

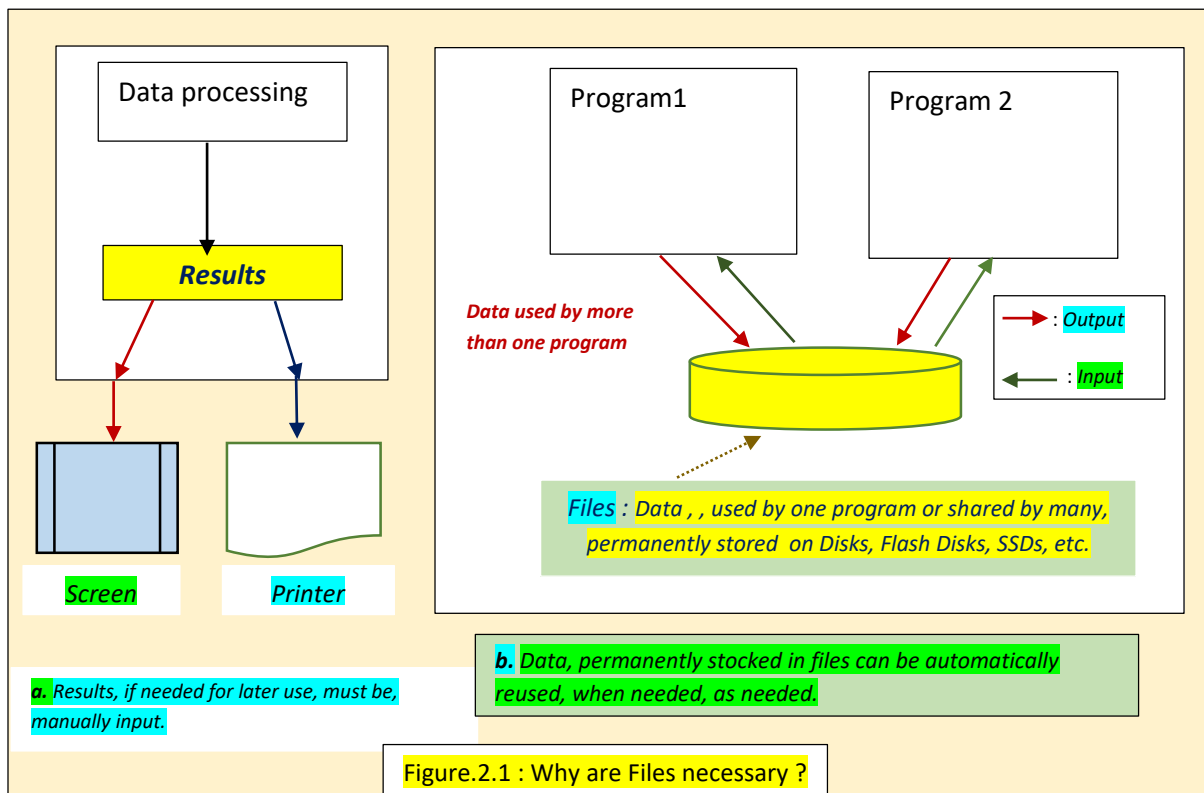
**Content :**

- Introduction
- Definitions
- Types of files
- Files usage

1. **Introduction :**

Until this point, the data used in programs was stored in RAM, only. Recall, however, that data stored in RAM (only) lasts only during the programs execution. The data stored in memory only is then said **volatile** data, in the sense that, these data will simply disappear when the program ends. On the other hand, it is well known for all that, part or all of the data stored in RAM only can be needed for **later use** by the same program (software) or other programs (software). In this case, these data must be re-entered to the programs (software) if needed (Fig.2.1a).

This solution can be adopted for data of small size. However, for data of big size, another solution is used (Fig.2.1b).



Indeed, **files are the common way to store data, permanently**. This solution allows **re-using data** by the same program (software), or, by **other programs** (software). This last case is referred to by **data sharing**.

Briefly, Files can be seen as:

- Particular **Data Structures**, used for **permanent storing of data**.
- Files are created and manipulated on external permanent data storing tools: Disks, Flash disks, **Solid-state drives (SSDs)**, etc.
- Data within **files** is then stored for **later use by programs** (software), **including programs that did not create them, once, or repeatedly**, as needed, when needed.
- Files are two types: **Text Files** and **Binary Files**.

In the following, in this Chapter 2, **files are presented, explained and illustrated**. This step allows us then to consider solving other kinds of problems, hence, opening for us **new horizons in the activity of programming**.

## 2. **Definitions** :

- **File**: A file is a collection of data identified by an **external name** (name for short) including a **path** allowing access to the file on an **external stocking device**: Hard Disks, Flash Disks, SSDs, etc. These storing tools are said to be **mass memory**, to distinguish them from RAM.
- File external name: is an identifier (string) that allows the **File Management System (FMS)**, a part of the Computer operating System (OS), to identify it and manage it. Following are examples of file names:
  - **Photo1.jpg** : File in **image** JPEG standard.
  - **Chapter2.pdf** : File of type .pdf (*Portable Document Format*), a well known and intensively used document format.
  - **Prog2.exe** : A file of type executable program.
  - **Prog2.c** : A file stocking a C language program.

