Chapter 2. Files

1) Introduction

Computer programs, like messengers of the digital universe, are constantly engaged in dialogues with the outside world. They exchange data to receive instructions, information, and emit results to communicate their achievements. Until now, our sole method of interaction with these programs has been rudimentary: keyboard input to provide data and screen output to present results. However, these fleeting exchanges conceal a fundamental reality.

The data manipulated by a program, the results generated, reside temporarily in the computer's main memory, also known as RAM (Random Access Memory). This memory, although extremely fast and essential for the proper functioning of running programs, is volatile. This means that its content is ephemeral and is erased at the end of the program's execution. Like a slate wiped clean at the end of the day, RAM forgets everything once the computer is turned off.

The crucial question that arises is: how can we preserve the fruits of our work, the data generated, and the results obtained, beyond the fleeting illumination of main memory? How can we, for example, save students' names and grades, or record the highest scores achieved in a game? How can we create a text editor if all the written text disappears when the program is stopped? This is where files come into play as protagonists in the quest for permanence in the world of programming.

A file, in this context, is not simply a mass of zeros and ones. It represents a time capsule that can capture and preserve a program's digital data, offering long-term memory. Imagine it as a digital library where each book, each file, bears a specific name and contains a story to tell. These books are stored on permanent storage media such as hard drives, USB drives, or other persistent storage devices.

Thus, a file becomes the preferred means to transcend the transience of main memory, to establish a connection with temporal continuity, and to preserve data beyond the ephemeral lifespan of a program. It offers us a way to store, organize, and retrieve these valuable data, enabling programs to remember, learn, and progress over time.

By exploring the intricacies of files in algorithmics and the C language, we will unveil the underlying mechanisms that allow these digital time capsules to play a vital role in the computing landscape. We will learn how to read and write in these digital "books," how to organize them effectively, and how to harness their power to create robust and enduring applications. Thus, through this chapter, we embark on a journey that transcends the fleeting limits of main memory, embracing the persistence offered by files.

2) Definitions

As we have seen, the lifespan of variables and information manipulated thus far has been equal to the runtime of the program. They are destroyed at the program's conclusion and therefore cannot be reused thereafter.

In most enterprise applications, the large volume of manipulated information needs to be stored longer than the program's lifespan. Files step in to fill this void. They serve to permanently store data between two program executions.

But what exactly is a file? That is what we will explore in this section.

2.1) File

A file is defined as an organized collection of numerical data grouped under a single name and carefully stored on a permanent storage medium such as a hard disk, USB drive, or optical disk, at a specific location. This versatile storage unit can host various types of information, ranging from text to images, programs, or other forms of data.

Each file is distinguished by a unique name, often accompanied by an extension indicating the type of data it contains. Importantly, unlike volatile memory, a file persists beyond the computer's shutdown, thereby preserving its content for future use.

Conceptually, a file can be likened to a table of records stored on a disk. Each record encompasses a structured collection of logical units of information, also referred to as fields or columns, providing an organized method for storing and organizing data (see Figure 2.1). It is important to note that when creating a file, its size is not necessarily specified, allowing it to dynamically adapt to the changing needs of the data it contains.

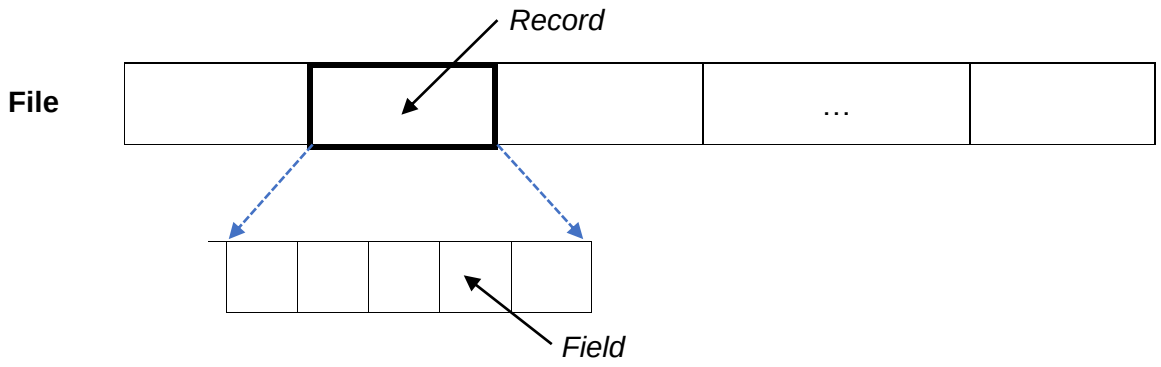


Figure 2.1. Structure of a file

2.2) Elements Attached to a File

Files are characterized by several notions, including:

2.2.1) File Name

The file name is the element that allows its identification. We distinguish between the internal (logical) name and the external (physical) name.

