PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

Ministry of Higher Education and Scientific Research

University of Tindouf

Faculty of Sciences and Technology
Department of Computer Science

DATA STRUCTURES AND ALGORITHMS 2

Major : Information systems

Level : first Year

Authored by :

Dr. DRAOUI Abdelghani

Academic Year : 2025-2026

TABLE OF CONTENT

TABLE OF CONTENT

TABLE OF CONTENT

TABLE OF CONTENT

General Introduction

General introduction

Data Structures and Algorithms form the foundation of computer science and software development. They provide systematic ways to organize data and efficient methods to process, store, and retrieve information. A strong understanding of these concepts enables programmers to design solutions that are both effective and optimized for performance.

This set of lecture notes is designed to introduce fundamental concepts that are essential for building structured and efficient programs. The material progresses from basic programming constructs to more advanced data organization techniques, ensuring a gradual and comprehensive understanding of the subject.

****Chapter One: Subprograms****

The first chapter focuses on subprograms, which are fundamental building blocks in structured programming. Subprograms (functions and procedures) allow programmers to divide complex problems into smaller, manageable units. This chapter covers the concept of modular programming, parameter passing, scope of variables, recursion, and the benefits of code reusability and maintainability.

****Chapter Two: Files****

The second chapter introduces file handling concepts. Files provide a way to store data permanently outside the main memory, enabling programs to handle large volumes of data efficiently. This chapter discusses file operations such as opening, reading, writing, appending, and closing files, as well as different file modes and basic file organization techniques.

****Chapter Three: Linked List Data Structure ****

The third chapter explores dynamic data structures, beginning with linked lists. Linked lists offer flexible memory usage and dynamic storage allocation compared to static data structures. The chapter explains different types of linked lists and their implementation. It further introduces stacks and queues as abstract data types, detailing their operations, applications, and implementations using linked lists.

By the end of these notes, students will understand how to design and implement fundamental data structures and apply structured problem-solving techniques. All algorithms discussed throughout these lecture notes are expressed in the C programming language.

Review of DSA 1

1 Review and Basic Definitions

1.1 Introduction

For a better understanding of concepts that will be covered in Data Structures and Algorithms 2 (DSA2), Fundamental concepts of the previous course (Data Structures and Algorithms 1) will be briefly reviewed. These concepts are the pre-requisites of this course, in case of having difficulties understanding it, student is advised to check for a detailed learning material.

1.2 Data structure

A data structure is a way to store and organize data in order to facilitate access and modifications. No single data structure works well for all purposes. Figure below shows the most popular data structures.

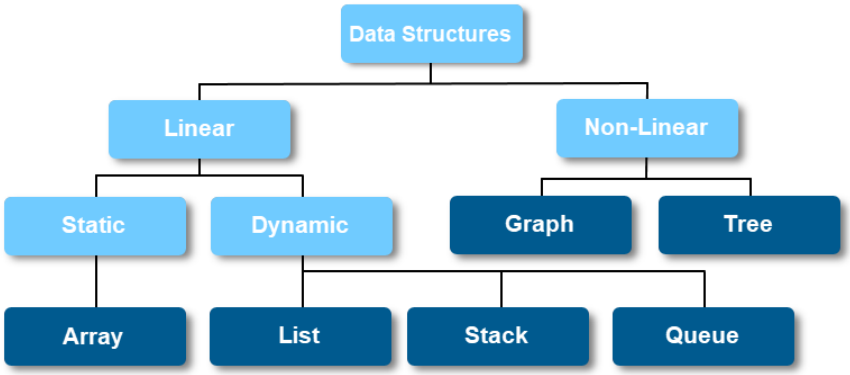


Fig 1. Classification of data structures

Note: Stack and Queue data structures can be implemented statically using arrays, as we will see later in this course. Non-linear data structures such as tree and graph are covered in the follow-up course, Data Structures and algorithms 3.

1.3 Algorithm

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output in a finite amount of time [1]. An algorithm is thus a sequence of computational steps that transform the input into the output.

1.4 Example of algorithms

We use algorithms in our daily life, here are some few examples:

- A food recipe (ingredients(as input) -> prepared meal(result)).
- Looking for meaning of a word in the dictionary (word -> definition).
- Sorting a sequence of elements (unsorted sequence -> sorted sequence).

For a sorting algorithm, if the input sequence <1,9,5,4> is given the algorithm should output the following output sequence <1,4,5,9>.

The input sequence <1,9,5,4> is called a problem instance.

The problem instance consists of a valid input needed to compute a solution to the problem.

1.5 Classification of algorithms

An algorithm is correct if, for every problem instance provided as input, it finishes its computing in finite time and output the correct solution.

An incorrect algorithm might not finish at all on some input instances, or it might finish with an incorrect answer. Contrary to what you might expect, incorrect algorithms can sometimes be useful, if you can control this error rate.

We will concern ourselves only with correct algorithms.

1.6 Evaluation of algorithms

Algorithms are evaluated according to their use of resources. Algorithm running time and consumption of memory (RAM) for its execution.

To evaluate and potentially improve our algorithm, we should ask ourselves these questions:

- Is my algorithm a correct one?
- How much is its running time?
- How much space in memory is consumed?
- Are there any way to do better?

Note: Estimating the running time and the space of memory consumed are covered in the advanced course DSA3.

1.7 How to express an algorithm?

An algorithm can be specified in natural language (Arabic, French, English …etc.), flow chart, pseudocode, or computer program. The only rule to be respected is that the algorithm should have clear meaning.

A pseudocode, it is a method of expressing algorithm where a mixture of expressions in natural language and expressions used in programming language (assignment, comparison, addition …etc) are used. It is intended for human to focus on understanding algorithms without worrying about the differences in syntax in the various programming languages.

Note: In this course we will use C programming language and the pseudocode to write algorithms.

1.8 Data types

Most algorithmic problems require storage and manipulation of data of a specific type, the most frequently used data types are summarized in the table below :

type	Example of possible values
integer	-400 -6 10000
Real	4.459 -3.1543
character	‘a’ ‘A’ ‘_’ ‘9’
string	‘’University of Tindouf’’
Boolean	True, False

1.9 Reading user input and displaying output

Values of variables -defined in the program- can be read from input of the user. Displaying the output of the program, is usually done, in a window. To read inputs and display output to the user, we use the read and print functions. Table below show an example of use of this functions:

Pseudocode	Example
read	read(a)
print	print('The value of b is : ',b)

1.10 Types of Operators

1.10.1 Assignment

Assigning values to variables can be performed using an assignment operator. In our pseudocode we adopt the combined symbol '**<-**' for this task.

Pseudocode	Explanation
A <- B	Value of B is assigned to A using the assignment operator <-

1.10.2 Arithmetic operators: +, -, *, /, div, mod

We can perform arithmetic operations on numerical data types such as integer and real numbers, the modulo (i.e. the remainder of division) is denoted by '**%**' symbol.

Operation	Result (Explanation)
5+9	14
60-20	40
15/2	7.5
15 **div** 2	7 (Euclidian division)
9 **%** 2	1 (remainder of division of 9 over 2)

Note: Performing arithmetic operation on variables of character type, corresponds to performing these operation on the corresponding ASCII code of these characters.

1.10.3 Priority rules

In an instruction containing more than one arithmetic operation, the following order of performing these operations has to be respected:

Order	Operators
01	(),[],{}
02	Exponents like : x^2 x^3
03	×, /
04	+, -

Note: For operators having same priority, we perform calculation from left to right.

Example: 100/4*5 , a is equal to 125 and not 5 !

1.10.4 Logical and relational operators

The logical operator (and, or, negation) along with relational operators leads always to true and false answer, we use 0 to denote false and 1 to denote true.

Operation	Result (Explanation)
5 = 9	0 (False)
60 ≠ 20	1 (True)
1 and 0	0 (False)
1 or 0	1 (True)
>, <, >=, <=	/

Note: In C language, we use 0 to express 'False' value and any other value to express True value.

1.11 Control flow structures

The order of execution of a program is from top instruction to bottom one, an instruction or a group of instructions can be prevented from execution using one of the conditional structures, if instructions are needed to be executed multiple times before proceeding to the execution of the subsequent lines one type of loops is used.

1.11.1 Conditional control structures

The instructions inside the conditional block is executed only if a condition or group of conditions are satisfied.

1.11.1.1 If structure

Pseudocode	Example

If Condition(s) then
  Instruction(s)
endif
	If (age > 18) then
	  print(' you are an adult')
	endif

1.11.1.2 If-else if –else structure

Execution of instruction can be restricted to satisfaction of one of a group of conditions.

Pseudocode	Example

if Condition(s) then
  Instruction(s)
Else if Condition(s) then
  Instruction(s)
Else Condition(s) then
  Instruction(s)
endIf
	if a > b then
	  Print('a is larger than b')
	Else if a < b
	  Print('b is larger than a')
	Else if a < b
	  Print('b is equal to a')
	endif

1.11.1.3 Switch case structure

Switch-case structure is preferred when dealing with many conditions.

Pseudocode	Example
Switch variable or expression **Case** value 1 : action1 **Case** value 2 : action2 … … **Otherwise** a default action **EndSwitch**	**Switch** Variable 1 **Case** 1: print('winter') **Case** 2: print('spring') **Case** 3: print('summer') **Case** 4: print('autumn') **Otherwise** print('Error') **EndSwitch**

1.11.2 Iterative structures

1.11.3 For loop

If the number of execution of some instructions is known, it is preferable to use the 'for loop'.

Pseudocode	Example

for i from .. to .. do
Instruction(s)
EndFor
	for i from 1 to 20 do
	Print('Hello')
	EndFor

1.11.4 While loop

In some cases, number of iterations of some group of instructions in not known, therefore, we use the while loop.

Pseudocode	Example
While condition(s) Instruction(s) EndWhile	while grade > 10 print('Congratulations') EndWhile

1.11.5 Do …. While

If a group of instructions is needed to be performed at least once, the 'do-while' loop is preferred

Pseudocode	Example
do Instruction Instruction **while** conditions(s)	**do** print('hi !') write('type N to terminate') **while** C = 'N'

Remark: Having a code written using one of the aforementioned loop types, we can rewrite the same code using the other type, below is an example

For loop	While loop

For i <- 2 to n **do**
Instruction(s)
EndFor
	i <- 2
	while (i ≤ n)
	Instruction(s)
	i <- i + 1
	EndWhile

1.12 Arrays

1.12.1 One Dimensional array

Question 1: How can we save grades of 100 students?

Question 2: Do we have to declare 100 separate variables, each storing a grade of one student?

Answer: The best way to store grades of 100 students is to declare one single variable whose type is one dimensional array.

In general, in order to store multiple values of variables of the same type, we can use one dimensional arrays. The drawbacks of using arrays to store data is that their size cannot be changed during the execution time and the elements of the array must have same data type.

A 1D-array can be declared following the syntax below :

var arrayName [0,..,size] : data type

where :

arrayName : denotes the name of the array.

Data type : denotes the data type of elements of the array.

To access the i^{th} element of the array arrayName, we use the following syntax : T[i].

```
Algorithm Filling_1D_table
Variable T[0,..,4] : array of integers, i :integer
Begin
  For i from 1 to 4 do
      Print('insert element index ',i)
      Read(T[i])
  EndFor
End
```

1.12.2 Multidimensional dimensional array

We would like to write a program that calculate the average of each student knowing that the class contains 20 students who studied 6 courses.

Question: How can store this type of data efficiently?

Answer: Ideally, we use two-dimensional array, where one dimension represent the students and the other represents the modules, in that case the syntax would be :

Var Grades[0,..,19 ;0,..,5] : array of real nbrs

To Access j^{th} grade of i^{th} student we adopt the following syntax Grades[i,j]

Example:

```
Algorithm filling of 2D array
Variables T[1,..,4 ;1,..,6] : array of real nbrs, i,j :integer
begin
        for i from 1 to 4 do
            for j from 1 to 6 do
               Print('insert grade', j, ' of student',i)
               Read(T[i ;j])
            EndFor
        EndFor
End
```

1.13 Structure

When dealing with entities having many attributes, where each attribute has different data type, in that case, we declare a variable of type structure. A structure is a compound data type, composed by multiple elementary data types.

In order to declare a variable of type structure, we need to define structure data type. We can access its attributes using the dot notation.

Structure syntax	Example
Type Identifier=structure Field1 : Type1 Field2 : Type2 ... ... Field N : Type N EndStructure	Type Car=Structure Manufacturer : string Color : string HorsePower : integer EndStructure

Use of created structure to declare variables	Example
Variable Var1 : Identifier begin Var1.Field1 <- Value1 Var1.Field2 <- Value2 Var1.Field3 <- Value3 End	Variable V : Car Begin V. Manufacturer <- 'BMW' V. Color <- 'Red' V. HorsePower <- 120 End

1.14 Exercises

1. Write an algorithm that asks the user for two integer numbers to output the remainder of division. (hint: use conditional structure to avoid division over zero).
2. Write a program that tells if an integer number given by the user is odd or even.
3. Write a program that display only the even numbers of a pre-defined array of integers.
4. Write a program that display all the divisors of an integer number given by the user.
5. Write a program that gives the roots of a quadratic equation.
6. Write a program that receives the three sides of a triangle, to check if this triangle is a right one.
7. Write a program that calculate the area of a disc of a given radius 'r'.
8. Define another subprogram that calculate the volume of a cylinder of a given radius 'r' and height 'h'.
9. We want to store data of student of our department, show how you can use structure to store data of each student with the following pieces of information: name, age, gender.

1.15 Conclusion

In this introductory part, a quick and brief review of fundamental concepts are exposed. These concepts represents the pre-requisite of this course. For a detailed explanation, reader may consult of the references mentioned below.

Chapter 01
Subprograms

2 Chapter 01: Subprograms

2.1 Introduction

Due to the complexity of most real world applications, code is decomposed into smaller parts to facilitate its manipulation. In addition, these applications may contain repeated blocks of code, refactoring this code by using subprograms and therefore avoiding code repetition will make the code maintenance easier. In this chapter, you will be introduced to the concept of subprogram.

2.2 Issue of code repetition

Consider the following code written in C language.

```c
#include <stdio.h>
int main()
{
    printf("Centre Univ Tindouf\n");
    printf("Institut de S & T\n");
    printf("department de MI\n");

    printf("Centre Univ Tindouf\n");
    printf("Institut de S & T\n");
    printf("department de geologie\n");
}
```

This program displays information related to the two departments at the institute of science and technology in the university center of Tindouf.

Question: How many lines do we need to modify to display these pieces of information in English?

Answer: Due to code repetition a total of six lines has to be modified.

To make this code easy to modify, we can re-write our program using a subprogram. The resulting code is

```c
#include <stdio.h>
void display(){
    printf("Centre Univ Tindouf\n");
    printf("Institut de S & T\n");
}
int main()
{
    display();
    printf("department de MI\n");

    display();
    printf("department de geologie\n");
}
```

A subprogram with a name of 'display' is defined above and outside of the main block. Then it is called twice, inside the main block.

If the message to be displayed in English, only 4 lines instead of 6 lines are to be modified. Thanks to the use of subprogram, the number of lines to be modified is reduced. The advantage is more pronounced if, for example, the code displays pieces of information about ten departments at the same institute and university. Hence, 12 lines of code to be modified instead of 30 lines !

