COMP1521

W8 - Files

Overview

Admin

Files

File operations

Admin

Assignment 2 released!!! More MIPS!!!!!!

W7 weekly test due on thursday

W8 weekly test released on thursday

W8 labs due next monday

Assignment 1 style marks probably some time in W9 (no promises)

Admin: Assignment 2

General advice:

- Complete this week’s lab before starting the assignment (especially read lit file)
- Start early (and if you want help from tutors, come to help sessions as early as you can)
- We will cover all content needed for the assignment today

Admin: Assignment 2

Some of my recommendations:

- Make a struct to represent the state of the program (memory, registers, etc)
- Write lots of helper functions (if you complete the entire assignment I would expect at least 50 helper functions. Generally, length of functions should not exceed 50 lines).
- Specific helper functions that are good to create:
    - Read an N byte little endian integer from a file
    - Read an N byte little endian value from IMPS memory
    - Write an N byte little endian value to IMPS memory
    - Extract the opcode from an instruction
    - Extract the first/second/third register from an instruction
    - Extract an immediate value from an instruction

Assignment 2: sign extension

In many parts of assignment 2 you will need to perform sign extension.

For example:

You have a 16 byte signed value stored inside uint16_t type.

You want to perform arithmetic and add it to a uint32_t.

If you just add them without any sign extension, a negative 16 byte value -1 would be treated as positive 65535 for example.

0b00000000_00000000_00000000_00000000 + 0b11111111_11111111

= 0b00000000_00000000_11111111_11111111 = 65535 (not -1)

Assignment 2: sign extension

In another example you might have a 1 byte signed value inside a uint8_t that you wish to sign extend into a 32 bit register.

In this example, the signed value -1 would be stored as positive 255:

```
uint8_t val = 0b11111111;

uint32_t result = val; // result = 0b00000000_00000000_00000000_11111111 =255
```

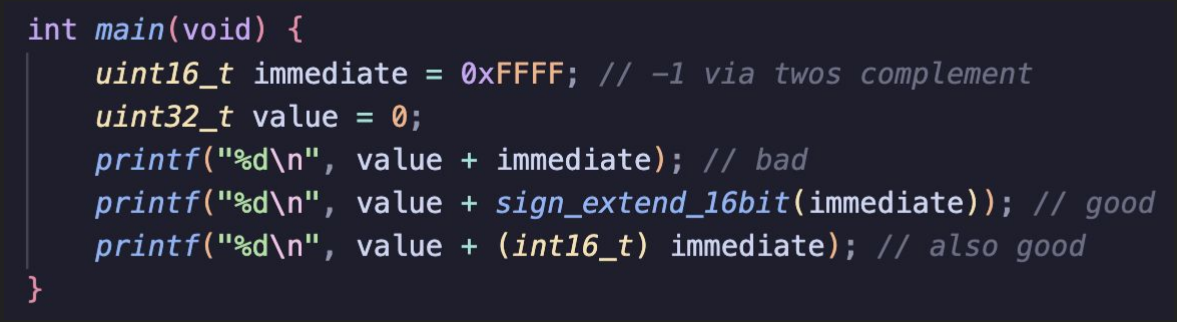
Assignment 2: sign extension

The issue arise because we are storing signed values inside unsigned types. This is sometimes something we do when working with low level values (like implementing memory access), so that we don’t get annoying issues with bitwise ops on signed types.

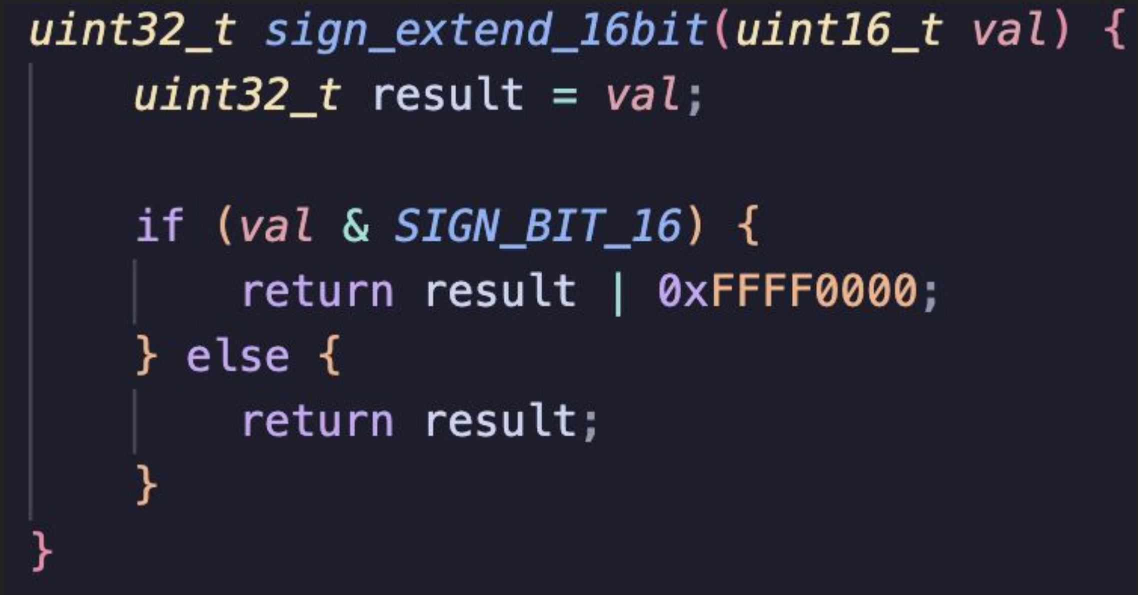
Solution 1: cast the value to a signed type (must be of the same size!!!!!!) - C automatically performs sign extension on signed types.

Solution 2: manually add the extra 1 bits (can leave as unsigned, twos complement will make the number act negative when it overflows even if it’s unsigned)

Example for 16 bit sign extension



```c
int main(void) {
    uint16_t immediate = 0xFFFF; // -1 via twos complement
    uint32_t value = 0;
    printf("%d\n", value + immediate); // bad
    printf("%d\n", value + sign_extend_16bit(immediate)); // good
    printf("%d\n", value + (int16_t) immediate); // also good
}
```



```c
uint32_t sign_extend_16bit(uint16_t val) {
    uint32_t result = val;

    if (val & SIGN_BIT_16) {
        return result | 0xFFFF0000;
    } else {
        return result;
    }
}
```

Files

What is a UNIX file

A file is a stream of data

This includes:

- ascii text
- c program
- compiled executable
- a directory (a small amount of data describing the contents of the directory)
- a network connection (socket; a stream of data received/sent over the internet)
- peripherals (the stream of data received from your keyboard)
- a symbolic link to another file (the stream of data in the other file)

The philosophy is that “everything is a file”

What is a UNIX file

Some special files that will be relevant are called “stdin”, “stdout” and “stderr”.