The internal or logical name of a file is the one under which it is identified within a program.

The external or physical name of a file, on the other hand, is the one under which the file is identified in secondary memory. This name consists of three parts:

- The identifier of the storage medium,
- The name of the file itself,
- An extension (see the following section)

For example, `"C:/Chapter 2.docx"` refers to a data file stored on partition C of the hard disk and named "Chapter 2".

2.2.2) File Extension

The file extension is a sequence of characters that follows the file name, separated by a period, as illustrated in Figure 2.2. Typically composed of a few letters or a word, it is of crucial importance as it provides information about the type of file. This extension provides details about the file format and often indicates the program capable of reading or manipulating it.

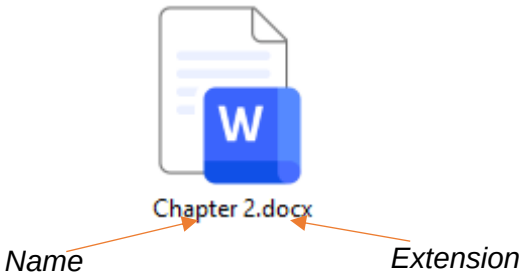
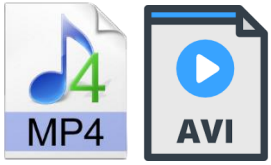


Figure 2.2. Example of File Name and Extension

The file extension is a crucial element for the operating system and software as it quickly identifies the type of content of the file, thereby facilitating appropriate file processing based on its format.

The following table presents some of the most common extensions, along with the corresponding file type and their associated icon:

Extension	File Type	Icon
.exe	Executable File	
.jpg,.jpeg,.png,.bmp,.gif	Image Files	
.txt, .doc, .docx	Text Files	
.mp3	Audio Files	
.mp4,.avi	Video Files	

.c, .pas, .java	Source Code Files	
.rar	Archive File	
.html	Web File	

Table 2.1. Some Examples of Files and Their Extensions

2.2.3) Access Modes

The access mode determines how the machine can retrieve information from a file. Generally, there are two main access modes: sequential access and direct access, with an additional mode known as indexed access.

a) Sequential Access

Sequential access involves processing information sequentially, meaning in the order in which it appears (or will appear) in the file. This implies that to reach the n^{th} piece of information, it is necessary to traverse the (n - 1) preceding pieces of information.

b) Direct Access

Direct access allows immediate access to the desired information without traversing the preceding information. You only need to know the position or number *n* of the desired item to access it. This position represents the location of the record relative to the beginning of the file.

This method is generally faster and more efficient.

c) Indexed Access

Indexed access combines the speed of direct access with the simplicity of sequential access (although it remains somewhat more complex). It is particularly suitable for processing large files.

In this mode, each record in the file is accessible via an access key, a unique field identifying a record. The key is associated with a record number, allowing rapid access to the corresponding record.

Remark :

Any file can be used with any of the three access modes. The choice of access mode does not concern the file itself but only how it will be processed by the machine. Thus, it is in the program, and only in the program, that the desired access type is chosen.

3) File Types

When discussing the concept of files, it is crucial to understand that there is not just one type of file. The significant criterion that distinguishes files is how the information is organized within them.

In algorithmics, two main types of files are typically distinguished based on how the file is organized: ***text files*** and ***binary files***.

3.1) Text Files

3.1.1) What is a Text File ?

A text file is a sequential file containing only characters (alphabetic, numeric, or control characters) encoded in ASCII and organized into lines. If each line contains the same type of information, the lines are referred to as ***records***, making the file structured.

The presentation in the form of « line » is achieved through the presence of an end-of-line indicator: either the carriage return, whose ASCII code is 13, or the line feed, whose ASCII code is 10.

Text files are readable by a simple text editor like Windows Notepad. This type of file is commonly used when storing information that can be likened to a database (structured data).

Let's take a classic example, that of an address book. The file is intended to store the contact information of a number of people. For each person, you would note the last name, first name, phone number, and email. In this case, the most suitable file type is the text file, where it may seem simpler to store one person per line of the file (per record). In other words, when you take a line, you can be sure that it contains the information about one person, and only that.

3.1.2) Organization of Text Files

Text files are structured into records, and it's important to understand how these records are structured in turn. There are two main approaches to structuring data within a text file: *delimited* and *fixed-width fields*.

a) Delimited

This structuring method uses a special character, known as a ***delimiter***, to mark the end of one field and the beginning of the next. The delimiter character should not appear within each field to maintain readability.

Delimited structuring has the advantage of being more memory-efficient; there is no wasted space, and a text file encoded in this way occupies the minimum amount of space possible. However, it has a major disadvantage, which is slower reading speed. Indeed, every time a line is retrieved from the file, each character must be scanned one by one to identify each occurrence of the delimiter character before the line can be split into different fields.

