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

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 pretty 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, building a research index, is appropriate to sharing Markdown files across several threads to build this index as fast as possible on the first startup. 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 app as the whole UI will quit, creating a bad experience for the user, this could happen in case a strange Markdown file containing binary data was opened and the parser wasn't robust enough to support this unusual situation. It's not a like a CLI where if you get an error, you are used to run it again with other arguments, people are going to start it via the start menu and when it crashes, no logs will be immediately visible.

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 enable big performance gain by enabling parallel execution of calculation tasks or doing tasks repartition to separate UIs and background processing. When we start doing concurrent programming, managing several threads of execution come with major challenges.

To understand them, let's image a person responsible to create decorations for a Christmas tree.

A classic program would be, a person working first on cutting the paper for the décoration, then fold it and hang on the tree. Rince and repeat. In conccurent programing, we would split the work. For example split it in four, so asking people 3 other people to help you. You each work on your decoration, then hang on the tree. SImple, no ?

Sadly, you only have one scissor, and only one of you can hang decoration on tree at a time. The scissor would be a shared ressource, and hanging decoration a tree a critical section.

Conccurent program always tried to address those issue. Mupltiple people hanging decoration on the tree could be dangerous, while there is only one scissor so the other have to wait on it.

A simple way to resolve this issue is using what we call a mutex. Mutex is like a lock on something, allowing only one person at a time working on it.

Memory allocation basics

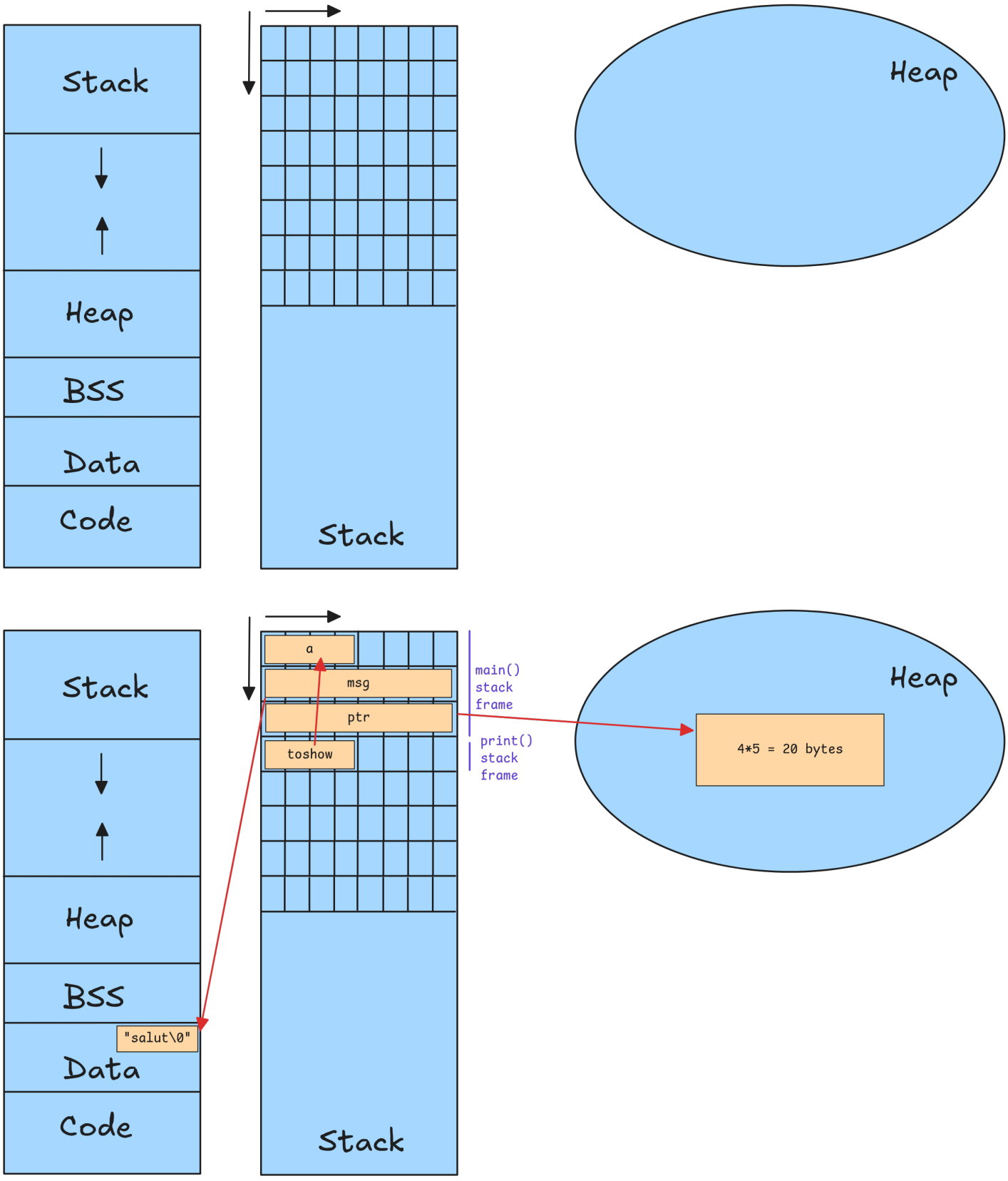
Give this simplified piece of C code, we find numerous allocation.

```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;
}
```

TODO explain quickly and explain why memory management is hard.