As can be noticed, a file name is composed of two parts, separated by a dot:

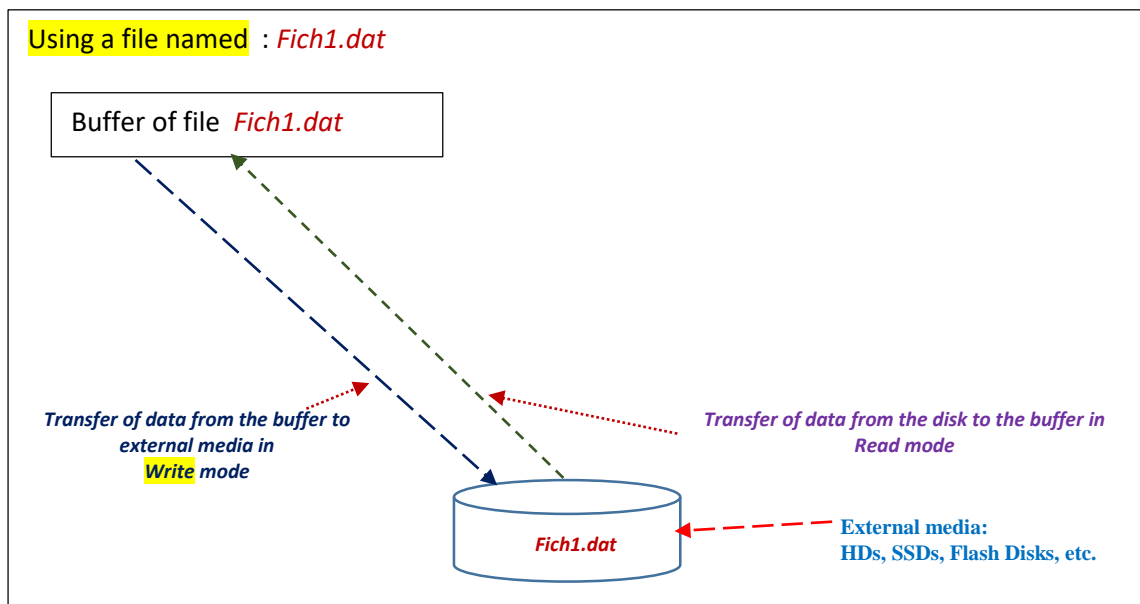
**FileName.Extension**

The file name is generally chosen by the programming person, the extension is determined by the nature of the file content. The extension helps automatically the FMS use the good software to open a file of specific type. For example, a .pdf file can be opened and manipulated by PDF-Reader software. Likewise, an image file can be opened, for example, by PHOTOS software of Windows OS.

- Data within files can be **homogeneous** (same type of data, e.g. text, integer, etc., exclusively), or **heterogeneous** (mixture of more than one type).

- Data within files are generally accessed in a **sequential mode** from the beginning to the end of the files. However, there exists also a **direct access mode** that allows targeting specific positions in files.
- Practically, accessing a file by programs requires a **logical name associated to the physical** (external) name of that file at the opening operation of it.
- The logical name of a file is in fact a pointer to the **file buffer**, a temporary memory zone where files content can be edited. Buffering is used to limit the number of files access on disks, since data transfer from disks (read/write) are generally slow, due to the mechanical nature of hard disks. However, SSDs show much more speed for data transfer since they are based on flash memory, whereas HDs are based on a mechanical arms with a read/write head that moves across spinning disks to access data.

Figure 2.2 illustrates file transfer between disks (HDs, SSDs, Flash Disks, etc.) and programs through buffers.



**Figure. 2.2 Data transfer between the files buffer and external media.**

- Operations on file are as follows :
  - File Opening in **Write mode**
  - File Opening in **Read mode**
  - File **Closing**

All these operations will be presented and illustrated in the C language, in what follows.

### 3. Files in the C Language :

Like other languages, files are of two types in the C language: Text Files and Binary Files.

- Text Files** : are files composed of characters (e.g. "L1-Computer Science-Section2(1.5)") and special characters (e.g: EOL (End Of Line), EOF (End Of File)), in ASCII code.

A text file is usually recognizable by the .TXT extension. However, in diverse OS systems, other extensions are used for specific types of text files, e.g.:

**Configuration files**: Saved with extensions like .cfg, .conf, .ini, or .xml, etc.

**Source code files**: Used for storing programs in various programming languages: e.g. .py for Python, .java for Java, .cpp for C++, etc.

And other type of files.

- Binary files** : Generally speaking, every non text file is a binary file. More specifically, a binary file is a sequence of 0's and 1's that no program can read, interpret and manipulate correctly, except programs knowing the format they have been created with.

**Examples of binary files**: As already said, all non-text files are typically binary files. E.g. Images (.jpg, .png), audio files (.mp3, .wav), video files (.mp4), executable files (.exe, .com), document files (.DOC, .DOCX, .XLS, .XLSX, .PDF), etc.

Generally speaking, computer files share many principles and notations. Notably, they share commonly known operations on files, like READ, WRITE, OPEN and CLOSE. These are presented in Table 2.1, below.

Table 2.1 : Files opening Modes, in the C Language				
File Type	Opening Mode Code	Opening for :	Used File Modified ?	Used File Suppressed ?
Text	'r'	Read	No	No
	'w'	Read/Write	Yes	Yes
	'a'	Write at the File End	Yes	No
	'r+'	Read/Write	Yes	No
	'w+'	Read/Write	Yes	Yes
	'a+'	Write at the File End	Yes	No
Binary	'rb'	Read	No	No
	'wb'	Read/Create	Yes	Yes
	'a'	Write at the File End	Yes	No
	'rb+'	Read/Write	Yes	No
	'wb+'	Read/Write	Yes	Yes
	'ab+'	Write at the File End	Yes	No

These modes are presented, detailed and illustrated, in the following.

i. **Using Text Files in C Language:**

The operations (functions and procedures) generally used on text files in C language are illustrated in Tables 2.2 and 2.3. For this, we consider a text file **Fich1.txt**, to be created, then read, then closed. All these operations require the file opening function.

To be able to use the file **Fich1.txt**, we use a **pointer to a buffer associated with Fich1.txt**, called: **Fich1Ptr**.