Example :

Let's consider the case of an address book, containing the last name, first name, phone number, and email. The data in the text file can be organized using the semicolon as the delimiter character as follows:

```
Aloui;Samir;038385214;asamir@yahoo.fr

Tadjin;Leila;0771501304;leila.tadjin@gmail.com

Abdoun;Mohammed;0559379218;moh_Abdoun@hotmail.com

Sghiri;Farid;0664490340;farid007@gmail.com
```

Figure 2.3. Example of a text file organized using a delimiter character

b) Fixed-width Fields

In this structuring method, there are no delimiter characters. Instead, we know that the first *n* characters of each line store the last name, the following *m* characters store the first name, and so forth. This approach requires ensuring that no information exceeds the predetermined width allocated for each field.

Unlike the delimiter-based structuring, fixed-width fields can lead to wasted memory space, as the file contains fixed-length records with unused spaces. However, retrieving different fields is very fast. When retrieving a line, it's simply a matter of splitting it into different strings of predefined length.

Example:

Using fixed-width fields structuring, the address book can be organized as follows:

```
Aloui       Samir          038385214         asamir@yahoo.fr

Tadjin      Leila          0771501304        leila.tadjin@gmail.com

Abdoun      Mohammed       0559379218        moh_Abdoun@hotmail.com

Sghiri      Farid          0664490340        farid007@gmail.com
```

Figure 2.4. Example of a text file organized with fixed-width fields structuring

Remark:

In the past, when memory space was costly, delimited structuring was often preferred. However, for many years now, the vast majority of software applications -and programmers- have opted for fixed-width fields structuring.

3.2) Binary Files

Unlike text files, binary files do not contain textual data and are not organized in records or lines. Instead, a binary file directly contains binary data organized as bytes, which are only meaningful to a specific program. When data is written to a binary file, it is stored exactly as it is encoded in binary in memory. This implies that a binary file cannot be read by a text editor or any other software unless a program specifically designed for it is used.

All files that do not store structured data are necessarily binary files. This includes files such as sound files, images, executable programs, etc.

3.3) Comparison Between the Two Types of Files

To summarize the difference between text files and binary files, here is a brief comparative table:

Criteria	Text Files	Binary Files
Used to store...	Structured data	Various types of data, including structured data
Structured as...	Lines (records)	No apparent structure; bytes written consecutively
Data is written as...	Characters	Directly in memory representation (binary)
Records are structured...	Optionally with a separator or in fixed-width fields	Fixed-width fields, if encoding records
Readability	Clearly readable with any text editor	Appears as an unintelligible sequence of bytes
Reading the file	Can only be read line by line	Can read bytes of choice (including the entire file at once)

Table 2.2. Differences Between Text Files and Binary Files

In this course, we will focus primarily on the basic type: sequential access text files.

4) File Manipulation

Firstly, it's important to distinguish between logical files and physical files. A logical file is a new data type, while a physical file is the hardware file placed in mass memory; the programmer does not need to know how it is structured, but only needs to know its physical name, which is a string of characters, the formation rules of which depend on the operating system being used.

Algorithmics and most current programming languages provide instructions for file manipulation. These instructions can be classified as follows:

- Opening the file.
- Processing the file (Reading, writing, and navigating within the file).
- Closing the file.

Of course, before initiating file manipulation, it is imperative to proceed with a declaration step.

Files are manipulated through a buffer. This buffer is a memory area used to temporarily manipulate data before placing it elsewhere.

When writing to a file, the data is first stored in the buffer. Once the buffer is full, its content is written to the file, and the process is repeated. This method minimizes the number of write accesses to the file, which is crucial in terms of resources (especially time). When reading, the data is copied from the file into the buffer before being transferred to the requesting reading program. Therefore, read and write operations do not occur directly in the file but in the buffer.

This buffering strategy, illustrated in the figure above, optimizes performance by reducing the number of direct accesses to the file.

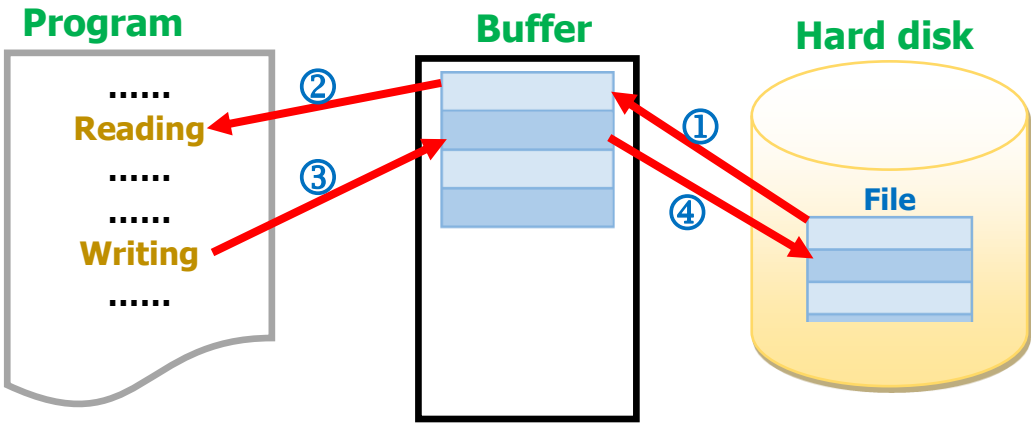


Figure 2.5. Buffering Principle

We will describe the manipulation steps in both algorithmics and the C language.