These files typically represent the console (they are typically connected to the console).

stdin represents the user input in the console.

stdout represents regular program output to the console

stderr represents error program output to the console

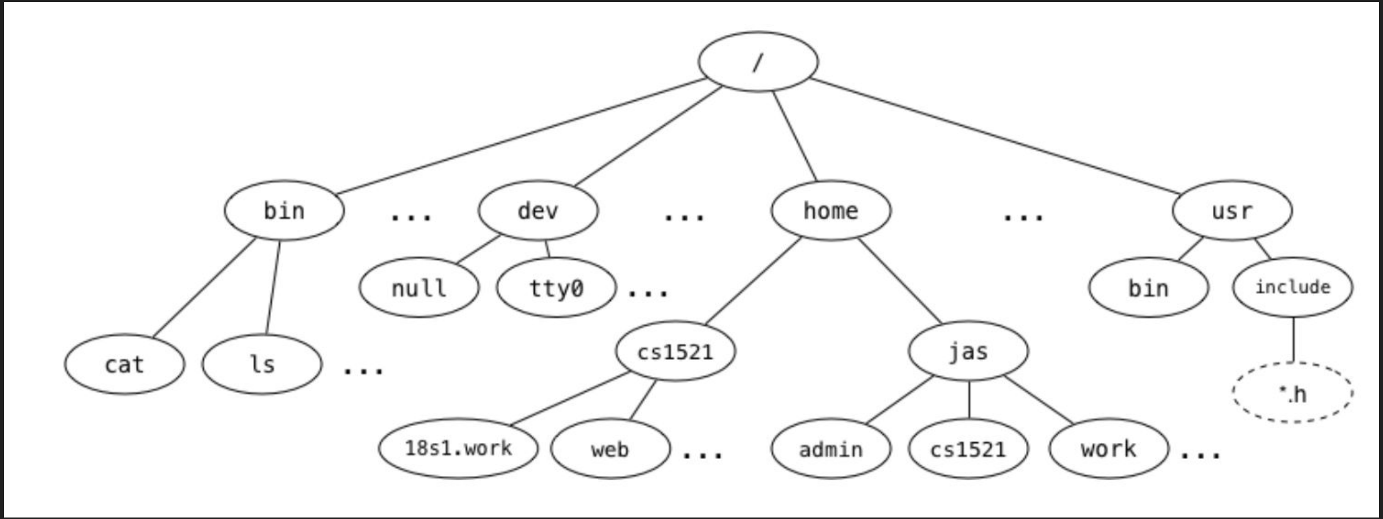
We can interact with these “files” the same way we interact with any other file.

How are files organized?

Most files are contained in a tree-like structure of directories.

The root directory (/) contains subdirectories home, dev, usr, etc

Those subdirectories can contain more subdirectories, as well as files.



How do we refer to files

A file can be referred to by its full “path” from the root directory

E.G: /import/kamen/4/z55555555/example.txt   (absolute path)

We can also use a relative path, relative to the directory we are currently inside.

For example, if I am in the directory `/import/kamen/4/z55555555/`, then I can just write `./example.txt` or even `example.txt` to refer to the above file.

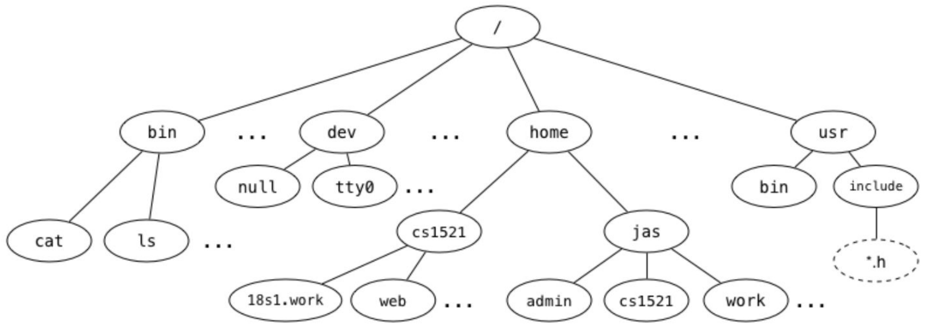
`.` is a special directory path that always refers to the current directory

`..` is a special directory path that always refers to the parent directory

`~` is a special directory path that always refers to the absolute path of the home directory for the current user (`~user` can be used if you want a certain user).

Tutorial Q1

1. We say that the Unix filesystem is *tree-structured*, with the directory called `/` as the root of the tree, e.g.,



Answer the following based on the above diagram:

a. What is the full pathname of COMP1521's `web` directory?

b. Which directory is `~jas/../..` ?

c. Links to the children of a given directory are stored as entries in the directory structure. Where is the link to the parent directory stored?

d. What kind of filesystem object is `cat` ?

e. What kind of filesystem object is `home` ?

f. What kind of filesystem object is `tty0` ?

g. What kind of filesystem object is a symbolic link? What value does it contain?

h. Symbolic links change the filesystem from a tree structure to a graph structure. How do they do this?

File operations

Low level file operations

At the lowest level, files can be read and modified using syscalls (Linux syscalls, as opposed to MIPS syscalls).

The level above that are the C functions open, close, read and write. These functions call the syscalls themselves.

The level above that (the functions we actually use) are the functions fopen, fclose, and a family of functions that read and write to files (fgetc, fread, fputc, fwrite, etc).

These functions call the lower level open/close/read/write functions, which in turn call the syscalls to actually do the operations.

File pointers

File pointers refer to specific files.

They also store information about how the file has been opened (ie, read or write), as well as where in the file the pointer is looking (1st byte, 2nd byte, etc).

Typically, a file pointer is initialised to point to the first byte of a file.

File pointers can be used to write and read bytes to and from files.

When writing or reading `n` bytes using a file pointer, the pointer is typically moved forwards by the number of bytes written/read.

