

UNIT 11

Data File Handling

(4 Hours) — C Programming (CACS151)

11.1 Introduction to Data File Handling

In every C program studied so far, data typed at the keyboard is used by the program while it runs, and any output is shown on the screen. The moment the program ends, all of that data is lost, because ordinary variables live only in the computer's memory (RAM), which is cleared when the program terminates. A data file solves this problem: it is a named collection of related information stored permanently on a secondary storage device such as a hard disk, so that it can be created once and then read, updated, or reused any number of times — even after the program that created it has finished running, or on a completely different day.

C provides a rich set of library functions, declared in the header file `<stdio.h>`, that let a program create a file, write information into it, read information back out of it, and finally close it so the data is safely saved to disk. Because C treats a data file simply as a stream of bytes, the same basic set of functions can be used whether the file holds plain text, numbers, structures, or any other kind of data.



General sequence followed by every C program that works with data files

Fig. 11.1 — The general life cycle followed while working with a data file in C

11.1.1 Why Use Data Files?

- Permanent storage — data survives after the program terminates and even after the computer is switched off.
- Large volume of data — files can hold far more information than can conveniently be typed in every time the program runs.
- Data sharing — a file created by one program can be read by another program, or by the same program at a later time.

- Faster repeated access — once stored, the data does not need to be re-entered from the keyboard for every run.

11.2 Types of Data Files

C supports two broad categories of data files, distinguished by how the information is stored on the disk.

11.2.1 Text Files

A text file stores data as a sequence of ASCII characters. Numbers, when written to a text file, are first converted into their character (digit) representation, and lines of data are separated by a newline character ('\n'). Because the content is plain readable characters, a text file can be opened and inspected directly in any text editor.

11.2.2 Binary Files

A binary file stores data exactly as it is represented inside the computer's memory — for example, an int is stored as the same sequence of bytes that represents it in RAM, with no character conversion. Binary files are more compact and preserve exact numeric precision, but their contents are not human-readable in a plain text editor; they must be read back using a C program.

TEXT FILE (.txt)	BINARY FILE (.bin/.dat)
Stores data as ASCII characters	Stores data exactly as it is held in memory
Human-readable in a plain text editor	Not directly readable — appears as raw bytes
Each line ends with a newline character	No automatic newline translation
Some numeric data may lose precision on conversion	Exact representation — no data loss
Opened using modes such as "r", "w", "a"	Opened using modes such as "rb", "wb", "ab"

Fig. 11.2 — Key differences between text files and binary files

Aspect	Text File	Binary File
Storage form	ASCII characters	Raw memory representation
Readability	Human-readable	Not human-readable
File extension (typical)	.txt, .dat	.bin, .dat
Typical open mode	"r", "w", "a"	"rb", "wb", "ab"
Data conversion overhead	Present (number → text)	None

Aspect	Text File	Binary File
Storage size	Usually larger	Usually smaller / exact

11.3 Opening and Closing a Data File

11.3.1 Declaring a File Pointer

Before a file can be used, a pointer of type FILE must be declared. FILE is a structure, defined in <stdio.h>, that stores information the system needs to manage the file — such as its current position, buffering details, and error/end-of-file status.

```
FILE *fp; /* fp will point to the file being used */
```

11.3.2 Opening a File — fopen()

The fopen() function opens a file and associates it with a file pointer. It takes two arguments: the file name (with path, if required) and the mode in which the file is to be opened. It returns a valid FILE pointer on success, or NULL if the file could not be opened (for example, if a file being opened for reading does not exist).

```
fp = fopen("student.dat", "r");
if (fp == NULL) {
    printf("Error: cannot open file\n");
    exit(1);
}
```

Mode	Meaning
"r"	Open an existing text file for reading only
"w"	Create a new text file for writing (overwrites if it already exists)
"a"	Open a text file for appending; data is added at the end (created if it does not exist)
"r+"	Open an existing text file for both reading and writing
"w+"	Create a new text file for both writing and reading (overwrites existing content)
"a+"	Open a text file for reading and appending
"rb", "wb", "ab", "rb+", "wb+", "ab+"	Same meanings as above, but the file is treated as binary

Note: If a file opened with mode "w" or "w+" already exists, its previous contents are erased. Use "a" or "a+" when existing data must be preserved.

11.3.3 Closing a File — fclose()

Every file that is opened must eventually be closed using `fclose()`. Closing a file flushes any buffered data out to the disk, releases the `FILE` structure, and frees the file pointer for reuse. Forgetting to close a file can result in loss of data that was still held in the buffer and never actually written to disk.

```
fclose(fp);
```

11.4 Reading and Writing Data To and From a Data File

C offers several pairs of functions for transferring data between a program and a file. The right choice depends on whether the data is character-by-character, line-by-line, formatted text, or raw binary blocks such as arrays and structures.