4.1) In Algorithmics

In this section, we will describe the steps for manipulating and using a file in algorithmics. The steps to follow for file manipulation, either for reading or writing, are as follows:

- Declaration of a file variable,
- Assignment of the variable to a physical file,
- Opening the file,
- End-of-file testing,
- Processing (reading, writing, etc.),
- Closing the file.

4.1.1) File Declaration

On a physical disk, a file is identified by a name (the physical or external name), for example, **`"D:/Students.txt"`**. This name can be encoded as a string of characters. To use the file throughout a program or within a subprogram, it must be identified by a variable.

This variable, known as a ***file pointer***, acts as a reference to the file in the file system. It is essential for establishing a connection between the program and the file, enabling operations such as reading, writing, and manipulation.

In fact, algorithmics has a predefined type called **`File`**, which allows file manipulation in a program.

The syntax for declaring a file variable is as follows:

```
Var <logical_name> : File;
```

Here, **`<logical_name>`** is the identifier used to reference the file variable. It serves as the internal or logical name associated with the file.

Example :

```
    Var f:File;
```

4.1.2) Assignment

As we've seen, to use a physical file in an algorithm, the algorithm must have a logical file or, in other words, a file variable. Thus, the logical file and the physical file do not necessarily have the same name.

However, the declaration alone of the file variable is not sufficient. It must be accompanied by the assignment of this variable to a real file on disk. The association between the logical file (the file variable) and the physical file (identified by an *external name* or *physical name*) is carried out through a process called ***assignment***, so that modifications made to the variable in the program will directly affect the physical file on its storage.

Assignment is performed using the Assign statement, with the following syntax:

```
Assign(<logical_name>,<physical_name>);
```

Here, **<logical_name>** is the name of the logical file (the variable used to reference the file), or what we have called the *internal name*. **<physical_name>** is the name of the physical file. The latter is the name of the file on the physical storage. This name can include the path (absolute or relative) of the file. The physical name can be a constant string or a string variable.

The assignment operation is mandatory; all accesses to the file will be made using the logical file name.

Remark:

It is possible to use the same variable to make modifications to two different physical files. To do this, simply perform the first assignment with the path of the first file, work with it, then close it, and perform a second assignment with the path of the second file.

Example :

```
Assign(f, "D:/Students.txt");
……
Assign(f, "E:/Documents/Contacts.txt");
……
```

4.1.3) Opening a File

After assigning a file, it must be opened before use. The file opening instruction must specify what will be done with it: reading, writing, or appending. This is known as the ***opening mode***.

The general syntax for this operation is as follows:

```
Open(<logical_name>,<mode>) ;
```

Where: **<logical_name>** is the logical name of the file. The **<mode>** parameter is a string indicating how the file should be opened. Thus, the possible opening modes are as follows:

- **"r"**: The file is opened in read-only mode without overwriting its content. This mode is used when the file already exists on disk and you only want to read its content. The pointer is positioned at the beginning of the file.
- **"w"**: The file is opened in write mode. This mode is used when you want to create a file that does not yet exist on disk. In this case, a file with the specified name and path during assignment is created. If the file already exists, its contents will be overwritten and lost.

- **"a"**: The file is opened in « append » mode. This mode is used when you want to open a file for adding data (only at the end of the file). Like the previous mode, if the file does not already exist, it will be created.

Examples :

Consider the following piece of code:

```
Var f1,f2:File;
Assign(f1, "D:/Students.txt");
Assign(f2, "E:/Documents/Contacts.txt");
Open(f1,"r");
Open(f2,"w");
Assign(f1, "D:/Students.txt");
Open(f1,"a");
```

In this code snippet, two file variables, **f1** and **f2**, are declared. These variables will be used as references to physical files on disk.

These variables are then assigned to the physical files **"D:/Students.txt"** and **"E:/Documents/Contacts.txt"**, respectively. This indicates which file each variable represents.

Next, the file associated with **f1** is opened in read mode, and the one associated with **f2** is opened in write mode. This step prepares the files for subsequent operations.

There is a new assignment of the file **"D:/Students.txt"** to the variable **f1**. Then, the file associated with **f1** is opened in append mode, meaning that the writing operations will occur at the end of the existing file, preserving its current content.

