Chapter 2: The Files

1. Introduction:

The computer processes information through the processor, where data is temporarily stored as variables in the central memory. This memory, often referred to as temporary or volatile memory, retains data only as long as there is an electric current. While this provides high throughput for the storage and retrieval of data, once the power is cut off, the information is lost.

To address this, computers require additional devices for permanent data storage that does not rely on electricity for retention. These devices are known as storage units and include options such as floppy disks, hard disks, CDs, USB sticks, memory cards, and more. Unlike temporary memory, these storage units offer substantial storage space and maintain their contents even when power is disconnected. Data stored in these units is typically organized in the form of files.

While this type of memory involves longer backup and recovery times, it is essential for storing large amounts of data reliably over the long term.

2. Definitions:

A file is a collection of structured information stored on volumes in a specific format. The operating system, such as Windows, plays a crucial role in managing these files—controlling how they are stored, retrieved, organized, and managed.

Key elements related to files include:

- **File Name**: A string of characters used to specify the location of a particular file stored on the computer. Users utilize the file name to save and retrieve files.
- **File Extension**: A set of letters appended to the file name, appearing after the last point. This extension aids both the operating system and the user in determining which program can process the file. For example, 'myFile.mp3' is played by an audio player, 'myFile.txt' is opened by a text editor, and 'myFile.doc' is opened by MS Word.
- **Directory**: A virtual file organization container capable of holding a group of files and/or other folders. This feature allows for the organization of files in a tree structure.
- **Path**: A set of folder names that precisely indicates the location of a specific file. Folder names are separated by '/' or '', depending on the operating system. The starting point is the main folder, where in Windows, the main folder is the name of the storage device, typically a Latin letter followed by a colon (e.g., :\myFolder1\myFolder2\myFile.doc).

<table>
<tr><td>C:</td><td>/</td><td>myFolder1</td><td>/</td><td>myFolder2</td><td>/</td><td>myFile</td><td>.</td><td>doc</td></tr>
<tr><td>Volume Name</td><td></td><td>folder name</td><td></td><td>folder name</td><td></td><td>Short file name</td><td></td><td>extension</td></tr>
<tr><td colspan="6">path</td><td colspan="3">filename</td></tr>
<tr><td colspan="9">The full name of the file</td></tr>
</table>

3. File Types:

Files are stored as bytes, differing only in how they are interpreted. From a user's perspective, files come in various types (audio, image, video, text, application, etc.), but programmers typically categorize them into two types: text files and binary files.

- **Text files:** All data in the file is stored as symbols (characters). Even numeric values (int, float, double, etc.) are formatted and stored as strings. These characters are often organized into lines, with each line ending in '\n'. If ASCII encoding is used, each character is stored in one byte. Text files can be opened with any text editor because each byte is translated into a character (char).
- **Binary files**: These files require interpretation by the program that created them or understands their specific formatting. Each byte or group of bytes can be translated in different ways, such as to different data types (int, float, bool, char, etc.).

Example:

- The number 25 can be stored as a number, requiring a single byte, or as text, requiring two bytes.

Files are typically categorized based on their extensions:

- Text files: txt, c, h, cpp, htm, xml, js
- Binary files: exe, bin, obj, zip, doc, mp3, jpg

Observations:

- In programming languages, all input and output devices, including the keyboard and monitor, are treated as files.
- Two special files are automatically configured for all programs:
  - `**stdin**` is the standard input file, typically associated with the keyboard.
  - `**stdout**` is the standard output file, typically associated with the display.

4. File Manipulation:

While the process of storing files varies among different storage devices and systems, often entailing complexity, the operating system shields users from these details. It provides a set of functions that facilitate file operations such as creation, opening, reading, and writing.

Conceptually, a file can be envisioned as a sequence of records stored consecutively in the order of entry. Reading can only occur sequentially, as random access to a record is not possible until all preceding records have been processed. When a file is opened, the read or write head starts at the first record. During subsequent reading or writing, the head moves directly to the next record. The end of the file is identified during playback when the special **EOF** symbol, representing 'End Of File,' is encountered.

