Chapter 7: Files

What happens to data stored in variables and arrays when a program finishes its execution? Can data stored in RAM be preserved after the program stops? How can a program reuse data created during a previous execution? How do real applications store large amounts of data permanently?

This chapter answers these questions.

7.1 Introduction

In the previous chapters, all data structures we studied (variables, arrays, records) were stored in main memory (RAM). However, RAM is a volatile memory: once the program terminates, all stored data is lost. In many real-world applications, data must be stored permanently, reused in future executions and shared between different programs. To overcome these limitations, programs use files, which allow data to be stored on a physical storage medium such as a hard disk or SSD[1].

7.2 Definition of a File

In algorithmics, a file is a physical data structure used to store data permanently on secondary memory. It is stored on a storage device composed of a sequence of data elements and preserved even after the program execution ends.

---

[1] SSD (Solid State Drive)
An SSD is a modern storage device that stores data using flash memory, with no moving parts. This allows SSDs to provide much faster access times, better durability, and lower power consumption compared to HDDs (Hard Disk Drive). SSDs are especially suited for operating systems and applications that require high performance.

7.3 Declaration of a File

The declaration of a file in algorithmics (C language) is performed as follows:

Syntax

In Algorithmic:

Variables file_id : **File of** Type;

In C language:

```
FILE *file_id;
```

Example: Declaration of a File containing a set of products

In Algorithmics:

Type Product = **Record**
ref, name: string[30];
price: real;
quantity : integer;
End;

Variables P: Product;
Fp: File of Product;

In C language:

```
typedef struct {
    char ref[30], name[30];
    float price;
    int quantity;
} Product;
Product P;
FILE *Fp;
```

7.4 Types of File

Files can be classified into two main types: text files and binary files.

7.4.1 Text Files

Text file stores data as characters, making it human-readable and easy to open with a text editor. The values in such files are usually separated by spaces, tabs, or line breaks. Because data is stored as readable characters, text files are generally larger and slower to read or write compared to binary files.

Reading from a text file is done either line by line or token by token, and the end-of-file is reached when no more characters remain.

7.4.2 Binary Files

A binary file stores data in its binary form, exactly as it exists in memory. Binary files are not human-readable, but they have the advantage of being smaller in size and faster to access and process. In binary files, data is read record by record or block by block, and the end-of-file is reached when no more records remain.

7.5 Operations on Files

File manipulation is based on a set of fundamental operations that allow a program to interact with data stored on secondary memory. The main operations performed on files are opening, reading, writing, appending, and closing. Among these operations, opening a file is essential, since no file operation can be performed before a file is successfully opened. Once a file is opened, the program can access its contents according to the specified mode, and when all operations are completed, the file must be closed to ensure proper resource management and data integrity.

7.5.1 Opening a File

Opening a file establishes a logical connection between the program and a physical file stored on disk. This operation associates a logical file identifier, used within the program, with a physical file, identified by its path and name. The opening operation is expressed by assigning the result of the Open instruction to the logical file identifier. The general syntax for opening a file is:

Syntax:

In Algorithmics :

file_id ← Open("path\filename.extension", "mode");

file_id : a file variable, the logical file identifier in the program.

path\filename.extension: the path of the physical file, including its name and extension, which indicates whether the file is a text file or a binary file.
Common examples of text file extensions include .txt, .csv, and .xml. and binary file extensions include: .dat, .bin

mode: specifies the opening mode, which determines how the file will be used by the program.

Several opening modes are commonly used, each corresponding to a specific type of operation:

- **The write mode (w):** creates a new file or overwrites an existing one, and positions the cursor at the beginning of the file.
- **The read mode (r):** allows access to an existing file for reading its contents without modification.

- **The append mode (a):** opens a file and positions the cursor at the end, allowing new data to be added after the existing content. If the file already exists, its previous data is preserved and new data is appended. If the file does not exist, append mode automatically creates the file before writing.

Note: append mode provides a safe way to add data to a file without risking the loss of existing information, while still ensuring that a new file is created when necessary.

