Ownership and lifetimes

How Rust's unique features will help us develop a stable, fast and multi-threaded desktop app

PLM - Paradigm analysis report

Table of Contents

Ownership and lifetimes 1
Project needs 1
Concurrent programming basics 2
Memory allocation basics 2
Why not just C++ or Java ? 3
Why memory safety is a big deal ? 4
Performance + Memory safety: the best of both world 4
Why is it possible to get both ? 4
The concept of ownership 5
The concept of lifetime 6
Why we don't need a garbage collector nor manual memory management ? 6
Fearless concurrency 8
The cost of the borrow-checker 9
Specifications for DME 10
Functional goals 10
Non-functional goals 10
Applying the paradigm on DME 11
Sources 11

Project needs

Before discussing this paradigm, let's briefly recall what DME (Delightful Markdown Experience) desktop app needs. As a Markdown previewer on steroids, we need to develop a optimized program to bring a great browsing experience. To achieve maximum speed, we need to implement multi-threading to the maximum, making all IOs tasks in separated threads to avoid waiting on hardware when we could move forward with computation.

Searching for Markdown files on the disk, reading their content, indexing the headings, and building a research index, need to be parallelized between several threads to accelerate the first application start. In addition to building this index, we want to have a very fast rendering on the displayed Markdown document. Generating the highlighted code snippets can take a bit of time considering we'll use Tree-Sitter to have top-quality tokenization, that's another work to be distributed among several threads.

In addition, we want to avoid crashing the UI in case binary or unsupported content is loaded in the application. GUI doesn't offer as many logs and feedback as CLI application.

Finally, as the app is going to be open for hours, like a PDF previewer or a web browser, we cannot tolerate memory leaks as it would slowly but surely eat all the available RAM…

Concurrent programming basics

Basic applications do not use the full power of modern processors when they only have one thread of execution. Having multiple CPU cores at disposition enables big performance gains by enabling parallel execution of calculation tasks or doing task distribution to separate UIs and background processing. When we start doing concurrent programming, managing several threads of execution comes with major challenges.

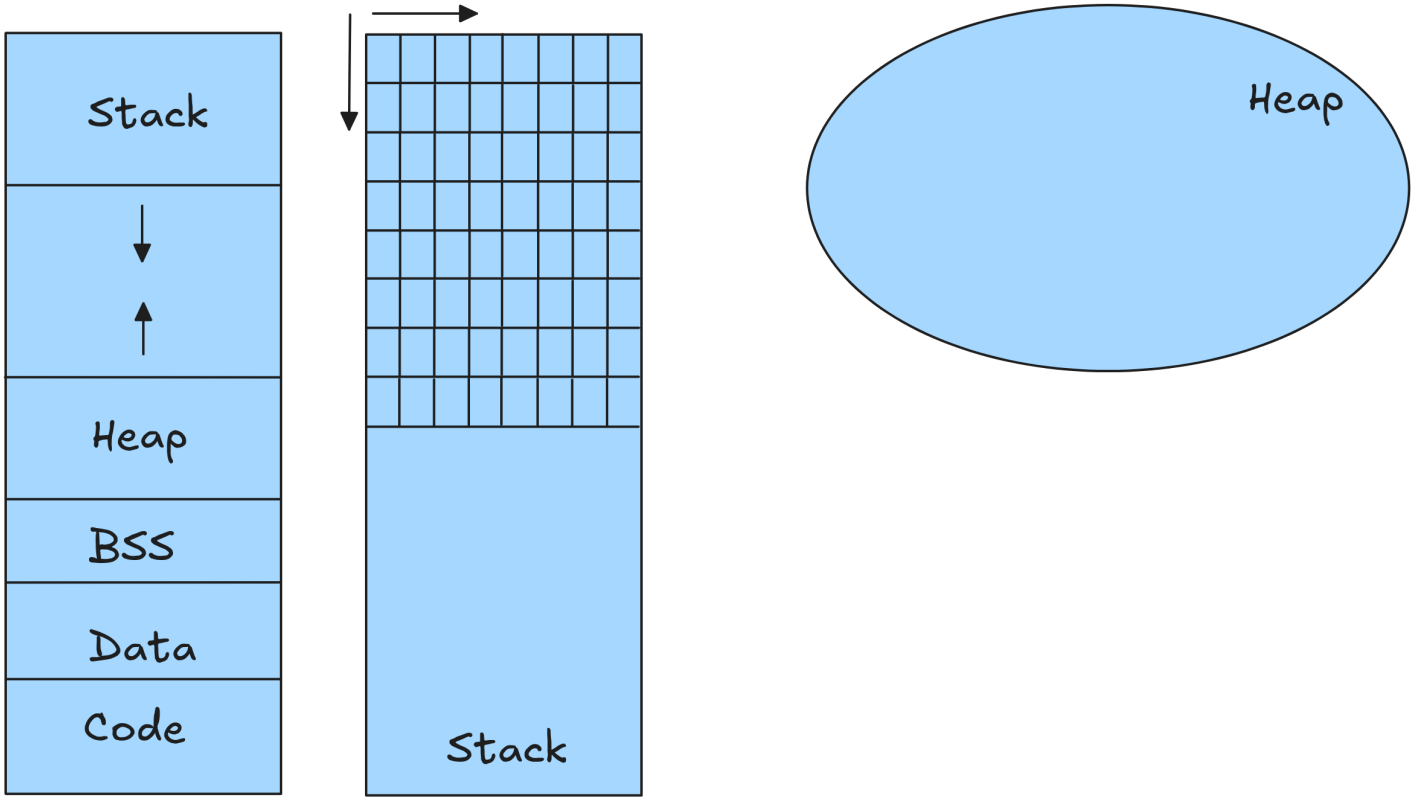
To understand them, let's imagine a person responsible for creating decorations for a Christmas tree.