In the code above, we notice that the third and the sixth line contains a repeated string which is '*department de*' while the only difference is the name of the department. We can include this repeated code in a subprogram, while giving the changing part as an input parameter to the subprogram. The newer code is shown.

```
#include <stdio.h>
void display(char dept[]){
    printf("Centre Univ Tindouf\n");
    printf("Institut de S & T\n");
    printf("department de %s\n",dept);
}
int main()
{
    display("MI");
    display("geologie");
}
```

In the newer code, code modification become easier as the entire repeated part is extracted as a 'common factor' in form of a subprogram.

2.3 Subprogram syntax

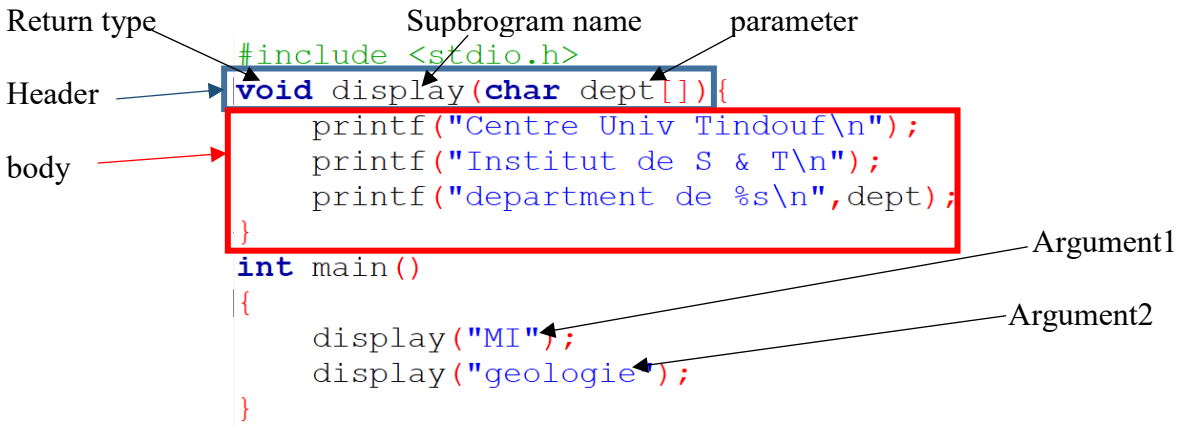
A subprogram is mainly composed of two parts:

1- **Subprogram definition** which represents the code above the main block.
2- **Subprogram call**, the subprogram name mentioned inside the main block.

Note: There is a third part called **subprogram declaration** that is not necessary, as we will later.

The subprogram definition itself is composed of the following parts shown in figure below:

1- **Subprogram header**: it is the upper part of subprogram definition where a **return type**, **subprogram name**, and one or several **parameters** are specified.
2- **Subprogram body**: contains the repeated code or the instructions to be executed when the subprogram is called elsewhere in the program.



- The return type '**void**' means that no return type is needed since the subprogram is used for only executing a set of instructions, no final value is needed as at the end of its execution.
- The function name is a string used to call the subprogram for execution of instructions inside it.
- To receive input, type and name of parameter (also referred as **formal parameter**) is specified after the function name, inside the parenthesis.
- The input given to the subprogram call is denoted as **argument** (or **actual parameter**), behavior of the subprogram depends in that case on the passed argument.
- So far, two types of subprograms are examined, subprograms with parameters and subprograms without parameters.

Assuming that we want to improve our program to display departments in different universities, in that case, we re-define our program with three parameters.

```c
#include <stdio.h>
void display(char univ[],char fac[],char dept[]){
    printf("University : %s\n",univ);
    printf("Faculty : %s\n", fac);
    printf("department :  %s\n", dept);
}
int main()
{
    display("Tindouf","ST","geology");
    display("Naama","SNV","biology");
}
```

```
C:\draoui_abdelghani\demos\bin\Debug\demo
University : Tindouf
Faculty : ST
department :  geology
University : Naama
Faculty : SNV
department :  biology
```

Note: The passed arguments when calling a subprogram, has to respect the same order of its corresponding parameters.

Example:

This program asks the user to input two integer values to display their sum. This process is repeated once.

```c
#include <stdio.h>
int main()
{
    int a,b,c;
    printf("a ?");scanf("%d",&a);
    printf("b ?");scanf("%d",&b);
    c = a + b;
    printf("the sum is : %d\n", c);

    printf("a ?");scanf("%d",&a);
    printf("b ?");scanf("%d",&b);
    c = a + b;
    printf("the sum is : %d\n", c);
}
```

Question: How many numbers are needed to be modified if we want to calculate the product instead of the sum of 'a' and 'b'?

Answer: A total of 4 lines are to be modified, the addition operator to be changed twice and the two messages '*the sum is* ' have to be modified to '*the product is*'.

We notice that code repetition demands modification of 4 lines. To solve this issue, we can re-write (refactor) our program using a subprogram as shown in the figure below.

```c
#include <stdio.h>
int a,b,c;

void operation(){
    printf("a ?");scanf("%d",&a);
    printf("b ?");scanf("%d",&b);
    c = a + b;
    printf("the sum is : %d\n", c);
}
int main()
{
    operation();
    operation();
}
```

Note that the integer variables ‘a’, ‘b’, and, ‘c’ are defined outside of main block and outside of operation subprogram. Therefore, these variables are common to both main and operation. These common variables are referred as **Global variables**.

Example:

We would like to compare a pair of numbers twice, to display in a message the larger number in each pair. We suggest solution given in the following code, having code repetition, we can re-write the code using subprograms.

```c
#include <stdio.h>
#include <stdlib.h>
int main(){

    int a,b,c,d,max1,max2;

    a = 7; b = 3;
    if (a>=b){
        max1 = a;
    } else {
        max1 = b;
    }

    printf("pass to step two\n");

    a = 2 ; b = 10;
    if (a>=b){
        max2 = a;
    } else {
        max2 = b;
    }

    printf("%d and %d are the largest",max1,max2);

}
```

```c
#include <stdio.h>
#include <stdlib.h>
int findMax(int a,int b){
    int result ;
    if (a>=b){
        result = a;
    } else {
        result = b;
    }
    return result;
}
int main(){

    int a,b,c,d,max1,max2;

    a = 7; b = 3;
    max1 = findMax(a,b);


    printf("pass to step two\n");

    a = 2 ; b = 10;
    max2 = findMax(a,b);

    printf("%d and %d are the largest",max1,max2);

}
```

Since the comparison results in returning the larger value, the subprogram is defined to return a value using the keyword ‘**return**’ in the body part of the subprogram definition. The return type is specified as ‘int’ instead of ‘void’ since the subprogram returns a value, which is the one of received two integers either the variable ‘a’ or/and ‘b’.

The subprogram call is written at the right hand side of an assignment operator and that it because the subprogram is returning a value. In other words, the result of the comparison (the returned value) will replace the subprogram call during the execution of the program.

2.4 Function or a procedure ?

So far, we can recognize two classes of subprograms:

1- A subprogram that execute a set of instructions without returning a value at the end of its execution
2- A subprogram that execute a set of instructions to return a single value at the end of execution.

For distinction, the first type of subprograms is referred as '**a procedure**' and the second type is referred as '**a function**'. In fact, a procedure is considered as a special function which has no return value. That is the reason why the words subprogram and function are used interchangeably.

Note: 'printf' and 'scanf' are just two built-in subprograms of type 'functions'. For instance, 'printf' displays the string received as input and returns the string length which can be exploited as an additional piece of information in our program.

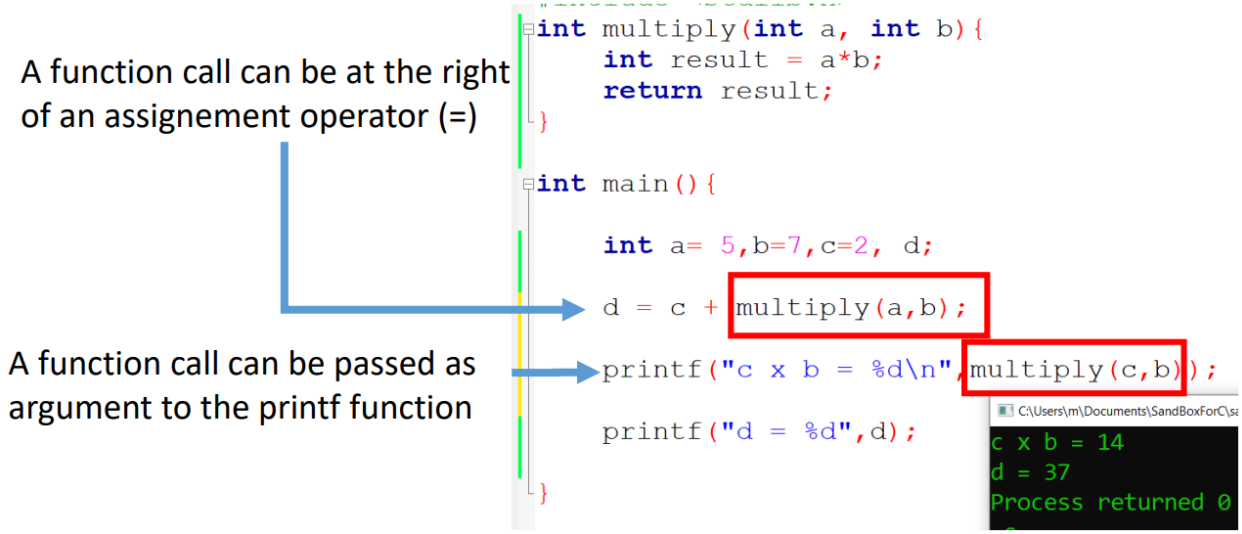
2.5 Calling subprograms

In addition, what differentiates a procedure from a function is that a procedure is called by specifying its name directly in the code as a single instruction. Since a function returns a value, the returned value can be exploited. The returned value is exploited by calling the function:

- At the right hand of an assignment operator.
- At one of the sides of a logical or relational operator
- Inside the printf function as one of its parameters.

Figures below show ways of calling a subprogram of type function.

Example: It shows calls the function 'multiply' at the right hand of the assignment operator and inside printf function.



Example: At one of the sides of a relational operator.

A function call can be inside condition of a while or do..while loop.

```
int isBiggerThanTen(int a){
    if(a > 10)
        return 1;
    else
        return 0;
}

int main(){

    int a= 14;

    while(isBiggerThanTen(a) == 1){
        printf("a = %d\n",a);
        a = a - 1;
    }

}
```

```
a = 14
a = 13
a = 12
a = 11
```

Example:

Returned value of isEven function is exploited in main, inside the condition of an if-block.

A function call can be inside a condition structure if or switch

```
int isEven(int b){
    if (b % 2 == 0)
        return 1;
    else
        return 0;
}

int main(){

    int b=16;

    if(isEven(b)== 1)
        printf(" b is even");
    else
        printf("b is odd");

}
```

```
b is even
```

2.6 Subprogram call methods

The examples shown so far, demonstrated subprogram call inside the main block. In reality, a subprogram can be also called inside any other subprogram.

```c
void saluer(){
    printf("Salam alaykum !\n");
}

void encourager(){
    printf("Bon courage cher(e) etudiant(e) !\n");
}

void saluezEtEncourager(){
    saluer();
    encourager();
}

int main(){
    saluezEtEncourager();
}
```

```
C:\Users\m\Documents\SandBoxForC\projet1\bin\Debu...
Salam alaykum !
Bon courage cher(e) etudiant(e) !
```

Note: In fact, a subprogram can be even called inside itself as we will see later when discussing the concept of recursion.

Example: write a program that receives a phone number to check whether this number is a valid or invalid. A valid mobile phone number has to satisfy the following conditions: - it contains 10 digits – it starts with digit 0 – its second digit is 5, 6, or, 7

Solution: We can write our program as a set of subprograms, a subprogram for each of the above conditions.

```c
int isValidLen(char n[]){

    if(strlen(n) == 10)
        return 1;
    else
        return 0;
}

int ZeroAtBeginning(char n[]){

    if(n[0] == '0')
        return 1;
    else
        return 0;
}

int isValidOperator(char n[]){
    if(n[1] == '5' || n[1] == '6' || n[1] == '7')
        return 1;
    else
        return 0;
}
```

```c
void VerifyNbr(char n[]){

    if (isValidLen(n) && ZeroAtBeginning(n) && isValidOperator(n))
        printf("Nbr is correct\n");
    else
        printf("Nbr is wrong !\n ");
}
```

In order to call only one subprogram in the main block, the 'VerifyNbr' procedure assembles the three conditions.

2.7 Common errors when using subprograms

2.7.1 Errors when defining a subprogram

Example: The return type of the function has to be the same as the type of the returned variable

```
int findAverage(float a , float b , float c){
    float avg ;
    avg = (a + b + c);
    return avg;
}
```

Example: A subprogram can be either a procedure or a function, therefore, void and return keywords cannot co-exist in the subprogram definition.

```
void findAverage(float a , float b , float c){
    float avg ;
    avg = (a + b + c);
    return avg;
}
```

2.7.2 Errors when calling a subprogram

```
void sayHello(){
    printf("Hello ! \n ");
    printf("have a nice day !");
}
float findAverage(float a , float b , float c){
    float avg ;
    avg = (a + b + c);
    return avg;
}

int main(){

    findAverage(13.5,4.0,7.25);
    Myaverage = findAverage(13.5,4.0,7.25);

    sayHello();


    printf("max is %d",maximum);
}
```

Calling function directly is useless since we have not received the returned value in a variable.

When calling a function and since it returns a value, this values should be received in a variable.

Since procedure does not return any value it is called directly.

2.8 Subprogram for code organization

Exercise: We would like to write a program that reads two grades of a student and then to tell student at the end whether he/she will pass or fail his/her exams.

```c
float CalculateSum(float grade1, float grade2){
    float total;
    total = grade1+grade2;
    return total;
}
float CalculateAverage(float sum){
    float avg ;
    avg = sum/2;
    return avg;
}
void DisplayDecision(float average){

    if(average >= 10){
        printf("You pass the exams");
    }else {
        printf("You failed the exams");
    }
}
```

```c
int main(){

    float grade1,grade2,sum,average;

    // 1- ask student for his grades
        printf("enter grade 1 : ");
        scanf("%f",&grade1);
        printf("enter grade 2 : ");
        scanf("%f",&grade2);
    // 2- compute sum
        sum = CalculateSum(grade1,grade2);

    // 3- calculate average
        average = CalculateAverage(sum);

    // 4-display decision
        DisplayDecision(average);

}
```

Assuming that each grade has different coefficient and the passing grade is 9.5, in this case, code inside the main block is not modified which demonstrates a clean and readable code.

```c
float CalculateSum(int grade1,int grade2){
    float sum ;
    sum = 2*grade1 + 3*grade2;
    return sum;
}
float CalculateAverage(int sum){
    float average ;
    average = sum/5;
    return average;
}
void DisplayDesicion(int average){
    if(average >= 9.5){
        printf("You passed exams");
    }else{
        printf("You failed exams ");
    }
}
```

```c
int main(){

    float grade1,grade2,sum,average;

    // 1-ask student for his grades
    printf("enter grade1 : ");
    scanf("%f",&grade1);
    printf("enter grade2 : ");
    scanf("%f",&grade2);

    // 2- compute sum
    sum = CalculateSum(grade1,grade2);


    // 3- calculate average
    average = CalculateAverage(sum);

    // 4- display decision
    DisplayDesicion(average);
```

2.9 Some built-in subprograms

We call subprograms that we define as programmers a '**user-defined**' subprograms. In addition to user-defined functions, each programming language is provided with multiple sets of **built-in** subprograms assembled into libraries. Each library contains subprograms of a given field. Math.h and string.h libraries are the most commonly used ones in C language.

