Chapter 2 Files

2.1 Introduction

During execution, algorithms store data in memory. This data disappears when the program ends. Files allow algorithms to:

- keep results for later execution,
- process data sets that are too large for memory,
- exchange data between different programs.

In C, file processing is done through the standard library `<stdio.h>` using the type `FILE`. A file is accessed through a **file stream**, represented in C by a pointer:

```
FILE *fp;
```

A program typically follows this pattern:

- Open the file.
- Read and/or write.
- Close the file.

2.2 Definition of a File

A file is stored on disk, but the program manipulates a **stream**:

- `FILE *fp` is a handle used by the program.
- the current position inside the file is tracked by the system (the “file cursor”).

A file must be opened to perform any operation. The **`fopen()`** function is used to create a new file or open an existing file. We need to specify the mode in which we want to open. Below is the syntax to open a file :

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

Where**, “**`filename”` is the name of the file to be opened, and “`mode”` defines the file's opening mode.

There are various file opening modes explained below, any one of them can be used during opening a file:

Mode	**Meaning**	**If file does not exist**	**If file exists**
"r"	read text	fails	reads from beginning
"w"	write text	creates	truncates (clears)
"a"	append text	creates	writes at end

"r+"	read/write text	fails	no truncation
"w+"	read/write text	creates	truncates
"a+"	read/append text	creates	writes at end

There are also modes like : "rb", "wb", "ab", "rb+", etc.

Example Opening a file

```
#include <stdio.h>
int main(void) {
    FILE *fp = fopen("data.txt", "r");
    if (fp == NULL) {
        printf("Cannot open file.\n");
        return 1;
    }
    return 0;
}
```

Each file must be closed after performing operations on it. The `fclose()` function closes an opened file. Below is the syntax of the this function :

```
int fclose(FILE *fp);
```

Example : Closing a file

```
#include <stdio.h>

int main() {
     FILE * file;

     file = fopen("file1.txt", "w");
     if (file == NULL) {
          printf("Error in opening file");
          return 1;
     }

fclose(file);

return 0;
}
```

2.3 File Types

2.3.1 Text Files

A **text file** stores information in the form of human-readable characters (such as digits, whitespace, and line breaks), which makes it straightforward to inspect and modify using standard text editors.

Typical input/output functions used with text files in the C programming language include:

- **Writing operations:** `fprintf, fputs, fputc`
- **Reading operations:** `fscanf, fgets, fgetc`

Example : write numbers to a text file

```
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("numbers.txt", "w");
    if (fp == NULL) return 1;

    fprintf(fp, "%d %d %d\n", 10, 20, 30);
    fprintf(fp, "%.2f\n", 3.14159);

    fclose(fp);
    return 0;
}
```

Example : read numbers from a text file

```
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("numbers.txt", "r");
    if (fp == NULL) return 1;

    int a, b, c;
    float x;

    fscanf(fp, "%d %d %d", &a, &b, &c);
```

```
    fscanf(fp, "%f", &x);

    printf("a=%d b=%d c=%d\n", a, b, c);
    printf("x=%.2f\n", x);

    fclose(fp);
    return 0;
}
```

N.B.

"**fscanf**" returns the number of input items successfully matched and assigned, **not** the number of characters read. It returns a negative value if an input failure occurs before any assignment.

"**fprintf**" returns the number of characters successfully written to the file. It returns a negative value if an output error occurs.

2.3.2 Binary Files

A **binary file** stores data in its raw internal (machine-level) representation rather than in a human-readable format. This approach is generally more compact and enables faster input/output operations.

Common C language functions used for handling binary files include:

- **Writing operations:** fwrite
- **Reading operations:** fread

Binary files are particularly suitable for storing structured data such as records.

Example : Writing and reading numbers

```
#include <stdio.h>

int main() {
    FILE *file;

    int i = 25;
    float f = 3.14f;
    double d = 9.87654321;
```

```
file = fopen("numbers.bin", "wb");
if (file == NULL) {
    printf("Error opening file for writing.\n");
    return 1;
}

fwrite(&i, sizeof(int), 1, file);
fwrite(&f, sizeof(float), 1, file);
fwrite(&d, sizeof(double), 1, file);

fclose(file);

int i_read;
float f_read;
double d_read;

file = fopen("numbers.bin", "rb");
if (file == NULL) {
    printf("Error opening file for reading.\n");
    return 1;
}

fread(&i_read, sizeof(int), 1, file);
fread(&f_read, sizeof(float), 1, file);
fread(&d_read, sizeof(double), 1, file);

fclose(file);

printf("Integer: %d\n", i_read);
printf("Float: %f\n", f_read);
```

```
    printf("Double: %lf\n", d_read);

    return 0;
}
```

Example : store and load records (binary)

```
#include <stdio.h>

typedef struct {
    int id;
    float score;
} Student;

int main(void) {
    Student s1 = {1001, 14.5f};
    Student s2 = {1002, 11.0f};

    /* Write */
    FILE *fw = fopen("students.bin", "wb");
    if (fw == NULL) return 1;
    fwrite(&s1, sizeof(Student), 1, fw);
    fwrite(&s2, sizeof(Student), 1, fw);
    fclose(fw);

    /* Read */
    FILE *fr = fopen("students.bin", "rb");
    if (fr == NULL) return 1;

    Student s;
    while (fread(&s, sizeof(Student), 1, fr) == 1) {
        printf("id=%d score=%.2f\n", s.id, s.score);
    }
```

```
    fclose(fr);
    return 0;
}
```

N.B.

"fread" returns the number of elements successfully read. If this number is less than expected, it may indicate end-of-file or a read error.

"fwrite" returns the number of elements successfully written. If the returned value is less than the requested count, a write error may have occurred.