el-kalam.com/cours/algorithms-and-data-structures/piles-et-files/