



الجمهورية الجزائرية الديمقراطية الشعبية
وزارة التعليم العالي والبحث العلمي
جامعة وهران للعلوم والتكنولوجيا محمد بوضياف
كلية الرياضيات والاعلام الالي
قسم الاعلام الالي

People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University of Science and Technology of Oran Mohamed BOUDIAF
Faculty of Mathematics and Computer Science
Department of Computer Science

**COURSE HANDOUT AND
SOLVED EXERCISES**

ALGORITHMS AND DATA STRUCTURES

Dr. YEDJOUR Hayat

Academic Year 2024-2025

Table of Contents

Foreword	1
Who is this course for?	1
Prerequisites	1
Chapter 1: Subprograms: Functions and Procedures	3
1. Introduction	4
2. Definitions	4
3. Local and global variables	7
4. Parameter passing	7
5. Recursion	12
Exercises	14
Solutions	17
Chapter 2: Files	36
1. Introduction	37
2. Types of files	37
3. File manipulation	38
4. Reading/Writing to a file	42
5. Moving within a file	51
Exercises	53
Solutions	55
Chapter 3: Linked Lists	70
1. Introduction	71
2. Comparison between arrays and Linked lists.....	71
3. Dynamic memory management	72
4. Singly Linked lists	76
5. Operations on linked lists	77
6. Variants of linked lists	80
Exercises	82
Solutions	83
Chapter 4: Stacks and Queues	92
1. Stack	93
• Applications of stacks	93
• Stack representations	94
• Basic stack operations	95
2. Queue	108
• Applications of queues	108
• Queue representations	108
• Basic queue operations	110
Exercises	114
Solutions	115
Conclusion	133
References	134

Foreword

Welcome to this course on algorithms and data structures with practical implementations using C++. This course material aims to provide a comprehensive and detailed introduction to both fundamental and advanced concepts in algorithms and the use of efficient data structures to solve complex problems. Algorithms form the core of computer science, defining the methods and processes by which we solve problems. Data structures, on the other hand, are the foundations on which these algorithms operate, offering ways to store and manage data optimally.

Course objectives

This course aims to equip students with the necessary skills to design, analyze, and implement efficient algorithms in C++. By the end of this course, you should be able to:

- Understand the fundamental principles of algorithms.
- Learn to analyze and evaluate the complexity of algorithms.
- Discover and use various data structures to enhance program efficiency.
- Develop programming skills in C++ by implementing various algorithms and data structures.

Who is this course for ?

This course is intended for first-year undergraduate and engineering students in the computer science department, as well as anyone wishing to deepen their knowledge of algorithms and data structures. A basic understanding of programming, particularly in C++, is recommended to get the most out of this course material.

Document structure

This course material is structured into several chapters, each covering a specific aspect of algorithms and data structures. Chapters 1 and 2 introduce basic concepts and provide simple examples to illustrate the ideas presented. Chapters 3 and 4 delve into more complex data structures and advanced algorithms with detailed implementations. Each chapter includes solved exercises in C programming to put the acquired knowledge into practice. This course material is divided into four parts:

1. Subprograms: functions and procedures
2. Files
3. Linked lists
4. Stacks and queues

Prerequisites

To get the most out of this course material, students should have a basic understanding of programming in C++, including variables, data types, loops, conditions, and functions. They should also have some knowledge of discrete mathematics and algebra, and be able to think algorithmically and logically.

CHAPTER 1

SUBPROGRAMS: FUNCTIONS AND PROCEDURES

1. Introduction :

A long program is often difficult to write and understand. Therefore, it is preferable to break it down into parts called **subprograms** or **modules**. For example, consider the following problem:

A teacher wants to determine the level of his class by calculating the class average, the lowest grade, and the highest grade. Write a program that displays the grades of a class in ascending order, followed by the lowest grade, the highest grade, and the average.

To solve this problem, we need to address the following subproblems:

- Enter the students' grades into an array.
- Display the students' grades.
- Find the smallest value in an array.
- Find the largest value in an array.
- Calculate the average of an array.

Each of these subproblems becomes a new problem to solve. If we can solve these subproblems, then the initial problem is almost solved.

Writing a program to solve a problem always involves writing subprograms that solve parts of the initial problem.

In algorithmics, there are two types of subprograms:

- Functions
- Procedures

Functions and **procedures** are independent modules (groups of instructions) identified by a name. They offer several advantages:

- Avoid rewriting the same process multiple times. Instead, we call the procedure or function at specified points.
- Organize the code and improve program readability.
- Facilitate code maintenance (modifying the code in one place is sufficient).
- These procedures and functions can potentially be reused in other programs.

2. Definitions

2.1 Functions :

In programming, a function is written outside the main program and its role is similar to a function in mathematics: it **returns a result** to the calling program.

The syntax is as follows:

```

Function function_name (parameters and their types) : function_type
Var // local variables
Begin
    Instructions constituting the function body
    return // the value to return
EndFunction

```

Where:

- **function_name** is an identifier.
- **function_type** is the type of the result returned.
- The **return** statement returns the result value.

2.1.1 Characteristics of functions :

- A function does not modify the values of its input arguments.
- It ends with a return statement that yields a result and only one result.
- A function is always used in an expression (assignment, display, etc.).

Examples of functions	
The following max function returns the larger of two real numbers x and y provided as arguments:	The pair function determines if a number is even:
Function max (x : real, y: real) : real var z : real; Begin z ← y; if (x > y) then z ← x endif; return (z); EndFunction	Function pair (n : integer) : boolean Begin return (n mod 2 = 0); EndFunction

2.1.2 Calling Functions:

Using a function is done by simply writing its name in the main program. The result being a value must be assigned or used in an expression. During a function call, the formal parameters are replaced by the actual parameters. For example, for the **max** and **pair** functions defined above, the call in the main program is as follows:

<p>Algorithm CallFunction var c : real, b : boolean; Begin b ← pair(3); c ← 5 * max(7, 2) + 1; write ("max(35*c+1) = ", max(35*c+1)); End</p>	<p>During the call pair(3), the formal parameter n is replaced by the actual parameter 3.</p>
---	---

2.2 Procedures :

When a task repeats in several places in the program and it does not calculate results or calculates multiple results at once, a procedure is used instead of a function. It is written outside the main program and its syntax is as follows:

```

Procedure procedure_name (parameters and their types)
Var // local variables
Begin
  Instructions constituting the procedure body
EndProcedure

```

2.2.1 Characteristics of procedures:

- A procedure is a subprogram similar to a function but **does not return anything**.
- A procedure may not have parameters.
- Unlike calling a function, we cannot assign the called procedure or use it in an expression.
- Calling a procedure is an autonomous instruction.

2.2.2 Calling procedures:

To call a procedure in the main program or another procedure, simply write an instruction indicating the name of the procedure.

```

Procedure ProcedureName (...)
  ...
EndProcedure

Algorithm CallProcedure
Begin
  ProcedureName (...)
  ...
End

```


3. Local and Global variables:

Two types of variables can be manipulated in a module (procedure or function): **local variables** and **global variables**. They differ in their scope (their "definition range," their "lifetime"). Generally, variables declared within a function or procedure are considered local variables, while global variables are declared in the main program.

- **A local variable** is only known inside the module where it is defined. It is created when the module is called and destroyed at the end of its execution.
- **A global variable** is known by all modules and the main program. It is defined for the entire application and can be used and modified by different modules of the program.

Advice: Use local variables as much as possible rather than global variables. This saves memory and ensures the independence of the procedure or function.

4. Parameter passing :

4.1 Formal and actual parameters :

- Parameters are used to exchange data between the main program (or calling procedure) and the called procedure.
- Parameters placed in the declaration of a procedure are called **formal parameters**. They are local variables to the procedure.
- Parameters placed in the call of a procedure are called **actual parameters**.
- The list of actual parameters must have the same number, order, and type of values as the list of formal parameters in the definition of the called procedure.

4.2 Different ways of passing parameters :

Passing parameters for functions is done either by **value** or by **address**:

1. **By Value:** The value of the actual parameter is copied to the formal parameter at the beginning of the procedure's execution. Changes to the formal parameter do not affect the actual parameter.

2. **By Reference:** The actual parameter and the formal parameter share the same memory location. Changes to the formal parameter directly affect the actual parameter.

Examples of parameter Passing:

- **by value example:**

```
Procedure Increment(x : integer)
Begin
  x ← x + 1;
  write(x);
EndProcedure
```

```
Algorithm TestByValue
var a : integer ;
Begin
  a ← 5 ;
  Increment(a) ;
  write(a) ; // Outputs : 5
End
```

In this example, x is a copy of a . Modifying x does not affect a .

- **by reference example :**

```
Procedure Increment(var x : integer)
Begin
  x ← x + 1 ;
  write(x) ;
EndProcedure
```

```
Algorithm TestByReference
var a : integer;
Begin
  a ← 5;
  Increment(a);
  write(a); // Outputs: 6
End
```

In this example, x and a are the same memory location. Modifying x affects a .

Example 1:

Write a procedure that increments an integer and the calling program.

```
Procedure increment (x: integer by value, y: integer by reference)
  x ← x + 1
  y ← y + 1
EndProcedure
```

```
Algorithm call_increment
Var n, m: integer
Begin
  n ← 3
  m ← 3
  increment(n, m)
  write ("n = ", n, " and m = ", m)
End
```

The result at the end: n = 3 and m = 4

Remark: The instruction $x \leftarrow x + 1$ has no effect when using pass by value.

Example 2:

Write a procedure that swaps two integers.

```
Procedure Swap (x: real by reference, y: real by reference)
Var t: real
  t ← x
  x ← y
  y ← t
EndProcedure
```

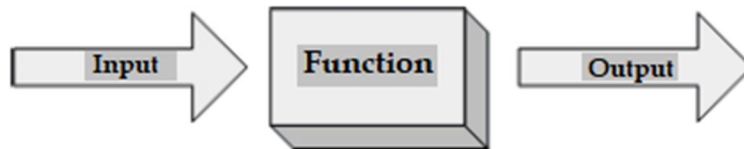
Example 3:

Write a procedure that calculates the sum and product of two integers.

```
Procedure SumProduct (x, y: integer by value, sum, product: integer
by reference)
  sum ← x + y
  product ← x * y
EndProcedure
```

4.3 Functions and procedures in C language

A function in C has an input and an output. When calling a function, there are three steps :



- a. Input: Information is passed into the function (providing it with data to work with).
- b. Processing: Using the input information, the function performs its operations.
- c. Output: After completing its calculations, the function returns a result. This is called the output or return value.

4.3.1 Characteristics of functions and procedures in C

- In C, a function takes **N** arguments and returns **a value** of a specified type.
- A procedure is a function that returns **void**; in this case, return is called without a parameter.
- The order, type, and number of arguments must be respected when calling the function.
- A function call must be placed after its declaration or its prototype.
- If the function does not return anything, specify the void type (this function is considered a procedure).

4.3.2 Examples of functions and procedures in C

Example 1: A function that finds the minimum of two integers :

```
int min(int a, int b);
```

```
void main() {  
    int c;  
    /* enter the values of a and b */  
    c = min(a, b);  
    printf("The min of %d and %d is: %d\n", a, b, c);  
}
```

```
int min(int a, int b) {  
    if (a < b) return a;  
    else return b;  
}
```

Example 2 :

```
#include <stdio.h>
```

```
int addition1(int a, int b) { // c = a + b
    int c;
    c = a + b;
    return c;
}
```

```
void addition2(int a, int b, int c) { // c = a + b
    c = a + b;
}
```

```
void addition3(int a, int b, int *c) { // c = a + b
    *c = a + b;
}
```

```
int main(void) {
    int x1 = 7, x2 = 8;
    int c_addition1 = addition1(x1, x2); // x1 and x2 are passed by value
    int c_addition2 = 0;
    addition2(x1, x2, c_addition2); // c_addition2 is passed by value
    int c_addition3;
    addition3(x1, x2, &c_addition3); // c_addition3 is passed by reference
    printf("c_addition1 = %d\nc_addition2 = %d\nc_addition3 = %d\n",
    c_addition1, c_addition2, c_addition3);
    return 0;
}
```

Execution:

```
c_addition1 = 15
c_addition2 = 0
c_addition3 = 15

-----
Process exited after 1.417 seconds with return value 0
Appuyez sur une touche pour continuer...
```

Remark:

A variable passed by value is not modified outside the function. To modify it, it must be used with pass by reference (&c_addition3).

5. Recursion

5.1 Introduction

To solve a problem in the general case, one assumes the ability to solve it in a slightly simpler case. This is the essence of recursion. Recursion relies on the recurrent definition of the problem, which involves breaking down a problem into subproblems. Here, the subproblem is the same as the initial problem but in a slightly simpler form. This reasoning leads to writing functions that contain calls to the function being defined.

- Recursion can be used to solve most problems and computations that have a repetitive nature.
- A module (function or procedure) can call itself: this is known as a recursive module.
- Every recursive module must have a base case (trivial case) that stops the recursion.

5.2 Writing a recursive function

To successfully write a recursive function, one must:

1. Present the recurrence formula in the general case.
2. Identify the particular cases for which the general formula does not apply and which are the stopping cases.
3. Ensure there is a strictly decreasing function associated with the parameters of the recursive function. This function, called the simplification function, signifies that the problem is indeed being solved by dividing it into subproblems simpler than the initial problem.

Example1 : Calculating the Factorial

1. Present the recurrence formula: $n! = n*(n-1)!$
2. Identify the particular case, which is the stopping case: $0! = 1$
3. Ensure there is a strictly decreasing function associated with the parameters of the recursive function: This function associates the value $n-1$ with the parameter n , and it is **strictly decreasing on natural integers**.

Function fact (n: integer): integer

If (n = 0) then

 return (1)

Else

 return (n * fact(n-1))

EndIf

EndFunction

In C Language:

```
int factorial(int n) {  
    int f;  
    if (n == 0)  
        f = 1;  
    else  
        f = n * factorial(n-1);  
    return f;  
}
```

Example 2: Write a Recursive Procedure to Display the Binary Value of an Integer n.

Procedure binary (n: integer)

If (n ≠ 0) then

 binary (n / 2);

 write (n mod 2);

EndIf

EndProcedure

Remark:

If the function's text calls the function itself, the function is recursive. At each recursive call, all local variables are saved on a stack and restored upon exiting the recursive call. The compiler manages the stack, but the user must ensure that recursive calls terminate. Otherwise, a runtime error occurs when the call stack overflows (stack overflow).

Exercises

Exercise 1:

Write a C function to calculate the n-th power of a value x and the calling program.

Exercise 2:

1. Write a C function to calculate the sum of the elements of a column in a matrix.
2. Write a C function to swap two columns in a matrix.
3. Write a C program that uses the previous functions to determine the index of the column with the minimum sum of elements and the index of the column with the maximum sum of elements, and swap these two columns.

Exercise 3:

Write a function to remove duplicate values in a sorted vector.

Exercise 4:

Write a function to construct the mirror word of a given word.

Exercise 5:

Write a recursive subroutine to calculate the greatest common divisor (GCD) of two numbers a and b.

Exercise 6:

- a) Write a recursive subroutine to check if a value val is in a vector.
- b) Write a recursive subroutine to check if a word is a palindrome.

Exercise 7:

Write a recursive function to calculate the n-th Fibonacci number.

$$f_0 = f_1 = 1,$$

$$f_{n+2} = f_n + f_{n+1}.$$

Exercise 8:

We have three pegs and 64 disks, each with a different radius and a hole in the center to pass through the pegs. Initially, all 64 disks are on the first peg, sorted by size with the largest at the bottom.

- ✓ The goal is to move these disks to the third peg following these rules:
- ✓ Only one disk can be moved at a time.
- ✓ At any moment, on any peg, a disk can only be placed on top of a larger disk.

Exercise 9:

Write a C recursive procedure that displays the binary value of an integer n (read from input), along with the main program that calls this procedure.

Exercise 10:

Given a vector T (a one-dimensional array) containing N integer numbers (where $N \leq 100$), write functions to:

1. Determine the minimum, maximum, and average of the elements in the array T .
2. Calculate the product of all elements in T and count the number of strictly positive values.
3. Calculate the sum and dot product of two vectors $T1$ and $T2$.
4. Determine the positions where a specific value appears in the vector T .
5. Reverse the contents of the vector T .
6. Remove all zero values from the vector T .
7. Move negative values to the beginning and positive values to the end of the vector using only one array.

Exercise 11:

1. Write and Test a Mirror Function: Implement a function that returns the mirror (reverse) of a given string. Ensure to test the function with various inputs.
2. Implement a void uppercase Function: Write a function that converts all lowercase letters in a string to uppercase.
3. Write a sub_word Function: Create a function that checks whether a given substring is present within a specified word.

Important: You are not allowed to use the functions strcpy, strcmp, or strncmp in your implementations. However, you may use the strlen function.

Exercise 12:

1. Write a Function occurrences: Implement a function that returns the number of times a character C appears in a character array T.
2. Check for Anagrams Using occurrences: Two words are anagrams if one word contains exactly the same characters as the other, in any order. For example, "carte" and "trace," or "sortie" and "toiser." Duplicates can exist within the words (each character in a word may appear multiple times). Using the occurrences function, write a function anagram that takes two character arrays T1 and T2 representing two words and returns true if the words are anagrams of each other and false otherwise.
3. Write a Function find: Create a function that determines whether the word M is found within the text T starting from a given position i.
4. Count Word Occurrences Using find: Using the find function, write a function count_words that counts the number of times a given word M appears in the text T.