4.1.4) Closing a File

At the end of processing a file, the algorithm must indicate that it no longer needs the file by closing it. Closing a file after any opening is indeed mandatory. A good programmer should never overlook this step, as it helps to avoid input/output errors, preserve the integrity of file data, and optimize an algorithm by using a minimum of internal variables with a maximum of external files.

Closing a file is the reverse operation of opening it. It notably destroys the link between the logical file and the physical file, thereby releasing the resources that the system had reserved for handling the file during its opening.

To close a file associated with the variable named **<logical_name>**, the **Close** primitive is used, with the following syntax:

```
Close(<logical_name>) ;
```

Example :

```
Close(f1);
```

4.1.5) End-of-File Test

During file access, it's essential to check for the end of the file. For example, it's not possible to read from a file if there's nothing left to read.

This test can be accomplished using the predefined function **EOF**, with the following syntax:

```
EOF(<logical_name>)
```

Here, **<logical_name>** represents the logical name of the file.

The function returns a Boolean result: **True** if the end of the file is reached and **False** otherwise.

4.1.6) Reading from a file

After opening the file, several actions are possible: reading the information it contains, modifying some of it, deleting some, or adding more.

It should be noted that a text file consists of a set of lines, each representing a record, and each line contains several pieces of information (or fields) separated by a delimiter character or structured in fixed width. Thus, the file is organized.

However, there are two modes of reading from a file: unformatted line reading and formatted reading.

a) Unformatted Line Reading

In this mode of reading, the file is read line by line. With each read operation, a complete line of the file is read and stored in a variable, which must be of type **String**.

To read the content of a complete line from a text file identified by **<logical_name>** and store this line in the variable **<name_variable>**, the **FReadLn** instruction is used as follows:

```
FReadLn(<logical_name>,<name_variable>) ;
```

Example:

Consider the file **"D:/Contact.txt"** illustrated in the previous figure 2.3. The following algorithm reads the file line by line and displays them on the screen:

```
Algorithm reading_File;
Var
    f: File;   // Declaration of the file variable
    line: String;   // Declaration of the variable to store a line
Begin
Assign(f, "D:/Contact.txt");   // Assigning the file to the variable f
Open(f, "r");   // Opening the file in read mode ("r")
// While the end of the file is not reached
While EOF(f) = False Do
   Begin
   FReadLn(f, line);   // Reading the current line into the variable line
   Write(line);   // Displaying the line on the screen
   End;
Close(f);   // Closing the file
End.
```

b) Formatted Reading

In this mode of reading, the file is read information by information rather than line by line. This is useful when the file is structured in a specific way, for example, when each line represents a record with distinct fields. In such cases, it is more useful for the data read from the text file to be formatted, meaning they are interpreted and extracted according to a predefined

organization. Formatting allows each field to be processed separately and orderly, thus facilitating data manipulation in the algorithm.

During formatted reading, each field of the file is read individually, allowing for the processing of data of different types, such as integers, real numbers, or other specific types.

Formatted reading from a text file involves storing the content of the current field of the file in a variable and preparing to read the next field if it exists. Like reading from the keyboard, the type of the variable must match the type of the field being read.

The instruction for formatted reading from a text file is **FRead**,, with the following syntax:

```
FRead(<logical_name>,<name_variable>) ;
```

This instruction reads a field from a file identified by **<logical_name>** and inserts the data it contains into the variable **<name_variable>**.

Example:

Let's consider the previous file **"D:/Contact.txt"**. The following algorithm reads the file and displays its content on the screen, but this time the reading is done field by field:

```
Algorithm reading_File;
Var
    f: File;   // Declaration of the file variable
    // Declare variables for fields of the file
    firstName,lastName,email: String;
  tel: Integer;
Begin
Assign(f, "D:/Contact.txt");   // Assigning the file to the variable f
Open(f, "r");   // Opening the file in read mode ("r")
// While the end of the file is not reached
While EOF(f) = False Do
  Begin
  // Read each field from the file and store it in the respective variables
  FRead(f, lastName);
  FRead(f, firstName);
  FRead(f, tel);
  FRead(f, email);
  // Display the fields on the screen
  Write(lastName,firstName,tel,email);
  End;
Close(f);   // Closing the file
End.
```

Remarks :

- To read from a file, the file must physically exist on the secondary memory at the indicated location assigned to the logical file, and it must be opened in read mode.
- Before performing a read operation, it is necessary to test if there is still something to read (using the **EOF** function). It is not allowed to perform a read if the end of the file is reached.

- After each read operation of a field or a complete line, the read head immediately moves to the next information. If there is no more information, the pointer position becomes the end of the file.
- If you want to read information before the pointer, you need to reopen the file and read it until the desired information is reached.
- Reading a sequential file from start to finish requires the use of a loop. Since the number of fields or lines that the file contains is rarely known in advance, the **`While`** loop is typically used in conjunction with the **`EOF`** function as a condition.

