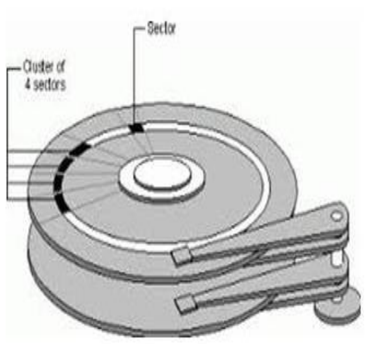
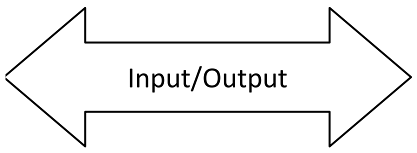
Files

Motivation (Why ?)

Until now we had only seen Inputs/Outputs (I/O) in the form of standard input (usually the keyboard, scanf) and standard output (usually the screen, printf).

This type of I/O quickly reaches its limits...
By the use of files: saving/restoring data between the program and the hard disk.





Program

```
Open File X ;
Read D from the file X ;
Close the file X ;
R ← Process (D) ;
Open File Y ;
Write R in File Y ;
Close File Y ;
```

I. Files Organisation

It is difficult to find an algorithmic notation for files. In effect, files represent an eteregenous concept of programming. There are categories, and in these categories there are sorts and so on. One principal criteria, which differentiate the two main categories of files is the following : The file is it organised or not as successive lines ? if yes, the file contains the same kind of information at each line. These lines are called records and the file is a structured one.

Let's take the classic case, that of an address book. The file is intended to store the contact details of a number of people. For each one, you will need to note the name, first name, telephone number and email. In this case, it may seem simpler to store one person per line of the file (per record). In other words, when we take a line, we will be sure that it contains information about a person, and only that person. Thus, a file encoded as records is called a text file.

In fact, after each record, the bytes corresponding to the CR (code Ascii 13) and LF characters (Ascii code 10), meaning a return to the beginning of the next line. Most often, the programming language, when it is a text file, will itself manage the reading and writing of these two characters at the end of each line: this is less than the programmer will have to take care of. The programmer will only have to tell the machine to read a line, or to write one. This type of file is commonly used when we need to store information that can be assimilated to a database (structured data).

The second type of files is defined differently: it brings together files that do not have a line structure (record). The bytes, whatever they may be, are written one after the other. These files are called **binary files**. Naturally, their different structure implies a different treatment by the programmer.
All files that store no structured data are necessarily binary files: this concerns for example a sound file, an image, a executable program, etc. . However, we will say a few words about it later, it is always possible to opt for a binary structure even in the case where the file represents structured data.

Another major difference between text files and binary files: in a text file, all Data is written as text (ascii characters). This means that the numbers are represented as a series of numbers (character strings).

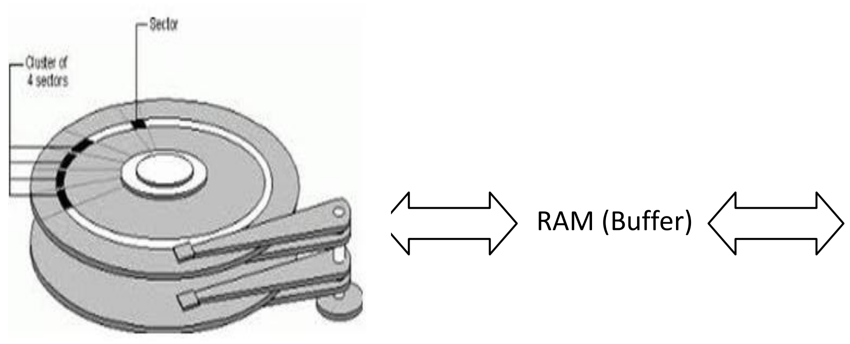
These **numbers must therefore be converted to strings** when writing to the file. Conversely, when reading the file, we will have **to convert these strings into numbers** if we want to be able to use them in calculations. On the other hand, in binary files, the data is written in the exact image of its encoding in RAM, which saves all these conversion operations.

