CSSE2310 Week 3 – File IO

1 Files

The header `stdio.h` defines the `FILE` type and a number of functions that use `FILE` pointers to manipulate files.

1.1 `FILE *`

`FILE` is a structure that contains

- A buffer for buffered input and output
- An end-of-file indicator, which is set after an unsuccessful read, see `feof`
- An error indicator, which is set when an error occurs, see `ferror`
- A position indicator, to mark the current position in the file

`FILE` is an *opaque type*, which means that rather than accessing the structure directly, everything is done via standard functions.

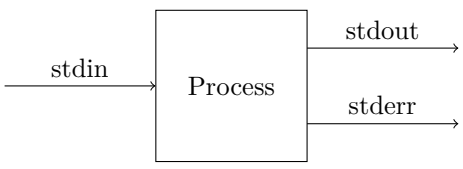
1.2 Functions

- `fopen(const char *, const char *)`
- `fgetc(FILE *)`
- `fgets(char *, int, FILE *)`
- `fprintf(FILE *, const char *, ...)`
- `fflush(FILE *)`
- `fclose(FILE *)`

See `man stdio` for a complete list of functions.

1.3 Standard streams

There are three predefined `FILE` pointers: `stdin`, `stdout`, and `stderr`.



When you run a program in the terminal, normally you won't be able to differentiate between `stdout` and `stderr`. Anything printed to either stream will be appear in the terminal.

Any function that takes a `FILE` pointer can accept `stdin`, `stdout`, or `stderr`. When writing your own functions that read or write to files, rather than opening the file inside the function, consider having the function take a `FILE` pointer so that the function works with the standard streams.

See `man stdin` for more details about the standard streams.

1.4 Opening and closing

A file can be opened using `fopen`, which will return either a `FILE` pointer or `NULL` if something goes wrong. You should always check for `NULL` in case an error occured.

`fopen` takes a "mode": a string that describes the manner in which the file should be opened. These modes are `"r"`, `"r+"`, `"w"`, `"w+"`, `"a"`, or `"a+"` and have the following meaning

`"r"` reads from the file and `"r+"` reads and writes.
`"w"` writes to the file and `"w+"` reads and writes. If the file exists, it will be overwritten.
`"a"` appends to the file and `"a+"` appends and reads.

When you've finished with the file, you should always `fclose` it to clean up resources.

See `man fopen` and `man fclose` for more details.

1.5 `fgetc`

Reads the file character-by-character. This is the function to use when you need precise control of how the input is parsed.

Common gotchas:

- Returns an *integer* not a character. `EOF` is a negative integer (usually -1). So, assuming `int`s are 4 bytes, EOF will be 0xffffffff,

which when cast to a `char` is truncated to 0xff. Consider what happens if the file contains a byte with value 255 (0xff).
- `feof` is set only after an attempt is made to read and there are no characters left (so if the file is "abc", the end-of-file indicator is set after the 4th call to `fgetc`)

See `man fgetc` for more details.

1.6 `fgets`

Reads the file chunk-by-chunk.

- reads at most `size - 1` characters so that it has space for the null terminator (`'\0'`)
- stops when it hits end-of-file or a newline
- if a newline is hit, it is stored into the buffer, so **sometimes** a newline will be stored and sometimes it won't be. This depends on how long the line is. If n in the following is `size - 1`, then
  - if line contains less than n characters, then the newline is stored
  - if line contains more than n characters or exactly n characters, then the newline is not stored, since it is not hit
  - if the last line in the file is not newline terminated, then no newline is stored

  A simple fix is to always check if the last character of the string is a newline and replace it with a null terminator.

Example If we wanted to read lines of no more than 80 characters, how long does the buffer have to be? Well, it's got to be at least 81 characters, because we have to account for the NUL character. And if the line is 79 characters or less, then the newline will be stored.

See `man fgets` for more details.

1.7 `fflush`

The `FILE` structure has an internal buffer. When the buffer becomes full, the contents of the buffer are "flushed" out to the stream. Under some circumstances, a print can get stuck in the buffer and we have to manually flush it using `fflush`.

Here are some rules of thumb

- Always print debugging messages to `stderr`, since it is not buffered (writes are immediate). We do this because if the program crashes, then any messages stuck in the buffer won't be printed, which can make debugging a headache.
- If you are expecting a response back (such as a prompt to the user), always flush the stream before reading, otherwise you might be left waiting for a response from the user, but the user hasn't actually received your request, and your program will hang forever.

1.8 A warning about `*scanf`

The `scanf` family of functions can be tricky.

- Code such as the following is a dangerous security hole and **will overflow** the buffer if the user enters too many characters.

  ```
  char buf[BUFFER_SIZE];
  scanf("%s", buf);
  ```

- the format string `"%d %d"` will permit any number of whitespace characters (such as tabs, spaces, or newlines) before either of the two numbers.
- using the format string `"%d %d"`, integer overflow will go undetected. That means scanning `4294967299` might give you `3`.

With care, `scanf` and friends can be used safely, but if you want exact control over how the input should be handled, use `fgetc` or `fgets`.