4.1.7) Writing to a File

Writing is the operation that involves recording data from the computer's memory to an external file on the disk at the current position.

To write to a file, we start by opening it for writing (using the `"w"` option), which will overwrite the file if it already exists. Alternatively, we can choose to write at the end of the existing file by using the `"a"` option instead of `"w"`. Then, we write strings of characters into the file.

Similar to reading, there are two modes of writing: unformatted line-by-line writing and formatted field-by-field writing.

a) Unformatted Writing

In unformatted writing mode, data is written line by line to the file. Each write operation adds a string of characters to a separate line in the file, followed by a line break. The instruction used to perform this operation is **`FWriteLn`** with the following syntax:

```
FWriteLn(<logical_name>,<text_line>) ;
```

This primitive copies the string **`<text_line>`** into the file on the secondary memory identified by **`<logical_name>`** at the current position and adds a carriage return.

The string to be written can be a constant in quotes or contained in a variable of type **`String`**.

Example:

The following algorithm creates a file named **`"Output.txt"`** on disk **`D`**, if it does not already exist. Then, it writes to the file a string of characters from a **`String`**-type variable, followed by another constant string, placing each on a separate line.

```
Algorithm writing;
Var f: File;
    text: String;
Begin
Assign(f, "D:/Output.txt");
Open(f, "w");
text ← "Hello students";
FWriteLn(f, text);
FWriteLn(f, "Files are very important in algorithmics");
Close(f);
End.
```

b) Formatted Writing

This mode allows formatting the data written to the file, meaning organizing the information according to a predefined structure. Unlike unformatted writing, which adds distinct lines for each write operation, formatted writing offers greater flexibility for organizing data in a specific way.

In this mode, data is written field by field rather than line by line. Formatted writing is particularly useful when the file needs to follow a specific structure, for example, when each line represents a record with distinct fields. It provides better control over the layout of data in the file, facilitating subsequent reading.

Thus, to write data **<data>** of any type to the file identified by **<logical_name>**, the instruction to use is as follows:

```
FWrite(<logical_name>,<data>) ;
```

Note that this instruction writes only the specified data without adding a carriage return.

Example:

The following algorithm adds a record to the end of the existing file **"D:/Contact.txt"**:

```
Algorithm reading_File;
Var
    f: File;   // Declaration of the file variable
    firstName,lastName,email: String;
  tel: Integer;
Begin
Assign(f, "D:/Contact.txt");   // Assigning the file to the variable f
Open(f, "a");   // Opening the file in append mode ("a")
lastName ← "Benaissa";
firstName ← "Kawthar";
tel ← 0795132045;
email ← benaissa@gmail.com;
// Writing information to the file
FWrite(f, lastName);
FWrite(f, firstName);
FWrite(f, tel);
FWrite(f, email);
Close(f);   // Closing the file
End.
```

4.2) In C language

In the C language, a file is not structured by nature; it is simply seen as a sequence of bytes. It's up to the programmer to give structure to this data.

A file in C is manipulated through a pointer to a particular data structure called **FILE**, and managed through a set of functions, all defined in the standard input/output library **<stdio.h>**. An object of type **FILE*** is called a data stream. The asterisk (*****) following the keyword **FILE** indicates that it is a pointer type, enabling indirect access to file-related operations and data. Pointers will be presented in detail in the following chapter.

To manipulate a file in the C language, the steps to follow are the same as those in pseudocode. The only difference is that, in C, the assignment is done during the file opening process and not at an earlier stage.

The steps to follow are as follows:

- Declaration of a variable of type **`FILE*`** which allows interfacing a program with the file system of the operating system.
- Opening the file.
- End-of-file test.
- Processing (Reading, writing, and moving within the file).
- Closing the file.

We detail each of these operations in the following sections.

4.2.1) Declaration

The first step is to declare a file pointer that will allow us to reference the file we want to manipulate. The declaration is as follows:

```
FILE* <file_pointer>;
```

Example:

```
FILE* fp;     //fp is a file pointer
```

The declared pointer must then be initialized by opening the file before it can be used.

3.2.2. Opening

Every file must be opened to access its contents for reading, writing, or modification. Opening a file involves associating an external object (the file) with the pre-declared file pointer.

A file is opened using the predefined function **`fopen()`**, with the following syntax:

```
<file_pointer> = fopen(<physical_name>,<mode>);
```

Where:

- **`<physical_name>`** is a string specifying the full name or path of the physical file to be opened. The path can be absolute or relative.
- **`<mode>`** is a string defining the opening mode of the file.