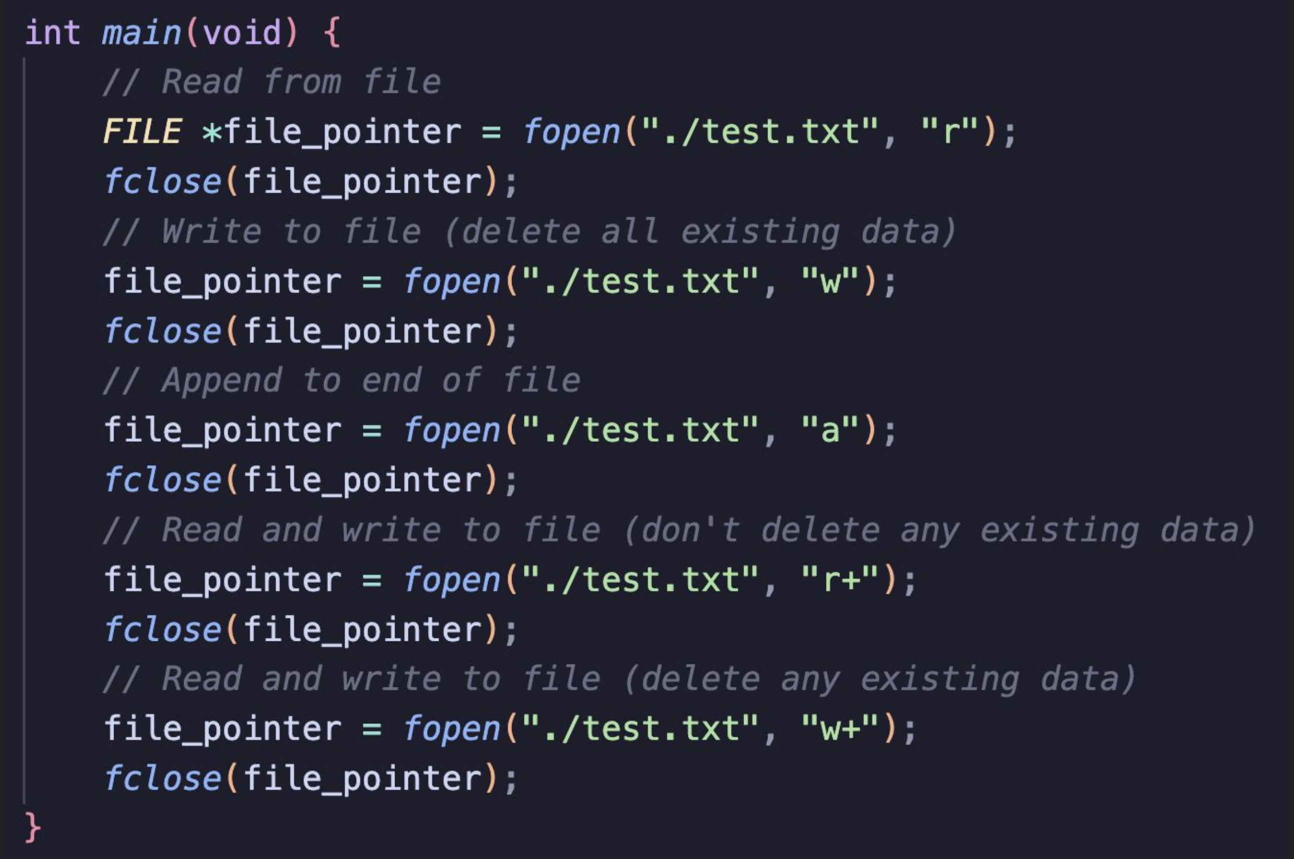
Opening and closing files

Using functions fopen and fclose.

Most useful modes to open file in are “r”, “w”, “a” and “r+”.

“r” and “r+”will fail if the file does not already exist.

Closing the file ensures all input is actually written to the file



```c
int main(void) {
    // Read from file
    FILE *file_pointer = fopen("./test.txt", "r");
    fclose(file_pointer);
    // Write to file (delete all existing data)
    file_pointer = fopen("./test.txt", "w");
    fclose(file_pointer);
    // Append to end of file
    file_pointer = fopen("./test.txt", "a");
    fclose(file_pointer);
    // Read and write to file (don't delete any existing data)
    file_pointer = fopen("./test.txt", "r+");
    fclose(file_pointer);
    // Read and write to file (delete any existing data)
    file_pointer = fopen("./test.txt", "w+");
    fclose(file_pointer);
}
```

Writing data to files

```c
int main(void) {
    // Open file in write mode
    FILE *file_pointer = fopen("./test.txt", "w");
    // Write a single byte to the file (and move the file pointer forward 1 byte)
    fputc('A', file_pointer);
    // Write multiple bytes to the file (and move the file pointer forward 4 bytes)
    fwrite("ABCD", 1, 4, file_pointer);
    // Write a formatted string to the file (and move the file pointer forward 3 bytes)
    fprintf(file_pointer, "%d\n", 42);
    // Close the file
    fclose(file_pointer);
}
```

Write a single byte using `fputc` or multiple bytes using `fwrite`.

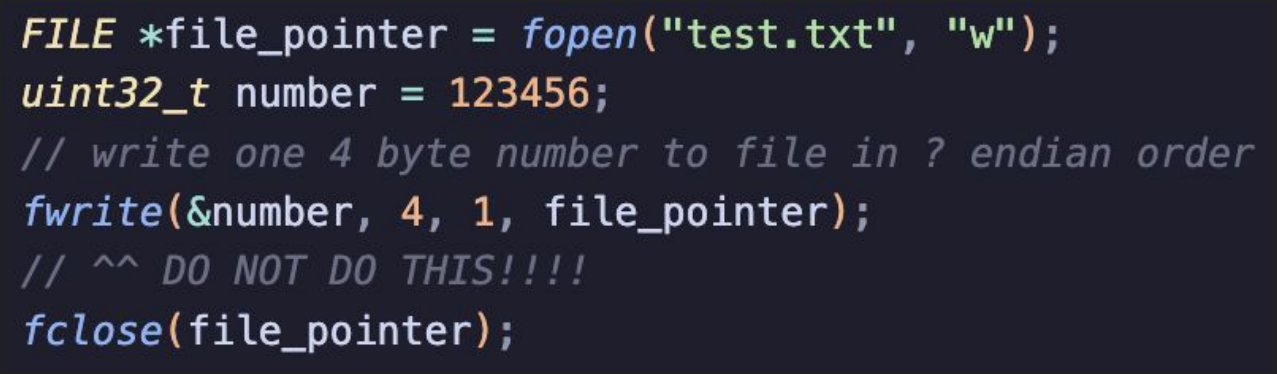
Write a formatted string using `fprintf`.

Portability of fwrite

fwrite should only be used with elements that have a size of 1 byte.

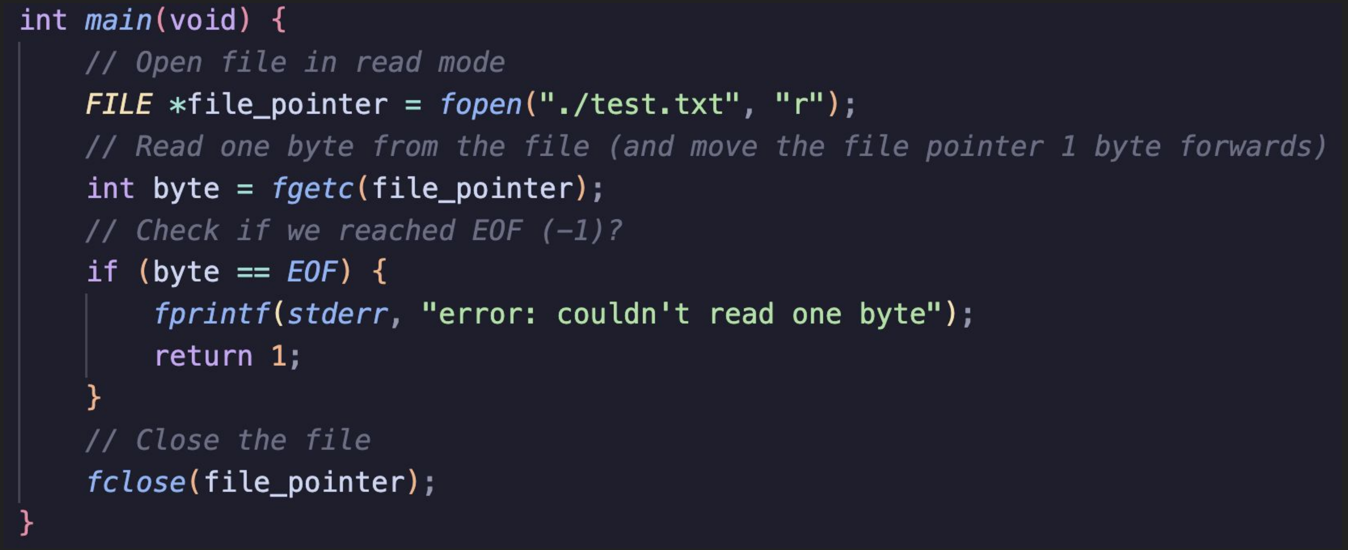
For larger elements, the endianness (BIG/LITTLE) used to store the elements will depend on the machine. This can lead to inconsistent behaviour.