We report here that, for writing, as well as for reading, we can access the data in an **unformatted or formatted** way.

In addition, for **binary files**, there is another type of access mode: **Direct access**. Let us develop these elements in the following:

a. **Unformatted Access:** This type of access can be carried out on text files according to the following two modes:

- **Character by character access** (Sequential, Unformatted).
- **String access by string** (Sequential, Unformatted).

b. **Formatted access:** In this case, there is reading/writing of variables of specific types: e.g. **int, float, char[]**.

- **Direct access** can be performed by the **fseek()** function.

**Table 2.2 Text Files Access Modes and Associated Functions**

Access Mode	Fonction utilisée	Rôle
Non-formated	<b>fputc()</b>	Writing one character at a time to the file
Non-formated	<b>fgetc()</b>	Reading one character at a time from the file
Formated	<b>fgets()</b>	Reading one string at a time from the file
Formated	<b>fputs()</b>	Writing one string at a time to the file
Formated	<b>fscanf()</b>	Reading a typed variable: (int, float, string or struct) to the file
Formated	<b>fprintf()</b>	Writing a simple typed variable (int, float, string) to the file
//	<b>fseek()</b>	Move the read/write head on the file to a desired position.

Tables 2.3. Functions associated with text files

Function	Usage on Fich1.txt	Explanations	Access Type	Illustrations
<code>fopen()</code>	<code>FILE* Fich1Ptr = fopen("Fich1.txt", "w");</code>	Opening the file for writing/creation	//	
<code>fprintf()</code>	<code>fprintf(Fich1Ptr, "%d", num);</code>	Writing the integer num in Fich1.txt	Formatted ( <code>int</code> )	
<code>fscanf()</code>	<code>fscanf(Fich1Ptr, "%d", &amp;num);</code>	Reading the integer num from Fich1.txt	Formatted ( <code>int</code> )	
<code>fclose()</code>	<code>fclose(Fich1Ptr);</code>	Closing the file Fich1.txt	//	
<code>feof()</code>	<code>feof(Fich1Ptr)</code>	File end reached ?	//	
<code>fseek()</code>	<code>fseek(Fich1Ptr, -15, SEEK_END);</code>	Move back 15 characters from the end of the file	//	

ii. Use of binary files:

As already mentioned, binary files are necessarily defined on the basis of typed data: `int`, `float`, `char`, `char[]` or `struct`. Therefore, two types of access are possible on these data, using the formats of their definitions: **Formatted-sequential access** or **formatted-direct access**.

a. **Formatted sequential access:** In this case, the file is browsed sequentially, from beginning to end, respecting the format of each data entered or to be entered in the file (`int`, `float`, `char`, `char[]`, `struct`).

**Note** here, in the case of `int` and `float`, the existence of several variants (**subtypes**):

`int`: `short int`, `int`, `long int`, etc.

`float`: `float`, `double`, `double double`, etc.

Clearly, respecting the formatting is achievable by using the appropriate **subtype** for correct encoding or decoding of the data in a binary file.

b. **Formatted Direct access:** In this access, the binary file is seen as a 1-dimensional array where an index leads directly to the desired element. In the binary file, there is use of movement according to three modes: **Start of file**, **Current position** or **from the End of file**, to directly access the desired data.

This access is performed by the C language function: `seek()`. Its complete syntax is as follows:

`int fseek(FILE *pointer, long int offset, int position);`

In this code :

`*pointer` : is the pointer to the binary file..

`long int offset` : is the displacement (in `long int`)

`int position` : is the displacement mode:

`0` or `SEEK_SET` : Relative to the start of the file

`1` or `SEEK_CUR` : Relative to the current position

`2` or `SEEK_END` : Relative to the end of the file.

Below, in Tables 2.4 and 2.5, are the operations performed on binary files:

Table 2.4 Modes d'Accès aux fichiers Binaires et fonctions associées		
Access Mode	Used Function	Role
Formatted	<code>fwrite()</code>	Writing a formatted data block (Buffer).
Formatted	<code>fread()</code>	Reading a formatted data block (Buffer).
//	<code>fseek()</code>	Move the Read/Write head on the file to a desired position.

Tables 2.5 Functions associated with Binary files				
Function	Usage on Fich2.bin	Explanations	Access Type	Illustrations
<code>fopen()</code>	<code>FILE* Fich2Ptr = fopen("Fich2.bin", "w");</code>	Opening the file for writing/creation	//	
<code>fwrite()</code>	<code>fwrite(&amp;j, sizeof(int), 1, Fich2Ptr);</code>	Writing in the file Fich2.bin of a 1 int j from its address &j	Formatted	
<code>fwrite()</code>	<code>fwrite(buf, sizeof(buf), 1, Fich2Ptr);</code>	Writing the whole contents of the buffer buf to the Fich2.bin file.	Formatted	
<code>fread()</code>	<code>fread(&amp;n, sizeof(int), 1, Fich2Ptr)</code>	Reading 1 variable n of type int from the current position in the file Fich2.bin.	Formatted	

<code>fread()</code>	<code>fread(buf, sizeof(buf), 1, Fich2Ptr);</code>	Reading 1 block of data into buffer buf from the current position in file Fich2.bin.	Formatted	
<code>fclose()</code>	<code>fclose(Fich2Ptr);</code>	Fich2.bin closing	//	
<code>fseek()</code>	<code>fseek(Fich2Ptr, -15, SEEK_END);</code>	Move back 15 bytes from the end of the file Fich2.bin	//	

#### 4. Illustrations on text files :

In the following, we illustrate the different operations for each type of file. Here are illustrations on text files:

##### a. Operation of creating a text file:

In this illustration, we want to

- create a text file named 'text.txt';
- close it immediately after its creation (the file is then empty);

Here is a program in C language, performing this task:

```
// Prog1:
#include <stdio.h>
#define FILE_NAME "text.txt"
int main(){
    FILE *file_ptr = fopen(FILE_NAME, "w"); // Creation of the file (mode 'w')
    fclose(file_ptr);
    return 0;
}
```

##### Notes:

- If the file 'text.txt' does not exist in the current directory, it will be created.
- If the file 'text.txt' already exists in the directory, it will be overwritten (emptied of its content).
- The file will be used in the different programs, through its pointer (here: file\_ptr).
- Once the file is closed by fclose(file\_ptr), it can no longer be used by any other operation, before it is opened again by the operation: fopen().