11.4.1 Character I/O — fputc() and fgetc()

`fputc(ch, fp)` writes a single character to the file pointed to by `fp`, while `fgetc(fp)` reads and returns a single character from the file (or the constant `EOF` when the end of the file has been reached).

Example 11.1 — Copy characters from keyboard into a text file

Read characters typed by the user until the '#' symbol, writing each one into a file named "notes.txt" using `fputc()`.

```
#include <stdio.h>
int main() {
    FILE *fp;
    char ch;
    fp = fopen("notes.txt", "w");
    printf("Enter text, end with # : \n");
    while ((ch = getchar()) != '#')
        fputc(ch, fp);
    fclose(fp);
    return 0;
}
```

Explanation

getchar() reads one character at a time from the keyboard; fputc() writes that same character into the file. The loop stops as soon as the user types '#', and fclose() then saves the file to disk.

11.4.2 Line I/O — fputs() and fgets()

fputs(str, fp) writes an entire string to a file, and fgets(str, n, fp) reads up to n-1 characters from the file (or up to the next newline) into the string str. Unlike gets(), fgets() is safe because it limits how many characters can be read, preventing buffer overflow.

```
char line[100];
fp = fopen("notes.txt", "r");
while (fgets(line, sizeof(line), fp) != NULL)
    printf("%s", line);
fclose(fp);
```

11.4.3 Formatted I/O — fprintf() and fscanf()

fprintf() and fscanf() work exactly like printf() and scanf(), except that the first argument is a file pointer instead of the standard input/output stream. They are the most convenient way to write and read mixed data types — such as a record containing a name, a roll number, and marks — in a readable text format.

Example 11.2 — Store student records using fprintf()

Write the roll number, name, and marks of several students into a file "student.dat", then read them back and display them on the screen.

```

#include <stdio.h>
int main() {
    FILE *fp;
    int roll; char name[30]; float marks;

    fp = fopen("student.dat", "w");
    printf("Enter roll, name, marks (0 roll to stop):\n");
    scanf("%d", &roll);
    while (roll != 0) {
        scanf("%s %f", name, &marks);
        fprintf(fp, "%d %s %.2f\n", roll, name, marks);
        scanf("%d", &roll);
    }
    fclose(fp);

    fp = fopen("student.dat", "r");
    printf("\nRoll\tName\tMarks\n");
    while (fscanf(fp, "%d %s %f", &roll, name, &marks) != EOF)
        printf("%d\t%s\t%.2f\n", roll, name, marks);
    fclose(fp);
    return 0;
}

```

Explanation

The first loop keeps writing formatted records to student.dat until roll number 0 is entered. The file is then reopened in read mode, and fscanf() reads one full record at a time; the loop naturally stops when fscanf() returns EOF at the end of the file.

11.4.4 Block I/O — fread() and fwrite()

When speed and exact storage matter — particularly for binary files holding arrays or structures — fwrite() and fread() transfer a block of raw bytes in a single call, rather than converting each value to and from text.

```

fwrite(&record, sizeof(record), 1, fp); /* write one record */
fread(&record, sizeof(record), 1, fp); /* read one record */

```

Example 11.3 — Storing a structure using fwrite() and fread()

Define a structure Student and write one record to a binary file "stud.bin", then read it back and print it.

```
#include <stdio.h>
struct Student { int roll; char name[30]; float marks; };

int main() {
    struct Student s = {1, "Anil Shrestha", 78.5};
    FILE *fp = fopen("stud.bin", "wb");
    fwrite(&s, sizeof(s), 1, fp);
    fclose(fp);

    struct Student r;
    fp = fopen("stud.bin", "rb");
    fread(&r, sizeof(r), 1, fp);
    printf("%d %s %.2f\n", r.roll, r.name, r.marks);
    fclose(fp);
    return 0;
}
```

Function	Purpose	Typical use
fputc() / fgetc()	One character at a time	Simple text copy, counting characters
fputs() / fgets()	One line/string at a time	Reading lines of text safely
fprintf() / fscanf()	Formatted, mixed data types	Human-readable records
fwrite() / fread()	Raw block of bytes	Structures, arrays, binary files

11.5 Updating a Data File

"Updating" a file means changing, adding to, or correcting data that has already been stored, without having to retype everything from scratch. C provides two common approaches depending on what kind of change is needed.

11.5.1 Appending New Data

To add new records to the end of an existing file without disturbing the data already present, the file is opened in append mode ("a" or "a+"). Every write operation then places new data after the last existing byte, regardless of where the file pointer would otherwise be.

```
fp = fopen("student.dat", "a");
fprintf(fp, "%d %s %.2f\n", roll, name, marks);
fclose(fp);
```

11.5.2 Modifying Existing Records

To change a record that is already stored partway through a file, the file is opened with a mode that allows both reading and writing — such as "r+" for text files or "rb+" for binary files. The program locates the record to be changed (commonly with `fseek()`, covered next), overwrites it with new data, and then continues.