The different possible modes for a file in the C language are the same as in algorithmics, with additional modes:

- **`"r"`** : Open for reading only. The file must exist.
- **`"w"`** : Open for writing only. The file is created if it does not exist, or it is truncated if it exists.
- **`"a"`** : Open for writing only, appending to the end of the file if it exists.
- **`"r+"`** : Open for both reading and writing. The file must exist.
- **`"w+"`** : Open for reading and writing. The file is created if it does not exist, or it is truncated if it exists.
- **`"a+"`** : Open for reading and writing, appending to the end of the file if it exists.

The function **fopen()** returns a pointer to **FILE**. The returned value is a valid pointer if the file opening is successful. In case of a file opening error (file not found, error in the file name, or permissions issue, in reading or writing, etc.), this function returns **NULL** pointer.

Example:

The following program declares a file pointer (**fp**), then opens the file **"liste.txt"** located in the directory **"d:/"** in read-only mode (**"r"**). If the file cannot be opened, it displays an error message indicating that the specified file cannot be opened.

```
main(){
  FILE* fp ; // Declare a file pointer
  fp = fopen("d:/liste.txt","r"); // Opening in read-only mode

  if(fp == NULL)
      printf("Unable to open the specified file ");
}
```

4.2.2) Closing

When all operations on a file are completed, it must be closed using the predefined function **fclose()** from the **<stdio.h>** library. Its syntax is:

```
fclose(<file_pointer>);
```

Example :

```
fclose(fp); // close the file pointed to by pf, previously opened
```

4.2.3) End-of-File Test

To check if we have reached the end of a file, we use the function **feof()**, which takes a file pointer as input and returns 1 if the end of the file has been reached, and 0 otherwise.

The syntax of this function is as follows:

```
feof(<file_pointer>);
```

4.2.4) Reading

Reading is the transfer of data from the file to the RAM. To read from a file, it must have been previously opened in **"r"**, **"r+"**, **"w+"**, or **"a+"** mode.

The C library offers several functions for reading text files, the most important of which is the formatted reading function **fscanf()**.

The **fscanf()** function, similar to **scanf()**, allows reading formatted data from a file and storing it at the addresses specified by the function's parameters. Its syntax is similar to that of **scanf()**:

```
fscanf(<file_pointer>, "<control_string>", &<variable1>,
                        &<variable2>,…);
```

Where:

- **<file_pointer>** is the file pointer returned by **fopen()**.

- **<control_string>** indicates the format in which the read data is converted. Thus, the formats here are the same as those used in the **scanf()** function, such as **%d, %f, %c,** …

The **fscanf()** function returns the number of variables actually read, which may be less than the number of variables requested to be read in case of an error or end of file. This allows detecting end-of-file or reading errors.

Example:

If we want to read and display the data from a file located at **"D:/list.txt"** containing the following information:

```
Amrani Kamel 32
Mohammedi Ilyes 28
Achouri Sofiane 34
End
```

We could use the following code:

```
int main(){
    FILE* fp;
    char lastName[20]; char firstName[20];int age;
    fp = fopen("d:/list.txt","r");
    if(fp==NULL)
       printf("File not found");
    else{
       while(fscanf(fp,"%s %s %d", &lastName,&firstName,&age)==3)
           printf("%s %s %d\n",lastName,firstName,age);
    }
    fclose(fp);
}
```

This program only displays the first three lines of the file. This is justified by the condition in the **while** loop, which tests the value returned by **fscanf()**. Since we request to read two strings and one integer, the **fscanf()** function must return 3 for a successful read. At the end of the file, reading fails in **fscanf()**, and the **fscanf()** function returns a value less than 3, causing the loop to exit without displaying the last line.

Remarks:

- With each **fscanf()** read, the file pointer automatically moves forward in the file.
- The code for reading a file depends on the file's format. Therefore, the designer of a program using files must carefully consider and specify exactly what data is in the file and in what order this data is arranged before writing the program.

a) Handling Delimiter Characters During Reading

The **fscanf** function, much like **scanf**, interrupts reading when it encounters a delimiter character such as a newline, space, or tab. Therefore, the program works correctly as long as the fields or information do not contain spaces or other delimiter characters. However, if one of the fields (e.g., the first name) contains a space, it will lead to a problem. The program will interpret this as reading more than 3 elements and will stop prematurely.

Let's return to the initial idea presented at the beginning of this chapter. Text files are typically organized into records, which have a homogeneous structure. When a delimiter character is used to separate different fields within a record, the same delimiter character is consistently applied throughout the entire file. Therefore, it is strongly recommended to choose a delimiter other than space, as the latter can frequently appear in various types of fields such as names, titles, addresses, etc.