**b. Writing text to a file:**

In this illustration, we want to:

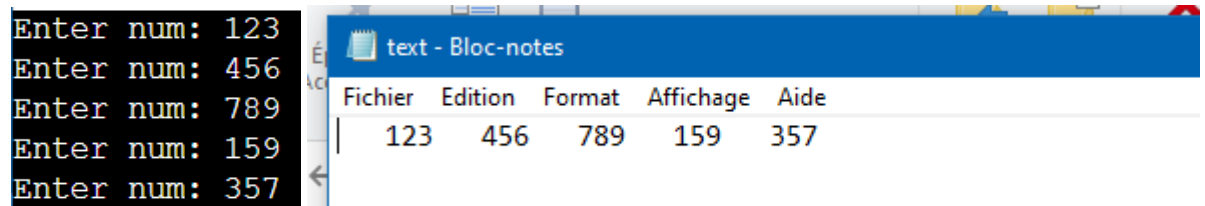
- Use the file 'text.txt' to store 5 integers
- Display each stored number (for later verification)

Here is a program that does this task:

**Prog2:**

```
#include <stdio.h>
#include <stdlib.h> // Contains the exit() procedure
int main(){
    int num;
    int i;
    FILE *fptr;
    //
    fptr = fopen("text.txt", "w"); // Opening file text.txt in writing mode
    if(fptr == NULL){
        printf("Error!");
        exit(1); // Something went wrong using file text.txt
    }
    for (i=1; i<=5; i++){
        printf("Enter num: ");
        scanf("%d", &num);
        fprintf(fptr, "%6d", num); // Writing entered numbers in file text.txt
    }
    fclose(fptr); // closing file text.txt
    return 0;
}
```

Execution:



The screenshot shows the program's execution output on the left and the contents of 'text.txt' on the right. The output displays five prompts 'Enter num:' followed by the numbers 123, 456, 789, 159, and 357. The file content shows the same five numbers separated by spaces.

Fichier	Edition	Format	Affichage	Aide
123	456	789	159	357

**Notes:**

- File 'text.txt' is opened in writing mode : `fptr = fopen("text.txt", "w");`

There is **association** between the file **external name**: **text.txt** and its **internal** name: **fptr**.

- The successive writing of the 5 numbers entered (from the keyboard) is done by the operation:

**fprintf(fptr,"%6d",num);** , in **text mode**, **sequentially** and from the **beginning of the file**.

Thus, any previously existing data are lost.

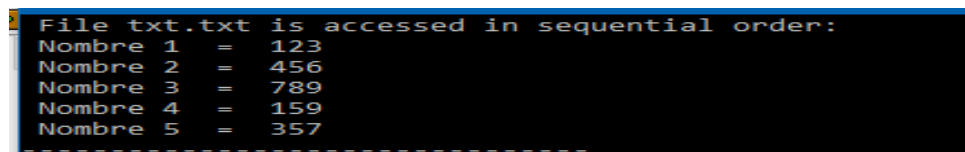
c. Reading from a text file, **element by element**:

In this illustration, the task is to read the elements written in a text file, element by element. We always illustrate on the already created text file: 'text.txt'. Therefore, practically, here, we will read the numbers previously written in this file: {123, 456, 789, 159, 357 }.

**Prog3 :** **Reading the elements entered in the file 'text.txt', element by element**

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int num,i;
    FILE *fptr;
    if ((fptr = fopen("text.txt","r")) == NULL){
        printf("Error opening the file");
        // Execution terminated if pointer fptr == NULL.
        exit(1);    // Program terminated
    }
    printf(" File txt.txt is accessed in sequential order:");
    i=0;           // Counter of numbers in file "text.txt"
    while (!feof(fptr)){    // feof : end of file
        fscanf(fptr,"%d", &num);    // Reading one number at a time
        printf("\n Nombre %d = %d",++i, num); // writing number num on the screen
    }
    fclose(fptr);    // file text.txt is closed here
    return 0;
}
```

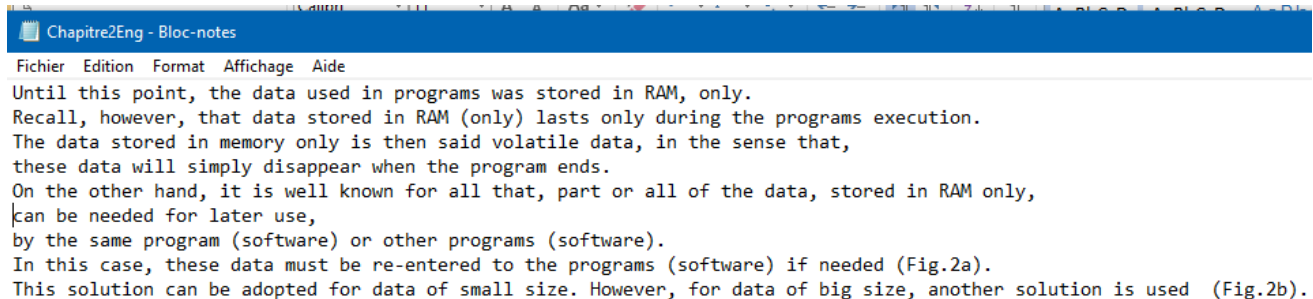
Execution:



```
File txt.txt is accessed in sequential order:
Nombre 1 = 123
Nombre 2 = 456
Nombre 3 = 789
Nombre 4 = 159
Nombre 5 = 357
```

**Prog.4 : Example : Writing a text file, line by line**

First, let's create a file named "Chapter2Eng.txt", using 'Bloc Notes' program of 'Windows'.



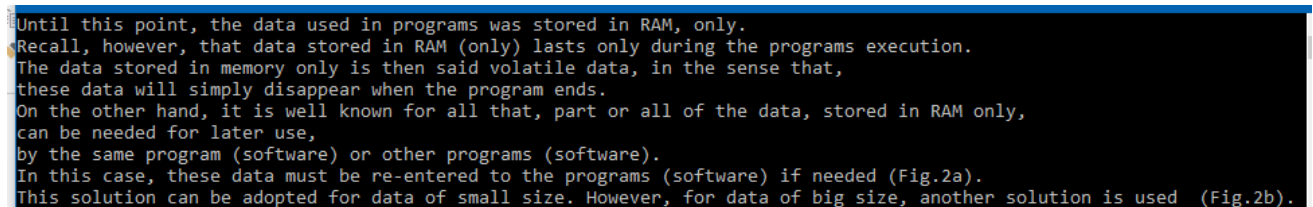
This file will now be used to illustrate how to write a text file LINE by LINE. Here is a C program that does the needed work:

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    // Use fp pointer "Chapitre2Eng.txt" in read mode.
    FILE* fp = fopen("Chapitre2Eng.txt", "r");

    char line[256];    // Buffer to store each line of the file.

    if (fp != NULL) { // file opened properly?
        while (fgets(line, sizeof(line), fp)) { // Read the file Line by Line, in Buffer line
            printf("%s", line);                // the read lines are then printed on the screen
        }
        fclose(fp);    // when the file end is reached, close it.
    }
    else {              // an error occurred when trying to open the file:
        printf("Unable to open file!\n");    // Print an error message
    }
    return 0;
}
```

**Execution :**

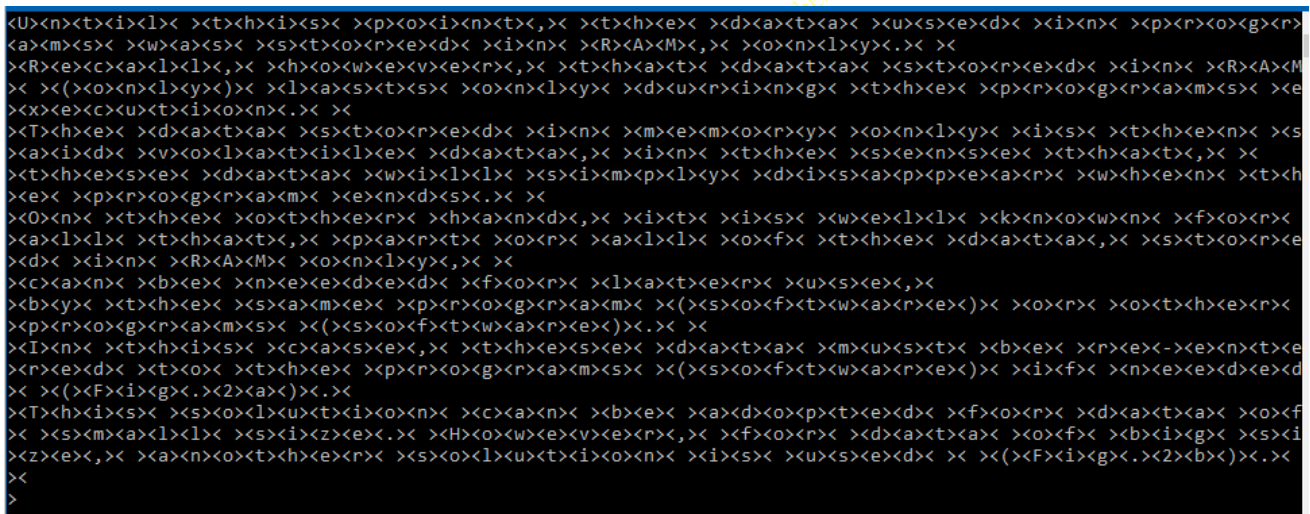


**Prog 5 :**

**Example:** Reading a text file, character by character using procedure `fgetc()`. Read content is displayed on screen. Each character read is enclosed in `<` and `>`.

```
#include <stdio.h>
#include <stdlib.h>
int main()
// fp : pointer to file "Chapitre2Eng.txt"
FILE* fp = fopen("Chapitre2.txt", "r"); // opening in read mode
if (fp != NULL) { // File, properly opened?
    char c = fgetc(fp); // reading 1 character at a time from the file using fgetc()
    while (c != EOF) { // End of the File "Chapter2Eng.txt" reached ?
        printf("<%c>", c); // read character is output to the screen
        c = fgetc(fp); // get another character, if possible.
    }
    fclose(fp); // File closing
}
else
    printf("Error opening file Chapter2Eng.txt \n");
}
```

**Execution :**



**Prog. 6 : Example :** Reading a text file, string by string using : fgets()

// In this illustration, file "Chapitre2Eng.txt" is read, 1 string, of length 30 characters, at a time.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h> // string library
int main() {
    int i=0; // strings counter
    FILE *fileptr;
    char str[30];
    fileptr = fopen("Chapitre2Eng.txt", "r"); // file opened in read mode

    if (NULL == fileptr) {
        printf("Error opening file \n");
        exit(1); // program terminated
    }
    printf(" File content – 30 characters at a time : \n");
    while (fgets(str, 30, fileptr) != NULL) {
        printf(" string %d : <%30s> \n", ++i, str);
    }
    fclose(fileptr);
    return 0;
}
```

**Execution :**