As such using multi-byte elements with fwrite is strongly discouraged, and will result in a deduction of style marks if you do this in the assignment.



```c
FILE *file_pointer = fopen("test.txt", "w");
uint32_t number = 123456;
// write one 4 byte number to file in ? endian order
fwrite(&number, 4, 1, file_pointer);
// ^^ DO NOT DO THIS!!!!
fclose(file_pointer);
```

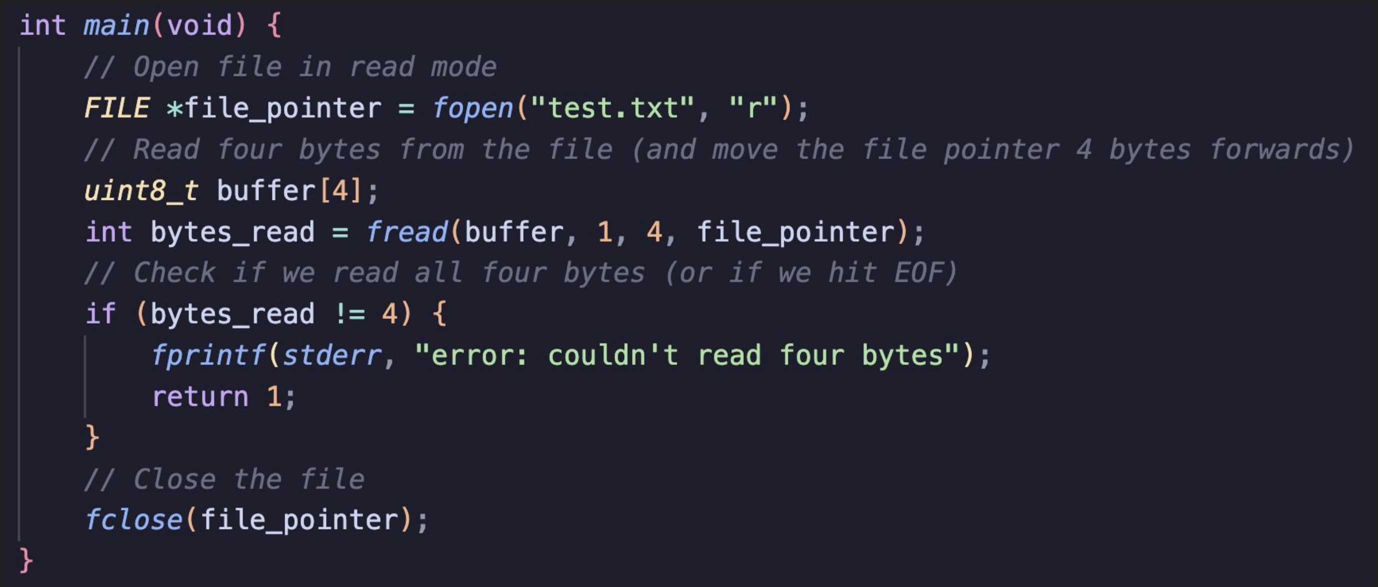
Reading data from files: fgetc



```c
int main(void) {
    // Open file in read mode
    FILE *file_pointer = fopen("./test.txt", "r");
    // Read one byte from the file (and move the file pointer 1 byte forwards)
    int byte = fgetc(file_pointer);
    // Check if we reached EOF (-1)?
    if (byte == EOF) {
        fprintf(stderr, "error: couldn't read one byte");
        return 1;
    }
    // Close the file
    fclose(file_pointer);
}
```

Note we stored the return value inside an `int`. This is necessary, otherwise it’s impossible to distinguish between an EOF (-1) and the byte 0xFF.

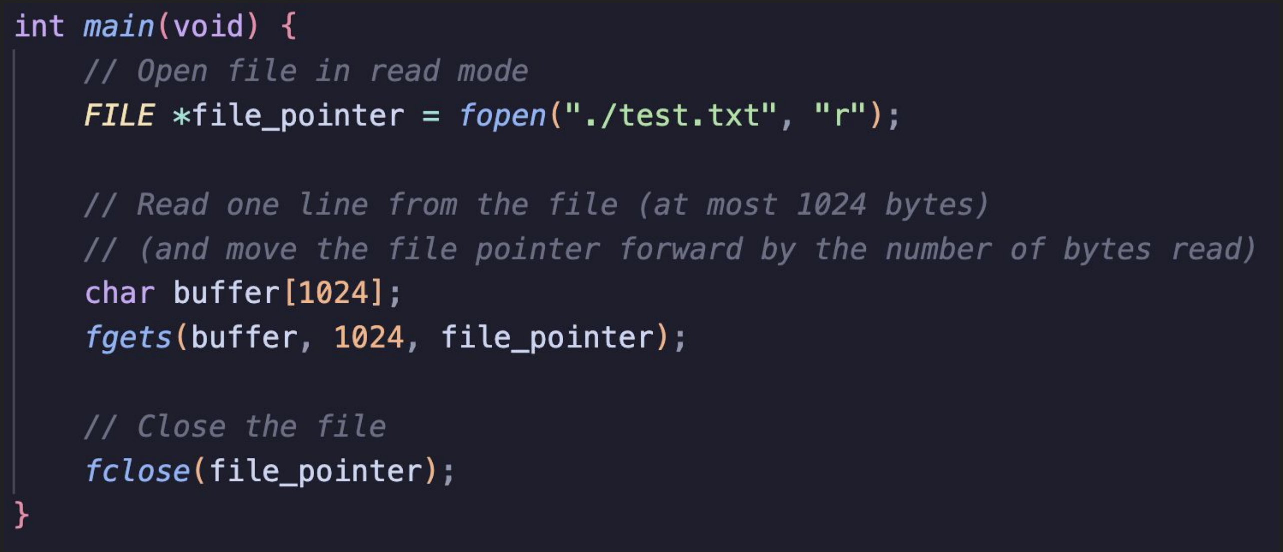
Reading data from files: fread



```c
int main(void) {
    // Open file in read mode
    FILE *file_pointer = fopen("test.txt", "r");
    // Read four bytes from the file (and move the file pointer 4 bytes forwards)
    uint8_t buffer[4];
    int bytes_read = fread(buffer, 1, 4, file_pointer);
    // Check if we read all four bytes (or if we hit EOF)
    if (bytes_read != 4) {
        fprintf(stderr, "error: couldn't read four bytes");
        return 1;
    }
    // Close the file
    fclose(file_pointer);
}
```

fread has the same portability issues as fwrite. Do not use an element size larger than 1 byte.