A classic program would be, a person working first on cutting the paper for the decoration, then fold it and hang it on the tree. Rince and repeat. In concurrent programming, we would split the work with your family. Asking 3 other people will help you finish the work faster. You work on your decoration, then hang it on the tree and everyone else do the same thing. Simple, no ? Sadly, you only have one scissor, and only one of you can hang decoration on tree at a time. In this metaphor, the scissor would be a shared resource, and hanging decorations on the tree is a critical section. How you going to manage this scissor without cutting the fingers of your mum when everyone want to use it ? We need protection mechanisms to make sure the scissor is used safely and the usage is interrupted by someone else as it could do damage to the tree or the fingers…

The root issue is that we don't control the order of execution, as the OS scheduler is the master in this situation. Standard solutions to control access to shared memory are mutex and semaphores but they can also be misused.

Memory allocation basics

We generally represent the virtual memory of a process, with this kind of diagram. At left, the different regions dedicated to store the loaded code, the static variables and constants in BSS and Data, and finally the stack and the heap.



Given this piece of C code, we find numerous allocations in several memory locations.

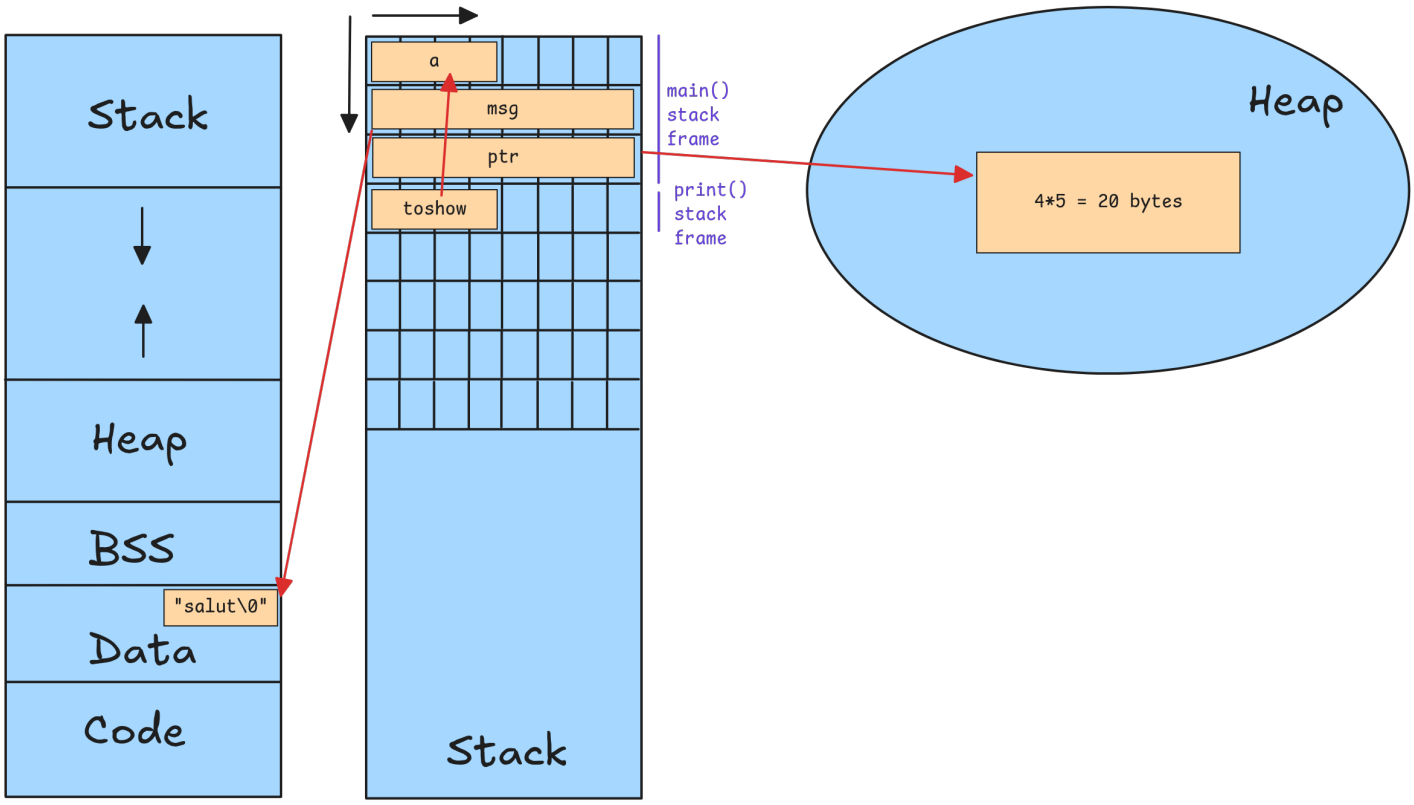
```c
#define SIZE 5

void print(int *toshow) {
    printf("%d", *toshow);
}

int main(void) {
    int a = 23;
    char *msg = "salut";
    char *ptr = malloc(sizeof(int) * SIZE);
    print(&a);
    free(ptr);
    return 0;
}
```

If we paused the program on `free`, the memory content would look like this. `a`, `msg`, `ptr` and `toshow` are local variables stored on the task. We have 3 pointers:

- `msg` towards read-only data section pointing on `salut\0`
- `ptr` the dynamically allocated zone on the heap via `malloc`
- `toshow` pointing to the other variable `a` on the stack as we gave its address when calling the function



In C, we manage memory ourselves, hence the `malloc` and `free`. We can access and manage raw pointers ourselves, and that's the core of memory unsafety because this is error-prone. If we want to avoid manual memory management, the question then is "but how do you know when and how to release memory ?? Is it really possible to determine that at compile time ?".

Why not just C++ or Java ?

"You want performance for a desktop app, that's would be easy to build a C++ desktop app with Qt no ?"

C++ would be a good option is terms of performance and object-oriented paradigms to manage and index the Markdown files. But there is an issue regarding concurrency. As we learned in the PCO course (Programmation Concurrente), we can spend hours reading small chunks of code managing mutex and semaphors to make sure it is **correct** in terms of safety. We spent a lot of time checking and reviewing our own code and still failing to get everything right, sometimes with complicated deadlock hard to detect at first sight.

The problem here is that the G++ cannot verify we are doing things correctly, as long as types are handled correctly, it can compile and the developer might detect nefarious bugs only in production. It's so easy to forget to protect a shared state, or associate a mutex in your head with 2 variables and forget a third one you just added.

For hot sections, where speed is key, when using low level functions from C, we regularly take the risk of forgetting to free heap allocated memory, leading to memory leaks.