In order to be able to use the math built-in functions, the math library should be included by adding #include <math.h> at the top of the program.

Method Signature	Return value
double sqrt(double x);	returns the square root of x
double pow(double x, double y)	returns x raised to the power of y
double fabs(double x)	returns the absolute value of x
double floor(double x)	returns the largest integer value less than or equal to x.
double ceil (double x)	returns the smallest integer value greater than or equal to x.

Example

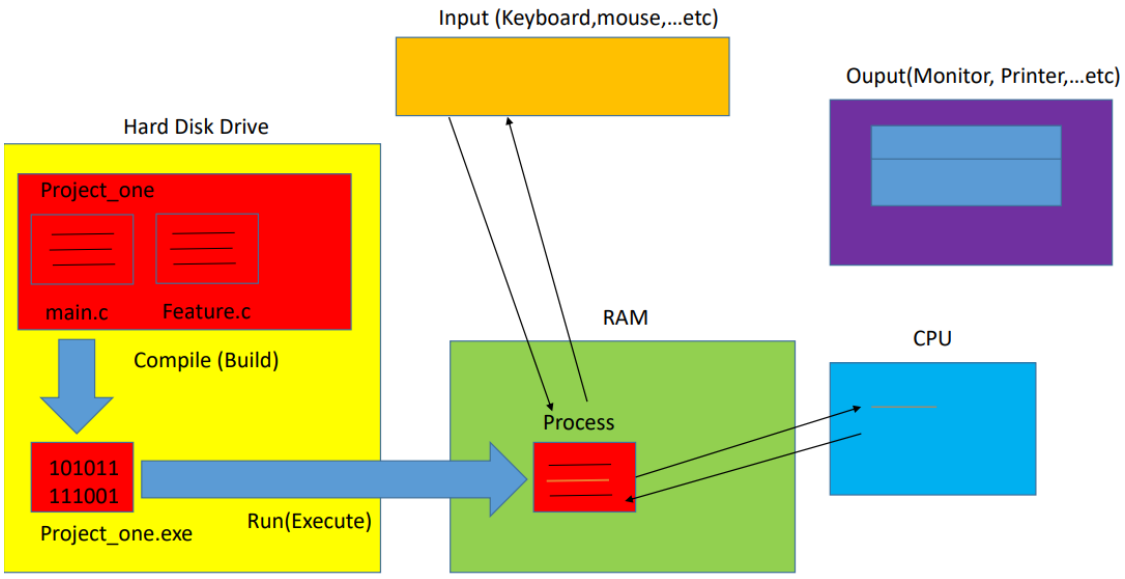
Given the legs values of a right triangle as input, write a subprogram that computes the hypotenuse of this triangle. We can exploit the built-in functions in math library. The *sqrt(x)* function returns the square root of the real variable x. while '*pow(x,y)*' receives real numbers 'a' and 'b' to return a^b

```c
#include <math.h>
float hypotenuse(float a,float b){
    int c;
    c = sqrt(pow(a,2)+pow(b,2));
    return c;
}

int main(){

    float a,b;
    a = 4;
    b = 3;
    printf("The lenght of the hypotenuse is %f",hypotenuse(a,b));
}
```

```
C:\Users\m\Documents\SandBoxForC\sandBox\bin\Debug\sandBox.exe
The lenght of the hypotenuse is 5.000000
```

2.10 Execution of program

A program is written in a text editor in a certain compiled programming language. If the code is complex, it is usually decomposed to a set of files, each file containing a group of subprogram. If the code has no syntax errors it can be easily converted into a machine code using a compiler, resulting in an executable file. Executing this program means that this program is loaded into RAM as a process. No matter how many subprograms the program contains. The execution is done sequentially starting from the main function. The main function is referred as the program entry point.



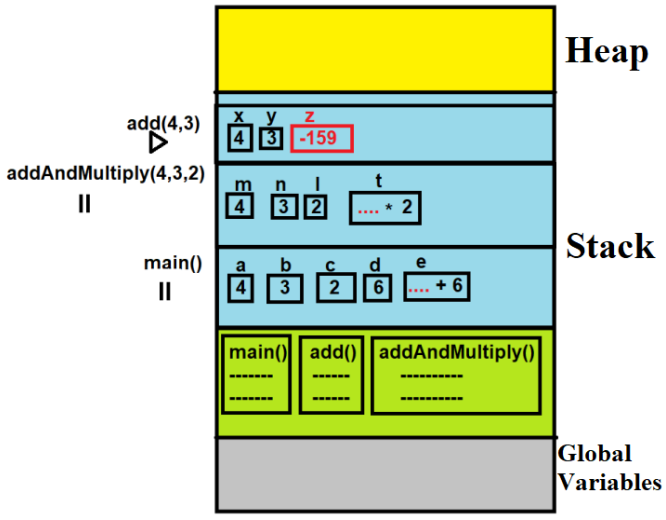
Usually, the text editor and compiler along with a debugger are built in a single software that is referred as an Integrated Development Environment (IDE). Code::Blocks and Turbo C are among the free IDEs available for coding in C/C++ language.

Program may be halted for reading user input via the keyboard. If the program is a console application, the output will be displayed in windows from the PC monitor.

2.11 Sequence of execution

Compilation process will result in an executable file. When this file is executed, program will be loaded into memory as a process. Consider the following example of a code composed of main function and two subprograms.

```
   #include <stdlib.h>
   #include <stdio.h>
1  int add(int x, int y){
2      int z;
3      z = x + y;
4      return z;
5  }
6  int addAndMultiply(int m,int n, int l){
7      int t ;
8      t = add(m,n) *l;
9      return t;
10 }
11 int main(){
12     int a = 4,b = 3,c=2,d=6,e;
13     e = addAndMultiply(a,b,c) + d;
14     printf("e = %d",e);
15 }
```



1- Being the entry point of the program, a stack frame for the main function is first created in the stack section of the process.
2- Local variables of main are created, with a random values assigned to variables that are not initialized.
3- When reaching an instruction containing a function call, the calling function(main) will pause execution letting the called function *addAndMultiply()* to run by creating a new stack frame for that called function right at the top of main's stack frame.
4- Execution flow passes to that new called function. Reaching line 8, another function is called in that instruction. Leading to pausing of execution of the calling function.
5- Execution flow passes then to the newly called '*add()*' function.
6- Reaching the return instruction, stack frame of *add* function is deleted and function call is replaced with the corresponding returned value.
7- Execution flow is passed back to the calling function '*addAndMultiply*'.
8- Execution will continue in the same fashion, eventually, displaying the value of variable 'e'.

2.12 Local and Global variables

Consider the code shown below, where a variable 'a' is created as local variable to main and an attempt to modify its value by another subprogram increment. Similarly, a local variable 'z' is created as local to function increment and an attempt to modify its value by 'main' function.

Note: The main block is itself a function, it is called the entry point of the program, since the execution starts from it.

```
1   #include <stdio.h>
2   int increment(){
3       int z = 5;
4       printf("a + 1 = %d",a+1);
5   }
6   int main()
7   {
8       int a =4;
9       z = z + 1 ;
10      increment();
11  }
```

In the code shown above, an integer variable 'a' is declared inside the main function, we say the variable 'a' is **local** to the main function. Being a local variable to main, any other subprogram cannot access or modify its value. Therefore, line 4 leads to a syntax error as the value of 'a' is to be modified by a function in which it is not declared.

Similarly, the variable 'z' is declared inside the increment procedure, leading to a syntax error in line 9.

In order correct this code and to allow for both functions to access and modify both variables, variables has to be declared above and outside of both functions. In that case, variable 'a' and 'z' are called **Global variables**. Solution is shown below

```
1   #include <stdio.h>
2   int a =4;
3   int z = 5;
4   int increment(){
5       printf("a + 1 = %d",a+1);
6   }
7   int main()
8   {
9       z = z + 1 ;
10      increment();
11  }
```

Note: A local variable to some function 'x', can be modified or accessed by another subprogram 'y', only if the address of this variable is passed as an argument to the subprogram 'y'. In this case,

a special variable holding the address of that variable has to be specified as a formal parameter. This special variable is referred as a pointer, further details on this variables will be discussed later.

2.13 Static variables

We would like to write a program with a variable counting the calling time of a given subprogram. Consider code below.

```c
#include <stdio.h>
int counter = 0;
void f(){
    counter++;
    printf("f fct is called %d time(s)\n",counter);
}
int main()
{
    f();
    f();
    f();
}
```

```
C:\draoui_abdelghani\demos\bin\Debug\demos.exe
f fct is called 1 time(s)
f fct is called 2 time(s)
f fct is called 3 time(s)
```

The counter variable, being a global variable, it can be modified by any other subprogram. Therefore, its values may not reflect the real number 'f()' is called. In addition, if the counter variable is declared as an *ordinary local* variable of the function 'f()'. Its value will be reset at each call. As a solution to such problems. The 'counter' variable is declared as a *static local* variable. A static variable is a variable that is shared only between function calls of a given function. Its initial value is taken into consideration only at the first call of that function. Above code can be re-written as follows.

```c
#include <stdio.h>
void f(){
    int static counter = 0;
    counter++;
    printf("f fct is called %d time(s)\n",counter);
}
int main()
{
    f();
    f();
    f();
}
```

```
C:\draoui_abdelghani\demos\bin\Debug\demos.exe
f fct is called 1 time(s)
f fct is called 2 time(s)
f fct is called 3 time(s)
```

2.14 Scopes of variables

Execution of a program is done sequentially, starting from the first line of the function 'main'. Assuming a variable is declared inside a given code block 'x'. This variable is created in memory(RAM) as soon as the execution reaches the block in which the variable is defined. This variable will be deleted from memory right after the execution passes outside the block in which the variable is created. In other words, we say that the scope of this variable is that block 'x'.

The scope of a global variable is the entire program, meaning that it will be created in memory from the start of program execution and it will remain in the program during the entire execution time.

The scope of variable b is the 'if block', therefore, this variable will be deleted from memory right after reaching the end of the execution of this block.

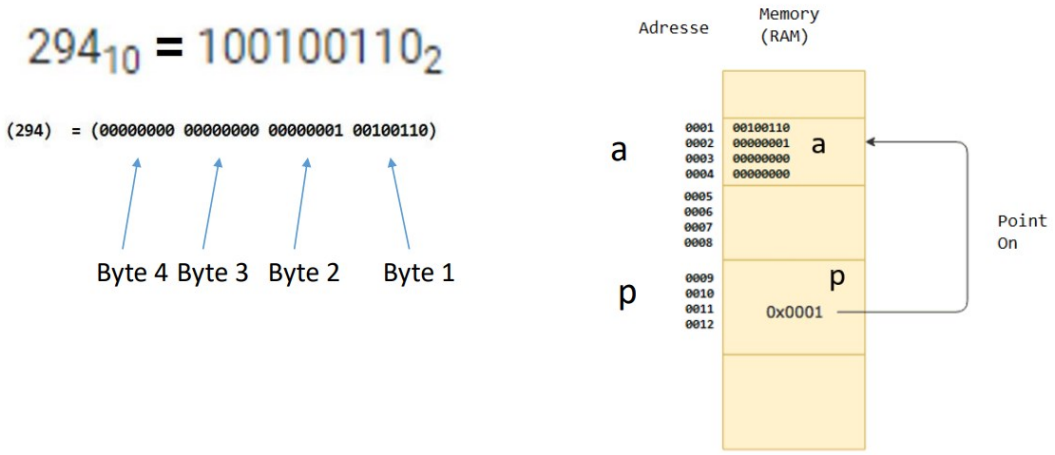
```
1  #include <stdio.h>
2  int main()
3  {
4      int a = 5;
5      if (a<9){
6          int b = 4;
7          printf("a =%d",a);
8      }
9      printf("b = %d",b);
10 }
```

Hence, the code shown on the right side, will yield a syntax error in line 9. The syntax error will state that a variable with name 'b' is used before its declaration, because it has been already deleted from memory.

2.15 Pointers

During the execution of a program, data are stored in RAM in binary form. In case of creation of a variable of type character, the binary form of its corresponding ASCII code is stored instead. Each datatype occupy a finite number of bytes in memory, hence, each datatype has a maximum and minimum value. Each byte in memory has its own address and each variable has an address which is the address of the first byte composing it.

A pointer is a variable defined to store the address of another variable of a same type. Any subprogram having the address of a variable in memory, it can be accessed/modified via the pointer holding its address.



In figure above, it demonstrate how 294 number is stored in memory (assuming it is declared as integer and assuming integer datatype is allocated 4 bytes in memory). A pointer 'p' is holding the address of the first byte of that variable 'a'.

We can get the number of bytes allocated for each datatype by using the built-in function *sizeof(x)*. Example below show, the number of bytes allocated for each datatype for the given compiler integrated inside the code::blocks IDE.

```
int main(){

    printf("short => %d octets\n",sizeof(short));
    printf("int   => %d octets\n",sizeof(int));
    printf("long  => %d octets\n",sizeof(long));
    printf("char  => %d octets\n",sizeof(char));
    printf("float =>%d octets\n",sizeof(float));
    printf("doube => %d octets\n",sizeof(double));

}
```

```
C:\Users\m\Documents\SandBoxForC
short => 2 octets
int   => 4 octets
long  => 4 octets
char  => 1 octets
float =>4 octets
doube => 8 octets
```

If our compiler allocates a space of 4 bytes (32 bits) for storing a variable of type int, possible values of any variable declared with type int are in the range -2^{32-1} to $2^{32-1} - 1$. Declaring a variable with a certain datatype while performing an operation which make the variable value exceed the maximum and minimum ranges of that datatype will result in false result.

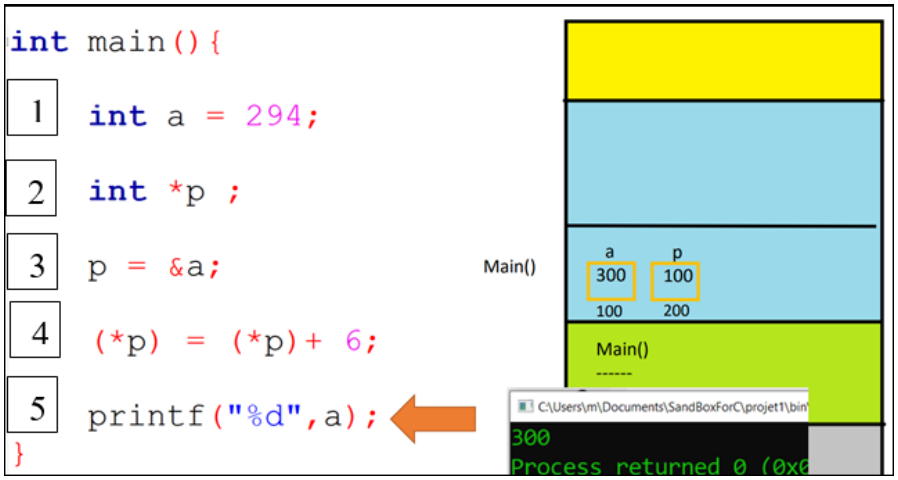
Short data type is usually allocated 2 bytes long (i.e. 16 bits long), hence, the maximum value that a variable of type short is $2^{16-1} - 1$ (=32 767). Given that piece of information what would be the result of the code below?