Solutions

Exercise 1:

```
#include <stdio.h>

double power(double x, int n) {
    double result = 1.0;
    for (int i = 0; i < n; i++) {
        result *= x;
    }
    return result;
}

int main() {
    double x;
    int n;
    printf("Enter the value of x: ");
    scanf("%lf", &x);
    printf("Enter the value of n: ");
    scanf("%d", &n);
    printf("The value of %.2lf^%d is: %.2lf\n", x, n, power(x,
n));
    return 0;
}
```

Exercise 2:

1. Sum of elements in a column :

```
#include <stdio.h>

int sum_column(int matrix[][N], int col, int rows) {
    int sum = 0;
    for (int i = 0; i < rows; i++) {
```

```

        sum += matrix[i][col];
    }
    return sum;
}

```

2. Swap two columns:

```

void swap_columns(int matrix[][N], int col1, int col2, int
rows) {
    for (int i = 0; i < rows; i++) {
        int temp = matrix[i][col1];
        matrix[i][col1] = matrix[i][col2];
        matrix[i][col2] = temp;
    }
}

```

3. Program to find columns with minimum and maximum sum and swap them:

```

int main() {
    int matrix[M][N]; // Assume M rows and N columns
    int min_col = 0, max_col = 0;
    int min_sum = sum_column(matrix, 0, M);
    int max_sum = min_sum;

    for (int j = 1; j < N; j++) {
        int col_sum = sum_column(matrix, j, M);
        if (col_sum < min_sum) {
            min_sum = col_sum;
            min_col = j;
        }
        if (col_sum > max_sum) {
            max_sum = col_sum;
        }
    }
}

```

```

        max_col = j;
    }
}

swap_columns(matrix, min_col, max_col, M);

// Print the matrix or any other operation
return 0;
}

```

Exercise 3:

```

#include <stdio.h>

int remove_duplicates(int arr[], int n) {
    if (n == 0 || n == 1) return n;

    int j = 0; // To store index of next unique element
    for (int i = 0; i < n-1; i++) {
        if (arr[i] != arr[i+1]) {
            arr[j++] = arr[i];
        }
    }
    arr[j++] = arr[n-1];

    return j;
}

int main() {
    int arr[] = {1, 2, 2, 3, 4, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    n = remove_duplicates(arr, n);
}

```

```

    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}

```

Exercise 4:

```

#include <stdio.h>
#include <string.h>

void mirror_word(char str[]) {
    int len = strlen(str);
    for (int i = len - 1; i >= 0; i--) {
        printf("%c", str[i]);
    }
}

int main() {
    char word[] = "example";
    printf("Mirror word of %s is: ", word);
    mirror_word(word);
    printf("\n");
    return 0;
}

```

Exercise 5:

```

#include <stdio.h>

int gcd(int a, int b) {
    if (b == 0)
        return a;
}

```

```

else
    return gcd(b, a % b);
}

int main() {
    int a = 56, b = 98;
    printf("GCD of %d and %d is %d\n", a, b, gcd(a, b));
    return 0;
}

```

Exercise 6:

a) Check if a value is in a vector :

```

#include <stdio.h>

int is_in_vector(int arr[], int n, int val) {
    if (n == 0)
        return 0;
    if (arr[n-1] == val)
        return 1;
    return is_in_vector(arr, n-1, val);
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    int val = 3;

    if (is_in_vector(arr, n, val))
        printf("%d is in the vector\n", val);
    else
        printf("%d is not in the vector\n", val);
}

```

```
    return 0;
}
```

b) Check if a word is a palindrome:

```
#include <stdio.h>
#include <string.h>

int is_palindrome(char str[], int start, int end) {
    if (start >= end)
        return 1;
    if (str[start] != str[end])
        return 0;
    return is_palindrome(str, start + 1, end - 1);
}

int main() {
    char word[] = "radar";
    int len = strlen(word);

    if (is_palindrome(word, 0, len - 1))
        printf("%s is a palindrome\n", word);
    else
        printf("%s is not a palindrome\n", word);

    return 0;
}
```

Exercise 7:

```
#include <stdio.h>

int fibonacci(int n) {
    if (n == 0 || n == 1)
```



```

        return 1;
    else
        return fibonacci (n-1) + fibonacci (n-2);
}

int main() {
    int n = 10;
    printf("Fibonacci number %d is %d\n", n, fibonacci (n));
    return 0;
}

```

Exercise 8:

```

#include <stdio.h>

void hanoi (int n, char from_peg, char to_peg, char aux_peg) {
    if (n == 1) {
        printf("Move disk 1 from peg %c to peg %c\n", from_peg,
to_peg);
        return;
    }
    hanoi (n - 1, from_peg, aux_peg, to_peg);
    printf("Move disk %d from peg %c to peg %c\n", n, from_peg,
to_peg);
    hanoi (n - 1, aux_peg, to_peg, from_peg);
}

int main() {
    int n = 3; // Number of disks
    hanoi (n, 'A', 'C', 'B'); // A, B and C are names of pegs
    return 0;
}

```

Exercise 9:

Algorithm:

- Base Case: If n is 0, print 0.
- Recursive Case:
 - If n is greater than 0, recursively call the procedure with $n/2$.
 - After returning from the recursion, print the remainder when n is divided by 2 ($n \% 2$).

C Program:

```
#include <stdio.h>

// Recursive procedure to print the binary representation of n
void printBinary(int n) {
    if (n == 0) {
        return;
    }
    printBinary(n / 2); // Recursive call
    printf("%d", n % 2); // Print the remainder
}

int main() {
    int n;

    printf("Enter an integer: ");
    scanf("%d", &n);

    if (n == 0) {
        printf("0");
    } else {
        printBinary(n);
    }

    printf("\n");
}
```

```
    return 0;
}
```

Exercise 10 :

1. Determine the Minimum, Maximum, and Average:

Algorithm:

- Initialize min to the first element, max to the first element, and sum to 0.
- Iterate over the array T:
 - Update min if the current element is smaller.
 - Update max if the current element is larger.
 - Add the current element to sum.
- Calculate the average as sum / N .

C Program:

```
#include <stdio.h>

void minMaxAvg(int T[], int N, int *min, int *max, float *avg)
{
    int sum = 0;
    *min = *max = T[0];

    for (int i = 0; i < N; i++) {
        if (T[i] < *min) *min = T[i];
        if (T[i] > *max) *max = T[i];
        sum += T[i];
    }

    *avg = (float)sum / N;
}
```

2. Calculate the Product and Count Positive Values:

Algorithm:

- Initialize product to 1 and countPos to 0.
- Iterate over the array T:
 - Multiply product by the current element.
 - Increment countPos if the current element is positive.

C Function:

```
void productAndCountPositives(int T[], int N, int *product, int
*countPos) {
    *product = 1;
    *countPos = 0;

    for (int i = 0; i < N; i++) {
        *product *= T[i];
        if (T[i] > 0) (*countPos)++;
    }
}
```

3. Calculate the Sum and Dot Product of Two Vectors:

Algorithm:

- Initialize sumT1, sumT2, and dotProduct to 0.
- Iterate over the arrays T1 and T2:
 - Add the corresponding elements to sumT1 and sumT2.
 - Multiply the corresponding elements and add to dotProduct.

C Function:

```
void sumAndDotProduct(int T1[], int T2[], int N, int *sumT1,
int *sumT2, int *dotProduct) {
```

```

*sumT1 = *sumT2 = *dotProduct = 0;

for (int i = 0; i < N; i++) {
    *sumT1 += T1[i];
    *sumT2 += T2[i];
    *dotProduct += T1[i] * T2[i];
}
}

```

4. Determine Positions of a Value in a Vector:

Algorithm:

- Iterate over the array T:
 - If the current element equals the value, record its position.

C Function:

```

void findPositions(int T[], int N, int value) {
    printf("Positions of %d: ", value);
    for (int i = 0; i < N; i++) {
        if (T[i] == value) {
            printf("%d ", i);
        }
    }
    printf("\n");
}

```

5. Reverse the Contents of a Vector:

Algorithm:

- Use two pointers, start and end, initialized at the beginning and end of the array.
- Swap the elements at start and end.
- Move the pointers toward each other and repeat until they meet.

C Function:

```
void reverseVector(int T[], int N) {
    int start = 0, end = N - 1, temp;

    while (start < end) {
        temp = T[start];
        T[start] = T[end];
        T[end] = temp;
        start++;
        end--;
    }
}
```

6. Remove All Zero Values from a Vector:

Algorithm:

- Use an index to keep track of the position in the new array.
- Iterate over T:
 - If the element is not zero, move it to the new position.

C Function:

```
int removeZeros(int T[], int N) {
    int j = 0;

    for (int i = 0; i < N; i++) {
        if (T[i] != 0) {
            T[j] = T[i];
            j++;
        }
    }

    return j; // New size of the array
}
```

7. Move negative values to the beginning and positive values to the end:

Algorithm:

- Use two pointers, left at the start and right at the end.
- Move the left pointer forward if it points to a negative number.
- Move the right pointer backward if it points to a positive number.
- Swap elements if left points to a positive number and right points to a negative number.

C Function:

```
void partitionNegativesPositives(int T[], int N) {
    int left = 0, right = N - 1, temp;

    while (left < right) {
        if (T[left] < 0) left++;
        else if (T[right] > 0) right--;
        else {
            temp = T[left];
            T[left] = T[right];
            T[right] = temp;
            left++;
            right--;
        }
    }
}
```

Exercise 11 :

```
#include <stdio.h>
#include <string.h>
```

// Function to reverse a string

```
void mirror(char *str) {
    int len = strlen(str);
    for (int i = 0; i < len / 2; i++) {
        char temp = str[i];
        str[i] = str[len - i - 1];
        str[len - i - 1] = temp;
    }
}
```

// Function to convert lowercase letters to uppercase

```
void uppercase (char *str) {
    for (int i = 0; str[i] != '\0'; i++) {
        if (str[i] >= 'a' && str[i] <= 'z') {
            str[i] = str[i] - ('a' - 'A');
        }
    }
}
```

// Function to check if a substring is part of a word

```
int sub_word(char *word, char *sub) {
    int len_word = strlen(word);
    int len_sub = strlen(sub);

    for (int i = 0; i <= len_word - len_sub; i++) {
        int j;
        for (j = 0; j < len_sub; j++) {
            if (word[i + j] != sub[j]) {
                break;
            }
        }
    }
}
```



```

        if (j == len_sub) {
            return 1; // Substring found
        }
    }
    return 0; // Substring not found
}

int main() {
    char str1[] = "example";
    char str2[] = "Hello World!";
    char word[] = "programming";
    char sub[] = "gram";

    // Testing mirror function
    mirror(str1);
    printf("Mirrored string: %s\n", str1);

    // Testing Uppercase function
    lowercase (str2);
    printf("Uppercase string: %s\n", str2);

    // Testing sub_word function
    int result = sub_word(word, sub);
    if (result) {
        printf("'%' is a substring of '%s'\n", sub, word);
    } else {
        printf("'%' is not a substring of '%s'\n", sub, word);
    }

    return 0;
}

```

Exercise 12 :

a. Solution Algorithm

1. Occurrences Algorithm:

- Initialize a counter to 0.
- Loop through the character array T.
- For each character, if it matches C, increment the counter.
- Return the counter.

2. Anagram Algorithm:

- For each character in the first word T1, check if it has the same number of occurrences in T2 using the occurrences function.
- If all characters match in frequency, return true; otherwise, return false.

3. Find Algorithm:

- Start from position i in T.
- Check if the substring of T starting at i matches the word M.
- If it matches, return true; otherwise, continue searching.
- If the end of the array is reached without finding M, return false.

4. Count_words Algorithm:

- Initialize a counter to 0.
- Use the find function to find the first occurrence of M in T.
- Increment the counter and continue searching from the next position until the end of T.
- Return the counter.

b. C Program Solution

```
#include <stdio.h>
#include <string.h>
```

// Function to count the occurrences of a character C in array T

```
int occurrences(char T[], char C) {
    int count = 0;
    for (int i = 0; T[i] != '\0'; i++) {
        if (T[i] == C) {
            count++;
        }
    }
    return count;
}
```

// Function to check if two words are anagrams

```
int anagram(char T1[], char T2[]) {
    if (strlen(T1) != strlen(T2)) {
        return 0; // Not anagrams if they have different lengths
    }
    for (int i = 0; T1[i] != '\0'; i++) {
        if (occurrences(T1, T1[i]) != occurrences(T2, T1[i])) {
            return 0; // Not anagrams if occurrences don't match
        }
    }
    return 1; // They are anagrams
}
```

// Function to find if the word M is in text T from position i

```
int find(char T[], char M[], int i) {
    int lenM = strlen(M);
    for (int j = 0; j < lenM; j++) {
        if (T[i + j] != M[j]) {
            return 0; // Mismatch found
        }
    }
    return 1; // Match found
}
```

```
// Function to count the number of occurrences of word M in text T
```

```
int count_words(char T[], char M[]) {  
    int count = 0;  
    int lenT = strlen(T);  
    int lenM = strlen(M);  
  
    for (int i = 0; i <= lenT - lenM; i++) {  
        if (find(T, M, i)) {  
            count++;  
        }  
    }  
    return count;  
}
```

```
int main() {  
    char T[] = "carte";  
    char T1[] = "trace";  
    char M[] = "sortie";  
    char T2[] = "toiser";  
    char text[] = "sortie sortie sortie";  
    char word[] = "sortie";  
  
    // Test occurrences function  
    printf("Occurrences of 'a' in 'carte': %d\n", occurrences(T,  
'a'));  
  
    // Test anagram function  
    if (anagram(T, T1)) {  
        printf("' %s' and '%s' are anagrams.\n", T, T1);  
    } else {  
        printf("' %s' and '%s' are not anagrams.\n", T, T1);  
    }  
}
```

```
if (anagram(M, T2)) {
    printf("' %s' and '%s' are anagrams.\n", M, T2);
} else {
    printf("' %s' and '%s' are not anagrams.\n", M, T2);
}

// Test count_words function
printf("Number of occurrences of '%s' in the text: %d\n",
word, count_words(text, word));

return 0;
}
```

CHAPTER 2

FILES

1. Introduction :

Until now, the values of the data used in a C program disappear once the execution is finished and we return to the program. The problem with these variables is that they exist only in RAM. Once your program stops, all your variables are deleted from memory, and it is not possible to retrieve their value afterward. How can we save these data and retrieve them when we stop the program?

The answer is that we can read from and write to files in the C language. These files will be written to the hard drive of your computer, so the advantage is that they remain there even if you stop the program or the computer. C provides a set of functions in the `stdio.h` header file for reading and writing data to and from a file.

Why are files necessary ?

- When a program finishes, all data is lost. Storing data in a file will preserve your data even if the program terminates.
- It takes a lot of time to input a large amount of data. Using a file containing all the data will facilitate access to the content of the file using a few commands in C.
- Data can be easily transferred from one computer to another without any modification.

2. Types of files

When dealing with files, you need to know about two types of files: Text files and binary files.

Text files are portable, meaning you can create a text file on Windows and open it on Linux without any issues. On the other hand, the size of data types and the byte order can vary from one system to another. However, binary files are not portable.

a. Text files

In text mode, data is stored as a line of characters terminated by a newline character (`'\n'`), where each character occupies 1 byte.

What is important to note in text mode:

- What is stored in memory is the binary equivalent of the ASCII number of the character.
- When you open these files, you can see all the content in the text file. You can easily modify or delete the content.

- They require minimal effort to maintain, are easily readable, and provide the least security.
- They take up more storage space.

b. Binary files

Binary files are mainly the .bin files on your computer.

In binary mode, data is stored on a disk in the same way it is represented in the computer's memory.

What is important to note in binary mode:

- They can contain a larger amount of data.
- They are not easily readable.
- They offer better security than text files.

3. File manipulation

In this section, we will learn how to perform input and output operations on a file.

3.1 Buffer

Reading and writing to and from files stored on the disk is a relatively slow process compared to reading and writing data stored in RAM. As a result, all standard input/output functions use a system called buffering to temporarily store data.

A buffer is a memory area where data is temporarily stored before being written to the file. When the buffer is full, the data is written (flushed) to the file. Additionally, when the file is closed, the data in the buffer is automatically written to the file, whether the buffer is full or not. This process is called **flushing the buffer**.

As soon as a file is opened, a buffer is automatically created. However, there are rare occasions when you may need to manually flush the buffer. If so, you need to call the **fflush()** function.

The main steps for file processing are as follows:

- 1- Opening the file
- 2- Error checking
- 3- Performing read/write operations on the file
- 4- Closing the file

3.2 Opening/Closing a file

When working with files, you need to declare a pointer of type FILE. This declaration is necessary for communication between the file and the program.

File *fic;

Opening a file is done using the fopen() function defined in the header file stdio.h. fopen returns a FILE*. It is extremely important to retrieve this pointer to be able to read from and write to the file. We will create a FILE pointer at the beginning of our function (for example, the main function):

```
int main()  
{  
    FILE* fichier = NULL;  
    return 0;  
}
```

The pointer is initialized to NULL from the beginning. It is a fundamental rule to initialize pointers to NULL from the start if you don't have another value to give them. If you don't, you significantly increase the risk of errors later on.

Then, you call the fopen function to get the value it returns into the fichier pointer. The syntax is as follows:

```
FILE * fopen(const char *filename, const char *mode);
```

- **filename:** string containing the name of the file.
- **mode:** specifies what you want to do with the file, i.e., read, write, append.

If successful, the fopen() function returns a pointer to the FILE structure. The FILE structure is defined in stdio.h and contains information about the file, such as the name, size, buffer size, current position, end of file, etc. If an error occurs, the fopen() function returns NULL.

The following code opens the file test.txt in "r+" (read/write) mode:

```
int main ()
```

```

{
    FILE* fichier = NULL;
    fichier = fopen("test.txt", "r+");
    return 0;
}

```

Or

```

int main ()
{
    FILE* fichier = fopen("test.txt", "r+");
    return 0;
}

```

The fichier pointer then becomes a pointer to test.txt. This test.txt file must be located in the same folder as the executable program.

The possible mode values are :

"w" (write) – This mode is used to write data to the file.