For example, opening a text file for writing can be expressed as:

Fp ← Open("D:\products.txt", "w");

In this example, Fp is the logical file identifier, "D:\products.txt" represents the physical file with its path and extension, and "w" indicates that the file is opened in write mode. Similarly, opening the same file in read mode would allow the program to access its contents, while opening it in append mode would allow adding new data at the end of the file.

Syntax in C language:
In C, the opening operation is performed using the fopen function.

file_id = fopen("path\\filename.extension", "mode");

For example, opening a text file for writing, in C language, can be expressed as:

Fp = fopen("D:\\products.txt", "w");

When working with binary files, the file extension does not change between algorithmics and the C language. In both cases, the file name commonly uses the extension .dat (Or .bin). The difference lies only in the opening mode. In algorithmics, the file is opened using the usual mode notation, whereas in the C language, the character *b* must be added to the opening mode to explicitly indicate that the file is binary.

Examples:
Fp = fopen("D:\\products.dat", "wb");

Fp = fopen("D:\\products.dat", "rb");

Fp = fopen("D:\\products.dat", "ab");

 **Notes:** In C language, the double backslash \\ are use in the path.

7.5.2 Writing to a File

Writing to a file allows data to be stored permanently. When a file is opened in write mode, any existing content may be replaced, whereas append mode ensures that new data is added after the existing content. In text files, writing is performed in a formatted manner, producing human-readable characters, while in binary files, data is written in its binary representation, preserving the structure of records and numeric values.

➢ Text Files

Syntax :

In algorithmics

Write(file_id, id_var);

In C language:

```
fprintf(file_id, "format_specifier", id_var);
```

file_id : the logical file (example: Fp)
id_var: a variable of the same type as the file elements

Important note: When writing a record to a text file, you must write each field separately, because the system only stores characters and does not know where one field ends and another begins.

Text files store data as human-readable characters, often separated by spaces, tabs, or line breaks. If a file contains records, reading it requires extracting each field individually.

For example, consider the previous Product record with fields ref, name, price, and quantity. In a text file, a record might be stored like this:

P001 Sugar 85.50 10
P002 Tea 140.75 15

Before writing to the file, the program must gather the data from the user, because write will take whatever is in memory.

```
Write("Enter reference: ");
Read(P.ref);

Write ("Enter name: ");
Read (P.name);

Write ("Enter price: ");
Read (P.price);

Write ("Enter quantity: ");
Read (P.quantity);
```

To write it correctly to the file, you must write each field individually:

Write(Fp, P.ref); → fprintf(Fp, "%s", P.ref);
Write (Fp, P.name); → fprintf (Fp, "%s", P.name);
Write (Fp, P.price); → fprintf (Fp, "%f", P.price);
Write (Fp, P.quantity); → fprintf (Fp, "%d\n", P.quantity);

Now the memory of P is copied exactly to the file.
Each *Write* writes one field at a time.

You cannot write the entire record at once, because the text file only stores characters, not the structured record format.

➢ Binary Files

Syntax :

In algorithmics

Write(file_id, id_var);

In C language:

```
fwrite(&id_var, size, count, file_id);
```

&id_var : the address of the variable or record that will receive the data (if id_var is an array, its name is used instead of &id_var)
size : size (in bytes) of one element (usually obtained with *sizeof(type)*)
count : number of elements to write
file_id : logical file

In binary files, the record is stored exactly as it exists memory. You can write the entire record at once:

```
Write(Fp, P);  // Writes all fields of the record in one operation
fwrite(&P, sizeof(Product), 1, Fp);
```

No need to write each field separately, because the binary file preserves the record's structure.

7.5.3 Reading from a File

Once a file is opened in read mode, data can be retrieved from it. The reading operation depends on the type of file. In text files, data is read as characters, words, or lines, while in binary files, data is read as complete records or blocks. Reading continues sequentially until the end-of-file is reached, indicating that no more data is available for reading.