"You want to avoid memory safety issues ? Stop managing memory yourself and use Java !"

Java would be good in terms of safety, as all memory bugs almost disappear as we don't manage memory ourselves and we cannot access raw pointers. As the JVM can generate exceptions, they can be catched to avoid a global crash, we can avoid most memory issues. This magic is coming from the garbage collector, regularly checking if our program has lost access so heap allocated data, to clean it itself. This is an annoying overhead that will not prevent us from reaching maximum speed and avoid using extra RAMs.

Using Java also comes at a cost of executing our program in a virtual machine, instead of running our code directly on the CPU. The cold-start is known to be slow compared to native program. Finally, the concurrency issues do not go away, that's totally possible to use a `ArrayList` which is not thread-safe, instead of a safe equivalent (like `CopyOnWriteArrayList`).

Why memory safety is a big deal ?

In a Microsoft presentation from 2019, we find that "~70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues". The Chromium projects also reports that "Around 70% of our high severity security bugs are memory unsafety problems (that is, mistakes with C/C++ pointers). Half of those are use-after-free bugs.".

To only cite a few, memory issues are use-after-free, buffer overflow, memory leaks, data race, … They can cause big security issue as seen above, and cause app crashes, segmentation faults or data corruption.

Performance + Memory safety: the best of both world

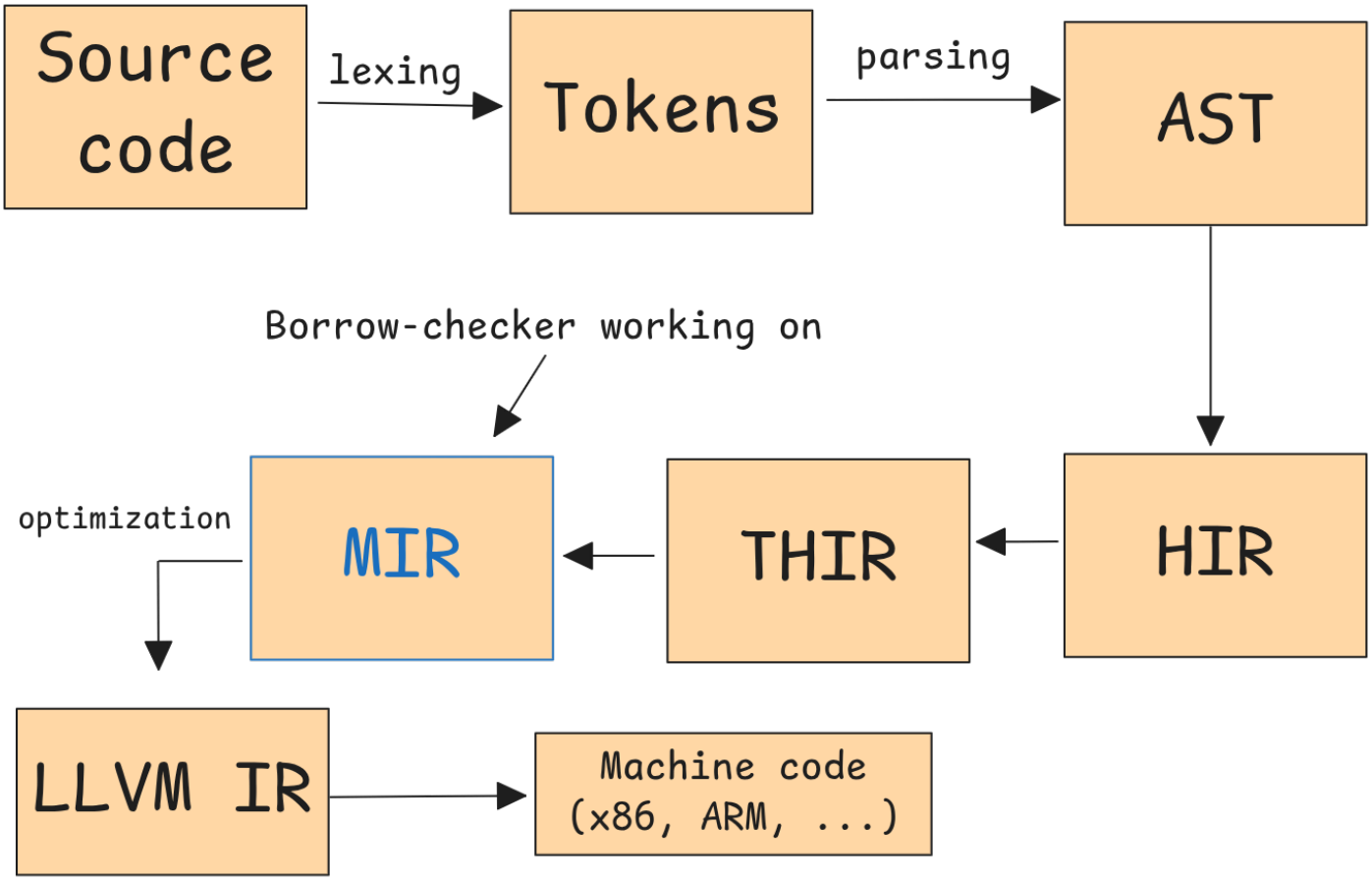
Rust is a strongly typed and compiled language, its strongest selling point is being the first language to bring the combination of speed and memory safety at the same time. It was designed for systems programming in mind (browsers, kernels, …) by a developer at Mozilla, released 1.0 in 2015, but now has reached almost all programming areas: mobile, embedded programming, desktop apps, GUIs, games, low level libraries, even web frontend via WebAssembly ! It's advanced static analysis at compilation time also removes some of the memory checks are runtime (such as pointer nullness checks).