```
File content : 30 characters at a time :
string 1 : < Until this point, the data us>
string 2 : < ed in programs was stored in >
string 3 : < RAM, only.
>
string 4 : < Recall, however, that data st>
string 5 : < ored in RAM (only) lasts only>
string 6 : < during the programs executio>
string 7 : < n.
>
string 8 : < The data stored in memory only>
string 9 : < y is then said volatile data,>
string 10 : < in the sense that,
>
string 11 : < these data will simply disapp>
string 12 : < ear when the program ends.
>
string 13 : < On the other hand, it is well>
string 14 : < known for all that, part or >
string 15 : < all of the data, stored in RA>
string 16 : < M only,
>
string 17 : < can be needed for later use,
>
string 18 : < by the same program (software>
string 19 : < ) or other programs (software>
string 20 : < ).
>
string 21 : < In this case, these data must>
string 22 : < be re-entered to the program>
string 23 : < s (software) if needed (Fig.2>
string 24 : < a).
>
string 25 : < This solution can be adopted >
string 26 : < for data of small size. Howev>
string 27 : < er, for data of big size, ano>
string 28 : < ther solution is used (Fig.2>
string 29 : < b).
>
string 30 : <
>
```

### 5. Writing to and Reading from Binary Files :

We first recall that **writing to a binary file** is done by the operation: **fwrite()** and **reading** from this kind of file is done by the operation: **fread()**.

Note that these two functions are part of the standard input/output library: **<stdio.h>**

We also recall the syntaxes of these operations, in C language:

Generic syntax of **fwrite()**, in C language:

```
fwrite(addressofData, sizeData, numberOfData, pointerToFile);
```

where :

**addressofData**: is the **RAM address** of the data to be written to the file

**sizeData**: is the **size** of data.

**numberOfData**: is the **number of data** to be written in the file

**pointerToFile** : is the **pointer** (RAM) to the file on **external media**

The **fwrite()** function then **transfers** **numbersData** of data, each data being of size **sizeData**, from the memory address (RAM) **addressData**, to the file pointed to by **pointerToFile**.

The **size** of the **transferred data** is then: **numbersData X sizeData**, in bytes.

- **Generic syntax of the fread() operation, in C language:**

```
fread(addressofData, sizeData, numberOfData,
```

**addressofData**: is the **RAM address** of the data (Data) where to **read** the data (Data) **from the file**.

**sizeData**: is the **size** of the data to be **written to the file**.

**numbersData**: is the **number** of data to be **written in the file**.

**pointerToFile**: is the **pointer** (RAM) to the file on **external media**.

The **fread()** function then proceeds to the transfer of **numbersData** of data, each data being of size **sizeData**, from the external file pointed to by **pointerToFile**, to the memory area (RAM) at address: **addressData**.

The **size** of the transferred data is then: **numbersData X sizeData**, in bytes.

Here are, then, illustrations of the `fwrite()` and `fread()` procedures.

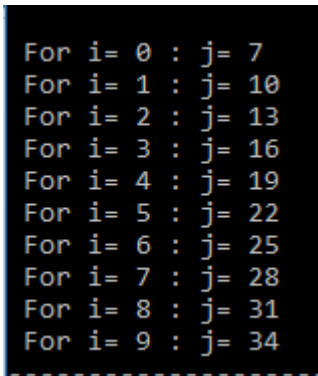
a. **Creating / writing to a binary file:**

In this illustration, we first create a binary file ('test.dat'), then, we write 10 integers (j), generated by a formula ( $j = 3*i + 7$ ,  $i = 1 \dots 10$ ), to avoid manual entry.

**Prog.7: Example 1: Writing 10 integers into a binary file, one at a time, sequentially:**

```
#include <stdio.h>
int main(){
    int i,j;
    FILE *fp;
    fp = fopen("test.dat", "wb"); // Creation of binary file "test.dat"
    for(i=0;i<10;i++){
        j=3*i+7; // Generating numbers j
        printf("\n For i= %d : j= %d ",i,j);
        fwrite(&j, sizeof(int), 1, fp); // writing 1 integer number j in binary file "test.dat"
    }
    fclose(fp);
    return 0;
}
```

**Execution:**



```
For i= 0 : j= 7
For i= 1 : j= 10
For i= 2 : j= 13
For i= 3 : j= 16
For i= 4 : j= 19
For i= 5 : j= 22
For i= 6 : j= 25
For i= 7 : j= 28
For i= 8 : j= 31
For i= 9 : j= 34
```

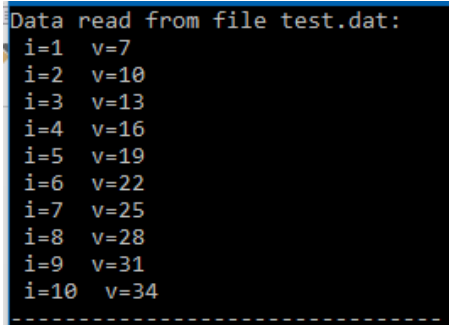
b. *Reading data sequentially from a binary file:*

In this illustration, we proceed to **reading** the 10 numbers previously entered in file 'test.dat', **sequentially**, starting from the **beginning of the file**.

**Prog.8 : Example 2:** Reading the 10 numbers entered in the 'test.dat' file, one by one, sequentially:

```
#include <stdio.h>
int main(){
    int i,v;
    FILE *fp;
    fp = fopen("test.dat", "rb"); // opening binary file test.dat in read mode
    i=0;
    printf("Data read from file test.dat:");
    while(fread(&v, sizeof(int), 1, fp)==1){ // 1 int read at a time from test.dat
        printf("\n i=%d v=%d", ++i,v);
    }
    fclose(fp);
    return 0;
}
```

**Exécution:**



```
Data read from file test.dat:
i=1 v=7
i=2 v=10
i=3 v=13
i=4 v=16
i=5 v=19
i=6 v=22
i=7 v=25
i=8 v=28
i=9 v=31
i=10 v=34
-----
```

c. *Reading data in bulk, from a binary file:*

In this illustration, we read the 10 numbers previously written in the file 'test.dat', in bulk, from the beginning of the file.

**Prog.9 : Example 3:** Reading in bulk the 10 numbers entered in the 'test.dat' file:

```
#include <stdio.h>
int main(){
    int i,j;
    int buf[10];
    FILE *fp;
    fp = fopen("test.dat", "rb"); // opening test.dat in read mode
    fread(buf, sizeof(buf), 1, fp); // transfert of 10 int numbers, in bulk
    printf("Data read in bulk from file test.dat:");
    for(i=0;i<10;i++)
        printf("\n i= %d buf[i]=%d",i,buf[i]);
    fclose(fp);
    return 0;
}
```



**Execution :**

```
Data read in bulk from file test.dat:
i= 0   buf[i]=7
i= 1   buf[i]=10
i= 2   buf[i]=13
i= 3   buf[i]=16
i= 4   buf[i]=19
i= 5   buf[i]=22
i= 6   buf[i]=25
i= 7   buf[i]=28
i= 8   buf[i]=31
i= 9   buf[i]=34
-----
```

d. Binary file defined by the struct structure (heterogeneous content):

**Prog.10 :**

**Example 4:** Writing Struct type records containing a single data (here, integer)

```
#include <stdio.h>
#include <stdlib.h>
struct entier {
    int val;
};
int main(){
    int n,i;
    struct entier e;
    FILE *fptr;
    if ((fptr = fopen("Data.dat","wb")) == NULL){
        printf("Erreur opening file Data.dat");
        // Arrêt du programme pour la valeur : NULL.
        exit(1);
    }
    printf(" Number of data in file : n = ");
    scanf("%d",&n);
    for(i = 1; i <= n; i++){
        e.val = 3*i + 1; // 3*i + 1 : Numbers generating function
        printf("\n i= %d  value =%d", i,e.val);
        fwrite(&e, sizeof(struct entier), 1, fptr);
    }
    fclose(fptr);
    return 0;
}
```

**Execution:**

```
Number of data in file : n = 12
i= 1  value =4
i= 2  value =7
i= 3  value =10
i= 4  value =13
i= 5  value =16
i= 6  value =19
i= 7  value =22
i= 8  value =25
i= 9  value =28
i= 10  value =31
i= 11  value =34
i= 12  value =37
-----
```

- e. Using structure **struct** for 3 int numbers (homogeneous data) :  
**Prog.11 :**

**Example 5: Writing numbers to a binary file: using 3-int number structs:**

```
#include <stdio.h>
#include <stdlib.h>
struct entier {
    int v1;
    int v2;
    int v3;
};
int main(){
    int n,i;
    struct entier e;
    FILE *fptr;
    if ((fptr = fopen("Data.dat","wb")) == NULL){
        printf("Error opening file Data.dat");
        exit(1);
    }
    printf(" Number of structs to write in file Data.dat: n = ");
    scanf("%d",&n);
    for(i = 1; i <= n; i++){
        e.v1 = 3*i + 1;
        e.v2 = 5*i + 1;
        e.v3 = 7*i + 1;
        printf("\n i= %d  e.v1=%d e.v2=%d e.v3=%d", i,e.v1,e.v2,e.v3);
        fwrite(&e, sizeof(struct entier), 1, fptr);
    }
    fclose(fptr);
    return 0;
}
```

**Execution :**

```

Number of structs to write in file Data.dat: n = 9

i= 1  e.v1=4  e.v2=6  e.v3=8
i= 2  e.v1=7  e.v2=11 e.v3=15
i= 3  e.v1=10 e.v2=16 e.v3=22
i= 4  e.v1=13 e.v2=21 e.v3=29
i= 5  e.v1=16 e.v2=26 e.v3=36
i= 6  e.v1=19 e.v2=31 e.v3=43
i= 7  e.v1=22 e.v2=36 e.v3=50
i= 8  e.v1=25 e.v2=41 e.v3=57
i= 9  e.v1=28 e.v2=46 e.v3=64
-----
    
```

f. **Reading the 3 numbers defined by struct**

In this illustration, we read the 9 struct records each containing 3 int numbers:

**Prog.12 :**

**Example 6: Writing numbers to a binary file: using 3- int numbers structs**

```

#include <stdio.h>
#include <stdlib.h>
typedef struct Integer3 { Type Integer3 definition
    int v1;
    int v2;
    int v3;
}TInt3; // Alias of Type Integer3

int main(){
    int i=0; // structs of type TInt3 counter
    TInt3 e; // Buffer for reading structs of type TInt3 from Binary file Data.dat
    FILE *fptr;
    if ((fptr = fopen("Data.dat", "rb")) == NULL){ // Binary File opened in Read Mode
        printf("Error opening file Data.dat");
        exit(1);
    }
    printf(" File Data.dat Content : ");
    while(fread(&e, sizeof(TInt3), 1, fptr) == 1){ // Reading TInt3 structs, one at a time
        i++;
        printf("\n Struct: i= %d  e.v1=%d  e.v2=%d  e.v3=%d", i, e.v1, e.v2, e.v3);
    }
    printf("\n Number of extracted TInt3 structs : %d", i);
    fclose(fptr);
    return 0;
}
    
