Lecture 13 - File I/O (last updated: 2023-10-19)

In the previous lecture, we encountered three channels with which programs can communicate with their environment: stdin, stdout, and stderr. They are declared in stdio.h as the following FILE pointers (i.e., their type is FILE *):

```
extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;
```

These FILE pointers are the same abstraction used to represent the files we're familiar with (e.g., text and binary files on our file system containing data); the core interface for standard I/O and file I/O is the same. In this lecture, we will explore that interface.

FILE pointers

The following declaration says, "stdin is a pointer to a FILE":

```
extern FILE *stdin;
```

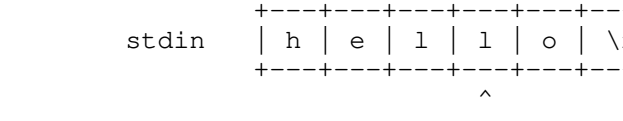
FILE is an "opaque type"; we are not supposed to know how exactly it is defined, only that it exists (as required by the C standard). It is usually defined with a typedef of some sort, e.g.:

```
typedef struct _IO_FILE FILE;
```

It doesn't matter what FILE is exactly -- it is different in different systems. A FILE pointer (i.e., a variable of type 'FILE *') represents a file that we are reading from or writing to, and we pass it to the standard library functions like fread() or fwrite() to indicate which file we want to operate on. So a FILE pointer is a "handle" to the file we are working with. And because we are not supposed to care about the internals of FILE, it's an "opaque handle."

The fact that stdin, stdout, and stderr are FILE pointers indicates one of the central design principles of the UNIX operating system. Under UNIX, the concept of a "file" goes far beyond the files on storage devices. Basically, anything we can read from or write to are treated as files. Those include hardware devices like keyboard, terminal screen, microphone, and speaker. They also include conceptual facilities like network connections between two processes running on two different computers.

A FILE pointer represents a common characteristic among all those things that UNIX considers a "file": a sequence of bytes with a position. For example, if we typed in "hello\n" to stdin, but we have only read 3 bytes, the file stream might look something like this:



Files (streams of bytes) differs from memory (an array of bytes) in that a stream is supposed to be accessed in sequence; e.g., if I read another 'l' from stdin, I will advance the position ("^"), and the next byte I read will be 'o'. In contrast, memory does not prescribe any such access order.

Though FILE pointers all share similar file stream semantics, specific FILE pointers may have capabilities that others do not. For example, some FILE pointers can only be read from (e.g., stdin); others may only be written to (e.g., stdout, stderr); some allow you to change the file stream position, while others do not.

Opening and closing files

stdin, stdout, and stderr are already available to all C programs, but to get a FILE pointer corresponding to any other file, we use fopen() ("file open"):

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

fopen() returns a non-NULL FILE pointer on success, and returns NULL on failure.

For example:

```
FILE *fp = fopen("myfile.txt", "r");
```

Both arguments are strings; the first argument is the path of the file we are trying to open, while the second is the "mode" we are trying to open this file in. Here are valid values for the mode argument (as documented in man fopen):

- "r": open file for reading; fail if file does not already exist. The stream is positioned at the beginning of the file.
- "w": open file for writing; create file if it does not exist, and overwrite it (i.e., delete old contents) if it already does. The stream is positioned at the beginning of the file.
- "a": open file for appending; create file if it does not exist, but keep old contents if it already does. The stream is positioned at the end of the file before each write operation.
- "r+": open file for reading and writing; fail if file does not already exist. The stream is positioned at the beginning of the file.
- "w+": open file for reading and writing; create file if it does not exist, and overwrite it (i.e., delete old contents) if it already does. The stream is positioned at the beginning of the file.
- "a+": open file for reading and appending; create file if it does not exist, but keep old contents if it already does. The stream is positioned at the end of the file before each write operation. The initial position of the stream is system-dependent.

You can add "b" to each of these modes (e.g., "wb", "r+b") to specify that you want to open the file in "binary" mode. The "b" option has no effect on Linux or any other UNIX-based systems. In a UNIX system, a file is just a sequence of bytes, so there are no fundamental differences between text and binary files. A text file is simply a file whose content consists of human-readable bytes.

In Windows, however, the "b" option disables a silent translation that the system performs when reading/writing newlines. Without the "b" option -- i.e., when reading/writing text files -- Windows perform the following translation:

- when reading, turn \r\n into \n
- when writing, turn \n into \r\n

This is because Windows represents a newline as a two-byte sequence \r\n in a text file. We know that a newline is simply '\n' in C, hence the need for a translation. When we work with a non-text file (a photo, for example), such a translation is obviously a bad idea.

In summary, in order to make our code portable across different platforms, we simply need to specify "b" when we open a file that is not a text file.