This has the further implication that **a text file is readable**, whereas **a binary file is not** (except of course by writing an appropriate program). If you open a text file by a text editor, like Windows Notepad, we will recognize all the information (these are characters, stored as such). The same thing with a binary file does not occur to us on screen than a series of incomprehensible characters.

Mechanics of file use

Access to a file from a mass storage medium (disk hard drive, optical disk, etc.) is expensive: transfer time, deteriorating mechanics, ...

Therefore, we must reduce the number of accesses $\Rightarrow$
    Use of a memory zone called: buffer zone (buffer).
Thus, a write (resp. read) instruction in the program will not immediately result in a write (resp. reading) on the disk but by writing (resp. reading) in the buffer, with writing (resp. reading) on disk when the buffer is full (resp. empty).



Program

```
Open File X ;
Read D from the file X ;
Close the file X ;
R ← Process (D) ;
Open File Y ;
Write R in File Y ;
Close File Y ;
```

In my program, the operating system does:

- Open a file.

  $\Rightarrow$ create a buffer (b) in RAM.

- Read/write to the open file.

  $\Rightarrow$ read/write in b.

- Close the file.

  $\Rightarrow$ flush the contents of b, free b,...

II. Records Structure

Some files may be structured as records. We must then know how these records are structured. However, there are two main possibilities. These two big variants for structuring data within a text file are the **delimitation** and the **fields of fixed width**.

Let's take the case of the address book, with the name, first name, telephone and email. The data, on the text file, can be organized as follows:

Structure #1

“Betayeb” ; “Zahra” ; 0142156487 ; “z.bet@yahoo.fr”

“BenMoh” ; “Ali” ; 0456912347; “A.ben@gmail.com”

“BenAli”; “Mohammed”; 0289765194; “M.Ben@uni-biskra.dz” or so: **Structure #2**

Betayeb	Zahra	0142156487	z.bet@yahoo.fr
BenMoh	Ali	0456912347	A.ben@gmail.com
BenAli	Mohammed	0289765194	M.Ben@uni-biskra.dz

- Structure n°1 is said to be **delimited**; It uses a special character, called a **delimitation character**, which allows you to identify when one field ends and the next begins. This delimiting character must be strictly prohibited inside each field, for lack of which causes the structure to become truly illegible.
- Structure no. 2 is said to have **fixed width fields**. There is no character of delimitation, but we know that the first x characters of each line store the name, the y following the first name, etc. This of course requires not entering any longer information than the field intended to accommodate it.

The advantage of structure no. 1 is its **small footprint in memory space**; there is no wasted space, and a text file encoded in this way takes up as little space as possible. But On the other hand, it has a major disadvantage, which is the **slowness of reading**. In fact, each time we retrieve a line in the file, we must then go through all the characters one by one to locate each occurrence of the separator character before being able to split this line into different fields.

Structure n°2, conversely, **wastes memory space**, since the file is a real Swiss cheese full of holes. But on the other hand, retrieving the different fields is **very fast**. When we retrieve a line, just split it into different strings of predefined length, and the task is realized.

At the time when memory space was expensive, the delimited structure was often favored. But for many years, almost all software – and programmers – have opted for fixed-width field structure. Also, unless otherwise stated, we will only work with files built on this structure. Note: when you choose to use fixed width fields, you can then very well opt for a binary file. The records will certainly be there one after the other, without anything reports the join between each record. But if we know how many bytes measure invariably each field, we therefore know how many bytes each record measures. And we can therefore very easily retrieve the information: if I know that in my address book, each individual occupies let's say 75 bytes, then in my binary file, I deduce that the individual n°1 occupies bytes 1 to 75, individual n°2 occupies bytes 76 to 150, individual n°3 occupies bytes 151 to 225, etc.

III. Access Types

We have just seen that the organization of data within the file records could be carried out according to two major strategic choices. But there is another file sharing line: the type of access, in other words the way in which the machine will be able to retrieve the information contained in the file. We distinguish:

1. **Sequential access**: we can only access the data following that which we have just read. We can't therefore access information only by having first examined the information that precedes it. In the case of a text file, this means that we read the file line by line (record by record).
2. **Direct (or random) access**: you can directly access the record of your choice, by specifying the number of this record. But this often means tedious management movements in the file.
3. **Indexed access**: to simplify, it combines the speed of direct access and the simplicity of sequential access (although remaining more complicated). It is particularly suitable for processing large files.