- If the file does not exist, this mode creates a new file.
- If the file already exists, this mode first erases the data in the file before writing anything.

"a" (append) – This mode is called append mode.

- If the file does not exist, this mode creates a new file.
- If the file already exists, this mode adds new data to the end of the file.

"r" (read) – This mode opens the file for reading. To open a file in this mode, the file must already exist. This mode does not modify the file content. Use this mode if you only want to read the file content.

"w+" (write + read) – This mode is identical to "w" but in this mode, you can also read data.

- If the file does not exist, this mode creates a new file.
- If the file already exists, the previous data is erased before writing new data.

"r+" (read + write) – This mode is identical to the "r" mode, but you can also modify the file content. To open the file in this mode, the file must already exist. You can modify the data in this mode, but the file content is not erased. This mode is also called update mode.

"a+" (append + read) – This mode is the same as the "a" mode but in this mode, you can also read the data from the file.

- If the file does not exist, a new file is created.
- If the file already exists, new data is added to the end of the file.
- Note that in this mode, you can add data but you cannot modify existing data.

To open the file in binary mode, you must add "b" to the mode as follows:

"wb": Open the file in binary mode.

"a+b" or "ab+": Open the binary file in (append + read) mode.

3.3 Error checking

The fichier pointer should contain the address of the FILE structure that serves as the file descriptor. This has been loaded into memory for you by the fopen() function. From there, two possibilities arise:

As mentioned earlier, fopen() returns NULL when an error is encountered while opening a file. Therefore, before performing any operations on the file, you must always check the file's opening.

```
FILE *fic;
fic = fopen("filename.txt", "r");

if (fic == NULL) {
    printf("Error opening file");
    exit(1);
} else {
    // we can read and write to the file
}
```

3.4 Closing a file

Always remember to close a file once you have finished working with it. This helps to free up memory. Both text and binary files should be closed after reading/writing operations. Closing a file is done using the fclose(fichier) function. This function takes as a parameter the pointer to the file and returns an int:

- ✓ 0: if the file was successfully closed
- ✓ EOF: if the file closure failed
- ✓ To close a file, you should write: fclose(fic);

Here, fic is a file pointer (FILE) associated with the file to be closed.

Example:

```
FILE *fic;
fic = fopen("filename.txt", "r");
```

```

if (fic == NULL) {
    printf("Error opening file");
    exit(1);
}
// operations on the file

fclose(fic);

```

4. Reading/Writing to a file in C

A file represents a sequence of bytes, whether it is a text file or a binary file. The C programming language provides functions to manage files on storage devices.

Reading from a file :

We can use almost the same functions as for writing:

- a. fgetc: reads a character;
- b. fgets: reads a string;
- c. fscanf: reads a formatted string.

Writing to a file :

There are several functions capable of writing to a file. The choice among them depends on the specific problem to be addressed. The three main functions for writing to a file are:

- a. fputc: writes a character to the file (ONE character at a time);
- b. fputs: writes a string to the file;
- c. printf: writes a formatted string to the file.

Character-by-Character: fgetc – fputc

fgetc : int fgetc(FILE *f);

- ✓ Reads a single character from the file and increments the file position pointer.
- ✓ The file must be opened in read mode.
- ✓ On success, it returns the ASCII value of the character.
- ✓ On failure or end of file, it returns EOF.

Example :

```

File *fi c;
fi c = fopen("test.txt", "r");
char ch ;
ch = (char)fgetc(fi c) ;
printf("%c", ch);
fclose(fi c);

```

or :

```
int nb ;  
nb = fgetc(fic) ;  
printf("%c", nb);
```

To read multiple characters, you can introduce a loop as follows:

```
while((ch=fgetc(fic))!=EOF)  
{  
    printf("%c", ch);  
}
```

Reminder:

- The function getchar() corresponds to the ASCII value of the character just read from the standard input.
- EOF is returned when an end-of-file character is encountered.
-

fputc : int fputc(int ch, FILE *fic);

- ✓ The function fputc() is used to write a single character specified by the first argument to a text file pointed to by the pointer fic.
- ✓ After writing a character to the text file, the internal file position pointer is incremented.
- ✓ If the write is successful, it returns the ASCII value of the character that was written.
- ✓ On error, it returns EOF.

Exemple : **fputc('b' , fic);**

By String: fgets – fputs

fgets : char *fgets(char *ch, int n, FILE *fic);

- The function reads a string from a file pointed to by fic into the memory pointed to by ch.
- It reads characters from the file until a newline character ('\n') is read, or until n-1 characters are read, or until the end of the file is encountered.
- After reading the string, it adds a null character ('\0') to terminate the string ch.
- On error or end of file, it returns **NULL**.

Example :

```
char str[20]; // str will hold the text read from the file
fi c = fopen("test.txt", "r");

if(fi c == NULL) {
    printf("Error opening the file");
    exit(1);
}

fgets(str, 20, fi c);
fclose(fi c);
```

To read until the end of the file:

```
char str[20];
while(fgets(str, 20, fi c) != NULL) {
    puts(str);
}
```

fputs : int fputs(char *str, FILE *fi c);

- The function **fputs()** writes the string str to the output stream referenced by fi c.
- It returns a **non-negative value** on success,
- Otherwise, **EOF** is returned in case of error.

Example 1 :

```
FILE *fi c;
char str[20];
fi c = fopen("test.txt", "w");

if(fi c == NULL) {
    printf("Error opening the file");
    exit(1);
}

gets(str);
fputs(str, fi c);
fputs("bonj our", fi c);

fclose(fi c);
```

Example 2 :

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    FILE *f;
    char str[20];
    f = fopen("test.txt", "w");

    if(f == NULL) {
        printf("Error opening the file");
        exit(1);
    }

    while(gets(str) != NULL) {
        fputs(str, f);
    }

    fclose(f);
}
```

Here are two important points to remember about the `gets()` function:

- The `gets()` function converts the entered newline character into a null character ('\0').
- When the end of file character is encountered, `gets()` returns NULL.

Reading/Writing Text Files: fscanf – fprintf

So far, we have seen how to read and write characters and strings to and from a file. In the real world, data consists of many different types. In this section, we will see how to read and write different types of data in a formatted manner.

fscanf : int fscanf(FILE *fp, const char *format [, argument, ...]);

- The `fscanf()` function is used to read formatted text from the file. It works like the `scanf()` function but instead of reading data from standard input, it reads data from the file.
- On success, this function returns the number of values read.
- On error or end of file, it returns EOF.

Example :

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    FILE *f;
    char nom[20];
    int age;

    f = fopen("test.txt", "r");

    if(f == NULL) {
        printf("Error opening the file");
        exit(1);
    }
    //read data from the file and store it in the variables nom and age
    fscanf(f, "Name: %s\tAge: %d\n", nom, &age); // file -> RAM

    // display the data
    printf("Name: %s\tAge: %d\n", nom, age); // RAM -> console

    fclose(f);
}
```

fprintf : int fprintf(FILE *fp, const char *format [, argument, ...]);

- The fprintf() function is identical to printf(), but instead of writing data to the console, it writes formatted data to the file.
- On success, it returns the total number of characters written to the file.
- On error, it returns EOF.

Example :

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    FILE *f;
    char nom[20];
    int age;

    f = fopen("test.txt", "w");

    if(f == NULL) {
        printf("Error opening the file");
        exit(1);
    }

    printf("Name: ");
```



```

scanf("%s", nom);
// console -> RAM (to the address nom using the gets function)
printf("Age: ");
scanf("%d", &age);

// I can choose the formatting or format I want
fprintf(fi c, "Name: %s\tAge: %d\n", nom, age); // RAM ->
file using fprintf

fclose(fi c);
}

```

Reading/Writing binary files: fread-fwrite

So far, we have been using text mode to read and write data to and from a file. In this section, we will learn how to read and write data to and from a file in binary mode. Remember that in binary mode, data is stored in the file the same way it is in memory, so no data transformation occurs in binary mode.

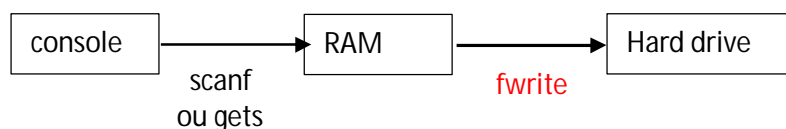
The `fread()` and `fwrite()` functions are commonly used to read and write binary data to and from a file, respectively. Although we can also use them with text mode.

`fwrite` : `int fwrite(void *source, int taille_type, int n, FILE *fi c);`

Allows writing data to a binary file. The function takes four arguments :

- (source): address of the data to be written to the file `fi c`
- `taille_type`: size of the data to be written to the file
- `n`: number of such data elements
- `fi c`: pointer to the file where we want to write

This will save the content from RAM to the file `fi c`, resulting in bytes (binary) that cannot be read by a text file.



Example 1 : Writing a variable

```
int a = 5;
fwrite(&a, sizeof(a), 1, fic); // fic: pointer to the file
```

- This instruction will save the value of a in a binary file.
- &a: pointer or address to the data to be written
- sizeof(a): size of the element in bytes
- 1: number of elements to write, in this case, it's 1
- fic: the file to which we write.

Example 2 : Writing an array

```
int tab[4]={2,5,7,8};
fwrite(tab, sizeof(tab), 1, fic); // fic : pointer to the file
```

Example 3 : Writing a structure

```
struct etudiant{
    char nom[20];
    int age;
};
struct etudiant etud={"mostafa", 34};
fwrite(&etud, sizeof(etud), 1, fic); // fic : pointer to the file
// This instruction saves 1 structure with 2 fields at once
```

Example 4 : Writing an array of structures

```
struct etudiant{
    char nom[20];
    int age;
};
struct etudiant etud[3]={{ "mostafa", 32}, {"ismail", 27}, {"dounia", 23}};
fwrite(etud, sizeof(etud), 1, fic); // fic : pointer to the file
```

Let's say we do not want to write all the elements of the array to the file. Instead, we want to write only the first and second elements of the array to the file.

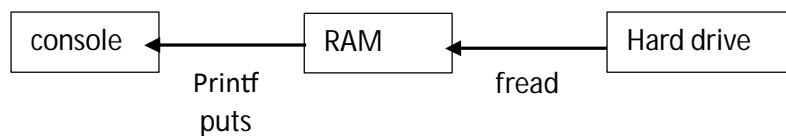
Example 5 :

```
struct etudiant{
    char nom[20];
    int age;
};
```

```
struct etudiant etud[3]={{"mostafa", 32}, {"ismail", 27}, {"dounia", 23}};
fwrite(etud, sizeof(struct etudiant), 2, fic); // fic : pointer to the file
```

fread : int fread(void *destination, int taille_type, int n, FILE *fic);

- fread() is commonly used to read binary data.
- It takes four arguments.
- It allows retrieving the content from the file fic and placing it in the destination of dimension n.
- To see the content of the string, we use the puts or printf function.



Example 6 : Reading a variable

```
int a;
fread(&a, sizeof(a), 1, f); // f: pointer to the file
printf("a = %d\n", a); // display what we retrieved
```

Example 7 : Reading an array

```
int tab[4]={2, 5, 7, 8};
fread(tab, sizeof(tab), 1, f); // f : pointer to the file
```

Example 8 : Reading a structure

```
struct etudiant{
    char nom[20];
    int age;
};
```

```
struct etudiant etd={"mostafa", 34};
fread(&etd, sizeof(etd), 1, f); // f : pointer to the file
```

Example 9 : Reading an array of structures

```
struct etudiant{
    char nom[20];
    int age;
};

struct etudiant etds[3];
fread(&etds, sizeof(etds), 5, f); // f : pointer to the file
```

This instruction reads the first 5 elements of type struct etudiant from the file and stores them in the variable etds.

putw(), getw()

- **Putw** : writes an integer value to a file
- **Getw** : reads an integer value from a file

Example :

```
#include <stdio.h>
int main ()
{
    FILE *fp;
    int i=1, j=2, k=3, num;
    fp = fopen ("test.c", "w");
    putw(i, fp);
    putw(j, fp);
    putw(k, fp);
    fclose(fp);

    fp = fopen ("test.c", "r");

    while(getw(fp)!=EOF)
    {
        num= getw(fp);
        printf("Data in test.c file is %d \n", num);
    }
    fclose(fp);
    return 0;
}
```

Moving within a file

Whenever you open a file, there is a cursor that indicates your position in the file. In summary, the cursor system allows you to read and write at a specific position within the file.

There are three functions to know:

- ✚ `ftell`: indicates your current position in the file;
- ✚ `fseek`: moves the cursor to a specific position;
- ✚ `rewind`: resets the cursor to the beginning of the file (equivalent to asking the `fseek` function to move the cursor to the beginning).

ftell : position in the file

This function is very simple to use. It returns the current position of the cursor as a long:

```
long ftell(FILE* filePointer);
```

The returned number indicates the cursor's position in the file.

fseek : moving within the file

The prototype of `fseek` is as follows:

```
int fseek(FILE* filePointer, long offset, int origin);
```

The `fseek` function allows you to move the cursor by a certain number of characters (indicated by `offset`) from the position indicated by `origin`.

The `offset` can be a positive number (to move forward), zero (= 0), or a negative number (to move backward).

As for the `origin` value, you can use one of the three constants (usually defined) listed below:

- **seek_set**: indicates the beginning of the file;
- **seek_cur**: indicates the current position of the cursor;
- **seek_end**: indicates the end of the file.

Other commands:

- Deleting a file: `int remove(char *filename);`
- Renaming a file: `int rename(char *oldname, char *newname);`
- Resetting the pointer to the beginning of the file: `void rewind(FILE *file);`

Example:

```
#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "r");
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    // Get the current position of the cursor
    long position = ftell(file);
    printf("Current position: %ld\n", position);

    // Move the cursor to the 10th byte from the beginning
    fseek(file, 10, SEEK_SET);
    position = ftell(file);
    printf("Position after fseek: %ld\n", position);

    // Reset the cursor to the beginning of the file
    rewind(file);
    position = ftell(file);
    printf("Position after rewind: %ld\n", position);

    fclose(file);
    return 0;
}
```

In this example:

- o ftell is used to get the current cursor position.
- o fseek is used to move the cursor 10 bytes from the beginning of the file.
- o rewind is used to reset the cursor to the beginning of the file.

Exercises

Exercise 1 :

1. Create a character file "essai1.dat" using an editor. Write a program to copy this text file to another file "essai2.dat". Test the program by checking the presence of the copied file in the directory after execution.
2. Calculate and display the number of characters in a character file (use any file from the directory).
3. Create and read a binary file of 10 integers.

Exercise 2 :

Write a C program that counts the number of words starting with the letter 'A' in a character file.

Exercise 3 :

Write a C program that constructs a file G from the elements of a character file F by replacing a sequence of spaces with a single space.

Exercise 4 :

Given two files F and G sorted in ascending order, write a C program to merge F and G into a sorted file H.

Exercise 5 :

The university wants to manage its employees. Each employee is identified by the following information:

(Num, Salary, First Name, Last Name, Gender, Department, Function "teacher or not")

Create a corresponding structure. Write a C program for file management with a main menu providing the following options:

1. Create the file.
2. Enter 10 employees.
3. Display the employee at position 6 in the file.
4. Count the number of female teachers.

Exercise 6 :

Given a text file whose name is entered via the keyboard, write a C program that:

- a. Determines the number of characters and the number of lines in this file.
- b. Uses two functions: `palindr` and `long_moy` to determine respectively the number of palindromes and the average word length in this file.
- c. Deletes a line specified by its number (entered via the keyboard) and displays the content of the file before and after deletion.

Exercise 7 :

Write a procedure that deletes the last element from a file F containing integers. Using this procedure, write an algorithm to empty an existing file F (named 'ESSAI.DAT') of integers, one element at a time. The algorithm should display the average of the remaining elements in F after each deletion.

Solutions

Exercise 1:

1. Copy a text file "essai1.dat" to "essai2.dat".

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *source, *dest;
    char ch;

    source = fopen("essai1.dat", "r");
    if (source == NULL) {
        printf("Error opening source file.\n");
        exit(1);
    }

    dest = fopen("essai2.dat", "w");
    if (dest == NULL) {
        printf("Error opening destination file.\n");
        fclose(source);
        exit(1);
    }

    while ((ch = fgetc(source)) != EOF) {
        fputc(ch, dest);
    }

    printf("File copied successfully.\n");

    fclose(source);
    fclose(dest);
    return 0;
}
```

2. Calculate and display the number of characters in a text file.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *file;
    char ch;
    int count = 0;
```

```

file = fopen("essai1.dat", "r");
if (file == NULL) {
    printf("Error opening file.\n");
    exit(1);
}

while ((ch = fgetc(file)) != EOF) {
    count++;
}

printf("Number of characters: %d\n", count);

fclose(file);
return 0;
}

```

3. Create and read a binary file of 10 integers.

```

#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *file;
    int numbers[10];
    for (int i = 0; i < 10; i++) {
        numbers[i] = i + 1;
    }

    // Write to binary file
    file = fopen("binary.dat", "wb");
    if (file == NULL) {
        printf("Error opening file for writing.\n");
        exit(1);
    }

    fwrite(numbers, sizeof(int), 10, file);
    fclose(file);

    // Read from binary file
    int read_numbers[10];
    file = fopen("binary.dat", "rb");
    if (file == NULL) {
        printf("Error opening file for reading.\n");
        exit(1);
    }

    fread(read_numbers, sizeof(int), 10, file);
    fclose(file);
}

```

```

// Display the numbers
printf("Numbers in binary file:\n");
for (int i = 0; i < 10; i++) {
    printf("%d ", read_numbers[i]);
}
printf("\n");

return 0;
}

```

Exercise 2:

Count the number of words starting with the letter 'A'.