Once you are done using a FILE, you should always fclose() it, e.g.:

```
fclose(fp);
```

You only need to fclose() FILE pointers you obtain from fopen(); you do not need to fclose() any of the standard FILEs. After you fclose() a FILE pointer, you should no longer use it (similar principle as a dangling pointer).

Writing output

To write to any FILE pointer, you can use fprintf():

```
fprintf(stdout, "write to stdout\n");
fprintf(stderr, "write to stderr\n");
fprintf(fp, "write to myfile.txt\n");
```

If you do not need string formatting (see end of this lecture note), you can also use fputs() to write a string, or fputc() to write a single character:

```
fputs("write to stdout\n", stdout);
fputs("write to stderr\n", stderr);
fputs("write to myfile.txt\n", fp);

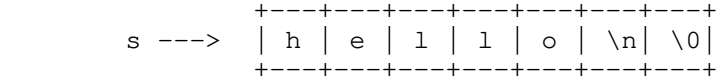
fputc('o', stdout);
fputc('e', stderr);
fputc('t', fp);
```

The first argument to fputs() has type char *; it is supposed to be a string, i.e., a pointer to the beginning of a null-terminated char array. fputs() will keep outputting characters until it reaches the null byte, but it will NOT output the null byte itself.

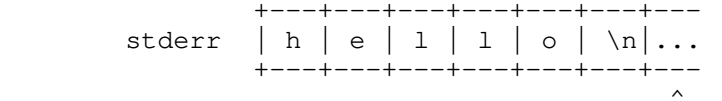
So if you call:

```
char *s = "hello\n"
fputs(s, stderr);
```

s points to the following null-terminated character array in memory:



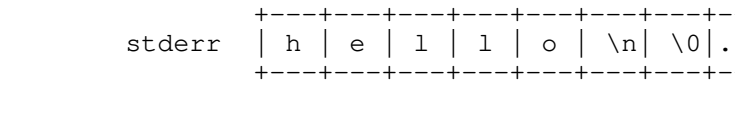
But only the following bytes are written to stderr:



If you would like to write arbitrary bytes out to a file stream, including null bytes, you should use fwrite():

```
fwrite(s, 1, 7, stderr);
```

fwrite() expects you to tell it how much to write out in terms of "items". The second parameter indicates the size of each item, while the third parameter indicates the number of items; the first parameter is a pointer to the first element of an array of those items. So in the above call, we are asking fwrite() to write 7 items, each 1 byte, from s, to stderr:



The difference between fputs() and fwrite() is that fwrite() allows you to explicitly convey the number of bytes you would like to write, without relying on a null byte to indicate when you would like to stop writing bytes. fputs() is usually only used to output textual data from strings, whereas fwrite() is used to output arbitrary binary data.

Note that text files do NOT include the null terminator. The null terminator is just what C uses to mark the end of a string in memory. A text file is simply a sequence of human-readable characters along with a few whitespace characters like spaces, tabs, and newlines.

Here are the full descriptions of fputs() and fwrite():

```
int fputs(const char *str, FILE *file)
```

- writes str to file.
- returns EOF on error.

```
size_t fwrite(const void *p, size_t size, size_t n, FILE *file)
```

- writes n objects, each size bytes long, from the memory location pointed to by p out to file.
- returns the number of objects successfully written, which will be less than n when there is an error.

Reading input

Reading input is more complicated than writing output because of blocking. Blocking happens when you ask the operating system for something it cannot return immediately, so it pauses the execution of your process until it is ready to "unblock". From the perspective of your program, it will seem as if the function call does not return until your program unblocks.

An example of blocking is sleep():

```
sleep(1);                       // Blocks until 1 second later.
```

When you read input that is not yet available, e.g., read from stdin before you've typed in anything, your program will block until it receives input:

```
        +---
stdin   |...        (nothing to read yet)
        +---
          ^
```

So if you call fgets() on stdin, it will block and your program will hang (not do anything) until you type something.

You can read string input using fgets() (opposite of fputs()), characters using fgetc() (opposite of fputc()), or binary input data using fread() (opposite of fwrite()).

fgetc() is the simplest: it just returns the character it read:

```
int c = fgetc(fp);  // we will discuss later why it returns int, not char
```

fgets() and fread() work by reading into a buffer you pass them:

```
char buf[1024];                 // Define 1024 bytes of space to read into

fgets(buf, sizeof(buf), fp);    // Read at most 1023 bytes to buf; add '\0'

fread(buf, 1, sizeof(buf), fp); // Read at most 1024 bytes (no additional 0)
```

The fgets() function is meant for reading a single line from a text file. It will return either when it encounters a newline or when its buffer fills up; either way, it guarantees that the buffer is null-terminated. The fread() function is better suited for arbitrary binary data.