Unlike the previous one, this typology is not reflected in the structure itself of the file. In fact, any file can be used with any of the three access types. The choice of type of access is not a choice that concerns the file itself, but only the way in which it will be processed by the machine. It is therefore in the program, and only in the program, that you choose the type of access you want. To conclude on all this, here is a small summary table:

	Text files	Binary files
We use them to store	Structured data	All, including structured data
They are structured as	Lines (records)	They have no apparent structure. These are written as a sequence of bytes.
The data are written	Exclusively as characters	As in RAM
Records are themselves structured as	of your choice, with a separator or in fields of fixed width	in fields of fixed width, if it is a file coding records
Readability	The file is clearly readable with any text editor	The file has the appearance of a sequence not readable bytes
Reading of the file	You can only read the file line by line	You can read the bytes of your choice (including the entire file in a single read)

As part of this course, we will voluntarily limit ourselves to the basic type: text files with sequential access. For more complete information on handling binary and other file types to access, you will need to study the documentation for the specific language in which you want to code.

IV. Instructions (text files in sequential access)

If you want to work on a file, the first thing to do is to open it. This is done by giving the file a local name (or a channel number in other languages). We cannot open only one file per channel, but whatever the language, we always have several channels, so no worries.

The important thing is that when we open a file, we stipulate what we are going to do with it: read, write or append. This is the **opening mode**.

- If you open a file for **reading**, you can only retrieve the information it contains, without modifying them in any way.
- If we open a file for **writing**, we can put all the information we want in it. But the previous information, if it exists, will be completely overwritten. And we won't be able to access information that previously existed.
- If you open a file for **appending**, you cannot read or modify the existing information. But will be able to add new lines (I remind you that instead of lines, we will prefer that of records).

These limitations are very strong. There aren't even instructions for deleting a record from a file! However, with a little practice, we realize that despite everything, even if it's tedious, you can still do whatever you want with these sequential files.

To open a text file, we would write for example:

```
Var myfile: file;
...
Assign (myfile, "Example.txt");
Open (myfile, “Read”);
...
```

myfile is the logical name of the file in the program and Example.txt is the "physical" name used by the operating system.

File Manipulation

Type

```
        Person = Record
                        Name, First_name , Email: strings [20] ;
                        Tel: string[10];
                  End;
Var
        x: person;

...
Read (myfile, x);
Write (x.Name, x.First_name, x.Email, x.Tel);
...
```

The ReadFile (or more simply Read) instruction retrieves in the specified variable the next record in the file ("next" is understood relative to the last read (record)). This is why the file is called sequential. In this case, we therefore recover the first line, therefore, the first record of the file, in the variable x.

Reading a sequential file from the beginning to the end requires programming a loop. As we know rarely in advance how many records the file has, we use the EOF function (acronym for End Of File). This function returns the value True if we have reached the end of the file (in which case further reading would trigger an error). The algorithm, ultra classic, in such a case is therefore:

```
Var
        x: person;
...
Assign (myfile, "Example.txt");
Open (myfile, “Read”);
While Not(EOF(myfile)) do
        Read (myfile, x);
        ...
Endwhile;
```

And nine times out of ten also, if you want to store the data in RAM as you go along, information read from the file, we use arrays. And since we didn't know in advance how many records there would be in the file, we also do not know how many locations should be in the array.

For a write or addition operation, you must first create a string equivalent to the new line of the file.
This string must therefore be “calibrated” in the right way, with the different fields that “fall” into the correct locations. The easiest way to avoid lengthy processing, you should proceed with correctly sized strings as soon as they are declared (most languages offer this possibility):

```
Var
X : person ;
…
Open(myfile, "Exemple.txt") ;
…
```

Such a declaration ensures that whatever the content of the Name variable, for example, it will always be 20 characters long. If its content is smaller, then the correct number of spaces will automatically be added to fill. If you try to enter content that is too long, it will be automatically truncated.

```
…
x.Name ← "Betayeb" ;
x.F_name ← "Zahra" ;
x.Tel ← "0348946532"
x.Mail ← "z.Bet@gmail.com" ;
write(myfile, x) ;
…
```

