# Week 09 Workshop

EXERCISE:
## Workshop 9 - Safety Shmafety

PRE-WORKSHOP PREPARATION:

You should watch the lectures, and review the answers to your theoretical lab questions.

IN-WORKSHOP REVISION:

This week, we'll be looking at some unsafe code. Before the workshop, we'll have a quick refresher on the following topics:

- Safe abstractions over unsafe code
- The `unsafe` keyword
- `*const T` / `*mut T`

## Discussion

First, we will do a short code review. This will take the form of choosing an exercise (probably `channels!`), and having everyone pull up their code. Much like in a pull-request, we will get students to swap computers and leave comments on things they like, don't like, or have questions about. Then, in a larger group, we will look through some of the interesting code and have a group discussion.

Second, we will have a discussion on the theoretical answers given to questions in the exercises discussed below. This will be the main opportunity for tutors to give feedback on that theoretical work. If you cannot make this weeks workshop, you will still get marks for submitting the theoretical work, but you will not get feedback (unless you ask us seperately). If you can turn up, you will get detailed feedback on those answers.

- Send and Sync
- optionally, Mitochondria (Investigating Cell)

## Week 9 Code

In groups, your task for this week is to build a `File` struct, using primitives from the `libc` crate. These use functions which are common in the C language, but which Rust does not use.

You should implement the following on the starter code:

1. Opening a file to read with the `read` function.
2. Reading a string a file with `read`.
3. Reading a type from a file with `read_i64`.
4. Reading a type from a file with `read_f64`.
5. Reading a type from a file with `read_char`.
6. When your file goes out of scope, close it automatically.

To do this, you will need to know about the following functions/enums/traits, mostly from `libc`:

- The FILE enum represents an open file.
- fopen opens a file (called `filename`). To open in "read" mode, you should use `"r"` as the mode. This returns a (possibly null) file-pointer, which represents an open file.
- fgets reads a whole line from a file. `buf` is a pointer to memory, which must be at least `n` bytes big. `stream` is a file-pointer. Note that fgets returns null if opening the file failed.
- fscanf reads a different type from the file `stream`, depending on `format`:
  - If the string is `"%d"`, the third argument to `fscanf` should be a `&mut libc::c_int`.
  - If the string is `" %c"` (note the leading space), the third argument to `fscanf` should be a `&mut libc::c_char`.
  - If the string is `"%lf"`, the third argument to `fscanf` should be a `&mut libc::c_double`.
- fclose should close the file pointer given to it. The Drop trait may be useful here.
- There are a variety of types ( `c_int`, `c_char`, `c_double`) which correspond to types from C. You will need these.

You should assume this code will only be run on unix-like systems, and that all paths will be ascii.

At the end of the activity, if there's time, look into the errno crate. This allows you to give the user more information about why an operation failed.

Afterwards, you might like to discuss the following points:

- Could your type be `Copy`, `Clone`, `Send` or `Sync`?
- Did you have to think about memory safety at all? Could you modify your code to cause a dangling pointer?
- How easy was interoperating with C functions?

---