Here are the full descriptions of the functions:

```
char *fgets(char *buffer, int size, FILE *file)
```

- reads at most size-1 characters into buffer, stopping if newline is read (the newline is included in the characters read), and terminating the buffer with '\0'
- returns NULL on EOF or error (you can call ferror() afterwards to find out if there was an error).

```
size_t fread(void *p, size_t size, size_t n, FILE *file)
```

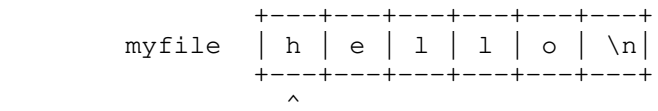
- reads n objects, each size bytes long, from file into the memory location pointed to by p.
- returns the number of objects successfully read, which may be less than the requested number n, in which case feof() and ferror() can be used to determine status.

End of file

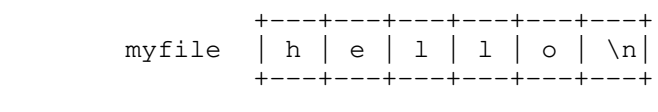
File reading functions like fgetc(), fgets(), and fread() will also unblock when they encounter some kind of error, or when they encounter an EOF ("end of file") condition. An EOF condition is not considered an error: it just happens when you try to read bytes, and the system tells you there cannot possibly be more bytes to read. For example, EOF happens when you try to read the 7th byte of a file that only contains 6 characters, e.g., one created like this:

```
    $ echo "hello" > myfile         # myfile is 6 bytes (including newline)
```

When we fopen() myfile, we get a handle to this stream:



If we read all six bytes, we will advance the cursor past the end of the stream:



If we try to read from myfile again, we will encounter an EOF condition, and fgetc()/fgets()/fread() will unblock immediately. The return value of each of these functions will differ from normal:

- fgetc() normally returns a byte that it read. When it encounters an error or an EOF condition, it needs to return a special value that is different from the normal byte that it returns. This is why fgets() returns an int. Normally it will return the byte that it read as an unsigned char (i.e., a value between 0 and 255). On error or EOF condition, it will return -1. The standard library has the following definition:

  ```
  #define  EOF  (-1)
  ```

  So a typical fgetc() call goes like this:

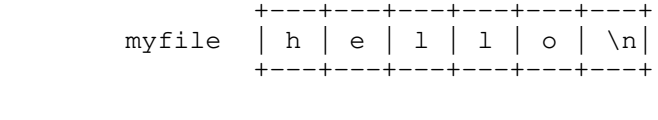
  ```
  int c = fgetc(fp);
  if (c == EOF) {
      // ...
  ```

- fgets() normally returns a pointer to the buffer it writes to; when it encounters EOF or an error, it will return a NULL pointer.

- fread() normally returns the number of items it read; when it encounters EOF or an error, it returns less than the number of items you asked for

To distinguish between EOF and errors, you can use feof() or ferror().

Note that EOF is only encountered when we try to read PAST the end of a stream. If the stream position is here:



feof() will still return zero for the FILE pointer of myfile (indicating that an EOF condition has not been raised) until we actually try to read more.

EOF conditions are not exclusive to real files; you can also have EOF on stdin. You can raise the EOF condition on stdin by typing Ctrl-D when there is no other pending input. (By the way, you can also use Ctrl-D to exit the shell; when you raise EOF in the shell, you are signaling the end of your input, and the shell will interpret it as your intention to log out.)

FILE output buffering

You may have noticed that printf() doesn't seem to do anything until you print a newline:

```
printf("hello...");
sleep(3);
printf("world\n");
```

The above program will not print "hello..." until after it sleeps; that is, "hello..." and "world" will appear to print at the same time.

This surprising behavior is due to FILE buffering. printf() (and other FILE output functions like fputc(), fputs(), and fwrite()) will actually buffer bytes given to stdout, and wait until they encounter a newline character to transmit the output to the terminal. This is called "line buffering". It makes terminal output operations more efficient without significantly effecting interactivity. When stdout is connected to the terminal screen, it is line buffered by default.

In contrast, stderr is "unbuffered" by default, meaning it will always send output immediately. This makes sense because stderr is typically used for error messages and debugging. So the following will behave as expected:

```
fprintf(stderr, "hello...");
sleep(3);
fprintf(stderr, "world\n");
```

All other files are block-buffered by default; instead of waiting for a newline, output is buffered until a fixed-size buffer is filled. This makes sense because, unlike the terminal screen, bytes written to files are usually not observed by the user.