Update need	Suggested mode
Add new records at the end only	"a" or "a+"
Change an existing record in place	"r+" (text) or "rb+" (binary)
Rebuild the whole file with fresh data	"w" or "w+" (erases old content first)

Note: Updating a record "in place" only works cleanly when every record occupies the same fixed number of bytes — which is exactly why binary files with `fwrite()/fread()` are preferred for update-heavy applications such as a student or employee database.

11.6 Random Accessing Files

The functions used so far — `fscanf()`, `fgets()`, `fread()`, and so on — read a file sequentially, one record after another from wherever the file pointer currently stands. Random access means jumping directly to any record in the file without having to read through all the records before it. This is essential when a file is large, or when only one specific record (say, the 50th student's data) needs to be examined or updated.

11.6.1 `fseek()` — Moving the File Pointer

`fseek(fp, offset, origin)` moves the file's internal position indicator by offset bytes, measured from a reference point given by origin, which must be one of three predefined constants.

Constant	Meaning	Value
SEEK_SET	Beginning of the file	0
SEEK_CUR	Current position of the file pointer	1
SEEK_END	End of the file	2



fseek() repositions the file pointer directly to any record — no need to read through preceding records sequentially

Fig. 11.3 — *fseek()* jumps the file pointer directly to any record, using a fixed record size

Example 11.4 — Jump directly to the 5th student record

Assuming every record in "stud.bin" is the same size (a struct Student), *fseek()* can move straight to the 5th record (index 4, since counting starts at 0) and read only that one record.

```
struct Student r;
FILE *fp = fopen("stud.bin", "rb");
fseek(fp, 4 * sizeof(struct Student), SEEK_SET);
fread(&r, sizeof(r), 1, fp);
printf("%d %s %.2f\n", r.roll, r.name, r.marks);
fclose(fp);
```

11.6.2 ftell() — Finding the Current Position

ftell(fp) returns the current position of the file pointer, expressed as the number of bytes from the beginning of the file. It is often used together with *fseek()* — for instance, to find out how large a file is by seeking to the end and then calling *ftell()*.

```
fseek(fp, 0, SEEK_END);
long size = ftell(fp); /* total size of the file, in bytes */
printf("File size = %ld bytes\n", size);
```

11.6.3 rewind() — Returning to the Start

rewind(fp) resets the file pointer back to the very beginning of the file. It has exactly the same effect as calling *fseek(fp, 0, SEEK_SET)*, but is shorter to write and communicates the intention more clearly.

```
rewind(fp); /* equivalent to: fseek(fp, 0, SEEK_SET); */
```

11.7 Printing a File

"Printing" a data file usually means displaying its stored contents — either on the screen or by sending it to a printer — after it has been read back from disk. The same read functions covered earlier (*fgetc()*, *fgets()*, *fscanf()*, *fread()*) are used; the values obtained are simply passed to *printf()* or written directly to a printer device instead of being stored in variables for further processing.

Example 11.5 — Print the entire contents of a text file to the screen

Open "student.dat" for reading and display every line until the end of the file is reached.

```
#include <stdio.h>
int main() {
    FILE *fp = fopen("student.dat", "r");
    char line[100];
    if (fp == NULL) {
        printf("File not found\n");
        return 1;
    }
    while (fgets(line, sizeof(line), fp) != NULL)
        printf("%s", line);
    fclose(fp);
    return 0;
}
```

Explanation

`fgets()` reads one line at a time into the array `line[]`; `printf("%s", line)` then displays that line exactly as it was stored. The loop ends naturally when `fgets()` returns `NULL` at end of file.

Note: To send output to an actual printer rather than the screen, the same data can be written to the special stream `stdprn` (available on some compilers) or, more portably, redirected through the operating system's `print` command after being written to a file.

Key Takeaways

Unit 11 — Summary

- ✓ A data file stores information permanently on disk so it survives after the program ends.
- ✓ Text files store ASCII-readable characters; binary files store the exact in-memory representation.
- ✓ `fopen()` opens a file with a chosen mode ("r", "w", "a", and their + / b variants); `fclose()` saves and releases it.
- ✓ `fputc()/fgetc()`, `fputs()/fgets()`, `fprintf()/fscanf()`, and `fwrite()/fread()` each transfer data at a different level of granularity.
- ✓ Files are updated by appending ("a"/"a+") or by opening for both reading and writing ("r+"/"rb+") and overwriting a record.
- ✓ `fseek()`, `ftell()`, and `rewind()` allow random access — jumping directly to, or measuring, any position in a file.
- ✓ Printing a file means reading its stored contents back and sending them to the screen or to a printer.

— *End of Unit 11* —

bcalink.com