The typical handling of a file involves the following steps:

1. Declare a variable of type `**FILE***` for each file intended for manipulation. This variable points to the memory address associated with the file on the storage device.
2. Open the file for reading or writing using the `**fopen**` function, assigning the returned address to the declared variable.
3. Utilize this variable for all read and write operations on the file.

4. Close the file using the `**fclose**` function.

In the C programming language, the input and output functions are part of the `<**stdio**>` library, necessitating inclusion through the directive:

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

The following tables summarize the most important functions for processing files:

Statement	`FILE* f;`
Explanation	To create a variable

Function	`FILE *fopen(const char *filename, const char *mode);`
Return	If fopen fails, it returns **NULL**.
Explanation	The file named by FileName is opened for read or write depending on the value of the second mode parameter, see the following table, and the system reserves the temporary memory needed to process the file and initialize the file information. Returns a pointer to it to use to locate the file. It should be used in place of the filename in the rest of the program.
Example	`f=fopen("c:/test.txt","r+");`

Function	`int fclose(FILE* f);`
Return	Returns 0 if the operation was successful.
Explanation	Closes the file to clear the temporary memory.
Example	`fclose(f);`

Functions for text files

Function	`int fscanf(FILE *f, const char *format[, address, ...]);`
Return	Returns the number of variables read (inputs) and, if the end of the file is reached, returns **EOF**.
Explanation	It reads and formats the input (converts it and stores it in variables) of the file, with spaces considered as separators between the data.
Example	`fscanf(f, "%d",&x);`

Function	`int fprintf(FILE *f, const char *format[, argument, ...]);`
Return	Returns the number of bytes written, and if it fails, EOF
Explanation	Writes formatted text to the file.
Example	`fprintf(f, "%.f",y);`

Function	`int fgetc(FILE *f);`
Return	Returns c and if the end of the file is reached or an error is reached, it returns **EOF**.
Explanation	Reads a character from the file.
Example	`x= fgetc(f);`

Function	`int fputc(int c, FILE *f);`

Return	If it fails, it returns EOF
Explanation	Writes a character to the file.
Example	`fputc(c, f);`

Function	`char *fgets(char *s, int n, FILE *f);`
Return	If the end of the file is reached or an error is reached, it returns **NULL**.
Explanation	It reads an array of n-1 characters and stores it in the variable s and adds '\0' to the end, if the function encounters '\n' (end of line) it stores it in s and stops playback.
Example	`fgets(s, 30, f);`

Function	`int fputs(const char *s, FILE *f);`
Return	If it fails, it returns EOF
Explanation	Writes an s string to the file, but it doesn't store the '\0' character and doesn't add '\n' (carriage return) if it doesn't exist.
Example	`fputs(s,f);`
Functions for binaries	
Function	`size_t fread(void *ptr, size_t size, size_t n, FILE *f);`
Return	Returns the number of items read.
Explanation	'fread' reads n data items from the file, each of them has size bytes, and stores them at the location specified by '`ptr`'. Where '`ptr`' is any pointer. The total size to be read is (n x size) bytes.
Example	`fread(&t, sizeof t, 1, f);`

Function	`size_t fwrite(const void *ptr, size_t size, size_t n, FILE *f);`
Return	Returns the number of items entered.
Explanation	'fwrite' writes 'n' data elements to the file, the size of each element is size byte, where 'ptr' refers to its location in memory.
Example	`fwrite(x, sizeof x, 1, f);`
Other Functions	
Function	`int feof(FILE *f);`
Return	0 means the end is reached
Explanation	To check if the end of the file is reached or not,
Example	`If(!feof(f))...`

Function	`int fseek(FILE *f, long offset, int whence);`
Return	Returns 0 on success
Explanation	Scans the file by a certain number of bytes or a certain number of '`offset`' characters from the beginning of the file if whence is equal to SEEK_SET.
Example	`fseek(f, 0, SEEK_SET);`

Function	`int fflush(FILE *f);`
Return	Returns 0 on success
Explanation	It writes the temporary memory data to the file.
Example	`fflush(f);`