```

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main() {
    FILE *file;
    char word[100];
    int count = 0;

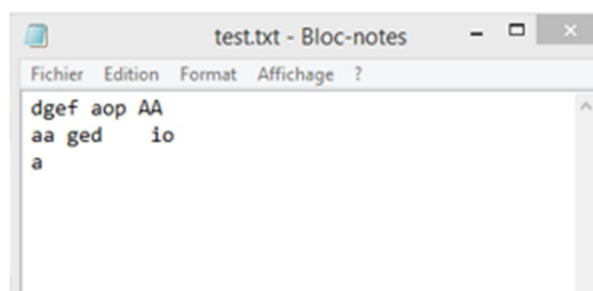
    file = fopen("essai1.dat", "r");
    if (file == NULL) {
        printf("Error opening file.\n");
        exit(1);
    }

    while (fscanf(file, "%s", word) != EOF) {
        if (toupper(word[0]) == 'A') {
            count++;
        }
    }

    printf("Number of words starting with 'A': %d\n", count);

    fclose(file);
    return 0;
}

```



```
nombre de mots qui commencent par la lettre (A/a) dans le fichier est 4
-----
Process exited after 1.797 seconds with return value 0
Appuyez sur une touche pour continuer...
```

Exercise 3:

Replace a sequence of spaces with a single space in a file.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *source, *dest;
    char ch, prev = '\0';

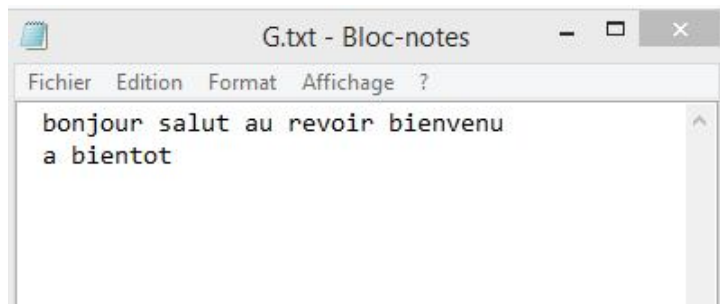
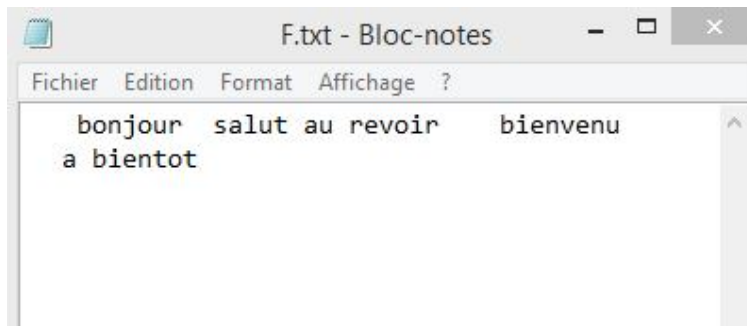
    source = fopen("essai1.dat", "r");
    if (source == NULL) {
        printf("Error opening source file.\n");
        exit(1);
    }

    dest = fopen("essai2.dat", "w");
    if (dest == NULL) {
        printf("Error opening destination file.\n");
        fclose(source);
        exit(1);
    }

    while ((ch = fgetc(source)) != EOF) {
        if (ch != ' ' || (ch == ' ' && prev != ' ')) {
            fputc(ch, dest);
        }
        prev = ch;
    }

    printf("Whitespaces compression completed.\n");

    fclose(source);
    fclose(dest);
    return 0;
}
```



Exercise 4: Merge two sorted files into a third sorted file.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *file1, *file2, *file3;
    int num1, num2;

    file1 = fopen("file1.dat", "r");
    file2 = fopen("file2.dat", "r");
    file3 = fopen("file3.dat", "w");

    if (file1 == NULL || file2 == NULL || file3 == NULL) {
        printf("Error opening files.\n");
        exit(1);
    }

    fscanf(file1, "%d", &num1);
    fscanf(file2, "%d", &num2);

    while (!feof(file1) && !feof(file2)) {
        if (num1 < num2) {
            fprintf(file3, "%d\n", num1);
            fscanf(file1, "%d", &num1);
        } else {
            fprintf(file3, "%d\n", num2);
            fscanf(file2, "%d", &num2);
        }
    }
}
```

```
while (!feof(file1)) {
    fprintf(file3, "%d\n", num1);
    fscanf(file1, "%d", &num1);
}

while (!feof(file2)) {
    fprintf(file3, "%d\n", num2);
    fscanf(file2, "%d", &num2);
}

printf("Files merged successfully.\n");

fclose(file1);
fclose(file2);
fclose(file3);
return 0;
}
```

Exercise 5: Employee management system

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define FILE_NAME "employees.dat"

typedef struct {
    int num;
    float salary;
    char firstName[20];
    char lastName[20];
    char gender; // 'M' for male, 'F' for female
    char department[20];
    int isTeacher; // 1 for teacher, 0 for non-teacher
} Employee;

void createFile() {
    FILE *file = fopen(FILE_NAME, "wb");
    if (!file) {
        printf("Error creating file.\n");
        return;
    }
    printf("File created successfully.\n");
    fclose(file);
}

void enterEmployees(int numberOfEmployees) {
    FILE *file = fopen(FILE_NAME, "ab");
    if (!file) {
        printf("Error opening file.\n");
        return;
    }

    for (int i = 0; i < numberOfEmployees; ++i) {
        Employee emp;
        printf("Enter details for employee %d\n", i + 1);
        printf("Number: "); scanf("%d", &emp.num);
        printf("Salary: "); scanf("%f", &emp.salary);
        printf("First Name: "); scanf("%s", emp.firstName);
        printf("Last Name: "); scanf("%s", emp.lastName);
        printf("Gender (M/F): "); scanf(" %c", &emp.gender);
        printf("Department: "); scanf("%s", emp.department);
        printf("Is Teacher (1 for Yes, 0 for No): ");
        scanf("%d", &emp.isTeacher);
        fwrite(&emp, sizeof(Employee), 1, file);
    }
}
```

```

    fclose(file);
}

void displayEmployeeAtPosition(int position) {
    FILE *file = fopen(FILE_NAME, "rb");
    if (!file) {
        printf("Error opening file.\n");
        return;
    }

    fseek(file, (position - 1) * sizeof(Employee), SEEK_SET);
    Employee emp;
    if (fread(&emp, sizeof(Employee), 1, file) == 1) {
        printf("Employee at position %d:\n", position);
        printf("Number: %d\n", emp.num);
        printf("Salary: %.2f\n", emp.salary);
        printf("First Name: %s\n", emp.firstName);
        printf("Last Name: %s\n", emp.lastName);
        printf("Gender: %c\n", emp.gender);
        printf("Department: %s\n", emp.department);
        printf("Is Teacher: %s\n", emp.isTeacher ? "Yes" :
"No");
    } else {
        printf("No employee found at position %d.\n",
position);
    }

    fclose(file);
}

void countFemaleTeachers() {
    FILE *file = fopen(FILE_NAME, "rb");
    if (!file) {
        printf("Error opening file.\n");
        return;
    }

    int count = 0;
    Employee emp;
    while (fread(&emp, sizeof(Employee), 1, file)) {
        if (emp.gender == 'F' && emp.isTeacher) {
            count++;
        }
    }

    printf("Number of female teachers: %d\n", count);

    fclose(file);
}

```



```

int main() {
    int choice;

    do {
        printf("Menu: \n");
        printf("1. Create the file\n");
        printf("2. Enter 10 employees\n");
        printf("3. Display the employee at position 6\n");
        printf("4. Count the number of female teachers\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                createFile();
                break;
            case 2:
                enterEmployees(10);
                break;
            case 3:
                displayEmployeeAtPosition(6);
                break;
            case 4:
                countFemaleTeachers();
                break;
            case 5:
                printf("Exiting... \n");
                break;
            default:
                printf("Invalid choice. Please try again. \n");
        }
    } while (choice != 5);

    return 0;
}

```

Explanation:

- a. Structure definition: The Employee structure holds the details of an employee.
- b. File operations:
 - createFile: Creates a new binary file.
 - enterEmployees: Appends new employees' data to the binary file.
 - displayEmployeeAtPosition: Displays the employee at a specific position in the file.

- countFemaleTeachers: Counts and displays the number of female teachers in the file.
- c. Menu: Provides a menu-driven interface to perform the operations.

Exercise 6:

- a. Counting characters and lines :

```
#include <stdio.h>

void count_chars_lines(const char *filename, int *char_count,
int *line_count) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        printf("Error opening file.\n");
        return;
    }

    int ch;
    *char_count = 0;
    *line_count = 0;

    while ((ch = fgetc(file)) != EOF) {
        (*char_count)++;
        if (ch == '\n') {
            (*line_count)++;
        }
    }

    fclose(file);
}

int main() {
    char filename[100];
    int char_count, line_count;

    printf("Enter the filename: ");
    scanf("%s", filename);

    count_chars_lines(filename, &char_count, &line_count);

    printf("Number of characters: %d\n", char_count);
    printf("Number of lines: %d\n", line_count);

    return 0;
}
```

b. Counting palindromes and average word length :

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int is_palindrome(const char *word) {
    int len = strlen(word);
    for (int i = 0; i < len / 2; i++) {
        if (tolower(word[i]) != tolower(word[len - i - 1])) {
            return 0;
        }
    }
    return 1;
}

void palindr_and_avg_word_length(const char *filename, int
*palindrome_count, float *avg_word_length) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        printf("Error opening file.\n");
        return;
    }

    char word[100];
    int word_count = 0;
    int total_word_length = 0;
    *palindrome_count = 0;

    while (fscanf(file, "%s", word) != EOF) {
        word_count++;
        total_word_length += strlen(word);
        if (is_palindrome(word)) {
            (*palindrome_count)++;
        }
    }

    if (word_count > 0) {
        *avg_word_length = (float)total_word_length / word_count;
    } else {
        *avg_word_length = 0;
    }

    fclose(file);
}
```

```

int main() {
    char filename[100];
    int palindrome_count;
    float avg_word_length;

    printf("Enter the filename: ");
    scanf("%s", filename);

    palindr_and_avg_word_length(filename, &palindrome_count,
    &avg_word_length);

    printf("Number of palindromes: %d\n", palindrome_count);
    printf("Average word length: %.2f\n", avg_word_length);

    return 0;
}

```

c. Deleting a Specific Line :

```

#include <stdio.h>
#include <stdlib.h>

void delete_line(const char *filename, int line_to_delete) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        printf("Error opening file.\n");
        return;
    }

    FILE *temp_file = fopen("temp.txt", "w");
    if (!temp_file) {
        printf("Error creating temporary file.\n");
        fclose(file);
        return;
    }

    char line[256];
    int current_line = 1;

    printf("Content before deletion:\n");
    rewind(file); // Move to the beginning of the file
    while (fgets(line, sizeof(line), file)) {
        printf("%s", line);
        if (current_line != line_to_delete) {
            fputs(line, temp_file);
        }
        current_line++;
    }
}

```

```

    fclose(file);
    fclose(temp_file);

    remove(filename);
    rename("temp.txt", filename);

    printf("\nContent after deletion:\n");
    file = fopen(filename, "r");
    while (fgets(line, sizeof(line), file)) {
        printf("%s", line);
    }

    fclose(file);
}

int main() {
    char filename[100];
    int line_to_delete;

    printf("Enter the filename: ");
    scanf("%s", filename);

    printf("Enter the line number to delete: ");
    scanf("%d", &line_to_delete);

    delete_line(filename, line_to_delete);

    return 0;
}

```

Exercise 7:

✚ The proposed algorithm is as follows :

Algorithm EmptyFile :

```

Var F: file of integers;
    X, S, Nb: integer;
    M: real;

```

```

Procedure SupD(In/Out: F: Fent);

```

```

-----

```

```

Begin
    Assign(F, 'ESSAI.DAT');
    Rewind(F);

```

```

While not EOF(F) Do
  SupD(F);
  Rewind(F);

  S ← 0; Nb ← 0;

  While not EOF(F) Do
    Read(F, X);
    S ← S + X;
    Nb ← Nb + 1;
  End;

  If Nb ≠ 0 Then
    M ← S / Nb;
    Write('Average = ', M);
  Else
    Write('The file is empty');
  EndIf;
End;

Close(F);
End.

```

✚ The proposed C program is as follows :

```

#include <stdio.h>
void SupD(FILE *F) {
  FILE *G;
  int x;

  rewind(F);

  if (fscanf(F, "%d", &x) != EOF) { // Check if file is not
empty
    G = fopen("G.dat", "w+");
    if (G == NULL) {
      printf("Error creating temporary file.\n");
      return;
    }

    int prev_x = x;
    while (fscanf(F, "%d", &x) != EOF) {
      fprintf(G, "%d\n", prev_x);
      prev_x = x;
    }

    fclose(F);
    fclose(G);

```

```

// Replace F with G
F = fopen("ESSAI.DAT", "w+");
G = fopen("G.dat", "r");

while (fscanf(G, "%d", &x) != EOF) {
    fprintf(F, "%d\n", x);
}

fclose(G);
}

fclose(F);
}

void EmptyFile() {
    FILE *F;
    int x, S, Nb;
    float M;
    F = fopen("ESSAI.DAT", "r+");
    if (F == NULL) {
        printf("Error opening file.\n");
        return;
    }
    rewind(F);
    while (fscanf(F, "%d", &x) != EOF) {
        SupD(F);
        rewind(F);

        S = 0;
        Nb = 0;

        while (fscanf(F, "%d", &x) != EOF) {
            S += x;
            Nb++;
        }
        if (Nb != 0) {
            M = (float)S / Nb;
            printf("Average = %.2f\n", M);
        } else {
            printf("The file is empty.\n");
        }
        rewind(F);
    }
    fclose(F);
}

int main() {
    EmptyFile();
    return 0;
}

```

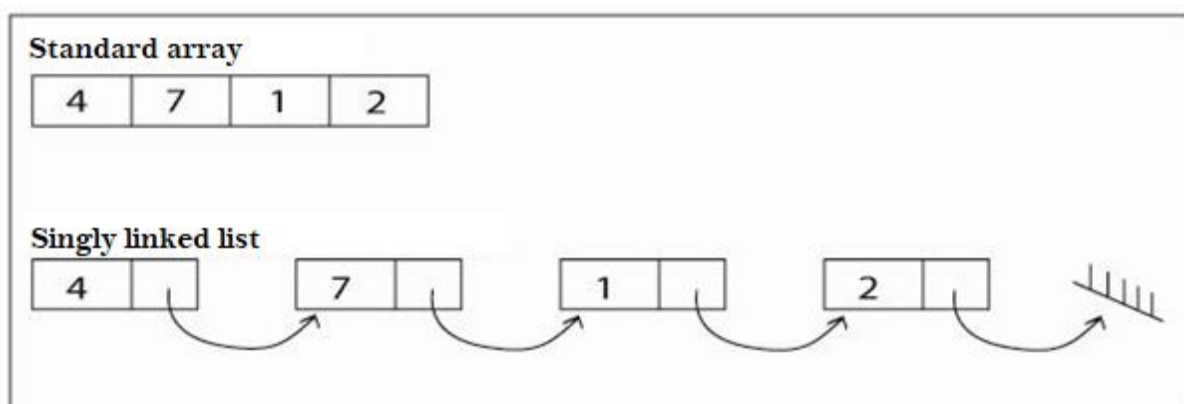
CHAPTER 3

LINKED LISTS

1. Generalities

As already seen in the first semester, when you create a standard array, its elements are placed contiguously in memory (one after the other). To create an array, you need to know its size. If you want to delete an element in the middle of the array, you need to temporarily copy the elements, reallocate memory for the array, and then fill it starting from the deleted element. In short, these are resource-intensive manipulations.

On the other hand, a singly linked list offers more flexibility. In a singly linked list, each element (or node) contains a value and a pointer to the next element in the list. This structure allows for efficient insertion and deletion of elements without the need for contiguous memory allocation or extensive copying and reallocating of memory.



In a linked list, unlike arrays, the elements of a list are distributed in memory (not contiguous) and linked together by pointers (to maintain the sequence of data). You can add and remove elements from a linked list at any position, at any time, without having to recreate the entire list. This is what marks the advantage of linked lists over arrays.

2. Comparison between Arrays and Linked Lists

Arrays are best suited for scenarios where frequent access by index is needed and the size of the dataset is known and fixed. Linked Lists are advantageous for scenarios where frequent insertion and deletion operations are required, and the dataset size can vary dynamically.

Arrays:

- Contiguous Memory Allocation: Elements are stored in contiguous memory locations.
- Fixed Size: The size of the array is fixed at the time of declaration.
- Efficient Indexing: Accessing elements by index is efficient, with a time complexity of $O(1)$.

- Insertion/Deletion: Inserting or deleting elements can be costly as it may require shifting elements, with a time complexity of $O(n)$.
- Memory Allocation: Requires reallocation of memory if the size needs to change.

Linked Lists:

- Non-contiguous Memory Allocation: Elements are distributed in memory and connected by pointers.
- Dynamic Size: The size of the list can grow or shrink dynamically.
- Efficient Insertion/Deletion: Inserting or deleting elements is efficient and can be done at any position, with a time complexity of $O(1)$ for insertion/deletion operations if the position is known.
- Inefficient Indexing: Accessing elements by index requires traversal from the head of the list, with a time complexity of $O(n)$.
- Memory Overhead: Requires additional memory for storing pointers.

Arrays are best suited for scenarios where frequent access by index is needed and the size of the dataset is known and fixed. Linked Lists are advantageous for scenarios where frequent insertion and deletion operations are required, and the dataset size can vary dynamically. The following table summarizes the main differences between them:

Array	Linked List
<ul style="list-style-type: none"> To declare an array, its size must be known. 	<p>The creation of a list is dynamic, and its size does not need to be known. The list can have as many elements as memory allows.</p>
<ul style="list-style-type: none"> The address of the first element is known, e.g., <code>array[0]</code>. 	<p>The address of the first element of a list is stored in the head of the list.</p>
<ul style="list-style-type: none"> It is possible to directly access the <i>i</i>-th element of an array, e.g., <code>array[i]</code>. 	<p>To reach an element in a list, you must traverse the list from the beginning.</p>
<ul style="list-style-type: none"> To delete or add an element in an array, a new array must be created and the old one deleted. 	<p>To add an element to a list, you need to call <code>malloc</code> to allocate memory. To delete an element from a list, call <code>free</code> to release the memory.</p>

3. Dynamic memory allocation

Dynamic memory allocation allows for reserving memory space for a program at runtime. To understand this principle, we will use the concept of an array. An array is a collection of elements stored in contiguous memory locations.

Elements	3	12	10	4	2	1	15	30
Indexes	0	1	2	3	4	5	6	7
Array size	: 8							
First index	: 0							
Last index	: 7							

As we can see, the length (size) of the array above is 8. For example, if we want to keep only 5 elements in this array, the remaining 3 indexes are just wasting memory. Therefore, it is necessary to reduce the length (size) of the array from 8 to 5. In another example, where we want to add 3 elements to an 8-element array with all 8 indexes filled, it is necessary to add 3 more elements to this array. In this case, 3 additional indexes are required. So, the length (size) of the array must be changed from 8 to 11. This procedure is called dynamic memory allocation.

Therefore, dynamic memory allocation can be defined as a procedure in which the size of a data structure (such as arrays) is changed during execution.

C provides certain functions to accomplish these tasks. There are 4 library functions provided by C, defined under the header file `<stdlib.h>`, to facilitate dynamic memory allocation in C programming. They are: **malloc()**, **calloc()**, **free()**, and **realloc()**. In this course, we will focus on 2 functions: `malloc()` and `free()`.

3.1 malloc() :

The `malloc` function, or "memory allocation," is used to dynamically allocate a single large block of memory with the specified size. It returns a void pointer, which can be cast to a pointer of any type.