The resulting answer is wrong because the value of the 'c' variable (40 000) exceeds the maximum value that short datatype can have. Therefore the value stored in the 'c' variable will be wrong.

```
int main(){

    short a = 1000;
    short b = 40;
    short c = a * b ;

    printf("%d",c);
}
```

C:\Users\m\Documents\San
-25536

2.15.1 Pointer syntax



As discussed before, a variable can be modified through the pointer holding its address. In the following example:

- **Line 1**: An integer variable 'a' is declared

- **Line 2**: A pointer on integer 'p' is declared as well. A star (*) symbol is used to designate the variable as a pointer.
- **Line 3**: Address value of the integer 'a' (assuming it is 100 as shown in memory) is stored in 'p'. In other words, address of any variable 'x' is specified using the '&' symbol as '&a'.
- **Line 4**: Variable whose value is stored in 'p' is updated.
- **Line 5**: New value of variable 'a' is displayed using the printf function.

The line p = &a represents a **referencing operation** which consist of assigning the pointer as reference to the variable. While the syntax (*p) represents the opposite operation referred as **dereferencing operation**, in which the variable pointed by 'p' is targeted.

In dereferencing operation, a number of bytes is deduced from the size of the pointer datatype. This number of targeted bytes is considered as a single value and returned. That is the main reason requiring the pointer having same type of the variable to which it is pointing.

2.15.2 Dereferencing operation issues

Given the binary form of 'a' that is 100000100, what is the output of the program?

```
int main(){

    int a = 260;
    char *p;

    p = &a;

    printf(" a = %d",*p);

}
```

```
C:\Users\m\Documents\Sa
a = 4
```

'p' is holding the address of 'a'. Since 'a' and 'p' don't have same data type, the result of the program is incorrect (4 instead of 260).

A dereferencing operation is preformed inside the printf statement. In other words, there is an attempt to display the value of 'a' through the pointer 'p'. Since the size of data type of 'p' is only one byte while 'a' is stored in 4 bytes, compiler will convert only the first byte of 'a' to a decimal value.

That is why the pointer type should follow the type of variable on which it is pointing.

2.15.3 Pointer arithmetic

Consider the code on the side where the ‘p’ pointer is incremented by one.

```
int main(){

    int a = 5;
    int *p ;
    p = &a;

    printf("the address of a is %d\n",p);

    p = p + 1;

    printf("the next address is %d", p);

}
```

```
the address of a is 6422036
the next address is 6422040
```

‘p’ is incremented by 1 but the its new value shows that it incremented by 4 !

In general, when adding ‘x’ value to the value of the pointer, a value of ‘x’*’n’ will be added to it. Where ‘n’ represents the number of bytes allocated for the data type of the pointer. Since the data type of the ‘p’ pointer is an integer and assuming our compiler allocates 4 bytes for int data type, then when adding 1 to ‘p’ old value, the new value of ‘p’ will be the addition of old value and 4.

Example:

The pointer ‘p’ is holding the address of the first byte of the first element in the array ‘notes’.

Incrementing the pointer value with 3 steps, means moving the pointer to the third element in the array. Decrementing by 1, result in moving the pointer to point on the second element in the array.

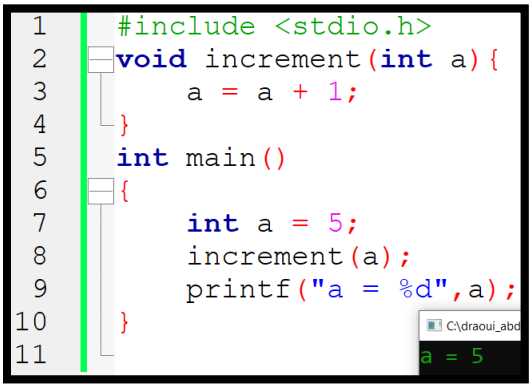
Hence, a pointer can be used to move between elements of the array.

```
int main(){

    float notes[] = {12.5, 9 , 11, 19};

    float *p = &notes;

    printf("value = %f\n",*p);

    p = p + 3;

    printf("new value is %f\n",*p);

    p = p - 1 ;

    printf("newest value is %f",*p);

}
```

```
value = 12.500000
new value is 19.000000
newest value is 11.000000
```

2.16 Passing by value and passing by address

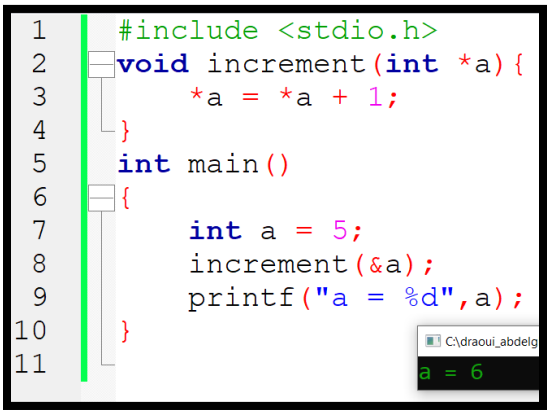
Example : A programmer has written the following code to increment the value of variable 'a' using a subprogram. However, the new value of 'a' has not be updated.



Explanation: Each of the two functions 'main' and 'increment' has its own variable 'a'. So in total we have two separate variables with same name 'a' declared in memory, each in its corresponding function stack frame.

Increment() as a function can only modify the value of its own local variable 'a'. In other words, the value of the 'a' variable in main function will not be affected by the execution of the increment function. In line 8, only the value of 'a' is passed as argument and not the variable itself. This method of passing parameters is referred as **passing by value**.

In order to allow the increment function to modify/access the value of the 'a' variable local to main, the calling main function has to pass the address of that variable to the called function increment. In that case, the formal parameters in the definition of the increment function will have to be modified to a variable type pointer to integer. The corrected code is shown below. This method of passing parameters is referred as **passing by address**.



Note: line 3, the block of memory -to which the local variable 'a' is pointing- is incremented.

2.17 Array and String as exceptions

In this example, the array 'a' is apparently passed by value, but considering the result of execution, we notice that the perm() function has been able to modify the values of element of the 'a' array.

```c
void perm(int a[], int taille){

    for(int i = 0 ; i<taille ; i++){
        a[i]= a[i] + 1;
    }

}

int main(){

    int a[] = {4,5,9};
    int t = 3;
    perm(a,t);
    for(int i = 0 ; i<t ; i++){
        printf("%d\t",a[i]);
    }
}
```

```
C:\Users\m\Documents\SandBoxForC\projet1\
5       6       10
```

In fact, although it appears from the syntax that the array is passed by value, it is passed by address (address of array is passed and not a copy of that array). Same rule apply when passing a string to a certain subprogram.

In summary, we can state that a variable of any datatype can be passed by value or by address, at the exception of arrays and string compound datatypes. By default, arrays and strings are passed by address, for storage efficiency, as this compound variables consume memory.

Note: As size of an array cannot be deduced from its address, each subprogram receiving the address of an array as an argument, size of that array has to be passed as well.

2.18 Recursion

Recursion is a powerful tool in algorithm design. It helps to break down complex problems into smaller sub-problems that are easier to solve. However, it is important to use recursion when appropriate and be aware of its limitations. With practice, student will develop an intuition for when to use recursion in the code.

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.

A recursive algorithm takes one step toward solution and then recursively call itself to further move. The algorithm stops once we reach the smallest version of the problem, called a base case. Since the called function may further call itself, this process might continue forever. So it is essential to provide **that base case** to terminate this recursion process.

2.18.1 Steps to Implement Recursion

Step1 - Define **a base case**: Identify the simplest (or base) case for which the solution is known or trivial. This is the stopping condition for the recursion, as it prevents the function from infinitely calling itself.

Step2 - Define **a recursive case**: Define the problem in terms of smaller subproblems. Break the problem down into smaller versions of itself, and call the function recursively to solve each subproblem.

Step3 - Ensure **the recursion eventually reaches the base case**: Make sure that the recursive function eventually reaches the base case, and does not enter an infinite loop.

Example : Given a natural number 'n', define a recursive subprogram that calculates n! (factorial of n)

n! = 1 * 2 * …….* (n-1)*n

= (n - 1)! * n

Factorial of the smallest natural number 'n' represents the base case, 0! = 1.

We notice that calculating factorial of 'n' requires calculating factorial of 'n-1'. Therefore, we are dealing with an algorithm which is recursive in nature. Fact(n) subprogram can be summarized in mathematical notation as :

$$Fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ Fact(n-1) * n & \text{if } n > 0 \end{cases}$$

Following the above mathematical formula, the recursive subprogram can be easily defined as

Base case

Recursive case

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

Note: Factorial recursive subprogram definition, contains one base case and one recursive case, we may have a recursive program that contains several base cases and/or several recursive case, as it will be shown later for the algorithm calculating fibonnacci sequence.

Example: Given a real number X and a natural number n, define a recursive subprogram that calculates X^n

Given that $X^n = X^{n-1} * X$ and $X^0 = 1$, we notice that calculating X^n requires calculating power of same real number X with a reduced parameter n-1. Among many options $X^0 = 1$ can be selected as a base case. Based on these remarks, we can define the required subprogram as follows.

```
float pui(float x, int n){
    if(n == 0){
        return 1;
    } else {
        return x*pui(x, n-1 );
    }
}
```

Example: Fibonacci number from a Fibonacci sequence represents the number of rabbit pairs in a given month 'n'. Fibonacci sequence can be summarized in the following table

n	1	2	3	4	5	6	7	..
Fib(n)	1	1	2	3	5	8	13	..

- In month one, we had one young pair of rabbits.
- In month two, this young gets older and it is able to give birth to a new pair of rabbits, each month.
- In month three, the pair of rabbits gives birth to new pair.
- Each new pair of rabbits gives a newer pair of rabbits each month, after passing one month to get older enough to give birth.

Question : What is the number of rabbit pair in a given month 'n'.

Answer : one can notice that the number of rabbit pairs in a given month 'n' is the sum of number rabbit pairs in the two previous months n-1 and n-2. Therefore, we can formulate the problem as follows.

$$fib(n) = \begin{cases} 1 & if\ n = 1\ or\ n = 2 \\ fib(n-1) + fib(n-2) & if\ n > 2 \end{cases}$$

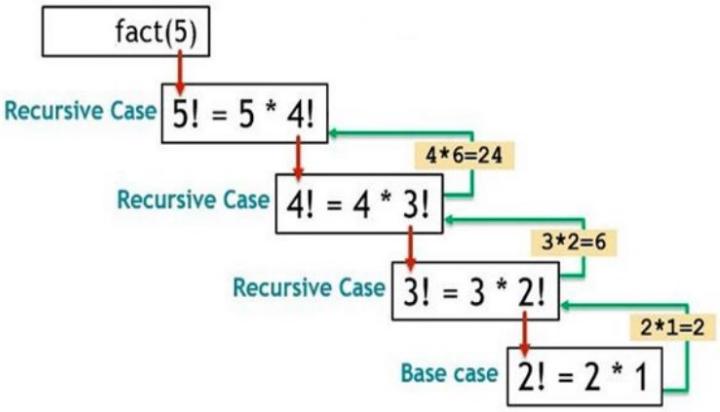
Hence, the recursive subprogram can be deduced easily for that formula.

```
int fib(int n){
    if(n == 0 || n == 1){
        return n;
    } else {
        return fib( n-2 ) + fib( n-1 );
    }
```

2.18.2 Recursion Tree

In order to deduce the output of a recursive function, we can draw the recursion tree. In addition, the number of function calls in a recursion tree is directly proportional to the size of memory occupied during the execution of that program with the given input size 'n'.

Example: Recursion tree of factorial recursive function



According to the definition of the recursive factorial subprogram defined earlier, 5! requires calculation of 4!. 4! requires calculation 3!. Going down to end at a base case, the phase in the recursion tree is called calling phase. When reaching the end of the calling phase, multiplication operation is performed in the reverse direction, giving the final result. The return phase starts from the base case to the display of final result.

2.18.3 Types of Recursion

There are several classifications of recursion. Tail or non-tail recursion is one of these classifications.

Tail recursion is defined as a recursive function in which the recursive call is the last statement that is executed by the function. So basically nothing is left to execute after the recursion call.

```c
#include <stdio.h>
void f(int a){
    if (a < 0)
        return;
    else {
        printf("%d",a);
        // recursive call
        // is the last statment
        f(a-1);
    }
}
```

Note: The factorial example, seen earlier, is an example of a non-tail recursion program.

2.18.3.1 Need for Tail Recursion

Tail-recursive functions are better than non-tail-recursive ones because they can be optimized by the compiler.

The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame can be avoided by simply sending the control back to the beginning of the function either using a loop or goto statement.

2.18.4 Advantages of Recursion

- **Simplifies code**: Recursion often results in cleaner and more readable code, especially when working with problems that have multiple sub-problems.
- **Solves complex problems**: Recursive solutions are intuitive for problems like tree traversal, where each node requires similar treatment.
- **Eliminates the need for complex loops**: Recursion can replace deep nested loops and reduce complexity in writing the code.

2.18.5 Drawbacks of Recursion

Despite its benefits, recursion has a few drawbacks:

Performance issues: Recursive algorithms can be inefficient in terms of time and memory. Every recursive call consumes stack space due to creation of new stack frame, which can lead to a **Stack Overflow Error** if the depth of recursion is too large.

Difficult to debug: Recursive code can sometimes be harder to debug, especially when there are issues with the base or recursive cases.

2.18.6 When Not to Use Recursion

Avoid recursion if:

- The problem can be solved iteratively with simpler code.
- The depth of recursion is too large and may cause memory overhead.
- Performance is critical, and an iterative solution offers a significant performance gain over recursion

2.19 Conclusion

In this chapter, the concept of subprograms has been examined. Subprograms are used extensively to avoid repetition and to organize code. Several types of subprograms has been explained and a brief demonstration of subprogram execution flow is introduced. Student, armed with this fundamental concepts can tackle more advanced concepts related to programming paradigms like the Object Oriented Programming. Details viewed about pointers will be used as a building block to understanding the remaining data structures in the upcoming chapters.

2.20 Exercises

1- Define and show how to call these following subprograms:

1.1. A subprogram that check if a given natural number is even or odd.

1.2.A subprogram that converts an amount of time in seconds to hours, minutes and, seconds.

1.3.A subprogram that calculate the sum of the first ‘n’ natural numbers.

1.4.A subprogram that calculates the average of real numbers stored in an array.