Why is it possible to get both ?

It doesn't use a garbage collector and doesn't ask the programmer to manually manage the memory. But how it is even possible ? How the program knows when to free heap allocated memory ?

The Rust compiler `rustc` implements a new paradigm, including the notion of ownership and lifetimes, checked by a part of the compiler called the **borrow-checker**. Instead of associating only a type and a variable to a resource, like most modern languages, it also tracks who has the ownership of this resource and how long the resource must exist. When the variable is the owner of a resource, the resource will be deallocated when the variable goes out of scope.

What are the steps of the Rust compiler ?



We can see the first standard steps (source code, tokens, AST) we have seen in PLP course, but we have several **Intermediate Representation** (IR), a **HIR** (High-level IR), **THIR** (Typed High-level IR), the **MIR** (Mid-level IR) and finally the target the Rust compiler the **LLVM IR**. LLVM is another big project serving as the backend for many compilers (including GHC), enabling to target many platforms and architectures, to finally get machine code in a binary file. The borrow-checker is reasoning on the MIR, we'll see an example below.

The concept of ownership

The first main concept of **ownership** is that in addition to a type and variable, each resource has an **owner**. This owner is responsible for the variable during its lifetime and the memory management. But the owner is not the only one that need to access, use or modify the associated resource. It has to share with certain conditions with other variables.

This would mean we can have a variable, scissor that we can share.

There are multiple ways to share a variable in Rust. The simple way is to move it. A simple assignment is enough to generate a move. Moving means changing the owner. The scissor are now under the responsability of `sam`. Because it was moved, `patrick` cannot use it anymore.

```
let patrick  = "scissor";
let sam = patrick;
```

Memory wise, `patrick` was responsible for the block of memory containing the scissor. Instead of moving it, we can instead lend it. In that case we use the `&` to take a reference of `patrick`:

```
let patrick  = "scissor";
let sam = &patrick;
```

Here `sam` asked for a reference, which implies a borrow instead of a move. The owner is accepting temporary access from `sam` in read-only. Memory-wise it would mean that the `sam` has an pointer to the memory block containing "scissor", this is just a pointer, but as pointer in C++ they are always valid (can be dereferenced at any time without any issue) and therefore cannot be null.

But what if the borrower wants to change the value ?