Fortunately, the C language offers us the ability to specify the delimiter character when reading from a file. This specification is made within the control string and follows the following syntax:

```
fscanf(<file_pointer>, "%[^<delimiter>] %*c %[^<delimiter>]
        %*c …", &<variable1>, &<variable2>, ...);
```

Such as:

- **`%[^<delimiter>]`** : This part allows reading a sequence of characters until the specified delimiter character (**`<delimiter>`**) is encountered. The read characters are stored in the corresponding variable.
- **`%*c`** : This reads and ignores the delimiter character itself, ensuring that it is not included in the variable.

Example :

For example, if the fields in the previous **`"D:/list.txt"`** file are separated by semicolons, it will have the following format:

```
Amrani;Kamel eddine;32
Mohammedi;Ilyes;28
Achouri;Sofiane;34
End
```

We can adjust the **`fscanf`** function and the program for reading the file will be as follows:

```
#include <stdio.h>
int main() {
    FILE* fp;
    char lastName[20], firstName[20];
    int age;
    fp = fopen("d:/list.txt", "r");
    if (fp == NULL)
        printf("Fichier introuvable");
    else {
      //Using the semicolon (;) as delimiter
      while(fscanf(fp,"%[^;] %*c %[^;] %*c %d", lastName,  firstName,
&age)==3)
            printf("%s %s %d\n", lastName, firstName, age);
    }
    fclose(fp);
    return 0;
}
```

4.2.5) Writing

Writing is the transfer of data from the main memory to the file. To write to a file, it must have been previously opened in **"w"**, **"a"**, **"r+"**, **"w+"** or **"a+"**.

The C library provides several functions for writing to text files, with the most important being the formatted writing function **fprintf()**.

The **fprintf()** function, similar to **printf()**, allows you to write data specified by the function parameters to a file. Its syntax is similar to that of **printf()**:

```
fprintf(<file_pointer>, "<control_string>", <expression1>,
                        <expression2>,…);
```

Where:

- **<file_pointer>** is the file pointer returned by **fopen()**.
- **<control_string>** contains the text to be displayed (which may contain escape characters) and format specifications corresponding to each expression in the parameter list. The format specifications are the same as those of the **printf()** function: **%d, %f, %c,**…

The **fprintf()** function returns the number of characters written, or a negative value if there was an input-output error.

Example :

The following code writes the data of the multiplication table for 5 into a text file named "multiplication table.txt":

```
main(){
 FILE* pf; int i;
 fp = fopen("multiplication table.txt","w");
 if (fp == NULL)
        printf("Error opening the file ");
 else{
   for(i=1; i<=10; i++)
     fprintf(fp,"5 * %d = %d\n", i, 5*i);
 }
 fclose(fp);
}
```

4.2.6) Moving the File Pointer

To navigate within a file, you can use the **fseek()** function, which allows you to position the file pointer at a specific location. The **fseek()** function operates either in absolute mode from the beginning or end of the file, or in relative mode from the current position. Its syntax is:

```
fseek(<file_pointer>, <offset>, <origin>);
```

Where:

- **<file_pointer>** is the file pointer returned by **fopen()**.
- **<offset>** is a **long int** specifying the offset relative to the origin.
- **<origin>** is an integer indicating the origin of the offset.

The **fseek()** function modifies the position of the **<file_pointer>** by a number of bytes equal to **<offset>** from **<origin>**. It returns 0 upon successful positioning, and a non-zero value otherwise.

The **<origin>** argument can take one of the following values (defined as symbolic constants in the **<stdio.h>** library):

- **SEEK_SET** : the offset applies from the beginning of the file.
- **SEEK_CUR** : the offset is relative to the current read/write position.
- **SEEK_END** : the offset is relative to the end of the file.

Example :

```
main(){
  FILE* fp;
  Fp = fopen("list.txt","r+");
  fseek(fp,0,SEEK_END);     //Moves to the end of the file
  fprintf(fp,"m1");  //Writes the word "m1" at the end of the file
  fseek(fp,0,SEEK_SET); //Moves back to the beginning of the file
  fprintf(fp,"m2"); //Writes the word "m2" at the beginning, overwriting the old word
  fseek(fp,2,SEEK_CUR); //"Jumps" 2 characters from the current position
  fprintf(fp,"m3"); //Writes the word "m3" at the position obtained after the offset
  fclose(fp);
}
```

5) Conclusion

In this chapter, we delved deep into the fascinating world of computer files. We began by defining files and discussing crucial concepts such as extensions, access types, and structuring. The distinction between text files and binary files was clearly established, highlighting their respective characteristics.

At the core of our approach, we immersed ourselves in the details of file manipulation, particularly text files, by adopting an algorithmic approach to understand the theoretical foundations. We then applied this knowledge practically in the C language. We meticulously detailed the essential steps, from declaring file variables to opening, processing data, and judiciously closing files to avoid potential issues.

The variety of file access techniques was presented in all its diversity, allowing readers to grasp multiple ways of working with stored data. Whether in reading or writing, with structured or unstructured files, the acquired knowledge offers valuable versatility in managing computer data.