1.5.A subprogram that calculate X^n, where ‘X’ is a real number and ‘n’ is a natural number.

1.6. A subprogram that receives an amount in Algerian dinars to convert it to euros. (Assuming 150 D.A = 1 euro).

1.7.A subprogram with integer parameters ‘a’ and ‘b’ to calculate the greatest common divisor (PGCD in french).

1.8.A subprogram with parameters ‘r’ and ‘h’ to calculate the volume of a cylinder.

1.9.A subprogram that receives the 3 sides of a triangle to display in a message whether it is right triangle is or not.

2- Using pointers correct the following program, to allow the ‘swap’ function to swap values of the variables ‘a’ and ‘b’.

```c
#include <stdio.h>
void swap(int a, int b){
     int temp;
     temp = a;
     a    = b;
     b    = temp;
}
int main()
{
    int a=1, b=5;
    swap(a,b);
    printf("a=%d,b = %d",a,b);
}
```

```
C:\draoui_abdelghani\demo
a=1,b = 5
```

3- For each question, define a recursive subprogram that :

2.1. Converts a number written in binary form to decimal form.

2.2. Counts the digits of a given natural number.

2.3. Counts the sum of the ‘n’ first natural numbers.

4- Write the iterative solution for finding the Fibonacci sequence.

5- For the two subprograms shown below:

4.1.Draw the recursion tree when calling functions m1(2), m1(2),m2(2,2) and, m2(2,3).

4.2. Suggest iterative forms of functions m1 and m2

Chapter 02

Files

3 Chapter 02: Files

3.1 Introduction

So far, the programs we have seen manipulate data that is stored in RAM. RAM is a volatile memory, in other words, all data saved in that memory is lost as soon as the program is closed or if the machine is restarted or turned off. In many cases, data -obtained after processing it in a given program- has to be stored permanently in a non-volatile memory such as hard disk drive. As examples, we mention data stored in databases, configuration files and, logs. Being used for demonstration purposes, the C programming language will be used to show how to open, create, read, update and, delete content of files.

3.2 Reading/Writing to files

Accessing a file -which is stored in a storage device- is a time costly operation. For this reason, data to be inserted to a file is written first in a specified zone in memory, referred as a buffer. Once, the buffer is full, it is unloaded by moving data stored in it to the file in a single access operation.

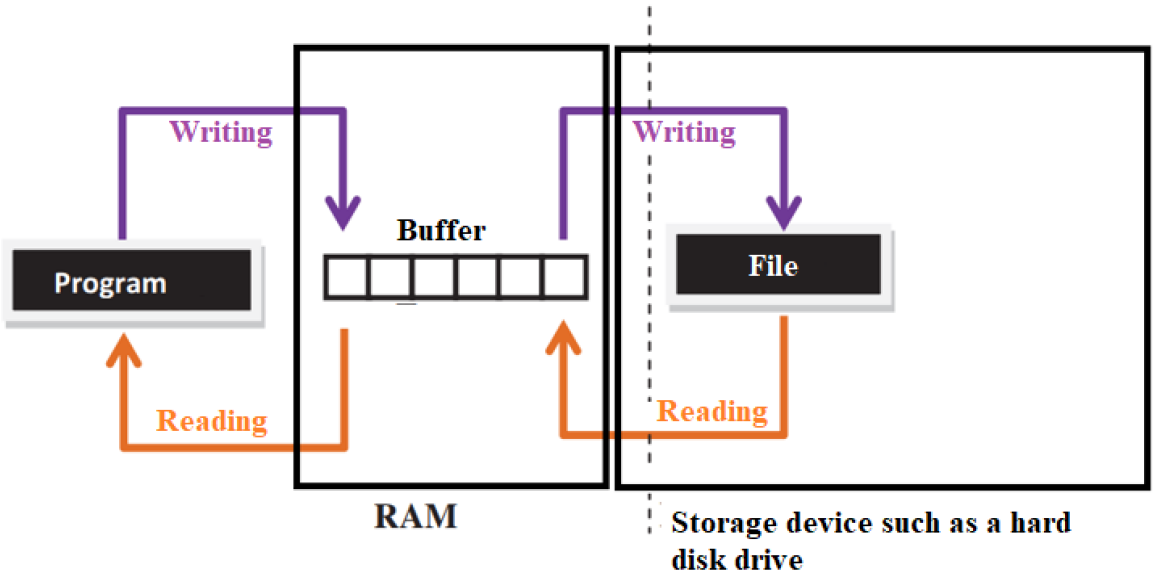


Figure 1 Buffer as intermediate between program and file

3.3 Types of files

We recognize two types of files :

Text file: Contains a text written in one of the character encoding standards such as ASCII. Even with numerical characters, the ten digits, the binary form of its corresponding ASCII code is used for representing these characters. Content of the text file is usually human-readable and can be viewed using any of the text editors.

Binary file: As its name suggests, a binary file contains block of binary numbers. Data is stored in binary format, which is more efficient for machine processing but not human-readable, such as images, executable files, etc.

Several built-in functions involved in the 'stdio.h' library can be used to write/read data from a file. It is often prefixed by the letter 'f' to distinguish it from other function. Fprintf(), fscanf() is an example of these functions.

3.4 Access methods to a file

A file is accessed for writing to or reading data from a file using different methods:

- **Sequential access**: Data can be inserted/read sequentially starting from the first character in the file. It can be insert/read character by character or as a complete string.
    - **Non-formatted sequential access**: data is inserted/read as a simple string that does not involves values of variables in any specific format.
    - **Formatted sequential access**: data is inserted/read as a string containing values of variables. In that case, type conversion (from string to the desired datatype) is performed during the read operation.
    - **Block sequential access**: In this access method, data inserted/read, is moved as a block between program and the targeted file.
- **Direct access**: reading/writing operation is performed from a specific position in file and not from the start of file.

Functions that will be used for accessing files are summarized in the following figure

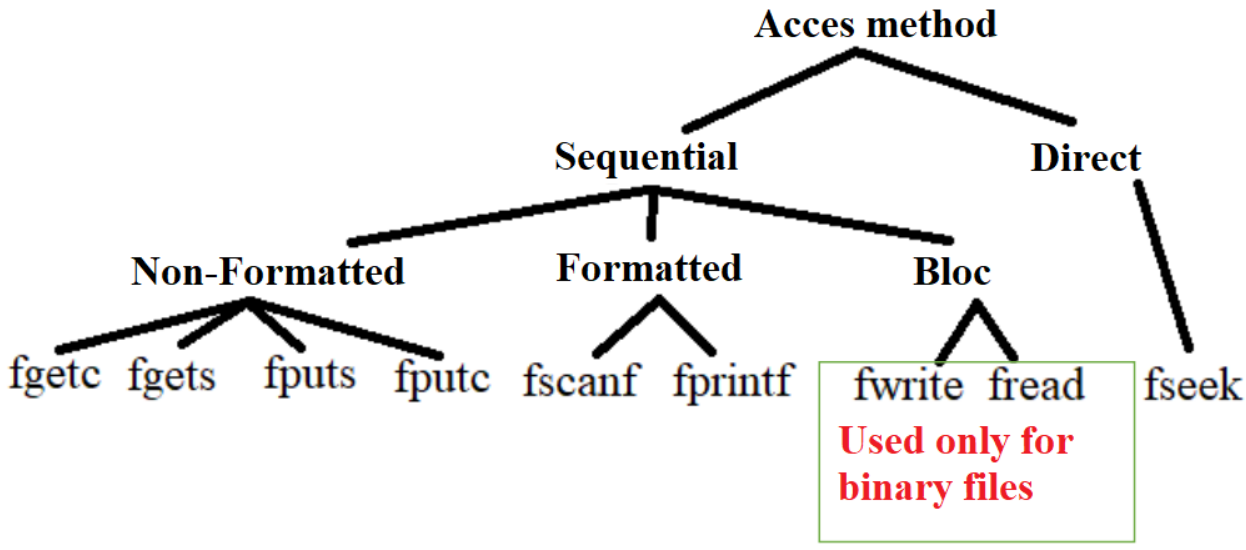


Figure 2 Built-in functions for reading/writing from/to file

3.5 Steps for data manipulation in files

For data manipulation in a file, file is opened, data are moved from/to file then file is closed.

File opening: It has to be opened first while indicating the reason for opening the file, either reading, overwriting or adding data to the existing data in file. The built-in function *Fopen(filepath, openingMode)*, is used where is file path is specified along with the opening mode, 'r' for reading mode, 'w' for writing mode, and 'a' for appending mode. For writing mode 'w', all the old data contained in the file is erased before insertion of new data. On the other hand, for appending mode 'a' new data is modifying the old data.

In case of successful file opening, the *fopen()* function returns an address of file structure. The file structure contains some necessary information about the targeted file like it size, current position of cursor, File address in the storage device…etc. File opening operation may fail due to many reasons such as opening a non-existent file for reading data from it or opening a file protected with a password. In that case, *fopen()* function returns null to indicate an error while opening file. Opening a non-existent file for writing to it, will lead to creation of that file.

For targeting the file for data writing/reading, a variable of type 'pointer to file structure' is created to receive the returned value from the fopen() function.

Data reading/writing : After opening the file, data is either read or written to the file.

Closing file: file is closed mainly to move all remaining data in the buffer to the file and that is in case of writing data to the file. If file is not closed the last data remaining in the buffer will be lost. Closing the file will also free memory allocated for the file structure created in RAM.

Code below show a boilerplate code for file manipulation inside main function.

```
int main(){

    FILE *fp;// pointeur vers le fichier

    // overture pour ecriture (w = write)
    fp = fopen("fichier1.txt","w");


    // traitement ( ecriture )


    // fermuture du fichier
    fclose(fp);
}
```

Figure 3 Opening, writing/read, closing file

In this example, file with a name 'fichier1.txt' is opened for writing new data in it. Since the opening mode 'w' is specified, all old data will be lost. If the file with such name does not exist, this file is created.

In this example, only the file name is specified and not the complete file path. It is a relative file path indicating the file in question exist in the same location of the project file.

3.6 Writing to a file

3.6.1 Sequential insertion of data character by character

Writing new data to a file while erasing all old data in that file requires indicating the file path along with the reading mode 'w'.

Writing this new data can be performed character by character right from the first line in the file. In that case, the built-in 'fputc(character,PtrToFileStructure)' is used. The latter function receives the character to be inserted as first input and the pointer to file structure as a second output, example below show code for insertion of the word 'Math' character by character.

Fputc() may cause an error if the storage device in which the file is full.

```
int main(){

    FILE *fp;// pointeur vers le fichier


    // overture pour ecriture (w = write)
    fp = fopen("fichier1.txt","w");


    // traitement
    fputc('M',fp);
    fputc('a',fp);
    fputc('t',fp);
    fputc('h',fp);


    // fermuture du fichier
    fclose(fp);
}
```

fichier1.txt
1 Math

Figure 4 example of writing to file word by word

3.6.2 Sequential Insertion of data string by string

Data can be inserted in a file a string by string using the fputs(str, pointerToFile) function. This function receives as input the string to be inserted (str) along with the address of the File in which the insertion is to be done.

```
int main()
{
    FILE *fp ;// pointeur vers un fichier

    fp = fopen("fichier1.txt","w");

    fputs("Centre ",fp);
    fputs("Univ ",fp);
    fputs("Tindouf ",fp);

    fclose(fp);
}
```

fichier1.txt
1 Centre Univ Tindouf

Figure 5 Example of writing to file string by string

3.6.3 Formatted Data insertion

The data to be inserted in file can involve values of variables read from the user input. In this case, we use the fprintf(PointerToFile, formattedString, variable(s)) function. This latter function receives as first input the address of the pointer to file, the formatting string containing format specifiers and the variables to be inserted in the formatted message. Example below show the insertion of formatted data obtained from the user input.

```c
int main()
{
    FILE *fp ;

    fp = fopen("fichier1.txt","w");

    int age,compteur=1;
    do {
        printf("enter age value or -1 to quit : ");
        scanf("%d",&age);
        if(age == -1) break;
        fprintf(fp,"age %d = %d\n",compteur,age);
        compteur++;
    } while (1);

    fclose(fp);
}
```

```
C:\Users\m\Desktop\files\file_manip\bin\Debug\file_manip.exe
enter age value or -1 to quit : 8
enter age value or -1 to quit : 14
enter age value or -1 to quit : 45
enter age value or -1 to quit : 25
enter age value or -1 to quit : -1
```

```
fichier1.txt
1  age 1 = 8
2  age 2 = 14
3  age 3 = 45
4  age 4 = 25
5
```

Figure 6 Formatted insertion of data in a file

3.6.4 Write and Append modes

If the append opening mode is specified (‘a’ given as argument instead of ‘w’), the new inserted data will be added to the already existing data in the file. Consider the following example where the file ‘fichier1.txt’ contains the string ‘Centre Univ’ and the new string ‘ Tindouf’ is added to it.

```c
int main(){

    FILE *fp;// pointeur vers le fichier

    // overture pour traitement r,w, ou a
    fp = fopen("fichier1.txt","a");


    // traitement
    fputs(" Tindouf",fp);

    // fermuture du fichier
    fclose(fp);
}
```

```
fichier1.txt
1  Centre Univ
```

```
fichier1.txt
1  Centre Univ Tindouf
```

Figure 7 difference between append and write modes

3.7 Reading data from a text file

We recall that opening a file for reading its content may not be succeed especially one the file is protected file or if the file is inexistent in the specified file path. The fopen() function returns a 'NULL' value in case of such errors. Therefore, code for reading the content of a file is restricted to the successful opening of a file (i.e. reading only if the fopen() function does not return NULL value).

3.7.1 Sequential reading of data character by character

File content can be read a character by character or a string by string. A text file has the special character ***'\n'*** at the end of each line while an End Of File (***EOF)*** special character is used to signal the end of content of that given file. The fgetc(PtrToFileStructure) built-in function is used to read content of a file, character by character. This function receives the address of the file structure related to the file in question. It returns the read character at the current position in file and increment the current position. It returns the EOF character to signal the end of file. The current position in file is among pieces of information contained in the File structure of the related File. Reading the entire content of the File is performed by including the fgetc() function in a loop, as shown in the example below.

```c
int main(){

    FILE *fp;// pointeur vers le fichier


    // overture pour ecriture (w = write)
    fp = fopen("fichier1.txt","r");


    // traitement
    if(fp == NULL){
        printf("fichier introuvable ou acces interdit");
    }else {
        char lettre = fgetc(fp);
        while(lettre != EOF){
            printf("%c",lettre);
            lettre = fgetc(fp);
        }
    }


    // fermuture du fichier
    fclose(fp);
}
```

fichier1.txt

```
1 centre univ tindouf
2 informatique et math
```

```
C:\Users\m\Desktop\tindouf\manipulation\b
centre univ tindouf
informatique et math
```

Figure 8 Reading content of file line by line

3.7.2 Sequential reading of unformatted data a string by string

Reading content of a text file can be performed in a string by string manner. The fgets(str,strLength, PtrToFileStructure) receives as input the variable in which the read string will be stored, the length of the string to be read from the File and, the address of File structure of the file in question. The fgets fuction returns the read string in the str variable and returns a non-Null value. In case of error or reaching the End of file, this function will return a NULL value. In the example below, content of a text file is read and displayed in the console window.

```c
int main(){

    FILE *fp;// pointeur vers le fichier

    // overture pour ecriture (w = write)
    fp = fopen("fichier1.txt","r");


    // traitement ( ecriture )
    if(fp == NULL)
        printf("fichier introuvable ou acces interdit");
    else{
        char chaine[100];
        char *drapeau = fgets(chaine,100,fp);

        while(drapeau != NULL){
            printf("%s",chaine);
            drapeau = fgets(chaine,100,fp);
        }

    }

    // fermuture du fichier
    fclose(fp);
}
```

fichier1.txt

```
1 centre univ tindouf
2 institut science et technologie
3 departement math et info
```

```
centre univ tindouf
institut science et technologie
departement math et info
```

Figure 9 Formatted reading of data from a file

Note: The fgets() function returns a string with the specified length, unless it reaches the end of line in which the string has started. In the latter case, the read string will be from the current position in file to the end of line character '\n'.