This borrow can also be mutable, meaning `sam` is allowed to modify it. This piece of code will fail to compile with an error on line 3 “cannot borrow `*sam` as mutable, as it is behind a `&` reference”, indicating that `push_str` (method to append chars at the end) cannot borrow the resource as mutable as this is an immutable reference !

```
let patrick = String::from("scissor");
let sam = &patrick;
sam.push_str("s")
```

By adding the keyword `mut` after the `&` reference symbol, we describe taking a mutable reference. This also means we need to have a mutable object at the start, adding a `mut` in the first line `let mut patrick` becomes also necessary.

```
let mut patrick = String::from("scissor");
let sam = &mut patrick;
sam.push_str("s")
```

Now, we know how the ownership, moves and borrows work. Remember that the owner is responsible for the memory, meaning also dropping (freeing) it. In this case, the String struct implement the `Drop` trait with a `drop` method freeing the dynamic memory containing the variable-length vector of chars.

In brief, the rules enforced by the borrow-checker are
1. Only one owner per resource
2. Only one mutable reference at a time
3. Or several immutable references, but not both
4. References must always be valid

The concept of lifetime

To decorate our Christmas tree, we decided to use candles. While a it is a really nice decoration, a candle end up burning out completely after a while. It reached it end of its lifetime. Afterwards, we would need to clean base, and afterwards reuse it. Program do the same, they allocated memory to a variable, like our base for our candle, then when this variable reached it end of its lifetime, the memory is freed so that the program can use it for something else, like reusing our base.

Most program use the scope to determine when a variable reached its end of its lifetime. Sometimes, a simple scope is not enough, so we need to extend the default lifetime of the variable. As a programmer you know when your variable is not needed any longer, like how you know how long the candle will last. Because it’s you who bought it, and you are its owner.

The goal to define the lifetime is to know when the candle is burnt out or the variable is unused. That way, the program can use the memory again. Instead of cleaning ourselves, we let the program do it, but we indicate to him when he can cleanup if it is unclear for him

Why we don’t need a garbage collector nor manual memory management ?

Compiler first define the owner of each variable. That way, the owner will be able to drop the variable when it reached the end of its lifetime, essentially freeing the memory allocated to for the variable. The owner can change via `move` but at the end we will only have one.

Behind the scene, when we allocate memory on the heap, the compiler will also determine when this memory will be freed, adding in the source code the drop instruction when the associated pointer lifetime ends. While it is not directly visible, displaying the code generated by the compiler allow us to see hidden instructions.

This piece of code just stores the value `2` on the heap (the smart pointer `Box` is essentially a forced dynamic allocation), and then move the ownership of this pointer to another variable `_a`, making usage of `m` afterwards unauthorized.

```
fn main() {
    let m = Box::new(2);
```

```
    let _a  = m;
}
```

If we look at the code display of the internal representation MIR (Mid-level Intermediate Representation), by running `rustc -Z unpretty="mir" src/main.rs` with the nightly compiler, we get this:

```
fn main() -> () {
    let mut _0: ();
    let _1: std::boxed::Box<i32>;
    scope 1 {
        debug m => _1;
        let _2: std::boxed::Box<i32>;
        scope 2 {
            debug _a => _2;
        }
    }

    bb0: {
        _1 = Box::<i32>::new(const 2_i32) -> [return: bb1, unwind continue];
    }

    bb1: {
        _2 = move _1;
        drop(_2) -> [return: bb2, unwind continue];
    }

    bb2: {
        return;
    }
}
```

We see the `m` variable used as `_1` and `_a` variable used as `_2`. We can see in `bb1` makes the `move` explicit, then there is an added instruction drop, calling the destructor on `_2` (`_a`) as it's the new owner and there is thus only one drop!

Fearless concurrency

Ownership and lifetimes are actually linked to how concurrency has been designed in Rust, even concurrency mechanism this is not necessarily handled by the borrow-checker.

The concurrency is also linked to smart pointers, we'll only touch a few here to show how far the safety has been pushed in the design of the standard library as well. What you need to know in short:

- There are 2 ways to communicate between threads:
  1. **with channels** like Go of various types such as `mpsc` (multiple producer, single consumer) allowing several threads to send something on a channel, but with **only one thread allowing to receive the messages**.
  2. **with shared memory** wrapped in `Mutex` or with special atomic types. Only types marked with the trait (interface in Rust) `Send` can be moved between from one to another thread. The `Sync` trait is used to mark types that can be synchronized with threads. Any non thread-safe type doesn't have the `Send` trait. These 2 traits are just marker, they don't ask to implement any method (that's similar to the `Cloneable` interface in Java).
- We cannot just send normal references between threads, because we don't have the static warrantees that the referenced object will live long enough for the reference to be always valid! If the owning thread finishes before another that has a reference, we could risk unallocated memory access! We need smart pointers for that such as `Arc`.