Reading data from files: fgets



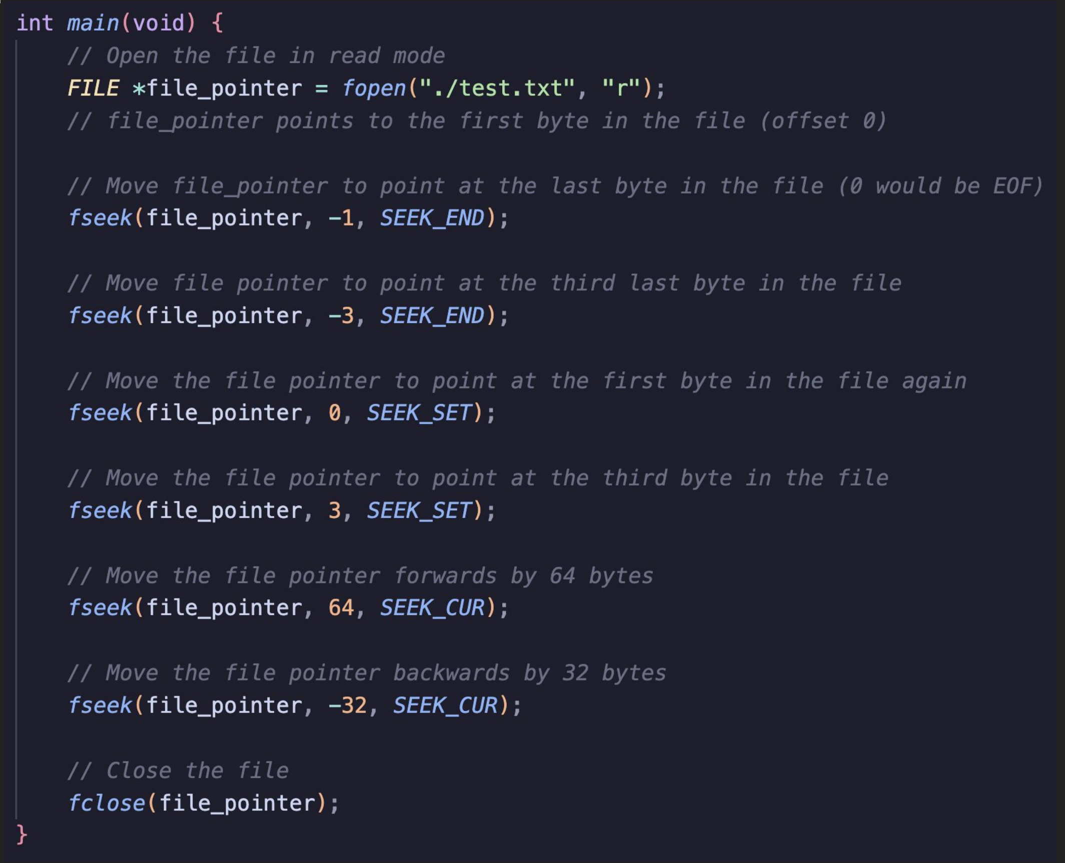
```c
int main(void) {
    // Open file in read mode
    FILE *file_pointer = fopen("./test.txt", "r");

    // Read one line from the file (at most 1024 bytes)
    // (and move the file pointer forward by the number of bytes read)
    char buffer[1024];
    fgets(buffer, 1024, file_pointer);

    // Close the file
    fclose(file_pointer);
}
```

Useful for text files, but not so much for reading raw bytes. Fgets stops reading when it sees a newline byte (0x0A) or reaches EOF (not the same thing as the null byte 0x00).

Manually moving the file pointer

Can seek relative to start, end or current position of file pointer.

SEEK_END confusingly starts seeking 1 byte past the end of the file.

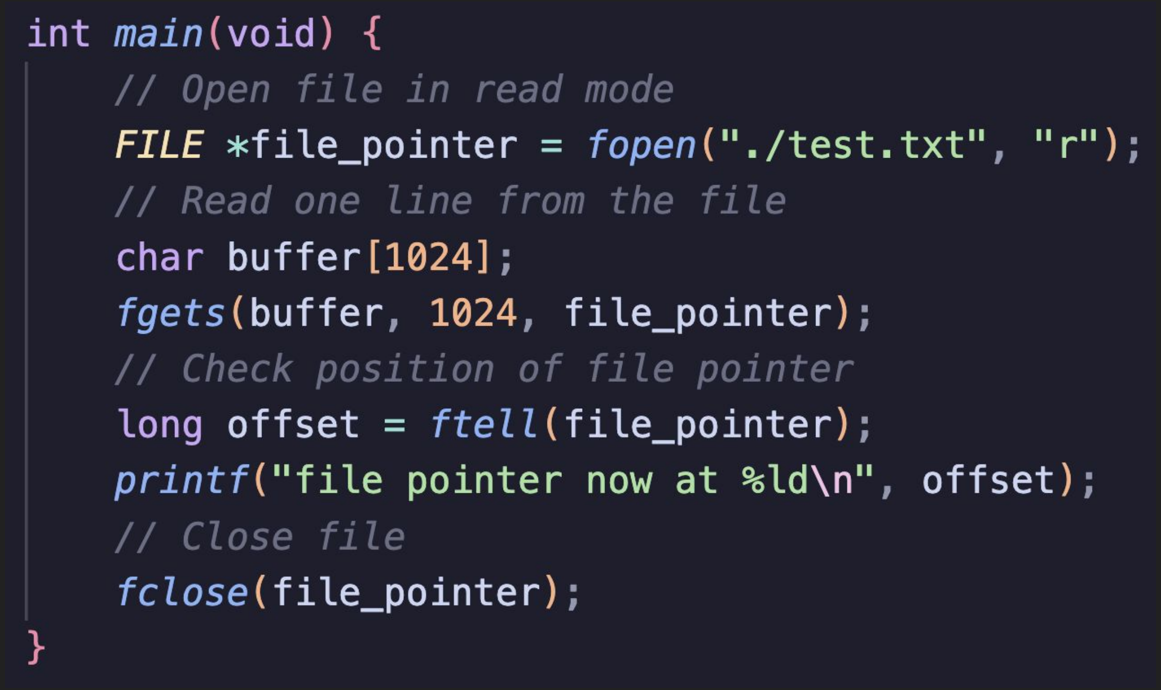


```c
int main(void) {
    // Open the file in read mode
    FILE *file_pointer = fopen("./test.txt", "r");
    // file_pointer points to the first byte in the file (offset 0)

    // Move file_pointer to point at the last byte in the file (0 would be EOF)
    fseek(file_pointer, -1, SEEK_END);

    // Move file pointer to point at the third last byte in the file
    fseek(file_pointer, -3, SEEK_END);

    // Move the file pointer to point at the first byte in the file again
    fseek(file_pointer, 0, SEEK_SET);

    // Move the file pointer to point at the third byte in the file
    fseek(file_pointer, 3, SEEK_SET);

    // Move the file pointer forwards by 64 bytes
    fseek(file_pointer, 64, SEEK_CUR);

    // Move the file pointer backwards by 32 bytes
    fseek(file_pointer, -32, SEEK_CUR);

    // Close the file
    fclose(file_pointer);
}
```

Checking the position of the file pointer

ftell returns a long (larger int).

This can be printed using formatted string with the %ld specifier (useful for debugging)

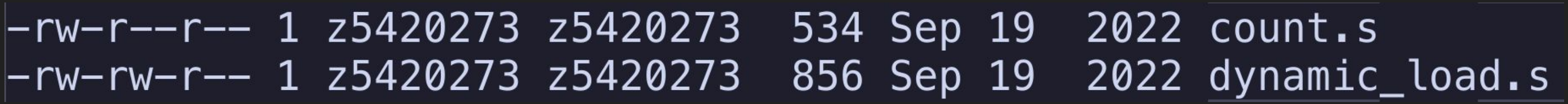


```c
int main(void) {
    // Open file in read mode
    FILE *file_pointer = fopen("./test.txt", "r");
    // Read one line from the file
    char buffer[1024];
    fgets(buffer, 1024, file_pointer);
    // Check position of file pointer
    long offset = ftell(file_pointer);
    printf("file pointer now at %ld\n", offset);
    // Close file
    fclose(file_pointer);
}
```

File permissions

File permissions are stored in a file’s metadata.