```
ptr = (type*) malloc(byte-size);
```

where :

- type: the pointer type.
- byte-size: the size of the block.

Example 1:

```
int *ptr;
ptr = (int*) malloc(10 * sizeof(int));
```

Since the size of int is 4 bytes, this statement allocates 40 bytes of memory. The pointer ptr holds the address of the first byte in the allocated memory. If the space is insufficient, the allocation fails and returns a NULL pointer.

Example 2 :

The code demonstrates how to dynamically allocate, use, and deallocate memory in C :

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *tab; // Pointer to hold the address of the allocated
memory
    int i;    // Loop variable

    // Allocate memory for an array of 5 integers
    tab = (int *)malloc(5 * sizeof(int));

    // Check if the memory allocation was successful
    if (tab == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    } else {
        // Inform that memory has been successfully allocated
        printf("Memory successfully allocated with malloc\n");

        // Initialize the allocated memory with values 0 to 4
        for (i = 0; i < 5; i++) {
            *(tab + i) = i; // or tab[i] = i;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < 5; i++) {
            printf("%d, ", *(tab + i)); // or printf("%d, ",
tab[i]);
        }
        printf("\n"); // Print a newline after the array
elements
    }

    // Free the allocated memory to avoid memory leaks
    free(tab);

    return 0;
}
```

Explanation:

a. Memory Allocation:

`tab = (int *)malloc(5 * sizeof(int));` allocates memory for an array of 5 integers. The `sizeof(int)` determines the size of an `int` in bytes. This ensures that the allocated memory can hold 5 integers.

b. Check for Allocation Success:

`if (tab == NULL)` checks if the `malloc` function successfully allocated the memory. If it returns `NULL`, the program prints an error message and exits.

c. Memory Initialization:

The `for` loop initializes the allocated memory with the values 0 through 4. `*(tab + i) = i;` is a way to access and set the value at each index, which is equivalent to `tab[i] = i;`

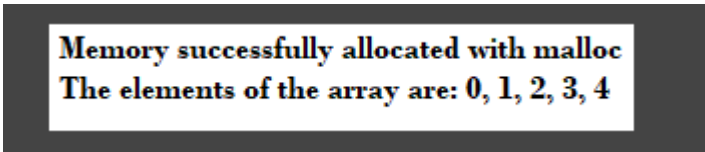
d. Print Array Elements:

The second `for` loop prints the elements of the array. It prints each value followed by a comma. `*(tab + i)` is used to access the value at each index, which is equivalent to `tab[i]`.

e. Free Memory:

`free(tab);` deallocates the previously allocated memory. This is important to avoid memory leaks, which occur when allocated memory is not released.

Execution:



```
Memory successfully allocated with malloc
The elements of the array are: 0, 1, 2, 3, 4
```

3.2 free()

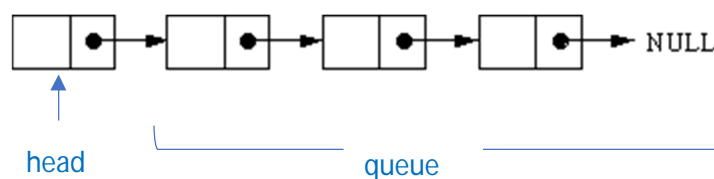
The `free` function is used to dynamically deallocate memory. Memory allocated using the `malloc()` and `calloc()` functions is not automatically deallocated. Therefore, the `free()` function is used whenever dynamic memory allocation occurs. This helps to reduce memory waste by releasing it.

The syntax is: `free(ptr);`

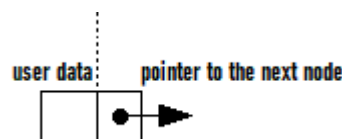
The `free()` function releases the memory that was allocated by `malloc()` or `calloc()`, making it available for future allocations. It's important to call `free()` for any dynamically allocated memory to prevent memory leaks, which occur when memory is allocated but not properly released.

4. Singly Linked Lists

A list is a data structure used to group data, represented by a sequence of nodes (also known as links, boxes, or list elements). Each node or link consists of two parts: its head, which corresponds to the most recently added element to the list, and its tail, whose last pointer points to an invalid address (NULL).



It is therefore only possible to traverse the list from the beginning to the end. The representation of an element in a singly linked list is as follows:



It is clear that a node represents a box or a structure composed of two elements that are not of the same type: the data to be stored and the pointer that will point to the next node if it exists. If there is no next node, the address will be NULL, indicating the end of the list.

As you can see, a linked list is a structured type, and defining a data structure allows us to create this linked list. In the C language, defining a linked list involves first defining the structure of a single node, which I will name `box`, as follows:

```
typedef struct box
{
    int data; // The data corresponding to the first part of the node, which I consider to be of type int
    struct box *next; // The next pointer points to the next node, so it is of type struct as well
} box;
```

Note: The definition of a node in a linked list always takes the same form, except that the type of the data field changes.

5. Operations on Singly Linked Lists

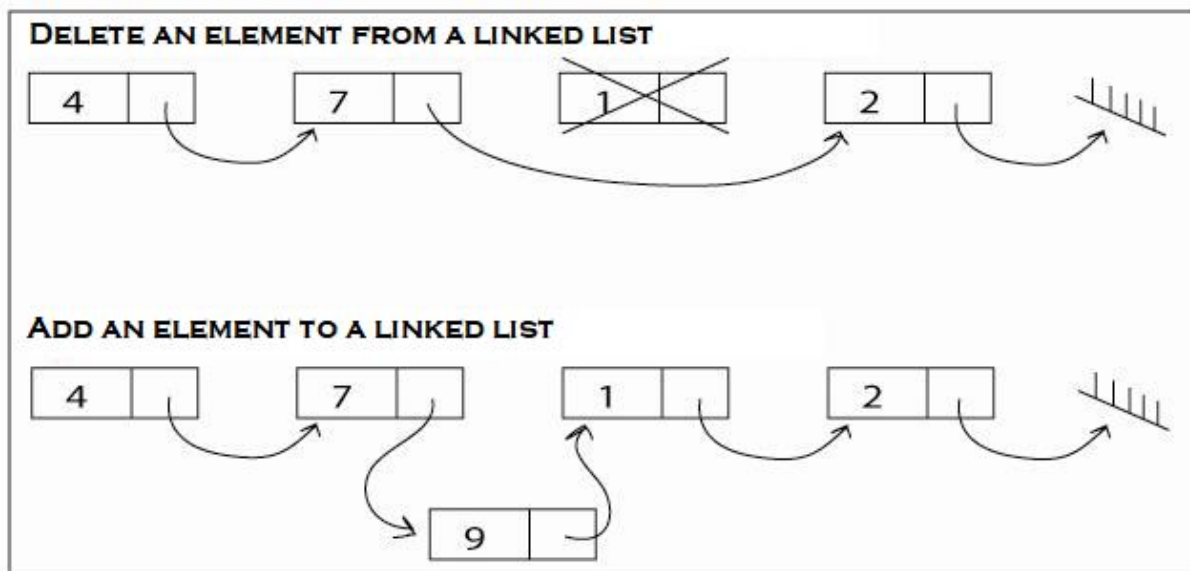
Here are some operations that can be performed on a list. It is best to use function calls:

- Create an empty list: `init()`
- Add an element `x` to a list `L`: `add(L, x)`
- Delete an element `x` from a list `L`: `delete(L, x)`
- Count the number of elements in a list `L`: The function `count(L)` is called to return the number of elements present in the list `L`.

These functions provide the basic operations needed to work with singly linked lists. Note that for inserting or deleting an element in this list, there are 3 cases:

- Insertion or deletion at the beginning of the list
- Insertion or deletion in the middle of the list
- Insertion or deletion at the end of the list

The general scheme for adding and removing an element in a singly linked list is as follows:



In a linked list, the last element points to nothing, meaning it is equal to NULL.

a. **Creating a new node:** To create a new node and initialize its data:

```
box* createNode(int value) {
    box* newNode = (box*)malloc(sizeof(box));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}
```

b. **Inserting a node at the beginning :** To insert a node at the beginning of the list:

```
void insertAtHead(box** head, int value) {
    box* newNode = createNode(value);
    newNode->next = *head;
    *head = newNode;
}
```

c. **Inserting a node at the end :** To insert a node at the end of the list:

```
void insertAtEnd(box** head, int value) {
    box* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
    } else {
        box* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}
```

d. **Traversing the List:** To print all nodes in the list:

```
void printList(box* head) {
    box* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```


e. **Finding a node:** To find a node with a specific value:

```
box* findNode(box* head, int value) {
    box* temp = head;
    while (temp != NULL) {
        if (temp->data == value) {
            return temp;
        }
        temp = temp->next;
    }
    return NULL; // Node not found
}
```

f. **Deleting a node:** To delete a node with a specific value:

```
void deleteNode(box** head, int value) {
    box *temp = *head, *prev = NULL;

    // If the head node itself holds the value to be deleted
    if (temp != NULL && temp->data == value) {
        *head = temp->next; // Change head
        free(temp); // Free old head
        return;
    }

    // Search for the node to be deleted
    while (temp != NULL && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }

    // If value was not present in the list
    if (temp == NULL) return;

    // Unlink the node from the linked list
    prev->next = temp->next;
    free(temp); // Free memory of the deleted node
}
```

6. Variants of linked lists

6.1 Doubly linked lists

6.1.1 Introduction

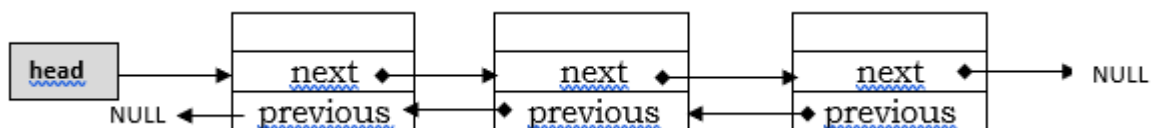
In a singly linked list, each node contains two parts:

1. A data part
2. The address of the node that follows it.

A list is called doubly linked if each node in the list contains:

1. A data part
2. The address of the node that follows it
3. The address of the node that precedes it

Therefore, a doubly linked list can be described as a singly linked list with an additional backward linkage, as indicated by the following diagram:



Note:

There is no element before the first node, so the value of the previous pointer is NULL.

There is no element after the last node, so the value of the next pointer is NULL.

6.1.2 Definition of the node structure

In C, defining a doubly linked list involves defining the structure of a single node, which I will name box, as follows:

```
typedef struct box
{
    int data;           // The data corresponding to the first part of the node,
                       // which I consider to be of type int
    struct box *next;  // The next pointer points to the following node,
                       // so it is also of type struct
    struct box *prev; // The previous pointer points to the preceding node,
                       // so it is also of type struct
} box;
```

Advantages:

- Allows traversal of the list in both directions.
- Saves time when navigating through the list.

6.1.3 Adding an element to a doubly linked list

To do this, we will use pass-by-address, so the add function is of type void. The code for adding an element in C is as follows:

```
void ajouter(liste *debut, int elem)
{
    box *b; // Create a pointer called b to allocate space for the new node dynamically
    b = (box *) malloc(sizeof(box)); // Call malloc to create a box (data + pointers)
    if (b == NULL) { // Check if malloc failed
        printf("Memory allocation failed.\n");
        return;
    }
    b->next = b->prev = NULL; // Initialize pointers to NULL
    b->data = elem; // Set the data of the box to elem
    b->next = *debut; // Point the new node's next to the current head
    if (*debut != NULL) // If the list is not empty
        (*debut)->prev = b; // Set the previous pointer of the current head to the new node
    *debut = b; // Update the head to point to the new node
}
```

Displaying a Doubly Linked List follows the same principle as a singly linked list.

Note: There are other variations of linked lists, such as singly and doubly circular linked lists. For more information, refer to the course materials.

Exercises

EXERCISE 1:

Develop a linked list of integers. Write the functions create, add, and main for a list of n elements.

EXERCISE 2:

Given a linked list of integers:

- a) Write a function to count positive values, zero values, and negative values.
- b) Write a recursive C function to search for a value val.
- c) Write a C function to calculate the sum of the elements in a list.

EXERCISE 3:

Given a sorted linked list of integer values, write a function to insert a value val into this list.

EXERCISE 4:

Given a doubly linked list of real numbers, write a C function to delete all negative values.

EXERCISE 5:

Write a function to reverse a linked list.

EXERCISE 6:

Given two lists of integers L1 and L2:

1. Write a function that checks if L1 and L2 are identical (contain the same elements in the same order).
2. Write a function that checks if L1 is a subset of L2 (all elements of L1 are found in L2, order does not matter).
3. Write a function that checks if L1 and L2 are disjoint (i.e., $L1 \cap L2 = \emptyset$).

Solutions

EXERCISE 1:

Develop a linked list of integers. Write the functions create, add, and main for a list of n elements.

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure of a node
typedef struct box {
    int data;
    struct box *next;
} box;

// Function to create a new node
box* createNode(int value) {
    box* newNode = (box*)malloc(sizeof(box));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Function to add a node to the end of the list
void addNode(box** head, int value) {
    box* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
    }
}
```

```

    } else {
        box* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

```

// Function to print the list

```

void printList(box* head) {
    box* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

```

// Main function

```

int main() {
    box* head = NULL;
    int n, value;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        printf("Enter value for node %d: ", i + 1);
        scanf("%d", &value);
        addNode(&head, value);
    }
}

```

```

printf("The linked list is: ");
printList(head);

return 0;
}

```

EXERCISE 2:

Given a linked list of integers:

a) Write a function to count positive values, zero values, and negative values.

```

void countValues(box* head, int* positive, int* zero, int*
negative) {
    *positive = *zero = *negative = 0;
    box* temp = head;
    while (temp != NULL) {
        if (temp->data > 0) {
            (*positive)++;
        } else if (temp->data == 0) {
            (*zero)++;
        } else {
            (*negative)++;
        }
        temp = temp->next;
    }
}

```

b) Write a recursive C function to search for a value val.

```

box* searchRecursive(box* head, int val) {
    if (head == NULL || head->data == val) {
        return head;
    }
}

```

```

        return searchRecursive(head->next, val);
    }

```

c) Write a C function to calculate the sum of the elements in a list.

```

int sumList(box* head) {
    int sum = 0;
    box* temp = head;
    while (temp != NULL) {
        sum += temp->data;
        temp = temp->next;
    }
    return sum;
}

```

EXERCISE 3:

Given a sorted linked list of integer values, write a function to insert a value val into this list.

```

void insertSorted(box** head, int val) {
    box* newNode = createNode(val);
    if (*head == NULL || (*head)->data >= val) {
        newNode->next = *head;
        *head = newNode;
    } else {
        box* current = *head;
        while (current->next != NULL && current->next->data <
val) {
            current = current->next;
        }
        newNode->next = current->next;
        current->next = newNode;
    }
}

```


EXERCISE 4:

Given a doubly linked list of real numbers, write a C function to delete all negative values.

Algorithm:

1. Initialize Pointers:
 - Set a pointer current to the head of the list.

2. Traverse the List:
 - a. While current is not NULL:
 - Check if current->data is less than 0.
 - If True:
 - Update the next pointer of current->prev to point to current->next (if current->prev is not NULL).
 - Update the prev pointer of current->next to point to current->prev (if current->next is not NULL).
 - If current is the head of the list, update the head to current->next.
 - Store the current node in a temporary pointer temp.
 - Move current to current->next.
 - Free the memory allocated for temp.
 - If False:
 - Move current to current->next.

3. End of List:
 - Continue the process until all nodes have been checked.

Program in C :

```
typedef struct box {  
    float data;  
    struct box *next;
```

```

    struct box *prev;
} box;

void deleteNegativeValues(box** head) {
    box* current = *head;
    while (current != NULL) {
        if (current->data < 0) {
            if (current->prev != NULL) {
                current->prev->next = current->next;
            } else {
                *head = current->next;
            }
            if (current->next != NULL) {
                current->next->prev = current->prev;
            }
            box* temp = current;
            current = current->next;
            free(temp);
        } else {
            current = current->next;
        }
    }
}

```

EXERCISE 5: Write a function to reverse a linked list.