3.7.3 Sequential reading of formatted content from a file

Assuming a numerical data in stored in a text file, reading strings representing these numbers and implicitly converting it for use in the program can be achieved using the fscanf(PtrToFileStructure,formatSpecifier, PtrToVariable) function. This function requires as input the address of the file structure of the related file, the format specifier of the given stored numbers and, the address of the variable in which the read number will be stored. Knowing the fscanf() function returns an EOF character to signal the end of the file, the Entire content of the file can be read by using the output of this function as a stopping condition for a loop as shown in example below. The example shows how to grades of a student as a real number to calculate the average of these grades.

```c
int main(){

    FILE *fp;// pointeur vers le fichier

    // overture pour traitement r,w, ou a
    fp = fopen("fichier1.txt","r");


    // traitement
    float somme=0,note,moyenne;
    int compteur=0;
    int drapeau = fscanf(fp,"%f",&note);

    while( drapeau != EOF){
        compteur++;
        somme = somme + note;
        drapeau = fscanf(fp,"%f",&note);

    }
    moyenne = somme/ compteur;
    printf("la moyenne des notes est %f",moyenne);


    // fermuture du fichier
    fclose(fp);
}
```

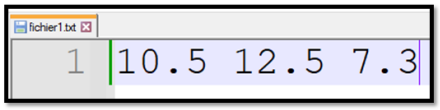
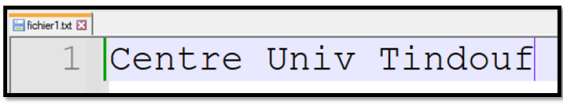


Figure 10 Example of reading formatted input from a file

3.8 Direct access to File

Unlike the sequential access examples seen so far, in a direct access, writing/reading content to/from a file can be achieved starting from a specified position in the file. In this case, moving the cursor (setting the current position in a file) is performed by the fseek() function. This function receives as argument, the address to the file structure of the related file, the incrementation step (the offset) starting from an origin given as third argument. The origin may be the beginning of file (0 given as argument), the end of file (2 given as argument and step has to be negative in this case) or, the current file position (1 is given as argument).

```c
int main(){

    FILE *fp;// pointeur vers le fichier

    // overture pour traitement r,w, ou a
    fp = fopen("fichier1.txt","r");


    // traitement

    long position = ftell(fp);
    printf("old position = %ld \n",position);

    fseek(fp,7,0);

    position = ftell(fp);
    printf("new position = %ld \n",position);

    char lettre = fgetc(fp);
    printf("lettre  = %c \n",lettre);

    rewind(fp);

    position = ftell(fp);
    printf("newest position = %ld \n",position);


    // fermuture du fichier
    fclose(fp);
}
```



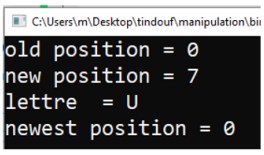


Figure 11 Direct access method for file editing

After moving the cursor to desired position reading/writing operation can be performed. In addition to the fseek() function, rewind() and ftell() functions are used frequently in direct access method. The rewind() function receives as argument the address of file structure of the related file. It resets the cursor current position to the beginning of the file. On the other hand, ftell() function receives the address of file structure of the file in question to return the current cursor position in that file. Example below shows how to use these three functions.

3.9 Renaming/Deleting a file

A file can be renamed or deleted using the rename() and remove() functions respectively. Example below show how to use these functions

```
// rename file having name fichier1 to file1
rename("fichier1.txt","file1.txt");

// to remove file2.txt file
remove("file2.txt");
```

Figure 12 Example of use of remove and rename functions

3.10 Conclusion

For saving data permanently in a storage device such as the hard disk drive, the C language provides a set of built-in function from the stdio library. Writing/Reading content of a file can be achieved using different access methods. From the basic examples exposed in this chapter, the student will be able to design a complete File management system.

3.11 Exercises

1- Using the built-in functions shown in this chapter, build a File Management System (FMS) that allow storing data of students of our CS department. The system has the following features :

- **a.** A file with name fms.txt is created in the project folder.
- **b.** The FMS allows insertion –in a new line- of new of pieces of information of a new student which are : register id number, first name , last name, age, gender
- **c.** The FMS allows updating pieces of information of a given student with a given student id number.
- **d.** The FMS allows deletion of a student record knowing his student id number.
- **e.** The FMS allows counting number of registered students.

Chapter 03

Linked List

4 Chapter 03: Linked List Data Structure

4.1 Introduction

Efficiency of an algorithm -in terms of execution time and memory usage- depends heavily on the way the data of that algorithm is stored. Depending on the complexity of the program, storing data can be achieved using a specific data structure or a combination of data structures. We recall that a data structure is a way of storing and organizing data in memory in order to facilitate access and modification of that data. The only data structure we have studied -so far- is the array. In this chapter, a new data structure referred as 'a Linked list' will be studied.

4.2 Storing data in arrays

If a data with a known size and data type is to be stored in our program, array data structure is used. Access to array elements is fast as only the index of the targeted element is needed for accomplishing this operation. However, in most cases, the exact number of elements to be stored in is not known and it may depend on the user input to the program. Declaring a large array as a preventive measure, may lead to unnecessary memory consumption, as too many cells will be allocated but unused. Thanks to the new data structure, linked list, slot of memory are allocated, only when it is needed. Its size is dynamic meaning it can be changed in the run time.

In addition, if data to be manipulated in the program, represents an entity with multiple attributes where each attribute has a different data type, storing a group of this type of data in an array is impossible. It is impossible because the datatype of the array elements has to be identical. Using a linked list, it elements can accommodate for variables with different data types.

In most programming languages, the attempt to change the size the array will raise a syntax error preventing the achievement of compilation process. The reason behind this restriction is that the array elements are stored in adjacent memory blocks and changing the array size dynamically may result in overwriting values of variable stored near the array in memory. Consider this example, which illustrate this issue.

```
10  int main(){
11
12      int a[] = {4,5,9};// addresses 104,108,112
13      int b = 5; // address 100
14      int c = 13;// address 116
15
16      int a[3] = 5;
17      printf("element 4 = ",a[3]);
18  }
19
```

```
Logs & others
Code::Blocks | Search results | Cccc | Build log | Build messages | CppCheck/Vera++ | CppCheck/Vera++ messages

File                Line  Message
                          === Build: Debug in sandBox (compiler: GNU GCC Compiler) ===
C:\Users\m\...            In function 'main':
C:\Users\m\...      16    error: redefinition of 'a'
C:\Users\m\...      12    note: previous definition of 'a' was here
```

Figure 1 Syntax Error due to attempt to change array size

In some programming languages, redefining the size the array is performed by creating a new array with a new size then moving the elements in the old array to the new one. However, this method is time costly especially when the new array size is very large.

In other words, we say that array are static data structure, meaning that its size cannot be changed in the run time or it is said that arrays cannot be changed dynamically.

4.3 Dynamic memory management

We have seen that local variables are created in the stack frame of the function in which it is used. The memory size to be allocated for storing variables in pre-defined in the compilation process. On the other hand, a space in a memory can be allocated dynamically for a variable with a desired size (i.e. in the run time of the program) and that is in the heap section of memory. The allocated space in memory can be accessed/modified through a pointer holding its address.

In C language this allocated space of memory has to be freed 'manually' by explicitly stating it the program, otherwise , a memory leak will occur due to occupation of an inaccessible space in memory. In other programming language such as Java, freeing occupied memory is performed automatically by the garbage collector (further details about this concept will be studied in Object Oriented Programming module).

In the example below, a simple program demonstrating the dynamic memory management is illustrated. Two slots of memory are created to manipulate. One slot via a variable while manipulating the other slot via a pointer. The latter slot is manipulated using a pointe because it is dynamically allocated in the heap section of memory. The scope of the second slot of memory is global as it can be changed by any subprogram having its address.

```
   #include <stdio.h>
   #include <stdlib.h>
1  int main(){
2    int a = 4;

3    int *p;

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

5    *p = 5;
6    printf(" valeur est %d",*p);

7    free(p);
8    p = NULL;

   }
```

```
C:\Users\m\Documents\SandBoxForC\sandBox\bir
valeur est 5
```

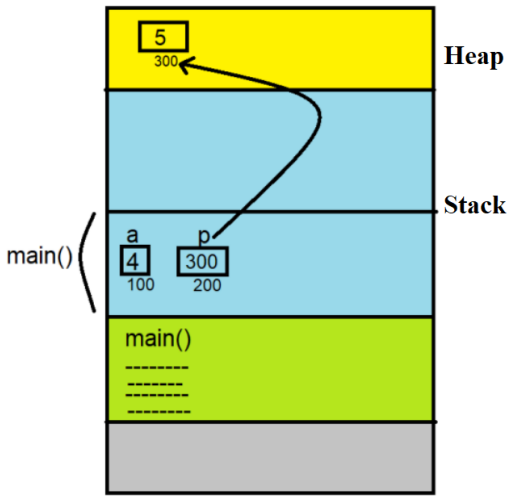


Figure 2 State of memory after execution of line 6

For dynamic allocation of memory in the heap section of the memory, the built-in 'malloc' function is used. It stands for **m**emory **alloc**ation. It allocate a memory of size given as an argument then it returns the address of that allocated memory. This function returns a general pointer, therefore, its returned value must be type casted (converted) to the same pointer type of the variable

receiving its value. This operation is necessary for correct dereferencing operation. Dereferencing is performed in line 6, to display the value of memory whose address is stored in the pointer 'p'. The built-in free() function is used to free up the memory slot previously allocated.

In this program, a space for use for storing an integer number is desired. Since that size allocated for variables of type 'int' varies depending on the compiler used, sizeof() function is used to specify the correct size of 'int' as set by the used compiler.

In summary, the example above shows how to allocate, exploit and, free a space in memory and that is in the run time. The size of the allocated memory space can be specified by the user input as well, an entry that cannot be predicted for the compiler to allocate it statically.

A space in memory can be allocated dynamically for more complex data types such as structures of a certain entity. Below is an example of a program along with the memory sections involved in the execution of the program.

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

struct wilaya {
    int matricule;
    float classement;
    char categorie;
};
int main(){
1   struct wilaya *p ; // pointeur vers structure wilaya
2   p = (struct wilaya *)malloc(sizeof(struct wilaya));
3   p->matricule =37;
4   p->categorie ='e';
5   p->classement =2.8;
6   p->classement =p->classement + 4 ;
7   printf("Categorie de cette wilaya est %c",p->categorie);
8   free(p);
9   p = NULL;
}
```