➢ Text Files

Syntax :

In algorithmics	**In C language:**
Read(file_id, id_var);	`fscanf(file_id, "format specifier", &id_var);`

file_id : the logical file (example: Fp)
id_var: a variable of the same type as the file elements

Important note: In text files, we must details the fields when reading from a file containing records.
When a file is a text file, the data is stored as human-readable characters, often separated by spaces, tabs, or line breaks. If the file contains records, reading it requires extracting each field separately, because the system does not know where one field ends and another begins.

For example, consider the previous Product record with fields ref, name, price, and quantity. In a text file, a record might be stored like this:

P001 Sugar 85.50 10
P002 Tea 140.75 15

To read it correctly, you must read each field individually:

Read(Fp, P.ref); → fscanf(Fp, "%s", P.ref);
Read (Fp, P.name); → fscanf(Fp, "%s", P.name);
Read (Fp, P.price); → fscanf(Fp, "%f", &P.price);
Read (Fp, P.quantity); → fscanf(Fp, "%d", &P.quantity);

Each *Read* retrieves one field at a time.

You cannot read the whole record at once, because the text file only stores characters, not the structured record format.

➢ Binary Files

Syntax :

In algorithmics

Read(file_id, id_var);

In C language:

```
fread(&id_var, size, count, file_id);
```

&id_var : the address of the variable or record that will receive the data (if id_var is an array, its name is used instead of &id_var)
size : size (in bytes) of one element (usually obtained with *sizeof(type)*)
count : number of elements to read
file_id : logical file

In binary files, the record is stored exactly as in memory. You can read the entire record at once:
Read(Fp, P); // Reads all fields of the record in one operation
fread(&P, sizeof(Product), 1, Fp);
No need to read each field separately, because the binary file preserves the record's structure.

Notes:
In the C language, the type of data stored in the file is determined by the operations used to read and write the data, not by the declaration itself.

7.5.4 Close a File

After opening and using a file (for reading and/or writing), it is mandatory to close it in order to release system resources, ensure that written data is properly saved, and avoid file access errors.
To close a file, we use the following syntax:

Syntax :

In algorithmics

Close(file_id);

In C language:

```
fclose(file_id);
```

7.5.5 Rename a File

The rename function allow to change (modify) the file name. It can only be applied to a file that is closed; renaming an open file may lead to errors or undefined behavior.
To rename a file, we use the following syntax:

Syntax :

In algorithmics

Rename("path\old_filename.extension", " path\new_filename.extension");

In C language:

```
rename("path\\old_filename.extension", " path\\new_filename.extension");
```

Note: If the old file contains a path and the new name does not contain a path, the file will be renamed and moved to the current working directory.

7.5.6 Remove a File

The remove function is used to delete a file permanently from the storage device. The file must be closed before removal

⚠️ Once removed, the file cannot be recovered by the program

Syntax :

In algorithmics

Remove("path\filename.extension");

In C language:

```
remove("path\\filename.extension");
```

7.6 EOF (End Of File)

EOF (End Of File) is a logical condition used to indicate that a read operation has reached the end of a file. It is not a variable and not stored inside the file. EOF (End Of File) is created automatically by the operating system or the file system when a program tries to read beyond the last data stored in a file.

When a read operation reaches the last byte (or last record) of the file, the next read attempt fails. This failure causes the system to signal the EOF condition.
In C, EOF is a symbolic constant defined in the standard library (stdio.h).

Checking for the end of file is essential to avoid reading beyond the available data. Programs commonly use EOF as a condition to control reading loops, ensuring that all data is processed correctly and that file access remains safe and reliable.

Syntax :

In algorithmics

EOF(file_id);

Returns:

true → end of file reached

false → data still available

In C language:

```
feof(file_id);
```

7.7 Conclusion

In this chapter, we introduced file handling and the main operations used to manage files. We explained how to safely access data using open, read, write, and close operations.
These concepts are fundamental for developing reliable programs that work with persistent data.