Algorithm:

- a. Initialize Pointers:
 - Set prev to NULL.
 - Set current to the head of the list.
 - Set next to NULL.

b. Traverse the List:

- While current is not NULL:
 - Save the next node in next (next = current->next).
 - Reverse the current node's pointer (current->next = prev).
 - Move prev to current (prev = current).
 - Move current to next (current = next).

c. Update Head:

- Once the traversal is complete, set the head of the list to prev (which is now the new head of the reversed list).

Program in C :

```
void reverseList(box** head) {
    box* prev = NULL;
    box* current = *head;
    box* next = NULL;
    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head = prev;
}
```

EXERCISE 6:

a. Proposed solution :

1. Check if L1 and L2 are identical:
 - Compare the lengths of L1 and L2.
 - If the lengths are not equal, they are not identical.
 - If the lengths are equal, compare each corresponding element of L1 and L2.

- If all elements match, the lists are identical; otherwise, they are not.
2. Check if L1 is a subset of L2:
 - For each element in L1, check if it exists in L2.
 - If all elements of L1 are found in L2, then L1 is a subset of L2.
 - If any element of L1 is not found in L2, then L1 is not a subset.
 3. Check if L1 and L2 are disjoint:
 - For each element in L1, check if it exists in L2.
 - If any element of L1 is found in L2, the lists are not disjoint.
 - If no elements of L1 are found in L2, the lists are disjoint.

b. Proposed C Program :

```
#include <stdio.h>
#include <stdbool.h>

// Function to check if L1 and L2 are identical
bool areIdentical(int L1[], int size1, int L2[], int size2) {
    if (size1 != size2) return false;

    for (int i = 0; i < size1; i++) {
        if (L1[i] != L2[i]) {
            return false;
        }
    }
    return true;
}

// Function to check if L1 is a subset of L2
bool isSubset(int L1[], int size1, int L2[], int size2) {
    for (int i = 0; i < size1; i++) {
        bool found = false;
        for (int j = 0; j < size2; j++) {
            if (L1[i] == L2[j]) {
                found = true;
                break;
            }
        }
        if (!found) return false;
    }
    return true;
}

// Function to check if L1 and L2 are disjoint
bool areDisjoint(int L1[], int size1, int L2[], int size2) {
```

```

        for (int j = 0; j < size2; j++) {
            if (L1[i] == L2[j]) {
                return false;
            }
        }
    }
    return true;
}

int main() {
    int L1[] = {1, 2, 3};
    int L2[] = {1, 2, 3};
    int size1 = sizeof(L1) / sizeof(L1[0]);
    int size2 = sizeof(L2) / sizeof(L2[0]);

    // Check if L1 and L2 are identical
    if (areIdentical(L1, size1, L2, size2)) {
        printf("L1 and L2 are identical.\n");
    } else {
        printf("L1 and L2 are not identical.\n");
    }

    // Check if L1 is a subset of L2
    if (isSubset(L1, size1, L2, size2)) {
        printf("L1 is a subset of L2.\n");
    } else {
        printf("L1 is not a subset of L2.\n");
    }

    // Check if L1 and L2 are disjoint
    if (areDisjoint(L1, size1, L2, size2)) {
        printf("L1 and L2 are disjoint.\n");
    } else {
        printf("L1 and L2 are not disjoint.\n");
    }

    return 0;
}

```

Explanation:

- areIdentical: This function checks if L1 and L2 are identical by comparing their lengths and each corresponding element.
- isSubset: This function checks if all elements of L1 are found in L2, regardless of order.
- areDisjoint: This function checks if L1 and L2 have no common elements.

CHAPTER 5

STACKS AND QUEUES

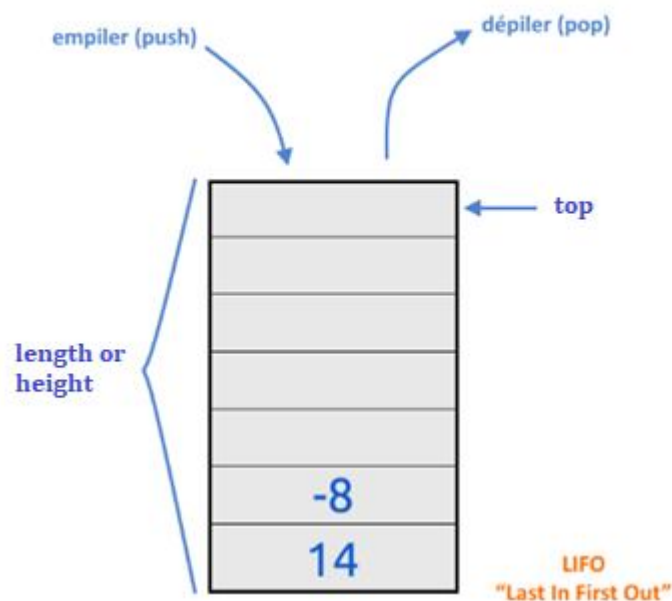
1. Stack

1.1 Definition:

A stack is a linear data structure in which elements are added and removed from only one end, called the top of the stack. This structure follows a Last In First Out (LIFO) principle, where the last element added is the first one to be removed.

1.2 Principle:

Last In First Out (**LIFO**): This means that the most recently added item is the first to be removed.



1.3 Examples of applications:

- ✓ Storing visited web pages: Browsers use a stack to keep track of the pages visited. When you press the back button, the browser pops the top page off the stack.
- ✓ Evaluating arithmetic expressions: Stacks are used to parse and evaluate expressions, especially in postfix or prefix notation.
- ✓ Function calls: Most programming languages use a stack to manage function calls. When a function is called, its execution context is pushed onto the stack. When the function returns, the context is popped off the stack.

1.4 Stack representations :

A stack can be represented in two main ways: static and dynamic. Each representation has its own advantages and use cases.

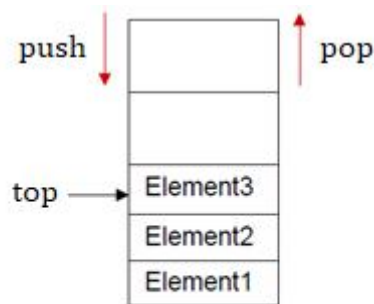
1.4.1 Static representation:

In a static representation, the stack is implemented using a fixed-size array. This approach is simple and straightforward, but it comes with the limitation of a fixed capacity. Once the array is full, no more elements can be added, and resizing the array dynamically is not possible. This method is useful when the maximum number of elements in the stack is known in advance.

Static stack declaration in C:

```
#define MAX 5
int stack[MAX];
int top = -1;
```

In a static representation, a stack is typically implemented using an array with a fixed size. This is useful when the maximum number of elements is known in advance.



a stack represented with an array of maximum size 5

The stack can be represented as a structure that contains a fixed-size array in C which stores the data of the stack and an index pointer which is used to track the top element of the stack.

Example:

```
struct stack {
    type arr[MAX_SIZE];
    int top;
}
```

We can use a utility function initialize the stack array along with the top pointer. The initial value of the top should be -1 representing that there are currently no elements in the stack.

Max size of the stack can be defined as per our requirements.

Basic Operations on a Stack:

Following are some basic operations in the stack that make it easy to manipulate the stack data structure:

1. Push: Add an element to the top of the stack.
2. Pop: Remove the element from the top of the stack.
3. Peek/Top: View the element at the top of the stack without removing it.
4. IsEmpty: Check if the stack is empty.
5. IsFull: Check if the stack is full (only applicable in static representation).

1. isFull Function :

The isFull() function provides the information about whether the stack have some space left or it is completely full. We know that the max capacity of the stack is MAX_SIZE elements. So, the max value of top can be MAX_SIZE - 1.

Algorithm of Stack isFull :

If top >= MAX_SIZE - 1, return true.
Else return false.

2. isEmpty Function :

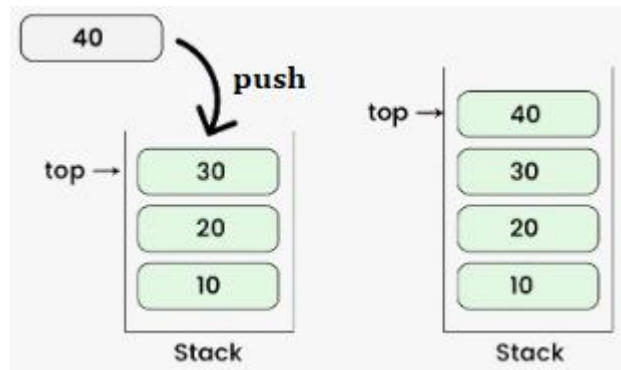
The isEmpty function will check whether the stack is empty or not. We know that when the stack is empty, the top is equal to -1.

Algorithm of Stack isEmpty :

If the top pointer == -1 return true
Else return false.

3. Push Function :

The push function will add (or push) an element to the stack. The edge case here will be when we try to add a new element when the stack is already full. It is called stack overflow and we have to check for it before inserted new element.



The push operation adds an element to the top of the stack. In a static implementation, it involves checking if the stack is full before adding the new element. In a dynamic implementation, if the stack is full, the stack can be resized to accommodate more elements.

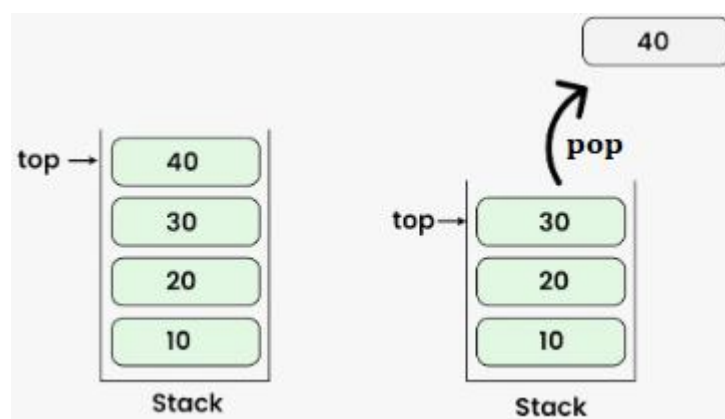
Algorithm of Stack Push :

Following is the algorithm for the push operation:

- a) Check whether if the stack is full.
- b) If stack is full then display the overflow message.
- c) If stack is not full then increment the top pointer.
- d) Add the new element to position pointed to by the top pointer.

3. Pop Function :

The push function will remove an element from the stack. One case that can occur here is when we try to remove the top using pop() function when the stack is already empty. Such condition is called stack underflow and can be easily checked.



The pop operation removes the top element from the stack. Before removing, it is essential to check if the stack is empty to avoid underflow errors. In both static and dynamic implementations, the pop operation retrieves the top element and adjusts the stack's top pointer or index accordingly.

Algorithm of Stack Pop

Following is the algorithm for the pop operation:

- a) Check whether if stack is empty.
- b) If stack is empty then display the underflow message.
- c) If stack is not empty then remove the element at top position
- d) Decrement the top pointer of the stack.

4. top Function :

The peek function will return the topmost element of the stack in constant time. If the stack is empty it returns -1.

Algorithm for Stack Top Function

Following is the algorithm for top operation on the stack:

- a) Check whether the stack is empty.
- b) If it is empty, return -1.
- c) Else return, stack.data[top] element.

Push/Pop static implementation in C:

The following program demonstrates how to implement a stack in C using a static array :

```
#include <stdio.h>
#define MAX 5

int stack[MAX];
int top = -1;

// Function to push an element onto the stack
void push(int value) {
    if (top == MAX - 1) {
        printf("Stack overflow\n");
        return;
    }
    stack[++top] = value;
    printf("%d pushed onto stack\n", value);
}

// Function to pop an element from the stack
int pop() {
    if (top == -1) {
        printf("Stack underflow\n");
```

```

        return -1;
    }
    int value = stack[top--];
    printf("%d popped from stack\n", value);
    return value;
}

int main() {
    push(10);
    push(20);
    push(30);
    push(40);
    push(50);
    push(60); // This should show "Stack overflow"

    pop();
    pop();
    pop();
    pop();
    pop();
    pop(); // This should show "Stack underflow"

    return 0;
}

```

Basic stack operations with static array implementation in C :

This program demonstrates basic operations on a stack using a static array-based implementation. The program efficiently demonstrates the push, pop, and peek operations on a stack implemented using a static array. Here's what each part of the program does:

Initialization:

- The Stack structure is defined with an array data of size MAX (100) and an integer top to keep track of the top element's index.
- The initStack function initializes the stack by setting top to -1, indicating that the stack is empty.

Check Functions:

- isEmpty: Checks if the stack is empty by verifying if top is -1.
- isFull: Checks if the stack is full by verifying if top is equal to MAX - 1 (99).

Push Operation:

- The push function adds a new element to the stack if it is not full. The top index is incremented, and the value is added at that index in the data array. If the stack is full, it prints a message "Stack is full!"

Pop Operation:

- The pop function removes and returns the top element of the stack if it is not empty. The top index is decremented to effectively remove the element from the stack. If the stack is empty, it prints a message "Stack is empty!" and returns -1.

Peek Operation:

- The peek function returns the top element of the stack without removing it, but only if the stack is not empty. If the stack is empty, it prints a message "Stack is empty!" and returns -1.

Main Function:

- The main function initializes a stack, pushes three elements (10, 20, 30) onto it, and then demonstrates the peek operation by printing the top element.
- It then pops and prints all the elements in the stack in LIFO (Last In, First Out) order until the stack is empty.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

typedef struct {
    int data[MAX];
    int top;
} Stack;

// Function to initialize the stack
void initStack(Stack *s) {
    s->top = -1;
}

// Function to check if the stack is empty
int isEmpty(Stack *s) {
    return s->top == -1;
}

// Function to check if the stack is full
int isFull(Stack *s) {
    return s->top == MAX - 1;
}

// Function to add an element to the stack
void push(Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack is full!\n");
    } else {
```

```

        s->data[++s->top] = value;
    }
}

// Function to remove an element from the stack
int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
        return -1;
    } else {
        return s->data[s->top--];
    }
}

// Function to view the top element of the stack
int peek(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
        return -1;
    } else {
        return s->data[s->top];
    }
}

// Main function to demonstrate stack operations
int main() {
    Stack s;
    initStack(&s);
    push(&s, 10);
    push(&s, 20);
    push(&s, 30);
    printf("Top element is %d\n", peek(&s));
    printf("Stack elements: \n");
    while (!isEmpty(&s)) {
        printf("%d\n", pop(&s));
    }
    return 0;
}

```

In this example:

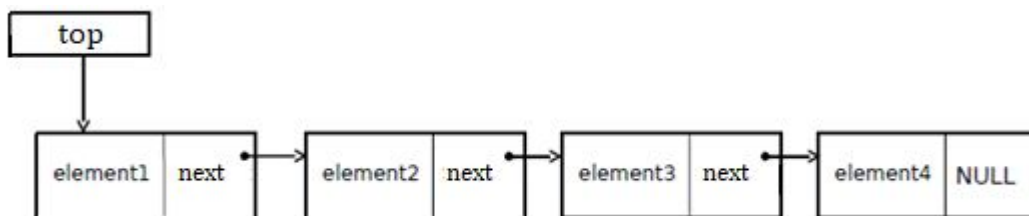
- We define a Stack structure with an array data and an integer top to keep track of the top element.
- We provide functions to initialize the stack, check if it's empty or full, push elements onto the stack, pop elements off the stack, and peek at the top element.
- The main function demonstrates how to use these stack operations.

Output Example:

Top element is 30
Stack elements:
30
20
10

1.4.2 Dynamic representation:

In a dynamic representation, the stack is implemented using dynamic memory allocation. This allows the stack to grow and shrink as needed, making it more flexible and efficient in terms of memory usage. Functions such as malloc, realloc, and free are used to manage the memory dynamically. This approach is beneficial when the number of elements is not known beforehand or can vary significantly.



Dynamic stack structure and initialization in C :