```
C:\Users\m\Documents\SandBoxForC\sandBox\bin\Debug\sandBox.exe
Categorie de cette wilaya est e
```

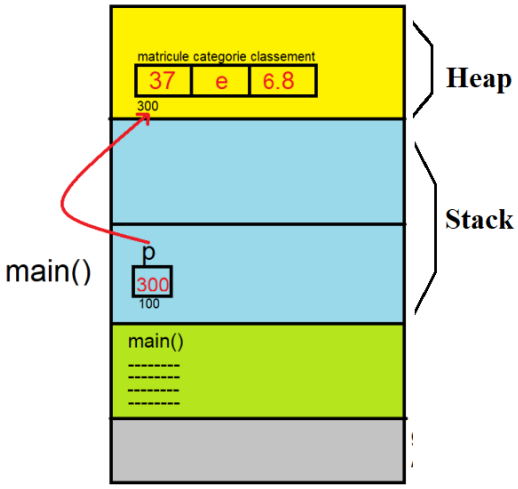


Figure 3 memory allocation for structure wilaya datatype

An entity 'wilaya' with attributes 'matricule', 'classement' and, 'category' is defined as a structure. A space having size of this structure is allocated in heap section of memory using the malloc() function. Attributes of this entity represented in a form of a structure are accessed/modified through the pointer 'p' which holds the address of this structure. As shown in lines 3 to 7, the access to any attribute is achieved using the arrow notation ('->'). This notation is a shortcut for dereferencing operation.

In short, for dynamic memory usage (i.e. for usage in the run time), memory is allocated using the malloc() function. That memory block is accessed through a pointer, and it is freed (deallocated) through the same pointer using the free() function.

Note: There are several alternatives of malloc() function with some minor differences in usage such as 'calloc()' and 'realloc()'.

4.4 Linked List as Abstract Data Structure

We recall that the array size cannot be changed in the run time and that the array can hold only elements with same data type. In addition, declaration of large size arrays for use in program, may leave too many array cells unused leading to inefficient use of memory. Thanks to the dynamic memory management, a block of memory is created and added to the list of old memory blocks to form a list with the optimal number of needed space.

Since array elements are stored in adjacent blocks of memory, knowing the address of the first element in the array is the only needed information for accessing the remaining elements through the use of pointer arithmetic seen earlier in chapter 2. The memory blocks allocated to form a list are not successive but have random values. To link between these blocks of memory created dynamically, each memory block is redefined as two fields to accommodate for two data. The additional field will be used to hold the address of memory block next to it. In that case, knowledge of the address of the first memory block is a sufficient information for accessing data stored in the remaining blocks of memory. This is **the linked list as an abstract data type**, where implementation details are not discussed at this stage and that is for sake of simplicity. The picture below shows how same collection of data is stored in an array and in a linked list.

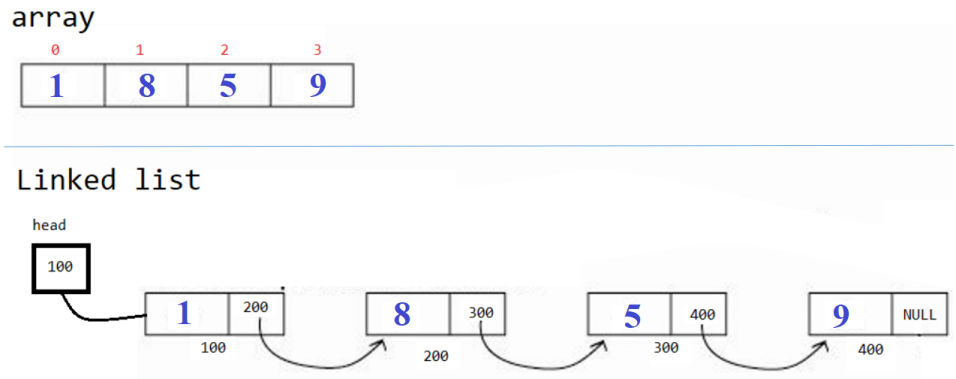


Figure 4 Same data stored in an array and in a linked list

The element of the linked list is denoted as a 'node'. Each node contains two fields a field for data and an additional field that stores the address of the node next to it. Obviously, since no node exist after the last node, the next node address field in the last node is 0 (or NULL). To access the elements of the linked list, the address of the first node is stored in a separate pointer variable called a head.

Moving between nodes in a linked list is performed by a variable of type pointer to a node. Its initial value is same as head value. To move forward the value of this pointer variable is updated with the value of the second field (field containing address of the next node) of the node to which it is pointing.

4.5 Types of linked lists

The example above shows a collection of data stored in a **Singly Linked List (SLL)**. It is qualified as *linear* since the elements of the linked list are linked sequentially (one after the other). It is ***singly*** linked list, since each node holds the address of one single node which is the node next to it. Knowing the address of first node -which is stored in the head pointer variable- we can move forward to access any node.

To move forward and backward in a linked list, we can redefine the nodes of the linked list as having three fields. A field that holds the data, a second field that holds the address of the next node and a third field that holds the address of the previous node. This type of linked list is referred as **Doubly Linked List (DLL)**, an example of this list is illustrated below.

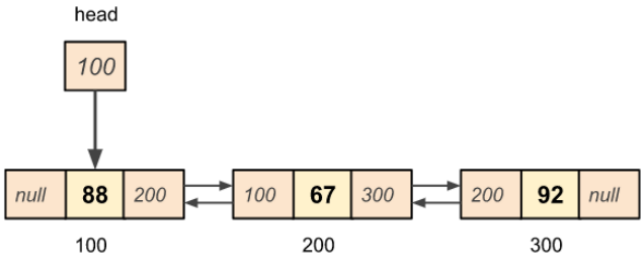


Figure 5 Example of a doubly linked list

Although storing data in a DLL allows moving easily from any node to any other node, the space required to store data in a DLL is tripled as compared to the size of an array storing same data. In addition, operations on this type of a linked list such as insertion of a new node or deletion, it demands modification of several fields in the linked list to preserve its circularity.

In an SLL, the address of next node field in the last field is always null, this field can be exploited by storing the address of the first node in it forming in a **Circular Linked List (CLL)**. Example below shows an example of a CLL.

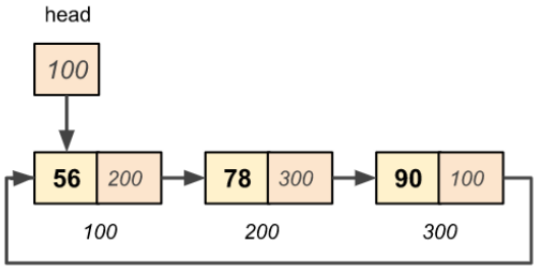


Figure 6 Example of a circular linked list

In a CLL as data structure and using a pointer variable, displacement from any node to any other node can be done easily. However, it is a time consuming operation if we want to move from a node to the node before it.

4.6 Implementation of Singly Linked List (SLL)

Discussing the concept of a linked list (LL) as an abstract data type, helps understanding the general idea behind the data structure. After that, passing to implementation is a necessary to benefit from this data structure in our program. In this section, we consider only the implementation of the SLL. Generalization to DLL and CLL is left as exercises to the student.

The SLL is a collection of interconnected nodes. The type and number data to be stored in each node depends on the nature of attributes of the entity which are representing as a node. For sake of simplicity, we assume that we are dealing an entity having one single attribute of type integer. In that case, the node must be defined to contain two fields, a data field of integer type and a field of type pointer to node for storing the address of the next node. Therefore, a complex variable of type structure can be used to define the node. The linked list is identified in the program by its head, a variable of type pointer to node and that is used for storing the address of the first node. An empty LL means the head value is NULL.

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

struct node *head;
```

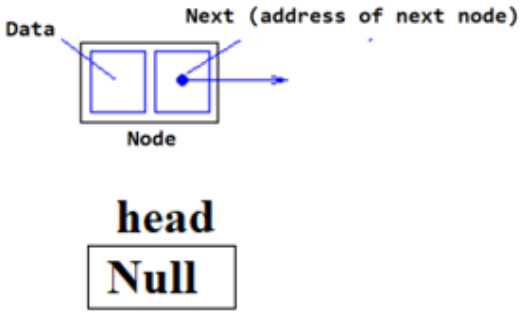


Figure 7 Definition of a node using structures

Note: Implementation of DLL requires redefinition of node as having three fields, field for storing data, a field for storing address of the next node and another field for storing the address of the previous node.

4.7 Frequent operations on SLL

Several operations can be performed on data stored in a linked list. The most frequent operations are insertion of a new node, deletion of a node, displaying data stored in nodes and, counting number of nodes. In this section, we will consider the implementation of insertion at the beginning of the LL, deletion of the first node and, displaying of data in nodes of the LL.

4.7.1 Insertion of a node at the beginning of the SLL

Insertion of a node at the beginning of the SLL, is achieved in several steps, steps are illustrated in figure below:

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

struct node *head;

void main(){

1   head = NULL;

2   struct node *temp = NULL;
3   temp = (struct node *)malloc(sizeof(struct node));

4   temp->data = 34;
5   temp->next = NULL;

6   head = temp;

}
```

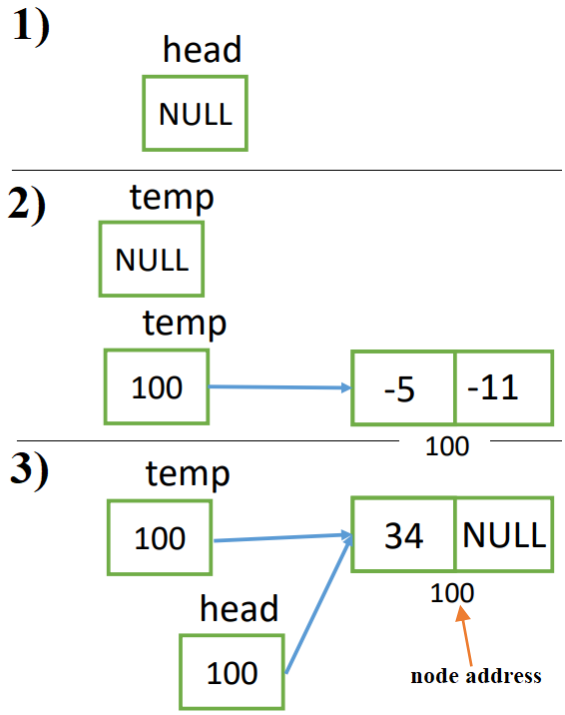


Figure 8 Example of Creation of a LL with one node

The steps

1. Initialization of head variable to NULL signaling creation an empty linked list.(see line 1 in code).
2. Creation of a new node with random field values. To achieve this step, a memory space of size equal to the size of structure node is allocated. Address of that node is saved in a temporary pointer variable 'temp' (see lines 2 and 3 in code).
3. Content of node fields are filled using the 'temp' pointer. (see lines 4 and 5 in code)
4. Head pointer is updated to the address of the newly created node (see line 6).

The illustration part 3 in the figure above represents a linked list created in the heap section of memory. It contains a linked list composed of one single variable. Insertion of multiple nodes to the linked list can be done by repeating the code inside main function multiple times. Or to avoid code repetition of that repeated code is written in a form of a subprogram as shown below

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

struct node *head;

void insertAtBeginning(int value){
    struct node *temp = NULL;

    temp = (struct node *)malloc(sizeof(struct node));

    temp->data = value;
    temp->next = head;

    head = temp;

}
void main(){

    head = NULL;

    insertAtBeginning(34);
    insertAtBeginning(74);

}
```

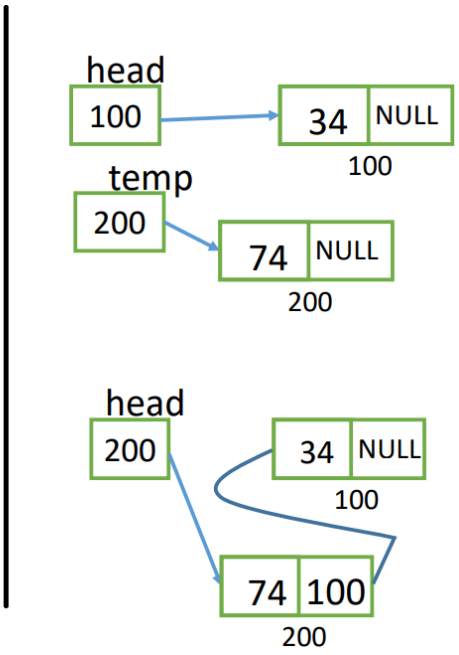


Figure 9 Subprogram creation for insertion of nodes at beginning

4.7.2 Displaying of data stored in a SLL

Just like the case when a collection of data is stored in an array, displaying of data stored in an SLL is performed starting from data in the first node to the last node. The idea is to create a pointer to node having an initial value same as head. We use that pointer to display data in that node and then update the pointer value to address of next node. Iteration is stopped when reaching the last node. This later is recognized by the NULL value of its second field, the next node address field.

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

struct node *head;

void insertAtBeginning(int value){
    struct node *temp = NULL;

    temp = (struct node *)malloc(sizeof(struct node));

    temp->data = value;
    temp->next = head;

    head = temp;

}
void displayList(){
    struct node *r=head;
    while(r != NULL){
        printf("%d\t",r->data);
        r = r->next;
    }
}
void main(){

    head = NULL;

    insertAtBeginning(34);
    insertAtBeginning(74);

    displayList();

}
```

```
C:\Users\rm\Documents\SandBoxFc
74      34
```

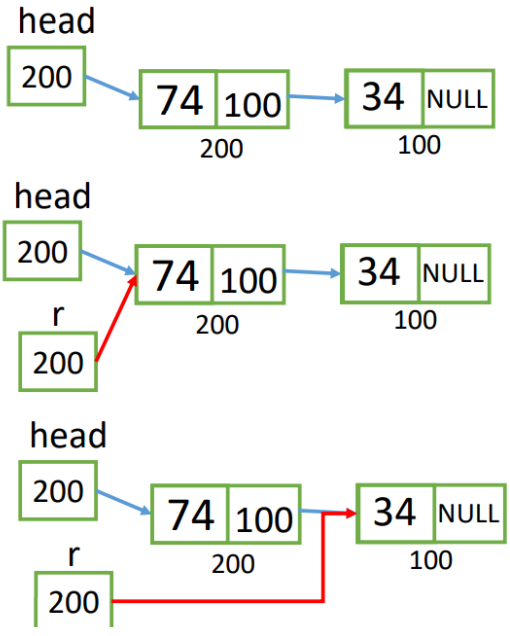


Figure 10 Function definition to display data in a LL

Example above demonstrates creation of a linked list with data values of 34 and 74. A display function is implemented to show this data.

4.7.3 Deletion of the first node in a SLL

In essence, deleting the first node in an SLL, consists of moving the head pointer to point on the second node, since this node will be the first node after deletion. However, moving the head pointer to point on second node will cause losing the address of the first node and therefore it cannot be targeted to be deleted using free() function. Hence, before moving the head to the second node, the address of the first node is stored in temporary pointer variable. Head pointer is moved to point on the second node and then the temporary pointer is used to delete the first node since it contains its address. Example below show the implementation of these steps in a form of a subprogram along with the content of heap section of memory.

```
struct node {

 struct node *head;

void insertAtBeginning(int value){

void displayList(){

void deleteFromBegining(){
    if(head == NULL){
        printf("list is empty !");
    } else {
        struct node *r=head->next;
        free(head);
        head = r;
    }
}

void main(){

    head = NULL;

    insertAtBeginning(34);
    insertAtBeginning(74);

    deleteFromBegining();

     displayList();
}
```

```
C:\Users\m\Documents\SandBoxFo
34
```

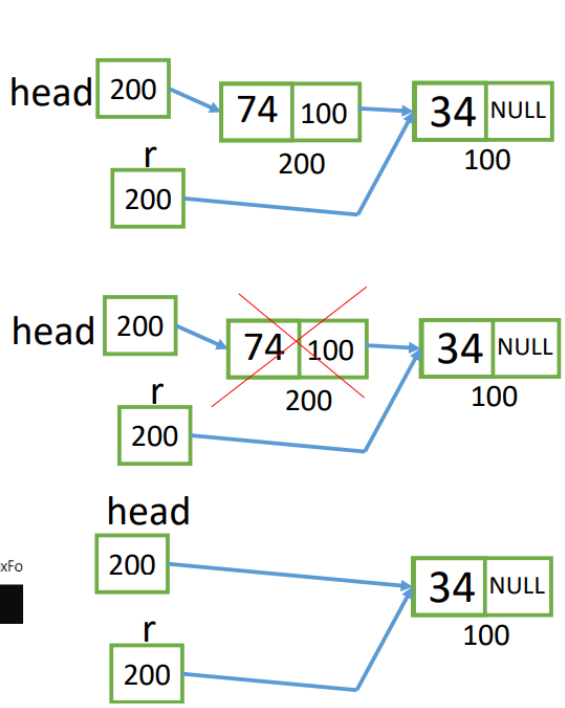


Figure 11 Function definition for deletion of first node

If the linked list is already empty, which can be deduced if head value is NULL. In that case, no node to delete, user is preferably informed via a message as shown in code above.

Note: If a given algorithm requires dealing with more than one linked list, function will be redefined such that the head of the targeted linked list is given as argument and head value is returned by every insertion and deletion functions. These functions redefinitions are required to target the linked list on which the operation will be done.

4.7.4 Comparison between Array and Singly linked List Data Structures

Tableau 1 Comparison of array and a SLL

Array	Simply Linked list
Elements of the array are stored in adjacent blocks of memory	Nodes are scattered in memory
Size is fixed (must be specified in compilation time)	Size can be changed dynamically (dynamically = at the running time)
Consume lesser space for storing data	Consume more space, since each node has address of node next to it.
Quick access to elements	Slow access (since we need to move from first node to the targeted node)
Insertion and deletion operations are slower	Deletion and insertion operations are faster

Exercise: On the light of the given implementation of operations above:

1- Define a subprogram to insert a node at the end of an SLL.
2- Define a subprogram to delete the last node from an SLL.
3- Define a subprogram to count number of nodes.
4- Define a subprogram to insert a node at the n^{th} position.

4.8 Physical and Logical data structures

Data structures can be classified into physical data structures and logical data structures. Physical data structures are the low-level, concrete representations of data in memory or storage (RAM, SSDs, Hard Drives), determining how data is physically laid out, addressed, and manipulated. Common types include arrays (contiguous memory) and linked lists (non-contiguous, pointer-based). Unlike physical structures, logical data structures define performance, storage efficiency, and memory management. Among the logical data structures we can cite the Stack and Queue data structures. Any one of these data structures can be implemented using any of the physical data structures.

4.9 Stack as Abstract Data Type

A stack is a linear data structure where insertion and deletion from this data structure is performed on one side only. Therefore, the last inserted element will be the first element to deleted. We say the stack is a Last In First Out (LIFO) data structure. Stack data structure is used to implement operations on more advanced data structures likes trees and graphs.

As stack can be visualized as an organized elements stored in a container with one of its side is closed, see figure below.

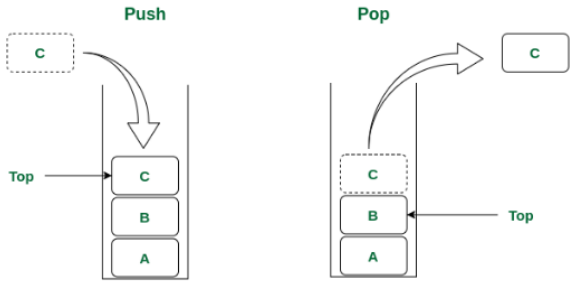


Figure 12 Stack as an abstract data type

There a several operations that can be performed on data stored in a stack:

- **Push operation** : it is an operation of insertion of a new element in stack.
- **Pop operation**: it consists of deleting the last inserted element from the stack and returning it for use in the program.

In addition to these two operations, the element at the top of the stack is tracked and the emptiness of the stack is tracked as well.

4.10 Stack implementation using array

A stack can be implemented statically using a one dimensional array. The drawback of this implementation are inherited from drawbacks of an array. Since the stack is implemented as an array, its size constant and cannot be changed dynamically. In figure below, three states of the stack are illustrated.

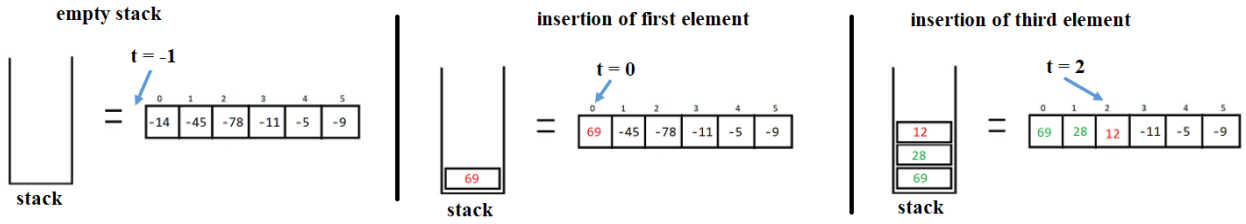


Figure 13 Stack implemented using array

An empty stack implemented as an array, is an array with arbitrary values. To track to the top of the stack, its index is stored in a separate variable 't'. In that case, 't' is equal to -1 when the stack is empty. If the stack is not empty, only the elements of the array whose indices are from 0 to 't' represent the elements of the stack.

4.11 Implementation of push operation

A push operation can be implemented in a form of a subprogram, push(value, stack). The push function receives as arguments the value of the element to be pushed and name of the stack in which the insertion to be performed. Figure above helps us to define this function. Insertion of value 'v' in stack 's' consist of incrementing 't' then placing 'v' as element index 't' in stack 's'. Since the size of the stack is limited, making sure that the 't' value does not exceed the last index value is a preliminary operation before proceeding to insertion. In that case, we would have a stack overflow. Pseudocode of the push() function is shown below.