If you insist on printing to stdout immediately before encountering a newline, or writing bytes to a file on disk before the buffer is filled, you can manually "flush" the buffer using fflush():

```
printf("hello...");
fflush(stdout);  // Immediately flushes out "hello..."
sleep(3);
printf("world\n");
```

Finally, if you want to turn off all buffering for a FILE pointer, and make it unbuffered like stderr, you can call setbuf() with NULL as the second argument. For example, the following call will make stdout unbuffered just like stderr:

```
setbuf(stdout, NULL);
```

File seeking

For some FILE pointers, we can change the current position of the underlying stream using fseek():

```
int fseek(FILE *file, long offset, int whence)
```

- Sets the file position for next read or write. The new position, measured in bytes, is obtained by adding offset bytes to the position specified by whence. If whence is set to SEEK_SET, SEEK_CUR, or SEEK_END, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively.

- returns 0 on success, non-zero on error

Here are some example usages:

```
fseek(fp, 0, SEEK_SET);   // Set stream position to the beginning
fseek(fp, 5, SEEK_CUR);   // Advance stream position by 5 bytes
fseek(fp, -3, SEEK_CUR);  // Retract stream position by 3 bytes
fseek(fp, 12, SEEK_SET);  // Set stream position to 12 bytes past beginning
fseek(fp, 0, SEEK_END);   // Set stream position to the end of file
```

However, this only works on certain FILE pointers. For example, this does not make sense for stdout, since you cannot "unprint" what has already been printed.

Formatted I/O

We have used printf() and fprintf(), which print output to stdout and other files according to a format string containing conversion specifiers like "%d" and "%s". Here are some common conversion specifiers:

```
%d: int
%u: unsigned int
%ld: long
%lu: unsigned long

%f: double
%g: double (trailing zeros not printed)
%s: string (char *)
%p: memory address (void *)
```

There are many more specifiers, and many modifiers that provide fine-grained controls of width and precision. (Type "man 3 printf" to see the full man page of the printf() family of functions in the Standard C library; the number "3" will pull up the page in Section 3, which contains the man pages for library functions, rather than the page for the printf command in Section 1.)

This is what their signature looks like:

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

The ellipsis ("...") means that more arguments can follow, depending on how many conversion specifiers are found in the first argument. Such functions that can take a variable number of arguments are called variadic functions.

scanf() and fscanf(), which read input from stdin or other files according to a format string, have similar signatures:

```
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

Sometimes we want to use the powerful formatting and parsing functionalities provided by printf() and scanf(), not for doing any I/O, but for simply converting native types like ints and doubles to and from strings. The Standard C Library provides the "s"-variants of those functions for that purpose:

```
int sscanf(const char *input_string, const char *format, ...)
```

- Parse input from input_string instead of stdin

```
int sprintf(char *output_buffer, const char *format, ...)
```

- Write output to output_buffer instead of stdout
- output_buffer is assumed to point to large enough memory; will result in a buffer overrun if that's not the case

```
int snprintf(char *output_buffer, size_t size, const char *format, ...)
```

- A safer version of sprintf(); it writes at most size bytes (including the null terminator)

Summary

By now, we've encountered a lot of C standard library functions, especially for file I/O. It sometimes can be helpful to see them all in one place and learn them in pairs:

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

  vs

  `int fclose(FILE *stream);`

- `char *gets(char *s);  // Do NOT use gets(); it is unsafe`
  `char *fgets(char *s, int size, FILE *stream);`

  vs

  ```
  int puts(const char *s);
  int fputs(const char *s, FILe *stream);
  ```

- `int getchar(void);`
  `int fgetc(FILE *stream);`

  vs

  ```
  int putchar(int c);
  int fputc(int c, FILE *stream);
  ```

- `size_t fread(void *ptr, size_t size, size_t n, FILE *stream);`

  vs

  `size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);`

- ```
  int printf(const char *format, ...);
  int fprintf(FILE *stream, const char *format, ...);
  int sprintf(char *output_buffer, const char *format, ...)
  int snprintf(char *output_buffer, size_t size, const char *format, ...)
  ```

  vs

  ```
  int scanf(const char *format, ...);
  int fscanf(FILE *stream, const char *format, ...);
  int sscanf(const char *input_string, const char *format, ...)
  ```

It can also be helpful to know that C standard library functions tend to follow similar naming conventions for single-character abbreviations:

- 'f' at the beginning stands for "FILE pointer," e.g., fgets(), fprintf(), fseek(); these functions take a FILE pointer as an argument

- 's' stands for "string," e.g., fgets(), fputs(), snprintf(); these functions use or produce C strings, i.e., null-terminated arrays of characters

- 'f' at the end stands for "format," e.g., printf(), fprintf(), snprintf(); these functions require a format string of some kind

- 'n' in string-related functions usually mean a size limit of n bytes, e.g., snprintf(), strncpy(), strncat()
  ```