Prog C — td8

version **#0.0.1-dirty**

Assistants C/Unix 2024 <assistants@tickets.forge.epita.fr>

Copyright

Contents

1 I/O **3**
1.1 Introduction . . . 3
1.2 Streams . . . 3
1.2.1 Standard streams . . . 3
1.2.2 FILE structure . . . 4
1.2.3 Printf . . . 4
1.3 Files . . . 4
1.3.1 Opening and Closing . . . 5
1.3.2 Practice . . . 6
1.3.3 Reading . . . 6
1.3.4 Writing . . . 7
1.3.5 Browsing a file . . . 8
1.3.6 Practice . . . 9

*https://intra.forge.epita.fr

1 I/O

1.1 Introduction

I/O (Input / Ouput) is the communication between an information processing system, such as a computer, and the outside world, possibly a human or another information processing system. You are used to performing I/O operations in your day to day life. Oral communication is an example. Using your keyboard, an I/O devices that perform Input operation, to type something is another. In computer science, there is this exact same concept with some little differences. In UNIX based operating system, we perfom I/O between **files** or **stream** to send or receive information. You have already performed some I/O operation in your **C** cursus through the usage of `printf(3)` or `puts(3)`.

1.2 Streams

A stream is a sequence of data elements that are available over time. You can compare this to the stream of a river. The water particules can be compared to the data elements.

In the context of I/O, streams are sequences of characters. You can think about the streaming services. Thanks to those service you can watch a movie without having to download the whole movie because you watch while the service send it to you. That is the same process.

In computing, streams are object from which you can read or to which you can write. This is thanks to them that you can edit or process a file.

Streams are an abstract concept above the notion of files that are used in UNIX based system and **C** or **C++** programing language. It allows an easier and simpler way to perform I/O operations, you will know more about files later.

1.2.1 Standard streams

You will use 3 streams usually, the 3 standard streams

- stdout: standard output stream
- stderr: standard error stream
- stdin: standard input stream

Those three streams are the main channel of interaction between a program and its users. You use them every time you run a command without even realising. By default, when you use `printf(3)` or `puts(3)`, you write on stdout, the standard output.

1.2.2 FILE structure

The `FILE` structure is used to represent I/O streams in C. It is defined in the header `stdio.h`. You can fine some documentation here. You can only manipulate and access this type object through pointer `FILE *`.

A lot of the functions you will see in this tutorials takes in parameter `FILE *`.

Keep in mind that you will use `FILE *` to manipulate streams. It is perfectly fine if you do not understand all the underlying concepts of the `FILE` structure.

1.2.3 Printf

Let's see `printf(3)`, you know that it is use to print string, but how does that works ?

Simply put, `printf(3)` write on the stdout stream, and your terminal display this stream. That is why you can see what is printed by `printf(3)`.

There is an equivalent to `printf(3)` that allows you to choose the stream in which you will write: `fprintf(3)`. Let's take a look at its prototype:

```
int fprintf(FILE *restrict stream,
            const char *restrict format, ...);
```

Going further...

The `restrict` keyword gives information on how the pointer will be used. It is useful for the compiler to optimize the code. You can read this page to learn more about this keyword.

You can see that it takes a `FILE *` in parameter.

To write on `stderr` you could do :

```
#include <stdio.h>

fprintf(stderr, "Hello World!\n");
```

The three main streams (`FILE *`): stdin, stdout and stderr are declared in `stdio.h`. They can be used in any operation using `stream`.

1.3 Files

We have seen what are `streams` and viewed the three main ones. However, how do we *create* new ones and edit some files ?

On UNIX-based systems, everything is a file. This includes, without being limited to, files, directories, hard drives, keyboards and even printers. In order to access them, you can use the set of functions which contains `fopen(3)`, `fclose(3)`, `fwrite(3)` and others.

1.3.1 Opening and Closing

`Opening` a file is the process of allocating ressources to *view* or *edit* your file. You can view this as opening a drawer before *viewing* its content or *taking* some things to *edit* its content. If we continue the analogy, `closing` a file is like closing your drawer. You deallocate every ressources needed to maintain a file open. As you do not want to let your drawer open, you **must** close each files you have opened.

```
$42sh ls
test open.c
```

```c
/**
** \file open.c
*/

#include <stdio.h>

int main(void)
{
    FILE *file = fopen("./test", "w"); /* Open a file in writing mode */

    if (NULL == file)
    {
        puts("Could not open test file\n");
        return 1;
    }

    fprintf(file, "Mind is the pain killer\n"); /* write something in the stream file */

    fclose(file); /* close the stream */

    return 0;
}
```

```
$42sh gcc -Wall -Wextra -pedantic -Wvla -Werror -std=c99 open.c -o open
$42sh ./open
$42sh cat -e test
Mind is the pain killer$
```

The previous code opened the file `test`, wrote one sentence in it and closed it. We will explain each line.

It starts by opening a file using `fopen(3)` which creates a stream (`FILE *`) corresponding to the file `test` in the current directory. Its prototype is the following:

```c
FILE *fopen(const char *restrict pathname, const char *restrict mode);
```

It takes two arguments: `pathname` and `mode`. The path can be either relative or absolute. The mode defines how the file is opened. The following table presents a non-exhaustive enumeration of legal values for `mode`:

`r`	reading mode
`r+`	reading and writing mode
`w`	writing mode
`w+`	reading and writing mode
`a`	appending in writing mode
`a+`	appending in reading and writing mode

Going further...