```
stack1[0..n-1] : array of integerts
n : integer
s : integer
s <- -1

procedure push(x:int, stack[]:array of int)
var
begin
    if( s = n-1)
        print('stack overflow')
        return
    EndIf
    s++
    stack[s] <- x
end
```

4.12 Definition of isEmpty() function

IsEmpty() function is an auxiliary function conceived to help us other operations on the stack. IsEmpty(stackName) function receives as an argument the stack name to return true or false values as answer to the question 'is the stack empty'. The 't' value is the flag that tells whether the stack is empty or not. IsEmpty() pseudocode is mentioned below:

```
fct IsEmpty(x:int, stack[]:array of int): boolean
var
begin
    if s = - 1 then
        return 1
    EndIf
    return 0
end
```

4.13 Definition of Top() function

The auxiliary Top() function, receives the stack name to return the value at the top, in other words the value of the last inserted element. If the stack is not empty, the top element corresponds to the element whose index is 't'

```
fct Top(stack[]:array of int): integer
var
begin
    if (isEmpty(stack)) then
        print('stack is empty')
        return -1
    EndIf
    return stack[s]
end
```

4.14 Implementation of pop operation

Pop operation consists of deleting the last inserted element from the stack and returning its value for use in the program. Pop(stackName) function receives as an input the stack name and returns the last inserted element. Checking that the stack is not empty before deletion then returning the top element is the basic logic behind the function implementation. 't' value is decremented to exclude to returned element from the stack. Both auxiliary functions top() and isEmpty() are exploited in the implementation of pop() function.

```
fct pop(stack[]:array of int): integer
var
begin
    if (isEmpty(stack)) then
        print('stack is already empty')
        return -1
    EndIf
    s--
    return stack[s+1]
end
```

4.15 Stack implementation using SLL

A stack can be implemented using a singly linked list. With the restriction of insertion and deletion from one side of the linked list. Implementing a stack as a linked list allows for efficient use of memory. In other words, an additional space of memory is allocated only when needed. In addition, size of the stack can change dynamically. Figure below shows this implementation

Implementation of push and pop operation corresponds to insertion and deletion at the beginning of the linked list or it can corresponds to insertion and deletion at the end of the linked list. For speed of execution, the first option is often adopted. Knowing whether the stack is empty or not corresponds to checking whether the value of head pointer is Null or not Null, respectively.

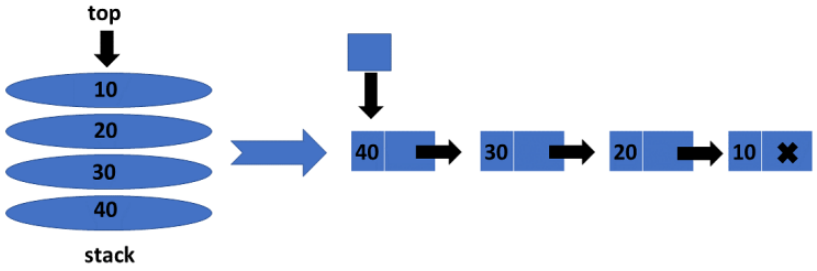


Figure 14 Stack implemented using a linked list

4.16 Some applications of stack

Stack as data structure is used in various application, undo operation in text editors is implemented using a stack. Undo operation consist of popping the last inserted character. On the hand function calls are stored in a stack in RAM as seen previously, during the execution of code. Checking the proper order of parenthesis by the compiler is also performed by a stack. '({[]})', '([)]' are examples of proper and wrong order of parenthesis.

4.17 Queue as an Abstract Data Type

A Queue data structure if viewed as an abstract data type. Is a liner data structure where insertion is performed on one side and deletion is performed on the opposite side. In that case we say that the first inserted element is the first element to be deleted. In other words, we say that Queue is a First In First Out (FIFO) data structure. Insertion operation in a queue is referred as 'enqueue' operation and deletion operation from a queue is denoted as 'dequeue'.

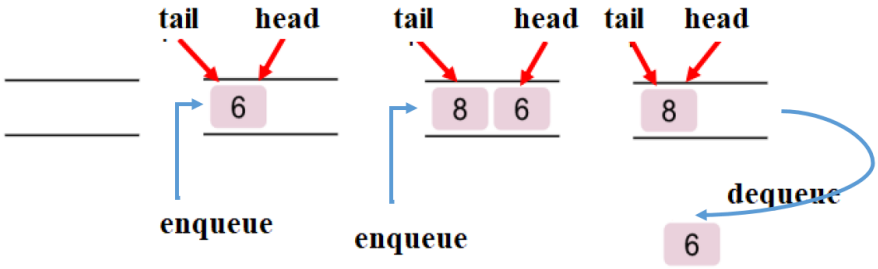


Figure 15 Queue as an abstract data type

4.18 Queue Implementation as an array

Just like the stack, a queue can be implemented as one dimensional-array. The first and last inserted elements are tracked and are referred as head and tail of a queue, respectively. Figure below illustrate this implementation.

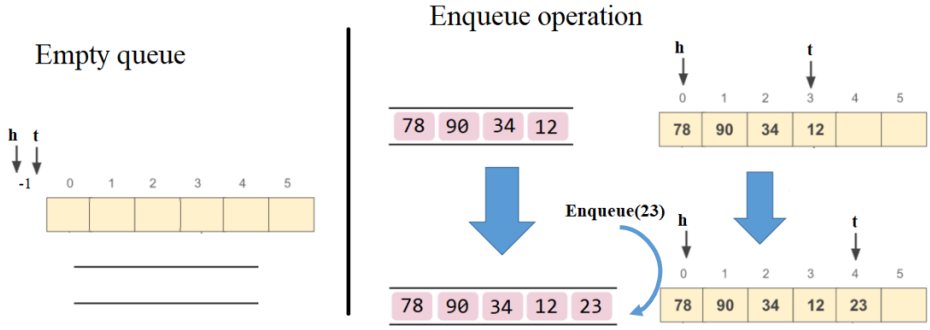


Figure 16 Queue implemented using array

4.19 Implementation IsEmpty() function

Very often, knowing whether the queue is empty or not is required. Therefore, an auxiliary function IsEmpty(QueueName) is defined. A queue is empty if both 'h' and 't' are equal to -1.

```
queue1[0..n-1] : array of integers
n, t , q : integer
t <- q <- -1

fct IsEmpty(x:int, queue[]:array of int): boolean
var
begin
    if t = - 1 and q = -1 then
        return true
    EndIf
    return false
end
```

4.20 Implementation of Enqueue operation

Insertion of an element can be performed by an Enqueue(value, queueName) function. This latter receives the value to be inserted 'v' in a queue 'q'. 'h' and 't' variables are used to track the indices of the head and queue element in the array. Enqueue of first element of queue consists of incrementing both indices 'h' and 't' then placing element 'v' as element index 't' in queue 'q'. Enqueue of any other element requires incrementation of 't' alone then placing the element 'v' as element index 't' in queue 'q'. As the queue is represented as an array, before proceeding to insertion, we have to make sure that the queue is not full (i.e. 't' should not be larger than the last index in the array). Pseudocode of this implementation is shown next.

```
proc enqueue(x:int, queue[]:array of int)
var
begin
    if(isEmpty(queue)) then
        t++ ; h++
        queue[h]
    elseif (h = n-1)
        print('queue overflow')
    else
        h++
        queue[h] <- x
end
```

4.21 Implementation Head() and Tail() functions

Head() and Tail() auxiliary functions return that first and last inserted elements, respectively. It corresponds to the elements whose indices are 'h' and 't' respectively.

```
fct head(queue:int) : int
vars
begin
    if(!isEmpty())
        return queue[h]
end
```

```
fct tail(queue:int) : int
vars
begin
    if(!isEmpty())
        return queue[t]
end
```

4.22 Implementation of Dequeue operation

Dequeue(QueueName) is a function that receives the Queue name as a input to delete the element at the head of the queue and returning it for use in program.

```
fct dequeue(queue[]:array of int) : integer
var temp : integer
begin
    if (isEmpty()) then
        print('queue already empty')
        return
    else if (t = h)
        temp <- h
        t<- h <- -1
        return queue[temp]
    else
        temp <- t
        t++
        return queue[temp]
end
```

4.23 Queue Implementation using SLL

Queue can be implemented using a SLL, where the only restriction is that the insertion (enqueuer operation) is done on one side and the deletion (dequeue operation) is done on the

opposite side of the SLL. In addition, two pointers are assigned to track the first and last elements of the queue. Implementation of Queue using SLL are left as exercises to the student.

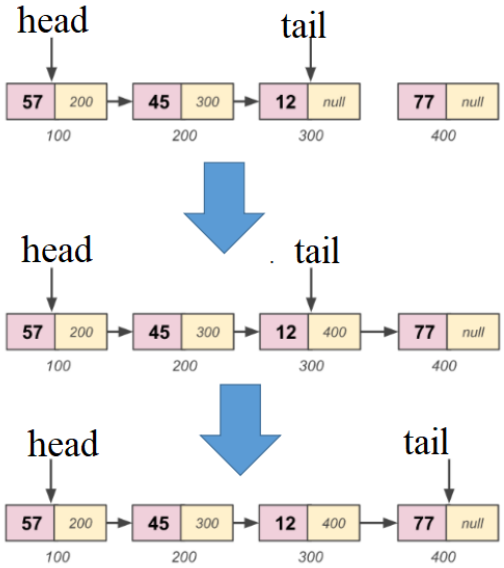


Figure 17 Queue implemented using linked list

4.24 Applications of Queue data structure

Here are some applications of queues :

- CPU scheduling- to keep track of processes for the CPU
- Handling website traffic - by implementing a virtual HTTP request queue
- Printer Spooling - to store print jobs
- In routers - to control how network packets are transmitted or discarded
- Traffic management - traffic signals use queues to manage intersections

4.25 Conclusion

In this chapter, various types of Linked List data structure have been examined. The advantages and drawbacks of each type are discussed. Logical data structures such as Stack and Queue are studied along with the different possible implementations and operations. Armed with this knowledge the student should be well prepared to study more advanced data structures such as trees and graphs.

4.26 Exercises

1. Write an algorithm that display the elements of a linked list in reversed order.
2. Write an algorithm that counts the number of elements in a linked list.
3. Write an algorithm for insertion of a new node at the beginning of a doubly linked list.
4. Write an algorithm for insertion of a new node at the beginning of a circular linked list.
5. Re-write the operations on stack and queue data structures
6. Implement a stack and its related operations using a singly linked list.
7. Implement a queue and its related operations using a singly linked list.

General Conclusion

General conclusion

In conclusion, these lecture notes have introduced the fundamental concepts of data structures and algorithms, beginning with the principles of modular programming through subprograms, progressing to file handling for permanent data storage, and culminating in the study of dynamic data structures such as linked lists, stacks, and queues.

Understanding subprograms provides the foundation for writing well-structured, modular, and maintainable programs. Knowledge of file operations equips students with the ability to manage persistent data efficiently. The study of linked lists, stacks, and queues strengthens the ability to organize and manipulate data dynamically, forming a basis for solving more complex computational problems.

Together, these topics establish essential programming skills and analytical thinking required in computer science. With a clear understanding of these core concepts and their implementation, students are well-prepared to explore more advanced data structures and algorithm design techniques. All algorithms presented in these notes are expressed using the C programming language, reinforcing both theoretical understanding and practical implementation skills.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms. MIT press.

[2] Djelloul BOUCHIHA. Algorithmics and C Programming: Lectures, Solved Exercises, and Practical Work. Publisher: LAP LAMBERT Academic Publishing. Copyright © 2024 Dodo Books Indian Ocean Ltd. and OmniScriptum S.R.L Publishing Group. ISBN: 978-620-7-47639-8. 353 p.

[3] Rémy Malgouyres, Rita Zrour et Fabien Feschet. Initiation à l'algorithmique et à la programmation en C. 2ième Edition. Dunod, Paris, 2011. ISBN : 978-2-10-055703-5.

[4] Nebra, Mathieu. Apprenez à programmer en C. OpenClassrooms, 2015.

[5] Schildt, H. (2018). *C: The Complete Reference* (4th ed.). McGraw-Hill Education.