And finally, once you have finished with a file, you must not forget to close this file. We thus free the channel that it occupied (and incidentally, we can use this channel in the rest of the program for another file… or for the same one).

```
close(myfile) ;
```

V. Treatment strategies

There are generally two ways to process text files:

- one consists of sticking to the file itself, that is to say directly modifying the information on the hard drive. It's sometimes a bit acrobatic when you want to delete an element of a file: we then program a loop with a test, which copies into a second file all the elements of the first file except one; and you must then completely copy the second file to the place of the first file…
- the other strategy consists, as we have seen, of going through one or more arrays. In fact, the fundamental principle of this approach is to start, before anything, by copying the entire starting file in RAM. Then, we only manipulate the array(s) in RAM. And when the processing is finished, we copy again in the other direction, from the memory to the original file.

The advantages of the second technique are numerous, and 99 times out of 100, this is how it will be handled:

- **Speed**: accesses to RAM are thousands of times faster (nanoseconds) than access to mass memories (milliseconds at best for a hard drive). By switching the file from the

start in an array, we minimize the subsequent number of disk accesses, all processing being then carried out in memory.
- **Ease of programming**: although it is necessary to write the instructions for copying the file into the array, as long as we have to fiddle with the information in all directions, it is much easier to do this with an array than with files.

But it is better to stick to files and not use arrays when using very large files. Indeed, copying a very large file into RAM requires resources that can reach considerable dimensions. So in the case of huge files (very rare, however), because for large databases, we use DBMS), copying into memory will be problematic.

VI. Files in C language

- In C language, the information necessary to maintain the association of program, buffer, hard disk are described in a FILE structure (stdio.h).
- Among the information stored in the FILE structure, we find :
  - the number of the file to open,
  - the type of opening (read/write),
  - the address of the associated buffer,
  - the position of the reading cursor,
  - the position of the writing cursor,
  - ...
- To use a file, you must start by declaring a variable of type FILE, or more precisely a pointer to FILE (FILE *), also called data flow:

  ```
  FILE * FilePointerName ;
  ```

Opening and closing a file

The C language offers two functions for opening and closing of a file:

- The fopen function: allows you to open a file, following a mode, and returns a flow (pointer to FILE).

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

  The function returns NULL if opening is not possible.

- The fclose function: allows you to close an opened file (a flow or stream).

  ```
  void fclose (FILE * pf)
  ```

The cases of stdin, stdout, and stderr

Three standard flows can be used in C without opening or closing :

- stdin (standard input): by default the keyboard;
- stdout (standard output): by default the screen;
- stderr (standard error): by default the screen;

Opening modes

The different modes of opening a file are:

Mode	Meaning
"r"	opening a text file for reading
"w"	opening a text file for writing
"a"	opening a text file for writing at the end
"rb"	opening a binary file for reading
"wb"	opening a binary file for writing
"ab"	opening a binary file for writing at the end
"r+"	opening a text file for reading/writing
"w+"	opening a text file for reading/writing
"a+"	opening a text file for reading/writing at the end
"r+b"	opening a binary file for read/write
"w+b"	opening a binary file for reading/writing
"a+b"	opening a binary file for reading/writing at the end

getc, putc, fscanf et fprintf : definition

Multiple functions for reading/writing from/to text files exist:

- int getc(FILE * pf): returns the next character of the flow pf. It returns the EOF constant if it meets the end of the file or in case of error.
- int putc(int c, FILE * pf): writes the character c into the file associated with pf. Returns the written character or EOF in case of error.

Remarks :

- getchar() ⇔ getc(stdin)
- putchar(c) ⇔ putc(c,stdout)

- int fscanf(FILE * pf, char * format, arg1,...): read the characters on the pf flow, interprets them according to the specifications included in format. Return the number of correctly read elements or EOF in case of error (before all readings).
- int fprintf(FILE * pf, char * format, arg1,...): converts, formats (depending on format) and prints its arguments in the pf flow. Returns the number of characters printed or EOF in case of error.

Remarks :

- scanf(char * format,...) ⇔ fscanf(stdin, char * format,...)
- printf(char * format,...) ⇔ fprintf(stdout, char * format,...)