They are displayed to the terminal when using commands like `ls`:



```
-rw-r--r-- 1 z5420273 z5420273  534 Sep 19  2022 count.s
-rw-rw-r-- 1 z5420273 z5420273  856 Sep 19  2022 dynamic_load.s
```

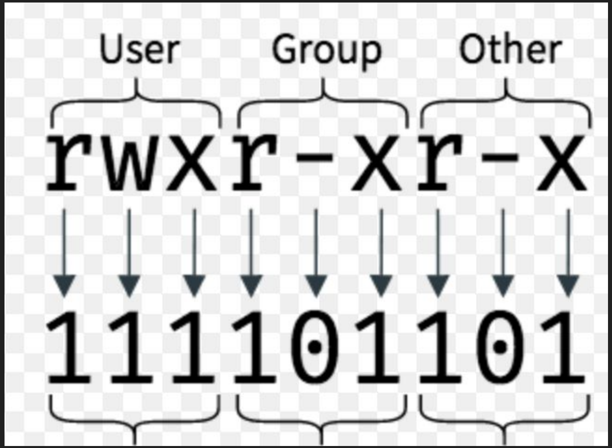
They are stored in a bitwise representation:

: ttttrwxrwxrwx

(first 4 bits represent file type)

File permissions can be read using `stat`.

File permissions can be modified using `chmod`.



User = you, Group = you (+ maybe friends), Other = everyone on the system.

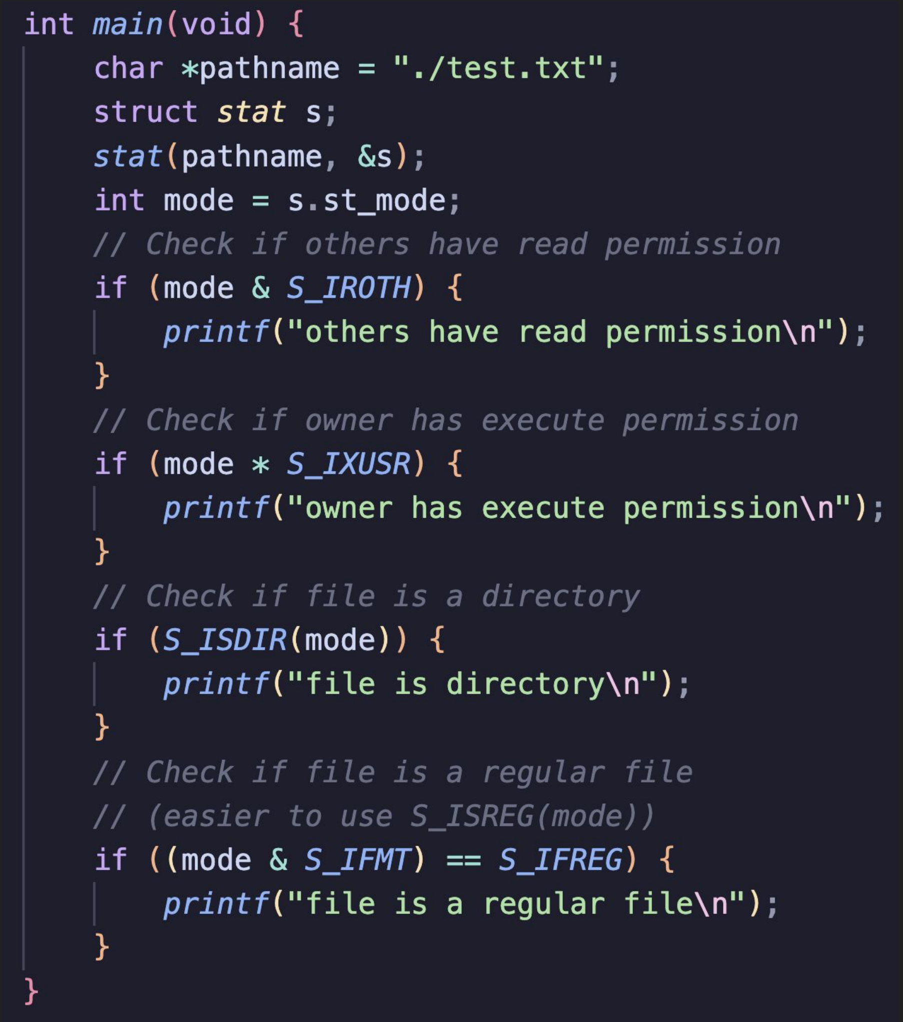
Reading file permissions

Use the defines in `man inode` to read the file mode.

S_I[R/W/X][USR/GRP/OTH]

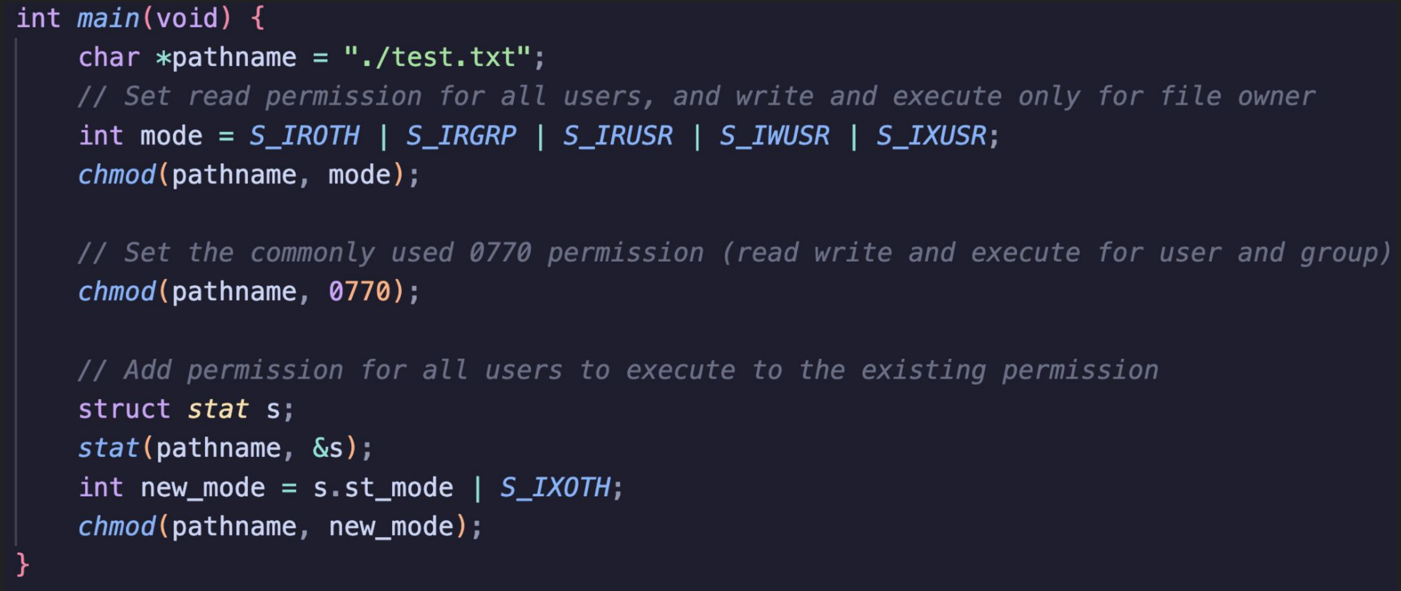
And use macros in same man page to check file type