```

Exécution :

```
File Data.dat Content :
Struct: i= 1   e.v1=4   e.v2=6   e.v3=8
Struct: i= 2   e.v1=7   e.v2=11  e.v3=15
Struct: i= 3   e.v1=10  e.v2=16  e.v3=22
Struct: i= 4   e.v1=13  e.v2=21  e.v3=29
Struct: i= 5   e.v1=16  e.v2=26  e.v3=36
Struct: i= 6   e.v1=19  e.v2=31  e.v3=43
Struct: i= 7   e.v1=22  e.v2=36  e.v3=50
Struct: i= 8   e.v1=25  e.v2=41  e.v3=57
Struct: i= 9   e.v1=28  e.v2=46  e.v3=64
Number of extracted TInt3 structs : 9
-----
```

g. Writing to a binary file defined by heterogeneous data struct

Example 7: Writing to a binary file using structs of 3 heterogeneous data: Car(Make: String, Weight(Ton): Float, Year: integer)

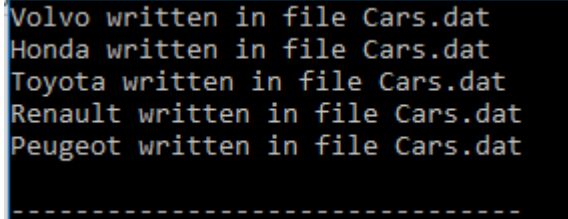
Prog 13 :

```
#include <stdio.h>
#include <stdlib.h>
typedef struct car {
    char mq[10];
    float po;
    int an;
}Tcar; // Type Tcar
int main(){
    int n,i;
    //struct voiture e;
    FILE *fptr;
    if ((fptr = fopen("Cars.dat","wb")) == NULL){ // File Cars.dat opening, creation mode
        printf("Error opening file");
        exit(1);
    }

    Tcar v1={"Volvo",2.1,2015};
    Tcar v2={"Honda",3.1,1999};
    Tcar v3={"Toyoyta",1.5,2019};
    Tcar v4={"Renault",2.2,2004};
    Tcar v5={"Peugeot",0.5,2017};
    fwrite(&v1, sizeof(Tcar), 1, fptr);
    printf("Volvo written in file Cars.dat \n");
    fwrite(&v2, sizeof(Tcar), 1, fptr);
    printf("Honda written in file Cars.dat \n");
```

```
    fwrite(&v3, sizeof(Tcar), 1, fptr);  
    printf("Toyota written in file Cars.dat \n");  
    fwrite(&v5, sizeof(Tcar), 1, fptr);  
    printf("Renault written in file Cars.dat \n");  
    fwrite(&v3, sizeof(Tcar), 1, fptr);  
    printf("Peugeot written in file Cars.dat \n");  
    fclose(fptr);  
    return 0;  
}
```

**Execution :**



```
Volvo written in file Cars.dat  
Honda written in file Cars.dat  
Toyota written in file Cars.dat  
Renault written in file Cars.dat  
Peugeot written in file Cars.dat  
-----
```

h. Reading data from a binary file defined by heterogeneous data in a struct Tcar:

**Prog. 14 :**

```
#include <stdio.h>  
#include <stdlib.h>  
typedef struct car {  
    char mq[10];  
    float po;  
    int an;  
}Tcar;  
int main(){  
    int n=2,i;  
    char m[10];  
    float p;  
    int a;  
    Tcar e;  
    FILE *fptr;  
    if ((fptr = fopen("Cars.dat", "rb")) == NULL){  
        printf("Error opening file Cars.dat");  
        exit(1);  
    }  
    printf(" Data read from binary file Cars.dat : \n ");  
    printf("\n Car : Make - Weight(Ton) - Year ");  
    while(fread(&e, sizeof(Tcar), 1, fptr)==1)  
        printf("\n %10s    %4.1f    %8d", e.mq, e.po, e.an);  
    fclose(fptr);  
    return 0;  
}
```

**Exécution :**

```
Data read from binary file Cars.dat :  
  
Car : Make - Weight(Ton) - Year  
      Volvo      2.1      2015  
      Honda      3.1      1999  
      Toyoyta     1.5      2019  
      Peugeot    0.5      2017  
      Toyoyta     1.5      2019  
-----
```

6. Illustration of the direct access function `fseek()`:

Example: `fseek()`: <https://www.geeksforgeeks.org/fseek-in-c-with-example/>

a. Syntax of `fseek()`

`int fseek(PointerToFile, long int offset, int offsetmode);`

où:

`PointerToFile`: is the pointer to the external file.

`offset`: Number of bytes to 'skip' from a position chosen by `offsetmode`.

When this number is positive: move forward.

When negative, move backwards.

`offsetmode`: There are 3 cases:

0 or `SEEK_SET`: with respect to the beginning of the file

1 ou `SEEK_CUR`: with respect to the current position

2 ou `SEEK_END`: with respect to the file end.

Returned values:

0: Operation was successful

1: Operation failed.

*Example: Illustration of fseek() function:*

The below C Program demonstrates one case use of fseek() function.

```
// C Program to demonstrate the use of fseek()
#include <stdio.h>
int main()
{
    FILE* fp;
    char str[30];
    fp = fopen("Chapitre2Eng.txt", "r");
    // Moving pointer to end
    fseek(fp, 18, SEEK_SET); // Move forward from file 'Chapter2Eng.txt'
    fgets(str, 30, fp);
    // Printing position of pointer
    printf(" Read string: <%s>", str);
    return 0;
}
Execution:
```

```
Read string: <the data used in programs was>
-----
```