The file can be opened for read, write, or update depending on the mode value passed to fopen. The following table shows a list of symbols used when opening a file, i.e. fopen, and what they mean.

value	Explanation
r	Opens a read-only file.
w	Creates a new file and opens it in write-only mode. If the file exists, it replaces it.
a	Opens a file to add (to be written to the end of the file) and if the file doesn't exist, it creates it.
r+	Open an existing file for update (read and write).
w+	Creates a new file for the update (read and write). If a file with this name exists, it will be replaced.
a+	Open a file to update (read and write). at the end of the file. If the file does not exist, it will be created.

To specify whether the file to be opened or created is of type text, the letter t can be added to the mode (rt, wt, r+t ...), but if it is a binary file, the letter b can be added to the mode string (rb, wb, w+b ...) .

The fprintf, fscanf, fputc, and fgetc functions are the same as the printf, scanf, putchar, and getchar functions if we specify stdin and stdout as input and output files, respectively.

fprintf(stdout,"text")⇔ printf("text");	fscanf(stdin,"%d",&x) ⇔ scanf("%d",&x);
fputc(c, stdout) ⇔ putchar (c);	c=fgetc (stdin) ⇔ c=getchar();
fputs, stdout) ⇔ puts;	fgets (s, n,stdin) تشبهها gets(s);

Advanced note: **NULL** and **EOF** are two constants defined in the stdio library where NULL is the same as 0 in int and '\0' in char while EOF is the same as -1 in int and -1=0xFFFF in system 16

5. Example:

`typedef struct {`
`  int num;`
`  char name[30], gender;`
`  float avg`
`} student;`
	Declare a data structure used to store each student's information and it contains:
	- `num` is a serial number
	- `name` is a string (array)
	- `gender` has either the letter ‘m’ for a man or ‘w’ for a woman
	- `avg` a real number for the mean

`Student E, T[10];` `char str[50];`	t is an array containing student records. No more than 10 students and `E` record contains one student's information. str A string that we use to read a line from a file.

`FILE *f;`	f A pointer indicating the file you want to open.
`for (int i=0; i<10; i++) { t[i].num=i; gets(t[i].name); t[i].gender=getchar(); scanf("%f", &t[i].avg); }`	Populate the table with the student's information entered by the user We use gets instead of scanf to enter the student's name because scanf stops at the first space and can't enter compound names. getchar to enter a single character.
`f=fopen("c:/test.txt","w+");`	Create the test.txt file, open it to read and write together, and return a pointer to it
`for (int i=0; i<10; i++) fprintf(f,"%d %s %c %.2f\n", t[i].num, t[i].name, t[i].gender, t[i].avg);`	Write student information as a group of formatted lines inside the file. The file can be opened with any text editor (e.g., notepad).
`fseek(f, 0, SEEK_SET);`	Move the cursor (file position indicator) to the beginning of the file
`while (fgets(str,50,f)) printf("%s",str);`	Read the file line by line and display on the screen until the end of the file (NULL return).
`fclose(f); f=fopen("c:/test.txt","w");`	for Close the file and recreate it again, everything in it will be lost.
`fwrite(&t, sizeof t, 1, f);`	Write table ‘t’ to file. In the case of a dynamically created table, the following statement can be used: fwrite(t, sizeof student, 10, f);
`fclose(f); f=fopen("c:/test.txt","r");`	Close and reopen the read-only file, which returns the read location to the beginning. Here, if we open the file with a text editor, we will find unreadable symbols because it is a binary file, not a text.
`while (fread(&e,sizeof(e),1,f)) printf("%d %s %c %.2f\n", e.num,e.name, e.sex, e.avg);`	The file is read record by record to the variable ‘e’ before being displayed formatted on the screen.
`fclose(f);`	Close the file.

Note : In the previous structure, the name was declared as a string created by a static array and not as a pointer

```
char name[30];
```

and not

```
char *name;
```

If we created it with a pointer, the name wouldn't be saved. (See the chapter on pointers for an understanding of pointers.)