S_IS[REG/DIR/…]



```c
int main(void) {
    char *pathname = "./test.txt";
    struct stat s;
    stat(pathname, &s);
    int mode = s.st_mode;
    // Check if others have read permission
    if (mode & S_IROTH) {
        printf("others have read permission\n");
    }
    // Check if owner has execute permission
    if (mode * S_IXUSR) {
        printf("owner has execute permission\n");
    }
    // Check if file is a directory
    if (S_ISDIR(mode)) {
        printf("file is directory\n");
    }
    // Check if file is a regular file
    // (easier to use S_ISREG(mode))
    if ((mode & S_IFMT) == S_IFREG) {
        printf("file is a regular file\n");
    }
}
```

Modifying file permissions



```c
int main(void) {
    char *pathname = "./test.txt";
    // Set read permission for all users, and write and execute only for file owner
    int mode = S_IROTH | S_IRGRP | S_IRUSR | S_IWUSR | S_IXUSR;
    chmod(pathname, mode);

    // Set the commonly used 0770 permission (read write and execute for user and group)
    chmod(pathname, 0770);

    // Add permission for all users to execute to the existing permission
    struct stat s;
    stat(pathname, &s);
    int new_mode = s.st_mode | S_IXOTH;
    chmod(pathname, new_mode);
}
```

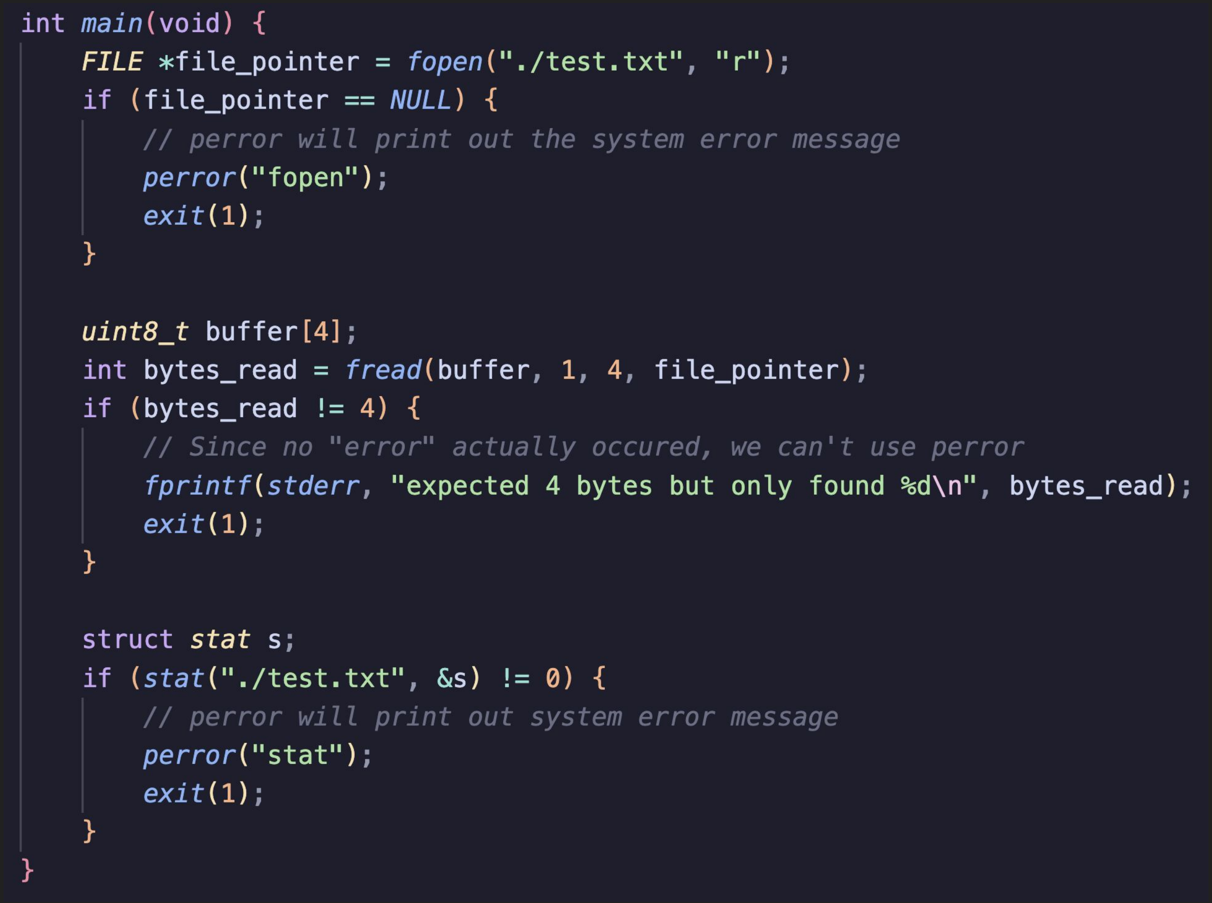
Use chmod. If you need to add/modify permission, it’s often useful to stat first and add permissions using bitwise OR.

Handling errors

Many low level functions can fail for a lot of reasons.

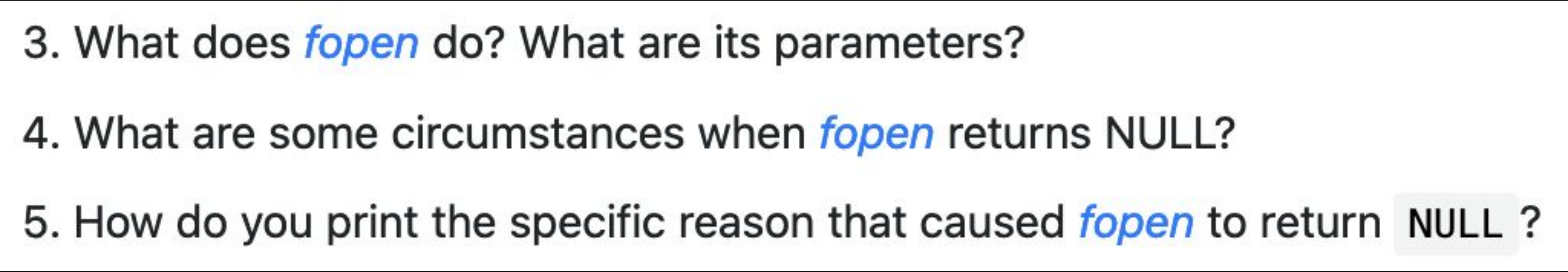
Functions that fail will set a system variable called `errno`. We can use perror to print a meaningful error message based on errno.

Only use perror if the failure would actually cause an error



```c
int main(void) {
    FILE *file_pointer = fopen("./test.txt", "r");
    if (file_pointer == NULL) {
        // perror will print out the system error message
        perror("fopen");
        exit(1);
    }

    uint8_t buffer[4];
    int bytes_read = fread(buffer, 1, 4, file_pointer);
    if (bytes_read != 4) {
        // Since no "error" actually occured, we can't use perror
        fprintf(stderr, "expected 4 bytes but only found %d\n", bytes_read);
        exit(1);
    }

    struct stat s;
    if (stat("./test.txt", &s) != 0) {
        // perror will print out system error message
        perror("stat");
        exit(1);
    }
}
```