In C++ in the PCO course, it was so easy to forget a mutex around a shared state as the compiler cannot warn us. Given an object of this `MegaCounter` class,

```cpp
class MegaCounter {
protected:
    int some_counter;

public:
    void save(int counter) {
        some_counter = counter;
    }
    int get() {
        return some_counter;
    }
};
```

In Rust, the data can be wrapped in a Mutex, which implement `Sync`, having a `MegaCounter` again here.

```rust
struct MegaCounter {
    some_counter: Mutex<i32>,
}
```

The `increment` implementation shows us that we have forced to call `lock` to get a `MutexGuard` to finally allow us to access the counter. The unlock of the mutex is done automatically in the `Drop` implementation on the `MutexGuard`, this will be run at the end of the block.

```rust
impl MegaCounter {
    fn new() -> Self { MegaCounter { some_counter: Mutex::new(0) } }

    fn increment(&self, add: i32) {
        // guard: MutexGuard<i32>
        let mut guard = self.some_counter.lock().unwrap();
        // mutable dereference to i32 via DerefMut trait
        *guard += add;
        // drop(guard);
    }

    fn get(&self) -> i32 {
      *self.some_counter.lock().unwrap()
    }
}
```

If we want to send a MegaCounter across threads, we need to use `Arc` (Atomically Reference Counter), it is like a `shared_ptr<T>` in C++, but the internal counter of references is managed atomically. The non atomic variant `Rc` is not `Send` and cannot be used here.

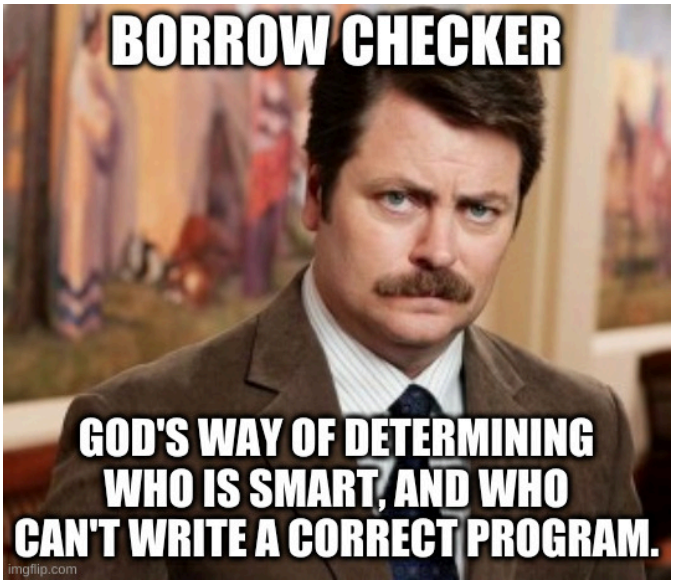
```
fn main() {
    let counter = Arc::new(MegaCounter::new());
    for i in 0..10 {
        let arc = counter.clone();
        thread::spawn(move || {
            arc.increment(i);
        });
    }
}
```

Rust is safe in many places but cannot garantee that deadlock can happen, if we call `lock` twice without the end of the block or without an explicit `drop` call, the thread will be stucked.

The cost of the borrow-checker

The numerous benefits we get with the borrow-checker, including the zero cost at runtime, also come at a cost elsewhere:

1. There is **learning curve steeper than other languages**, this is commonly referred as "fighting the borrow-checker", if you never worked with memory management before it will take time to get into it and start thinking about lifetimes and memory accesses yourself. Some developers might have never thought about these important details before and might be confused in the beginning.