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 a big issue regarding to concurrency. As we learned in the PCO course (Programmation Concurrente), we can spend hours reading small chunks of code managing mutexes 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 critical section where speed is really 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 ourself 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 reach 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. 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 causes big security issues 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 bringing the combination of speed and memory safety at the same time. It was designed for systems programming (browsers, kernels, …) but now has reached almost all programming areas, even web frontend via webassembly.

Why it is 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` implement 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 go out of scope.

the borrow checker

The concept of ownership

Now, we know what is an shared ressource. While the usage of a mutex it can be a bit heavy. In the end you could simple borrow the scissor then give it back when your finished.

Ownership is a programing concept introduced Each variable has an owner. This owner is responsible for the variable its lifetime and the memory management. This would mean we can have a variable, scissor that we can share.

There is mupltple way to share a variable in Rust. The simple way is to move it.

```
let owner_1  = "scissor";
let owner_2 = owner;
```

A simple assigment is enough to generate a move. Moving means changing the owner. The scissor are now under the responsibity of owner_2. Because it was moved, owner_1 cannot use it again. Memory wise,

owner_1 was responsible for the block of memory containtg the scissor. Instead of moving it, we can instead lent it. In that case:

```
let owner_1  = "scissor";
let owner_2 = &owner;
```

Here owner_2 asked for a reference, which imply a borrow instead of a move. Memory-wise it would mean that the owner_2 has an access to the memory block containg “scissor”

This borrow can be mutable or not, meanig owner_2 is allowed to modify it or not.

```
let mut owner_1  = "scissor";
let owner_2 = &owner;
//Is allowed because we said it's a mutable borrow
owner_2.append("s")
```

```
let mut owner_1  = "scissor";
let owner_2 = mut & owner;
//Raise a compilation error because it is not a mutable borrow
owner_2.append("s")
```

Now, we know how borrow works. Remember that the owner is responsible for the memory, meaning also dropping (freeing) it.

The concept of lifetime

To decorate our Chrismas tree, we decided to use candles. While a it is a really nice decoration, a candle end up burning out completly 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 dertermine 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 your 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 ourself, 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 ?

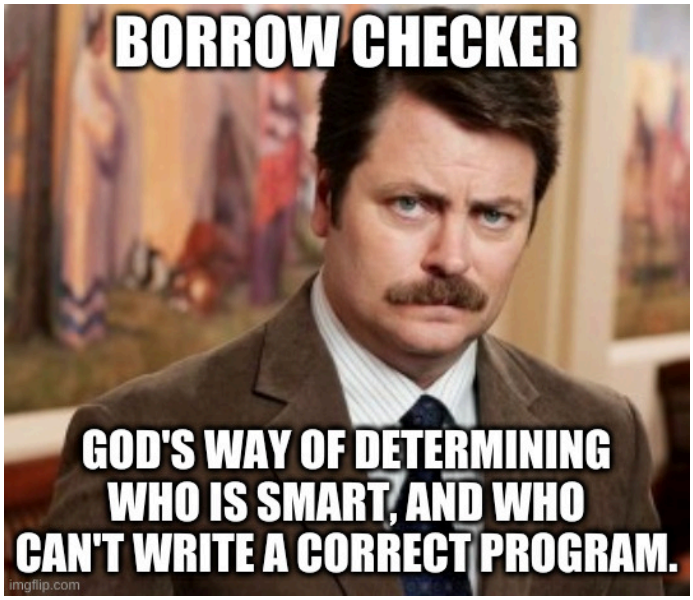
todo mieux

comment se passe une allocation dynamique, linteraction entre owner et lifetime. -> appel du drop

The cost of the borrow-checker

The numerous benefits we get with the borrow-checker also come at a cost:

- 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.



Meme from Reddit

- Some approaches like building graph data structures inherently have cross references, 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 ourself 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 ourself. 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.
- **Rust is not the fastest way to build prototypes**, as it forces us to write safe code, before creating 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".

gestion état partagé et communication inter threads

compilateur va utiliser les lifetimes pour savoir quand ya plus de ref le mutex et pour savoir quand libérer la mémoire.

auto unlock au drop du mutex

Specifications for DME

Here the specifications of features we'll develop along the following weeks.

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 document, 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 mentionned 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 conccurently, to maximize the speed of each part.