Tutorial Questions 3,4,5

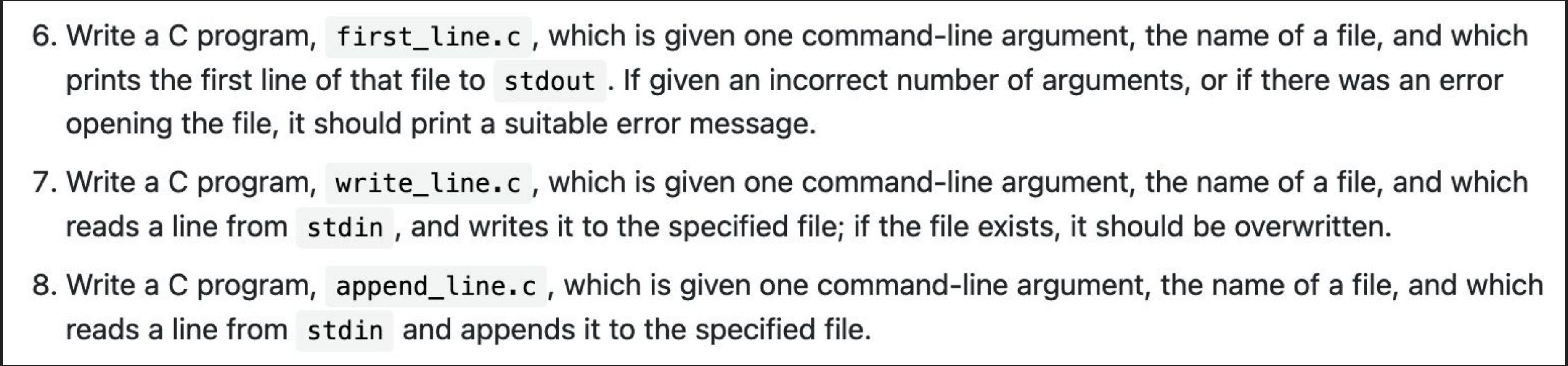


3. What does *fopen* do? What are its parameters?

4. What are some circumstances when *fopen* returns NULL?

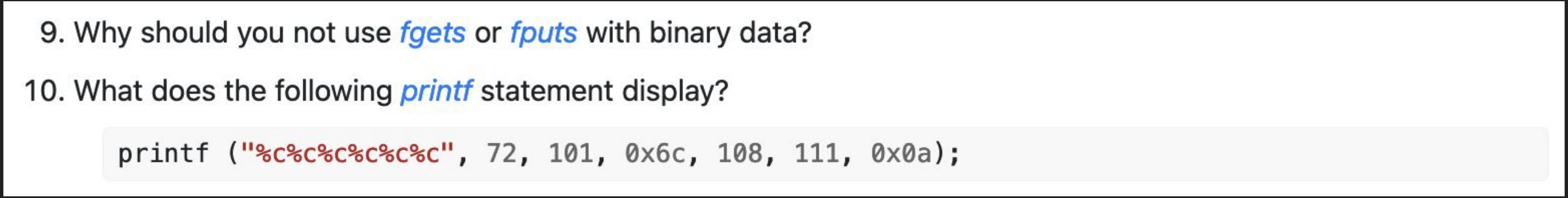
5. How do you print the specific reason that caused *fopen* to return `NULL` ?

Tutorial Questions 6,7,8



6. Write a C program, `first_line.c`, which is given one command-line argument, the name of a file, and which prints the first line of that file to `stdout`. If given an incorrect number of arguments, or if there was an error opening the file, it should print a suitable error message.
7. Write a C program, `write_line.c`, which is given one command-line argument, the name of a file, and which reads a line from `stdin`, and writes it to the specified file; if the file exists, it should be overwritten.
8. Write a C program, `append_line.c`, which is given one command-line argument, the name of a file, and which reads a line from `stdin` and appends it to the specified file.

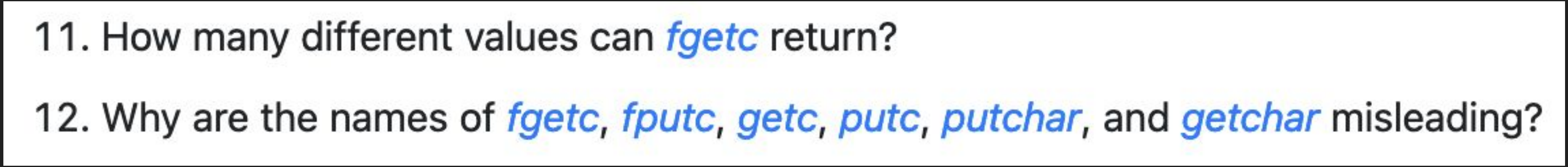
Tutorial Questions 9,10,11, 12



9. Why should you not use *fgets* or *fputs* with binary data?

10. What does the following *printf* statement display?

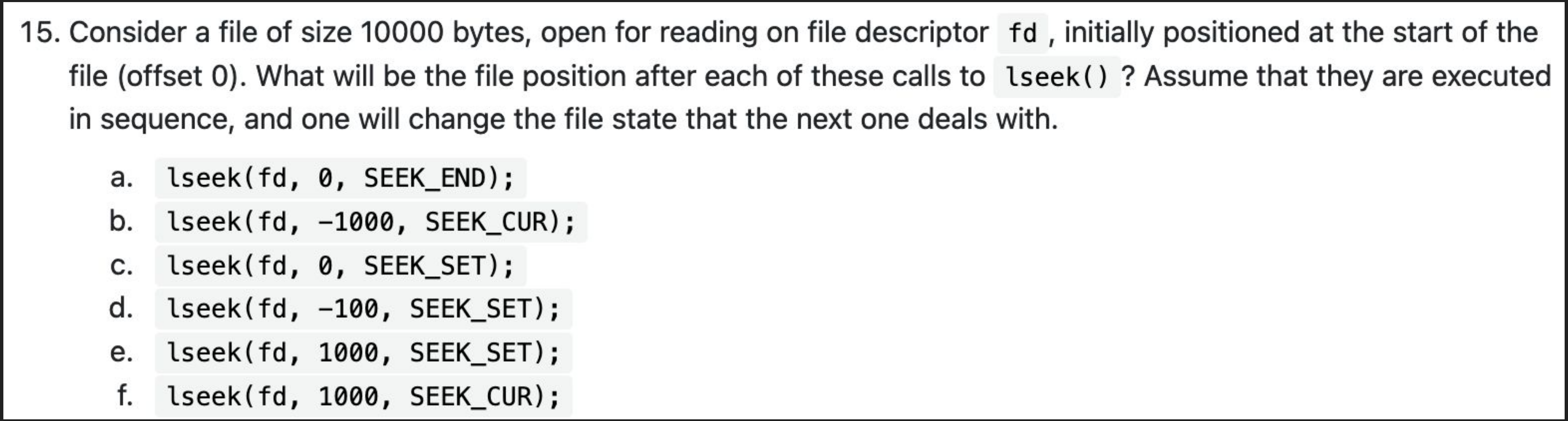
```
printf ("%c%c%c%c%c%c", 72, 101, 0x6c, 108, 111, 0x0a);
```



11. How many different values can *fgetc* return?

12. Why are the names of *fgetc*, *fputc*, *getc*, *putc*, *putchar*, and *getchar* misleading?

Tutorial Q15



15. Consider a file of size 10000 bytes, open for reading on file descriptor `fd`, initially positioned at the start of the file (offset 0). What will be the file position after each of these calls to `lseek()`? Assume that they are executed in sequence, and one will change the file state that the next one deals with.

    a. `lseek(fd, 0, SEEK_END);`
    b. `lseek(fd, -1000, SEEK_CUR);`
    c. `lseek(fd, 0, SEEK_SET);`
    d. `lseek(fd, -100, SEEK_SET);`
    e. `lseek(fd, 1000, SEEK_SET);`
    f. `lseek(fd, 1000, SEEK_CUR);`