Meme from Reddit

2. Some approaches like building graph data structures inherently have cross references, as we cannot have mutable references and immutable references at the same time, it means we could never change the graph content. As **the borrow-checker is conservative** it prefers to reject correct programs instead of accepting wrong code, if one the rule is not respected it will not compile.
   There is an escape hatch with the `unsafe` keyword is that access some unsafe operations that need careful consideration, but it doesn't disable the other rest of the compiler checks. We can use `unsafe` blocks ourselves or use wrapper types of the standard library that maintain `unsafe` sections for us. As the hardware is inherently unsafe, if when we want to write at a precise memory address, we need to manage some memory ourselves. There is obviously the possibility to do errors in this unsafe code and create memory issues, but the surface area to check is far smaller that the entire codebase in a C codebase.
3. **Rust is not the fastest way to build prototypes**, as it forces us to write safe code, before creating a binary. This issue can be partially mitigated with the "Rust easy mode", where you can clone everything with `.clone()` instead of using references, where you call `.unwrap()` on all `Result` or `Option` just to make it work: "it should not be any errors, and just crash if it's not the case".

Specifications for DME

Here are the specifications of features we'll develop along the following weeks. We will work with Tree-Sitter and Tauri.

Functional goals

Preview

1. We can preview any Markdown file with DME, by double-clicking on any `.md` file or running `dme test.md` in a GUI desktop app. Images and tables are supported.
2. The preview of code is done via Tree-Sitter: the syntaxes for C, C++, Java, Rust, Bash are built into DME (other languages are not supported for the start)
3. When the loaded file is changed on disk, the preview must refresh itself

PDF export

1. It's possible to export a PDF file with the same look as the preview via `dme export test.md`, including highlighted code with Tree-Sitter
2. It's possible to export several files at the same time just by giving more files like this `dme export test.md resume.md sample.md`

Research

1. It's possible to filter the list of Markdown with simple regexes in a configuration file in TOML, to avoid indexing hundreds of files unnecessary (i.e. when we cloned docs repository)
2. There is a way to quickly search among all Markdown files present on the disk, the search is matching the path fuzzily or matching terms inside headings
3. Finding a heading matching some keywords and choosing it will open the file as the current preview and jump to the matched heading
4. The research results are reloaded on every keypress, unlike a search engine
5. All documents under the user home directory can be found, except those present in folder starting with a dot (ignored `~/.config` i.e.)

Non-functional goals

All time measures must be made on Samuel's machine with a 12 cores processor and 16GB of RAM, with at least 5GB of unused RAM.

Preview

1. The preview of a 10 pages documents, with 50 pieces of code in different languages, must load under **300ms**
2. The refresh duration between when the file is saved and when the preview is updated, must be under **500ms**

PDF export

1. The PDF export of multiple files should be handled in parallel
2. The PDF export of a 10 pages document must take under 5s at maximum
3. The PDF export of 5 documents should not take more than 10s

Research

1. Building the full index of all headings of MDN content (Mozilla Developers Network documentation) should take less than **20s** (MDN GitHub repository, containing 13000 `.md` files with 95000 headings)
2. Searching for "Array constructor with a single parameter" (found on this page) by copy-paste must take less than 500ms to find the page on `Array() constructor`, showing the section `Array constructor with a single parameter`.
3. The partial search of "Array constructor single" should also list in the result the same section mentioned in the previous point

Applying the paradigm on DME

Keeping in the safe Rust subset (not using any `unsafe`), we'll be "forced" to follow of the ownership and lifetimes principles enforced by the borrow-checker. We will implement the 3 bigs features concurrently, to maximize the speed of each part.

Sources

Our work is mainly based on our experience, reading several articles, documentations and watching videos. Here is what were the most useful to us in our research

- The Rust book - https://doc.rust-lang.org/book/ - mostly chapters 4 + 10.3 + 14 + 15
- The friendly and contextual error messages, once we learned the basic vocabulary, they really help to understand why the borrow-checker is not happy
- Rust book experiment - https://rust-book.cs.brown.edu/ - mostly chapter 4 with better explanations and visualisations of memory, ownership and lifetimes
- How Rust went from a side project to the world's most-loved programming language
- The Rust Borrow Checker - A Deep Dive - Nell Shamrell-Harrington, Microsoft - Conference on Youtube
- How the Rust Compiler Works, a Deep Dive - RareSkills conf
- Microsoft presentation from 2019 on memory safety CVE
- The Chromium Project - safety issues measures