Those options can be used together. They can also be used with the `b` option that opens the file in binary mode. For more information about the possible values for `mode`, check the man page of `fopen(3)`.

In the above example, we opened the file `test` with the mode `w`. It truncates the file or creates it if it does not exist. It gives the stream write access on the file. Then, the file is used with `fprintf(3)` and closed using `fclose(3)`.

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

Be careful!

When you open a file using `fopen(3)`, you must always close it using `fclose(3)`. When you allocate resources through your code you must ensure that it is correctly deallocated. In the case of opening a file, you need to close it.

1.3.2 Practice

Goal

Create a file with permissions 755.

1.3.3 Reading

After opening a file, you may want to read it. This can be achieved through multiple functions. Each one of them has its advantages and drawback. You are strongly advised to read their respective man pages.

```
int fgetc(FILE *stream);
char *fgets(char *restrict s, int size, FILE *restrict stream);
size_t fread(void *restrict ptr, size_t size, size_t nmemb,
             FILE *restrict stream);
```

- `fgetc` gets a `char` from `stream` at its current position.
- `fgets` reads a string `s` of max length `size` from `stream`.
- `fread` is equivalent to `fgets(3)` but in binary mode. Therefore, the file needs to be opened using `fopen(3)` with at least the option `b`. You need to specify the number of elements to read `nmemb` and their `size` in bytes.

```c
/**
** \file fgets_example.c
*/

#include <stdio.h>

int main(void)
{
    FILE *file = fopen("test", "r");

    if (NULL == file)
    {
        /* Handle the error case */
    }

    char buf[24];

    if (NULL == fgets(buf, 24, file)) /* fill buf with the content of file */
                                      /* with at most 24 element */
        printf("Error occured while reading\n");

    puts(buf);

    fclose(file);
}
```

```
$42sh gcc -Wall -Wextra -pedantic -Wvla -Werror -std=c99 fgets_example.c -o fgets_example
$42sh ./fgets_example
Mind is the pain killer
```

1.3.4 Writing

The following functions allow you to write text into your `FILE *` stream.

```c
int fputc(int c, FILE *stream);
int fputs(const char *restrict s, FILE *restrict stream);
size_t fwrite(const void *restrict ptr, size_t size, size_t nmemb,
              FILE *restrict stream);
```

- `fputc(3)` writes char `c` to `stream` at its current position.
- `fputs(3)` dumps `s` into `stream`.
- `fwrite(3)` is equivalent to `fputs(3)` but in binary mode. Hence, you need to specify the number of elements and their size. Do not forget that the file must be opened using at least the `b` option.

```c
/**
** \file fwrite_example.c
*/

#include <stdio.h>

int main(void)
```

(continues on next page)

(continued from previous page)

```c
{
    FILE *file = fopen("test", "ab+");

    if (NULL == file)
    {
        /* Handle the error case */
    }

    const char *buf = "I will face my fear\n";

    fwrite(buf, sizeof(char), 20,  file); /* Write at most 20 char from buf */
                                          /* to file */

    fclose(file);
}
```

```
$42sh gcc -Wall -Wextra -Wvla -pedantic -Werror -std=c99 fwrite_example.c -o fwrite_example
$42sh ./fwrite_example; cat -e test
Mind is the pain killer$
I will face my fear$
```

1.3.5 Browsing a file

For now, you have only learned how to open a file and closing. However, how can you write or read something at a specific place in the file ? In the `FILE` structure, there is a field named `fpos` that corresponds to the actual location of the stream. When opening a file with `r` option, `fpos` is the beginning of the file, whereas for `a` option, it is a the end.

You can also deplace the current position of your stream arbitraly using `fseek(3)`. You are really advised to check the man page. Its prototype is :

```c
int fseek(FILE *stream, long offset, int whence);
```

`whence` can be either `SEEK_SET`, `SEEK_END`, `SEEK_CUR` that corresponds respectively to the beginning, the end or the current position of the file.

```c
/**
** \file offset.c
*/

#include <stdio.h>

int main(void)
{
    FILE* file = fopen("test", "r");

    if (fseek(file, 5, SEEK_SET) == -1) /* move the current position */
    {
        puts("Error on fseek\n");
    }

```

(continues on next page)

(continued from previous page)

```c
    char buf[18];

    if (NULL == fgets(buf, 18, file)) /* fill buf with the content of file */
    {
        puts("Could not get content of file\n");
    }

    puts(buf);

    fclose(file);
}
```

```
$42sh gcc -Wall -Wextra -Wvla -pedantic -Werror -std=c99 offset.c -o offset
$42sh ./offset
is the pain killer
```

1.3.6 Practice

Goal

Write a function that returns the number of lines in file `file_in`. If any error occurs, returns `-1` (for instance, if the file does not exist). Please note that an empty file is considered as having one line, and the `\n` character does not create a new line if it is at the end of the file.

You must open the file with read-only permission.

```c
int count_lines(const char *file_in);
```

I must not fear. Fear is the mind-killer.