```
typedef struct {
    int *data;
    int top;
    int capacity;
} Stack;

void initStack(Stack *s, int capacity) {
    s->capacity = capacity;
    s->top = -1;
    s->data = (int *)malloc(s->capacity * sizeof(int));
}
```

Push/pop dynamic implementation :

This program demonstrates the dynamic implementation of a stack using a pointer-based array. This program effectively demonstrates a dynamic stack implementation that can grow as needed, unlike the static stack that has a fixed size. Here's what each part of the program does:

Dynamic Stack Structure:

- The Stack structure is defined with a pointer data that will point to the dynamically allocated array, an integer top to track the top element's index, and an integer capacity to represent the current capacity of the stack.

Initialization:

- The initStack function initializes the stack by setting top to -1 (indicating the stack is empty) and dynamically allocating memory for the array data based on the initial capacity provided. If the memory allocation fails, the program exits with an error message.

Resizing the Stack:

- The resize function is called when the stack is full. It doubles the stack's capacity and reallocates memory for the data array to accommodate more elements. If memory reallocation fails, the program exits with an error message.

Push Operation:

- The push function adds a new element to the stack. If the stack is full (i.e., top equals capacity - 1), it calls the resize function to increase the capacity. The value is then added to the stack, and top is incremented. The program prints a message indicating that the value has been pushed onto the stack.

Pop Operation:

- The pop function removes and returns the top element of the stack. If the stack is empty (i.e., top is -1), it prints "Stack underflow" and returns -1. If the stack is not empty, the top value is removed, top is decremented, and the value is returned and printed.

Freeing Memory:

- The freeStack function frees the dynamically allocated memory for the stack's data array, sets the pointer to NULL, and resets the top and capacity values.

Main Function:

- The main function initializes a stack with an initial capacity of 2 and performs several push operations. When a third element is pushed onto the stack, the resize function is triggered to double the stack's capacity.
- The function then performs several pop operations, demonstrating the stack's LIFO (Last In, First Out) behavior. An attempt to pop from an empty stack shows the "Stack underflow" message.
- Finally, the freeStack function is called to free the dynamically allocated memory.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int *data;
    int top;
    int capacity;
} Stack;

// Function to initialize the stack
void initStack(Stack *s, int capacity) {
    s->capacity = capacity;
    s->top = -1;
    s->data = (int *)malloc(s->capacity * sizeof(int));
    if (s->data == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
}

// Function to resize the stack
void resize(Stack *s) {
    s->capacity *= 2;
    s->data = (int *)realloc(s->data, s->capacity *
sizeof(int));
    if (s->data == NULL) {
        printf("Memory reallocation failed!\n");
        exit(1);
    }
}

// Function to push an element onto the stack
void push(Stack *s, int value) {
    if (s->top == s->capacity - 1) {
        resize(s);
    }
    s->data[++s->top] = value;
    printf("%d pushed onto stack\n", value);
}
```

```

// Function to pop an element from the stack
int pop(Stack *s) {
    if (s->top == -1) {
        printf("Stack underflow\n");
        return -1;
    }
    int value = s->data[s->top--];
    printf("%d popped from stack\n", value);
    return value;
}

// Function to free the allocated memory
void freeStack(Stack *s) {
    free(s->data);
    s->data = NULL;
    s->top = -1;
    s->capacity = 0;
}

int main() {
    Stack s;
    initStack(&s, 2); // Initialize stack with initial
    capacity of 2

    push(&s, 10);
    push(&s, 20);
    push(&s, 30); // This will trigger a resize

    pop(&s);
    pop(&s);
    pop(&s);
    pop(&s); // This should show "Stack underflow"

    freeStack(&s); // Free the allocated memory

    return 0;
}

```

Output Example:

```

10 pushed onto stack
20 pushed onto stack
30 pushed onto stack
30 popped from stack
20 popped from stack
10 popped from stack
Stack underflow

```

Dynamic Stack implementation in C:

In this implementation, the stack structure and core operations are further refined to include additional features such as checking if the stack is empty or full, and a peek operation to view the top element without removing it.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int *data;
    int top;
    int capacity;
} Stack;

// Function to initialize the stack
void initStack(Stack *s, int capacity) {
    s->capacity = capacity;
    s->top = -1;
    s->data = (int *)malloc(s->capacity * sizeof(int));
    if (s->data == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
}

// Function to check if the stack is empty
int isEmpty(Stack *s) {
    return s->top == -1;
}

// Function to check if the stack is full
int isFull(Stack *s) {
    return s->top == s->capacity - 1;
}

// Function to resize the stack
void resize(Stack *s) {
    s->capacity *= 2;
    s->data = (int *)realloc(s->data, s->capacity *
sizeof(int));
    if (s->data == NULL) {
        printf("Memory reallocation failed!\n");
        exit(1);
    }
}

// Function to add an element to the stack
void push(Stack *s, int value) {
```

```

        if (isFull(s)) {
            resize(s);
        }
        s->data[++s->top] = value;
    }

// Function to remove an element from the stack
int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
        return -1;
    } else {
        return s->data[s->top--];
    }
}

// Function to view the top element of the stack
int peek(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty!\n");
        return -1;
    } else {
        return s->data[s->top];
    }
}

// Function to free the allocated memory
void freeStack(Stack *s) {
    free(s->data);
    s->data = NULL;
    s->top = -1;
    s->capacity = 0;
}

// Main function to demonstrate stack operations
int main() {
    Stack s;
    initStack(&s, 2); // Initialize stack with initial
    capacity of 2

    push(&s, 10);
    push(&s, 20);
    push(&s, 30); // This will trigger a resize

    printf("Top element is %d\n", peek(&s));

    printf("Stack elements: \n");
    while (!isEmpty(&s)) {
        printf("%d\n", pop(&s));
    }
}

```

```
        freeStack(&s); // Free the allocated memory
        return 0;
    }
```

In this code:

- The `initStack` function initializes the stack with an initial capacity specified by the user.
- The `resize` function doubles the stack's capacity when it is full.
- The `push`, `pop`, `peek`, `isEmpty`, and `isFull` functions allow manipulation of the stack.
- The `freeStack` function frees the dynamically allocated memory for the stack.

Output Example:

```
Top element is 30
Stack elements:
30
20
10
```

This implementation demonstrates how dynamic memory management enhances the flexibility and efficiency of stack operations in C.

2. Queue

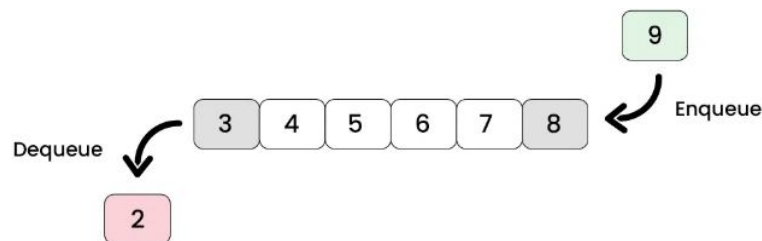
2.1 Definition:

A queue is a linear data structure that follows the First In, First Out (**FIFO**) principle. This means that the first element added to the queue will be the first one to be removed. Queues are analogous to lines of people waiting for service, where the first person in line is the first to be served, and new arrivals join the back of the line.

2.2 Principle:

The fundamental principle of a queue is FIFO. Elements are added to the rear (enqueue) and removed from the front (dequeue). This order ensures that the oldest elements are processed first, making queues ideal for scenarios where order and fairness are important. The two main operations in a queue are:

- a. Enqueue: Adding an element to the rear of the queue.
- b. Dequeue: Removing an element from the front of the queue.



2.3 Examples of applications:

Queues are widely used in various real-world and computational applications, including:

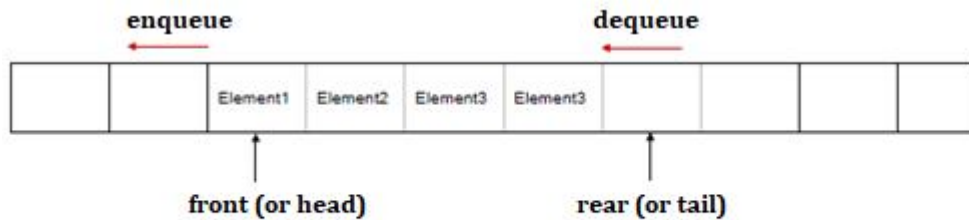
- ✓ Scheduling: In operating systems, queues manage processes by scheduling tasks according to their arrival time.
- ✓ Print Spooling: Print jobs are queued so that the first submitted document is printed first.
- ✓ Breadth-First Search (BFS): Queues are used in graph traversal algorithms like BFS to explore nodes level by level.
- ✓ Network Buffers: Data packets are queued in networking devices like routers and switches before processing.

2.4 Queue representations:

Queues can be represented in several ways, each suited to different use cases:

- a. **Static representation (array-based queue):** This representation uses a fixed-size array to store queue elements. The front and rear pointers

track where elements should be dequeued and enqueued. While simple to implement, it has a limited capacity and can lead to wasted space if not managed correctly, such as when the queue becomes full but still has empty slots due to elements being dequeued.



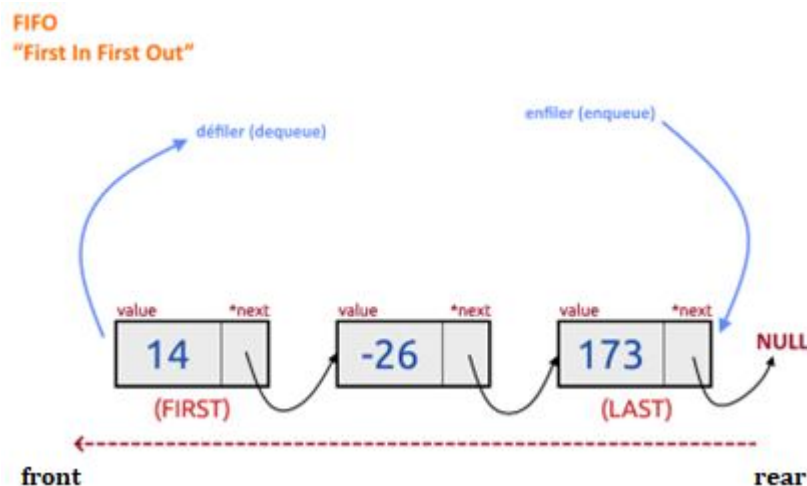
Queue represented by an array with a maximum size of 10

- ❖ Declaration of a queue in C, represented by an array with a maximum size of 10:

```
#define MAX 10

typedef struct {
    int data[MAX];
    int front;
    int rear;
} Queue;
```

- Dynamic representation (Linked list-based queue):** A more flexible approach is to use a linked list, where each node represents an element in the queue. This dynamic representation allows the queue to grow and shrink as needed, with no fixed size, making it ideal for situations where the number of elements is not known in advance or can vary significantly. The head pointer represents the front, and the tail pointer represents the rear of the queue, facilitating efficient enqueue and dequeue operations.



- ❖ Declaration for a queue using a linked list in C:

```
typedef struct Node {
    int data;
    struct Node* next;
} Node;

typedef struct {
    Node* front;
    Node* rear;
} Queue;
```

Each representation has its own advantages and trade-offs, making it essential to choose the right one based on the specific requirements of the application.

2.5 Queue operations:

Queues are a fundamental data structure used to manage and organize elements in a sequential manner. They operate on a First-In-First-Out (FIFO) principle, where the first element added is the first one to be removed. Managing queues involves several core operations: creating the queue, checking if it is empty or full, enqueueing a new element (adding it to the end), and dequeuing an element (removing it from the front). These operations can be implemented using different representations in C, such as using a fixed-size array or a dynamic linked list. Here's how you can implement these operations:

1. Create the Queue: Initialize the queue to prepare it for use.
2. Check if the Queue is Empty: Determine if the queue has no elements.
3. Check if the Queue is Full: Determine if the queue has reached its maximum capacity.
4. Enqueue a New Element: Add a new element to the end of the queue.
5. Dequeue an Element: Remove the element from the front of the queue.

In this section, we provide a comprehensive overview of queue operations using both static and dynamic representations in C, offering flexibility and efficiency depending on the application's needs.

2.5.1 Using an array representation :

Create the Queue:

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 10

typedef struct {
    int data[MAX];
    int front;
    int rear;
} Queue;

void initQueue(Queue *q) {
    q->front = 0;
    q->rear = -1;
}
```

Check if the Queue is Empty:

```
int isEmpty(Queue *q) {
    return q->rear < q->front;
}
```

Check if the Queue is Full:

```
int isFull(Queue *q) {
    return q->rear == MAX - 1;
}
```

Enqueue an Element:

```
void enqueue(Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue is full!\n");
    } else {
        q->data[++q->rear] = value;
    }
}
```

Dequeue an Element:

```
int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return -1;
    } else {
        return q->data[q->front++];
    }
}
```

2.5.2 Using a dynamic linked list representation :

Create the Queue:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
    struct Node *next;
} Node;

typedef struct {
    Node *front;
    Node *rear;
} Queue;

void initQueue(Queue *q) {
    q->front = NULL;
    q->rear = NULL;
}
```

Check if the Queue is Empty:

```
int isEmpty(Queue *q) {
    return q->front == NULL;
}
```

Enqueue an Element:

```
void enqueue(Queue *q, int value) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = NULL;
    if (isEmpty(q)) {
        q->front = newNode;
    } else {
        q->rear->next = newNode;
    }
    q->rear = newNode;
}
```

Dequeue an Element:

```
int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return -1;
    } else {
```

```

    Node *temp = q->front;
    int value = temp->data;
    q->front = q->front->next;
    if (q->front == NULL) {
        q->rear = NULL;
    }
    free(temp);
    return value;
}
}

```

Implementing queues using arrays offers simplicity but with fixed size constraints, while dynamic linked lists provide flexibility and scalability at the cost of more complex memory management. Each method has its benefits, and choosing the right one depends on the specific needs of your application. Understanding both approaches equips you to handle various data processing tasks effectively.

Exercises

EXERCISE 1:

Given a stack of integers, write a function to remove all negative values.

EXERCISE 2:

- a. Given a stack of real numbers, write a function to search for a value and check if it exists in the stack. If it does not exist, insert it.
- b. Rewrite this function for the case where the stack is sorted.

EXERCISE 3:

Given two stacks P1 and P2 containing ordered integer values, write a function to create another stack P3 containing the elements of stack P1 that are not present in P2.

EXERCISE 4: Queues

Write a C program to manage a queue:

1. Write the primitives enqueue, dequeue, and displayQueue.
2. Write a program to:
 - a. Create a queue with 5 elements (5, 3, 7, 2, 9).
 - b. Perform the following tasks: one dequeue, one enqueue of 15, one dequeue, one enqueue of 4, one dequeue, and one more dequeue.
 - c. Display the remaining elements after each task.

EXERCISE 5 :

Given a matrix A of integers with dimensions $N \times M$, write an algorithm to generate two lists from this matrix:

1. The first list contains the minimum values from each row (FIFO - First In, First Out).
2. The second list contains the sum of each column (LIFO - Last In, First Out).

Solutions

EXERCISE 1:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Define the structure of a node
typedef struct boite {
    int elem;
    struct boite *sui v;
} boite;

// Define the structure of the stack
typedef struct {
    boite *sommet;
    int nbr_elements; // number of elements in the stack
} pile;

// Function to create and initialize a stack
pile cree_init() {
    pile p;
    p.sommet = NULL;
    p.nbr_elements = 0;
    return p;
}

// Function to check if the stack is empty
bool vide(pile p) {
    return p.sommet == NULL;
}

// Function to return the top element of the stack
int sommet(pile p) {
    if (vide(p)) {
        printf("The stack is empty... no top element\n");
        return -1;
    }
    return p.sommet->elem;
}

// Function to display the stack
void affiche(pile p) {
    boite *temp = p.sommet;
    while (temp != NULL) {
        printf("%d ", temp->elem);
    }
}
```

```

        temp = temp->sui v;
    }
    printf("\n");
}

// Function to push an element onto the stack
void empiler(pile *p, int elem) {
    boi te *new_node = (boi te *)mal loc(si zeof(boi te));
    if (new_node == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    new_node->el em = el em;
    new_node->sui v = p->sommet;
    p->sommet = new_node;
    p->nbr_el ements++;
}

// Function to remove all negative values from the stack
void sup_neg(pile *p) {
    boi te *temp = p->sommet;
    boi te *temp_prec = NULL;

    while (temp != NULL) {
        if (temp->el em < 0) {
            if (temp_prec == NULL) {
                // Removing the first element
                p->sommet = temp->sui v;
            } else {
                // Removing any other element
                temp_prec->sui v = temp->sui v;
            }
            boi te *temp1 = temp;
            temp = temp->sui v;
            free(temp1);
            p->nbr_el ements--;
        } else {
            temp_prec = temp;
            temp = temp->sui v;
        }
    }
}

// Main function
int main() {
    pile f1 = cree_init();
    int nbr_el em, n;

    printf("Enter the number of elements in the stack: ");

```

```

scanf("%d", &nbr_el em);

printf("\nPushing elements\n");
for (int i = 0; i < nbr_el em; i++) {
    scanf("%d", &n);
    empiler(&f1, n);
}

printf("\nNumber of elements in the stack: %d\n",
f1.nbr_el ements);
affiche(f1);

printf("\nRemoving negative numbers... \n");
sup_neg(&f1);

printf("\nRemain ing number of elements: %d\n",
f1.nbr_el ements);
affiche(f1);

return 0;
}

```

In the provided solution :

- Initialization: The stack is properly initialized with cree_init.
- Push: The empiler function adds elements to the stack.
- Display: The affiche function displays the stack's content.
- Remove Negatives: The sup_neg function effectively removes negative elements.

EXERCISE 2 :

a) Searching for a value and inserting if not found:

- affiche Function: The affiche function currently modifies the stack as it prints elements. It should use a temporary pointer to traverse the stack instead of modifying p.sommet.
- rech_val Function: The recursive call in rech_val does not properly return the result of the recursive function. This could lead to undefined behavior. You should ensure that the function returns the correct result in each case.
- Stack Traversal: In functions that traverse the stack (affiche, rech_val), use a temporary pointer instead of modifying the sommet pointer to avoid unintended changes to the stack structure.

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Define the structure of a node
typedef struct boite {
    float elem;
    struct boite *sui v;
} boite;

// Define the structure of the stack
typedef struct {
    boite *sommet;
    int nbr_elements;
} pile;

// Function to create and initialize a stack
pile cree_init() {
    pile p;
    p.sommet = NULL;
    p.nbr_elements = 0;
    return p;
}

// Function to check if the stack is empty
bool vide(pile p) {
    return p.sommet == NULL;
}

// Function to display the stack
void affiche(pile p) {
    if (vide(p)) {
        printf("Pile vide ... \n");
        return;
    }

    boite *temp = p.sommet; // Use a temporary pointer to
    traverse the stack
    while (temp != NULL) {
        printf("[%f]\n", temp->elem);
        temp = temp->sui v;
    }
}

// Function to push an element onto the stack
void empiler(pile *p, float elt) {
    boite *b = (boite *)mal loc(sizeof(boite));
    if (b == NULL) {
        printf("Erreur allocation...");
    }
}

```



```

        exit(EXIT_FAILURE);
    }
    b->elem = elt;
    b->sui v = p->sommet;
    p->sommet = b;
    p->nbr_el ements++;
}

// Function to pop an element from the stack
void depiler(pile *p) {
    if (vide(*p)) {
        printf("La pile est vide, impossible de dépiler\n");
        return;
    }
    boite *temp = p->sommet;
    p->sommet = p->sommet->sui v;
    free(temp);
    p->nbr_el ements--;
}

// Function to search for a value in the stack
bool rech_val(pile p, float v) {
    boite *temp = p.sommet;
    while (temp != NULL) {
        if (temp->elem == v) {
            return true;
        }
        temp = temp->sui v;
    }
    return false;
}

int main() {
    pile f1 = cree_init();
    int nbr_el em;
    float n, val;

    printf("Entrer le nombre d'éléments dans la liste: ");
    scanf("%d", &nbr_el em);

    printf("\n\tEmpiler\n");
    for (int i = 0; i < nbr_el em; i++) {
        scanf("%f", &n);
        empiler(&f1, n);
    }

    printf("\n");
    affiche(f1);
}

```

```

printf("\nEntrer la valeur à chercher: ");
scanf("%f", &val);

printf("\nDébut de recherche...\n");
if (rech_val(f1, val)) {
    printf("\n\tLa valeur %.2f existe\n", val);
} else {
    printf("\n\tLa valeur %.2f n'existe pas\n", val);
    empiler(&f1, val);
    printf("\nLa nouvelle pile\n");
    affiche(f1);
}

return 0;
}

```

b) Searching and inserting in a sorted stack:

- **rech_val Function:** You correctly handle the sorted nature of the stack by stopping the search when the current element is smaller than the value being searched.
- **inserer Function:** The insertion logic is mostly correct, but there are some edge cases where the code might fail, especially when the stack is empty or if the new element should be inserted at the end.
- **Edge Case Handling:** Ensure that all edge cases are handled, such as inserting into an empty stack, or inserting an element smaller than all existing elements.

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

```

// Define the structure of a node

```

typedef struct boite {
    float elem;
    struct boite *sui v;
} boite;

```

// Define the structure of the stack

```

typedef struct {
    boite *sommet;
    int nbr_elements;
} pile;

```

// Function to create and initialize a stack

```

pile cree_init() {
    pile p;

```

```

    p.sommet = NULL;
    p.nbr_elements = 0;
    return p;
}

// Function to check if the stack is empty
bool vide(pile p) {
    return p.sommet == NULL;
}

// Function to display the stack
void affiche(pile p) {
    if (vide(p)) {
        printf("Pile vide ... \n");
        return;
    }

    boite *temp = p.sommet;
    while (temp != NULL) {
        printf("[%f]\n", temp->elem);
        temp = temp->sui v;
    }
}

// Function to push an element onto the stack
void empiler(pile *p, float elt) {
    boite *b = (boite *)malloc(sizeof(boite));
    if (b == NULL) {
        printf("Erreur allocation...");
        exit(EXIT_FAILURE);
    }
    b->elem = elt;
    b->sui v = p->sommet;
    p->sommet = b;
    p->nbr_elements++;
}

// Function to pop an element from the stack
void depiler(pile *p) {
    if (vide(*p)) {
        printf("La pile est vide, impossible de dépiler\n");
        return;
    }
    boite *temp = p->sommet;
    p->sommet = p->sommet->sui v;
    free(temp);
    p->nbr_elements--;
}

```

```

// Function to search for a value in a sorted stack
bool rech_val(pile p, float v) {
    boi te *temp = p.sommet;
    while (temp != NULL) {
        if (temp->elem == v) {
            return true;
        } else if (temp->elem < v) {
            return false; // Since the stack is sorted, no
need to search further
        }
        temp = temp->sui v;
    }
    return false;
}

// Function to insert an element into a sorted stack
void inserer(pile *p, float elt) {
    boi te *b = (boi te *)mal loc(sizeof(boi te));
    if (b == NULL) {
        printf("Erreur allocation...");
        exit(EXIT_FAI LURE);
    }
    b->el em = elt;
    b->sui v = NULL;

    // Handle insertion when the stack is empty or the element
is larger than the top element
    if (vide(*p) || p->sommet->el em < elt) {
        empiler(p, elt);
        return;
    }

    boi te *temp = p->sommet;
    boi te *temp_prec = NULL;

    // Traverse to find the correct insertion point
    while (temp != NULL && temp->el em > elt) {
        temp_prec = temp;
        temp = temp->sui v;
    }

    // Insert the element in the correct position
    b->sui v = temp;
    if (temp_prec != NULL) {
        temp_prec->sui v = b;
    } else {
        p->sommet = b; // In case the element is the new top
element
    }
}

```

```

    p->nbr_elements++;
}

int main() {
    pile f1 = cree_init();
    int nbr_elem;
    float n, val;

    printf("Entrer le nombre d'éléments dans la liste: ");
    scanf("%d", &nbr_elem);

    printf("\n\tEmpiler\n");
    for (int i = 0; i < nbr_elem; i++) {
        scanf("%f", &n);
        empiler(&f1, n);
    }

    printf("\nLe nombre d'éléments est: %d\n",
f1.nbr_elements);
    affiche(f1);

    printf("\tEntrer la valeur à chercher: ");
    scanf("%f", &val);

    printf("\nDébut de recherche...\n");
    if (rech_val(f1, val)) {
        printf("La valeur %.2f existe\n", val);
    } else {
        printf("\tLa valeur %.2f n'existe pas\n", val);
        inserer(&f1, val);
        printf("\nLa nouvelle pile\n");
        affiche(f1);
    }

    return 0;
}

```

EXERCICE 3:

Main Flow:

- Populate the two stacks P1 and P2.
- Calculate the difference, storing it in P3.
- Display the elements of P3, which contains elements from P1 that are not present in P2.

```

#include <stdio.h>
#include <stdlib.h>

```

```

// Define the structure of a node
typedef struct boite {
    int elem;
    struct boite *sui v;
} boite;

// Define the structure of the stack
typedef struct {
    boite *somet;
    int nbr_elements;
} pile;
// Function to create and initialize a stack
pile cree_init() {
    pile p;
    p.somet = NULL;
    p.nbr_elements = 0;
    return p;
}
// Function to check if a stack is empty
bool vide(pile p) {
    return p.somet == NULL;
}

// Function to push an element onto the stack
void empiler(pile *p, int elt) {
    boite *b = (boite *)malloc(sizeof(boite));
    if (b == NULL) {
        printf("Erreur allocation...");
        exit(EXIT_FAILURE);
    }
    b->elem = elt;
    b->sui v = p->somet;
    p->somet = b;
    p->nbr_elements++;
}
// Function to pop an element from the stack
int depiler(pile *p) {
    if (vide(*p)) {
        printf("Erreur: pile vide\n");
        exit(EXIT_FAILURE);
    }
    boite *temp = p->somet;
    int elem = temp->elem;
    p->somet = p->somet->sui v;
    free(temp);
    p->nbr_elements--;
    return elem;
}

```

```

// Function to search for a value in the stack
bool rech_val(pile p, int v) {
    while (!vide(p)) {
        if (p.sommet->elem == v) return true;
        p.sommet = p.sommet->suiv;
    }
    return false;
}

// Function to create a stack P3 containing elements of P1 that
are not present in P2
pile difference(pile P1, pile P2) {
    pile P3 = cree_init();
    pile temp = cree_init();

    while (!vide(P1)) {
        int elt = depiler(&P1);
        if (!rech_val(P2, elt)) {
            empiler(&P3, elt);
        }
        empiler(&temp, elt); // Save element in temporary stack
    }

    // Restore the original P1 stack
    while (!vide(temp)) {
        empiler(&P1, depiler(&temp));
    }

    return P3;
}

// Function to display the stack
void affiche(pile p) {
    while (!vide(p)) {
        printf("[ %d ]\n", p.sommet->elem);
        p.sommet = p.sommet->suiv;
    }
}

int main() {
    pile P1 = cree_init();
    pile P2 = cree_init();
    pile P3;

    int nbr_elem, n;

    // Populate P1
    printf("Entrer le nombre elements de la pile P1: ");
    scanf("%d", &nbr_elem);

```

```

for (int i = 0; i < nbr_elem; i++) {
    printf("Entrez un entier pour P1: ");
    scanf("%d", &n);
    empiler(&P1, n);
}

// Populate P2
printf("Entrez le nombre elements de la pile P2: ");
scanf("%d", &nbr_elem);
for (int i = 0; i < nbr_elem; i++) {
    printf("Entrez un entier pour P2: ");
    scanf("%d", &n);
    empiler(&P2, n);
}

// Calculate the difference and store it in P3
P3 = difference(P1, P2);

// Display the result
printf("\nLes elements de P1 qui ne sont pas dans P2
sont: \n");
affiche(P3);

return 0;
}

```

Explanation:

- Structures: The `boite` structure defines the node in the stack, and the `pile` structure manages the stack.
- `cree_init` Function: Initializes an empty stack.
- `vide` Function: Checks if the stack is empty.
- `empiler` Function: Adds an element to the stack.
- `depiler` Function: Removes an element from the stack and returns it.
- `rech_val` Function: Searches for an element in the stack.
- `difference` Function: Compares stacks P1 and P2. If an element in P1 is not in P2, it adds it to P3. The original stack P1 is restored after comparison.
- `affiche` Function: Displays the elements of a stack.

EXERCICE 4:

1. Queue management primitives : The code for managing a queue with enqueue, dequeue, and display Queue functions:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Define the structure of a node
typedef struct node {
    int elem;
    struct node *next;
} node;

// Define the structure of the queue
typedef struct {
    node *head;
    node *tail;
    int max_size;
    int num_elements;
} queue;

// Function to create and initialize a queue
queue create_init() {
    queue q;
    q.head = q.tail = NULL;
    q.max_size = 5; // Set max size to 5 as specified in the
exercise
    q.num_elements = 0;
    return q;
}

// Check if the queue is empty
bool is_empty(queue q) {
    return q.head == NULL;
}

// Display the elements in the queue
void display(queue q) {
    if (is_empty(q)) {
        printf("Nothing to display... The queue is empty!\n");
        return;
    }
    node *temp = q.head;
    while (temp != NULL) {
        printf("%d --> ", temp->elem);
        temp = temp->next;
    }
}
```

```

    printf("NULL\n");
}

// Enqueue (add an element to the end of the queue)
void enqueue(queue *q, int v) {
    node *n = (node *)malloc(sizeof(node));
    if (n == NULL) {
        printf("Memory allocation error...\n");
        exit(EXIT_FAILURE);
    }
    n->elem = v;
    n->next = NULL;

    if (is_empty(*q)) {
        q->head = n;
        q->tail = n;
    } else {
        q->tail->next = n;
        q->tail = n;
    }
    q->num_elements++;
}

// Dequeue (remove the element from the front of the queue)
void dequeue(queue *q) {
    if (is_empty(*q)) {
        printf("Queue is empty...\n");
        return;
    }
    node *temp = q->head;
    if (q->head == q->tail) { // If the queue has only one
element
        q->head = q->tail = NULL;
    } else {
        q->head = q->head->next;
    }
    free(temp);
    q->num_elements--;
}

```

2. Program to perform tasks :

- a. Create a queue with 5 elements: (5, 3, 7, 2, 9) :

```

int main() {
    queue q1 = create_init();
    int elements[5] = {5, 3, 7, 2, 9};
    printf("Initial queue:\n");
}

```

```

    for (int i = 0; i < 5; i++) {
        enqueue(&q1, elements[i]);
    }
    display(q1);
    return 0;
}

```

b. Perform the specified tasks :

```

int main() {
    queue q1 = create_init();
    int elements[5] = {5, 3, 7, 2, 9};

    // Create the initial queue
    for (int i = 0; i < 5; i++) {
        enqueue(&q1, elements[i]);
    }
    printf("Queue after initialization:\n");
    display(q1);

    // 1. Dequeue (remove 5)
    dequeue(&q1);
    printf("After dequeue:\n");
    display(q1);

    // 2. Enqueue 15
    enqueue(&q1, 15);
    printf("After enqueue 15:\n");
    display(q1);

    // 3. Dequeue (remove 3)
    dequeue(&q1);
    printf("After dequeue:\n");
    display(q1);

    // 4. Enqueue 4
    enqueue(&q1, 4);
    printf("After enqueue 4:\n");
    display(q1);

    // 5. Dequeue (remove 7)
    dequeue(&q1);
    printf("After dequeue:\n");
    display(q1);

    // 6. Dequeue (remove 2)
    dequeue(&q1);
    printf("After dequeue:\n");
    display(q1);
}

```

```

        printf("Number of elements remaining in the queue: %d\n",
q1.num_elements);

        return 0;
}

```

After dequeue:

9 --> 15 --> 4 --> NULL The above main function in part b already includes the display of remaining elements after each operation. You can execute this code to see the output after each step :

Initial Setup :

You create a queue with elements 5, 3, 7, 2, 9.

Execution Steps and Expected Results :

1. Initial queue display: Queue after initialization: 5 --> 3 --> 7 --> 2 --> 9 --> NULL
2. After dequeue (removes 5): After dequeue: 3 --> 7 --> 2 --> 9 --> NULL
3. After enqueue 15: After enqueue 15: 3 --> 7 --> 2 --> 9 --> 15 --> NULL
4. After dequeue (removes 3): After dequeue: 7 --> 2 --> 9 --> 15 --> NULL
5. After enqueue 4: After enqueue 4: 7 --> 2 --> 9 --> 15 --> 4 --> NULL
6. After dequeue (removes 7): After dequeue: 2 --> 9 --> 15 --> 4 --> NULL
7. After dequeue (removes 2): After dequeue: 9 --> 15 --> 4 --> NULL
8. Final number of elements remaining in the queue: Number of elements remaining in the queue: 3

After performing all the operations, the remaining elements in the queue are 9, 15, 4, and there are 3 elements left.

EXERCISE 5 :

1. Row Minimums (FIFO): For each row in the matrix, find the minimum value and store it in the rowMins array. This maintains the order of rows, which corresponds to FIFO (First In, First Out) for the minimum values.
2. Column Sums (LIFO): Calculate the sum of each column and store it in the colSums array. To meet the LIFO (Last In, First Out) requirement, the column sums are printed in reverse order.

```

#include <stdio.h>
#include <limits.h> // For INT_MAX

#define N 4 // Number of rows
#define M 3 // Number of columns

```

```

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Function to generate the list of row minimums (FIFO)
void getRowMinimums(int matrix[N][M], int rowMins[]) {
    for (int i = 0; i < N; i++) {
        int min = INT_MAX;
        for (int j = 0; j < M; j++) {
            if (matrix[i][j] < min) {
                min = matrix[i][j];
            }
        }
        rowMins[i] = min;
    }
}

// Function to generate the list of column sums (LIFO)
void getColumnSums(int matrix[N][M], int colSums[]) {
    for (int j = 0; j < M; j++) {
        int sum = 0;
        for (int i = 0; i < N; i++) {
            sum += matrix[i][j];
        }
        colSums[j] = sum;
    }
}

int main() {
    int matrix[N][M] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9},
        {10, 11, 12}
    };

    int rowMins[N];
    int colSums[M];

    // Generate row minimums
    getRowMinimums(matrix, rowMins);
    printf("Row minimums (FIFO): ");
    printArray(rowMins, N);

    // Generate column sums

```

```

    getColumnSums(matrix, colSums);
    printf("Column sums (LIFO): ");
    // Print column sums in LIFO order
    for (int i = M - 1; i >= 0; i--) {
        printf("%d ", colSums[i]);
    }
    printf("\n");

    return 0;
}

```

The expected output for the provided C program when executed:

➤ Matrix Input :

```

1 2 3
4 5 6
7 8 9
10 11 12

```

➤ Output

1. Row Minimums (FIFO): The minimum value of each row:

- Row 1: Minimum is 1
- Row 2: Minimum is 4
- Row 3: Minimum is 7
- Row 4: Minimum is 10

Output: Row minimums (FIFO): 1 4 7 10

2. Column Sums (LIFO): The sum of each column:

- Column 1: Sum is $1 + 4 + 7 + 10 = 22$
- Column 2: Sum is $2 + 5 + 8 + 11 = 26$
- Column 3: Sum is $3 + 6 + 9 + 12 = 30$

Output (printed in LIFO order, which is reverse order of calculation):
Column sums (LIFO): 30 26 22

Row minimums (FIFO): 1 4 7 10
Column sums (LIFO): 30 26 22

The result shows the row minimums in the order they were calculated, and the column sums in reverse order from the way they were calculated, meeting the FIFO and LIFO requirements respectively.

Conclusion

This course material has guided you through the fundamental and advanced concepts of algorithms and data structures, with a focus on implementations in C. The early chapters introduced essential basics, while the subsequent chapters explored more complex data structures and sophisticated algorithms. The summary exercises at the end of the document allow you to put the acquired knowledge into practice and reinforce your understanding.

By mastering these concepts, you will be well-prepared to tackle complex computing problems, optimize your programs, and design efficient solutions. We hope this course has been beneficial to you and has provided you with the necessary skills to succeed in the field of computer science.

REFERENCES

1. Sedgewick, R., & Wayne, K. (2015). Algorithms (4th ed.). Addison-Wesley.
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
3. Goodrich, M. T., & Tamassia, R. (2014). Data Structures and Algorithms in C (2nd ed.). Wiley.
4. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.
5. Knuth, D. E. (1997). The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd ed.). Addison-Wesley.
6. Harbison, S. P., & Steele, G. L. (2002). C: A Reference Manual (5th ed.). Prentice Hall.
7. Schaefer, M. (2014). Programming in C. Wiley.
8. Kernighan, B. W., & Ritchie, D. M. (1988). The C Programming Language (2nd ed.). Prentice Hall.
9. Thomas, P. (2019). Programmation en C avec Exercices et Corrigés (2nd ed.). Dunod.
10. Delannoy, C. (2018). Programmer en C: Cours et Exercices* (5th ed.). Eyrolles.