

`cargo new the-delightful-markdown-experience`

# Ownership and lifetimes

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

# HPC Lab 2 - Report

Author: Samuel Roland

## My code

Setup [Likwid](#) too. For this lab, I maintain both `CMakeLists.txt` and `xmake.lua`, but I use xmake generated binaries in my report.

## Via CMake

Setup the `fftw` library and `libsnd`, on Fedora here are the DNF packages

```
sudo dnf install fftw fftw-devel libsndfile-devel
```

Setup [Likwid](#) too.

Compile

```
cmake . -Bbuild && cmake --build build/ -j 8
```

And run the buffers variant

```
./build/dtmf_encdec_buffers decode trivial-alphabet.wav
```

Or run the fft variant

```
./build/dtmf_encdec_fft decode trivial-alphabet.wav
```

## 1. 3 main features

- Research
- Preview
- PDF export

## 2. Maximum of parallelisation

## 3. Stability and low memory footprint



TreeSitter



Tauri

Tree-Sitter generated HTML example

```
for (sf_count_t i = 0; i < TONE_SAMPLES_COUNT; i++) {  
    sum += fabs(audio_chunk[i] - reference_tone[i]);  
}  
  
return sum / TONE_SAMPLES_COUNT;
```

<pre>

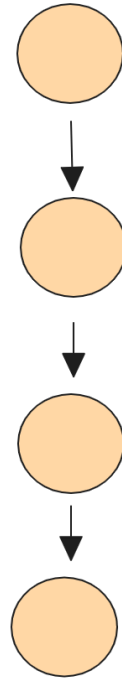
```
<code class="language-c">  
  <span class="keyword control repeat">for</span>  
  <span class="punctuation bracket">(</span>  
  <span class="type">sf_count_t</span>  
  <span class="variable">i</span>  
  <span class="operator">=</span>  
  <span class="constant numeric">0</span>  
  <span class="punctuation delimiter">;</span>  
  <span class="variable">i</span>
```

...

```
</code>
```

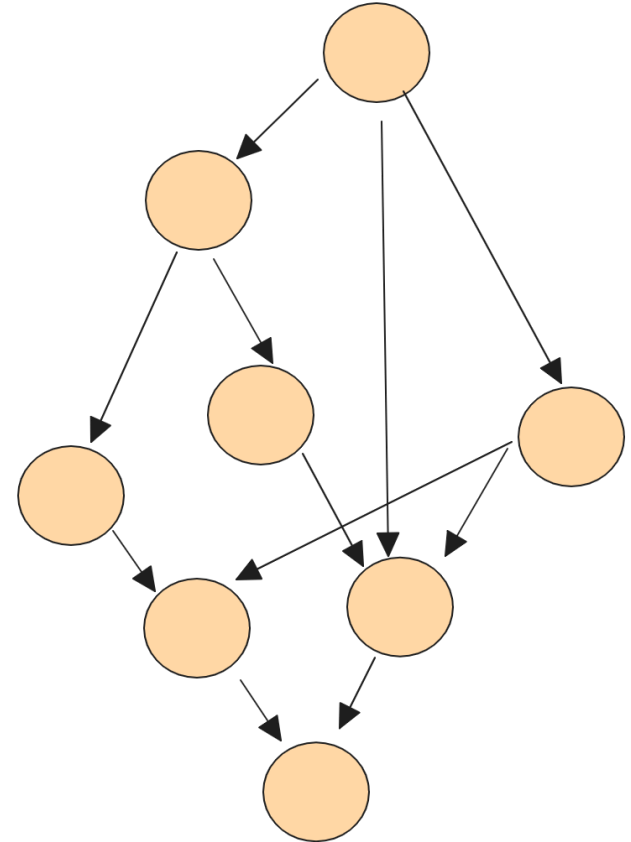
</pre>

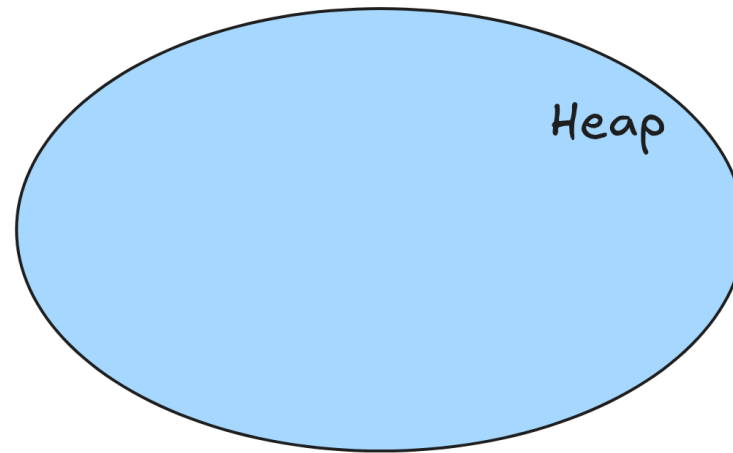
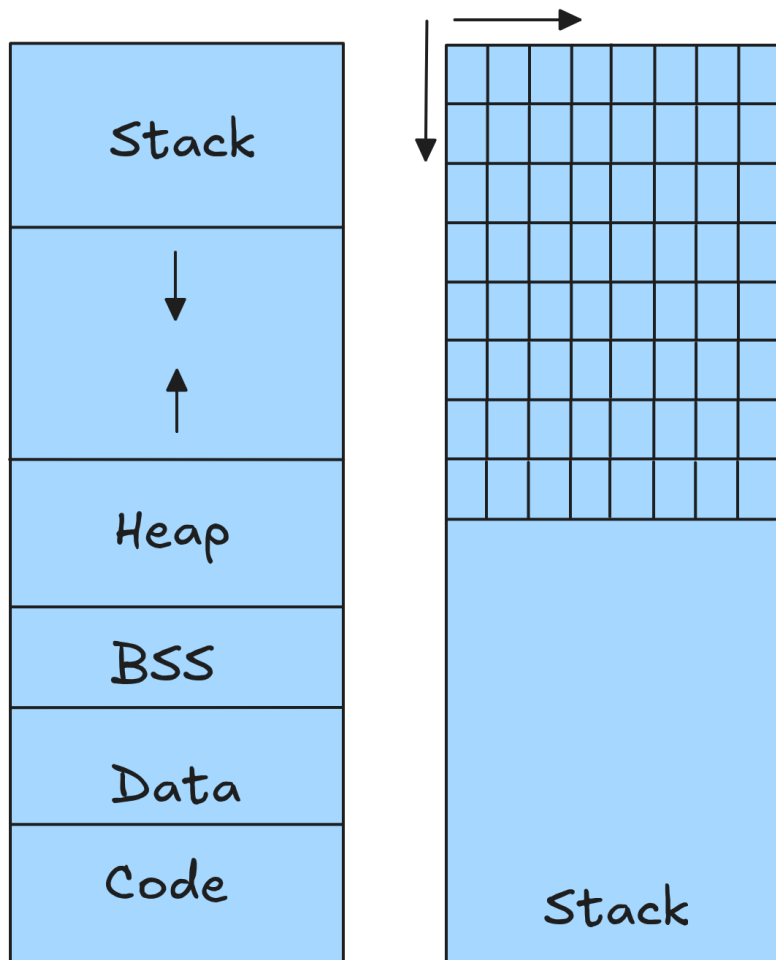
Serial



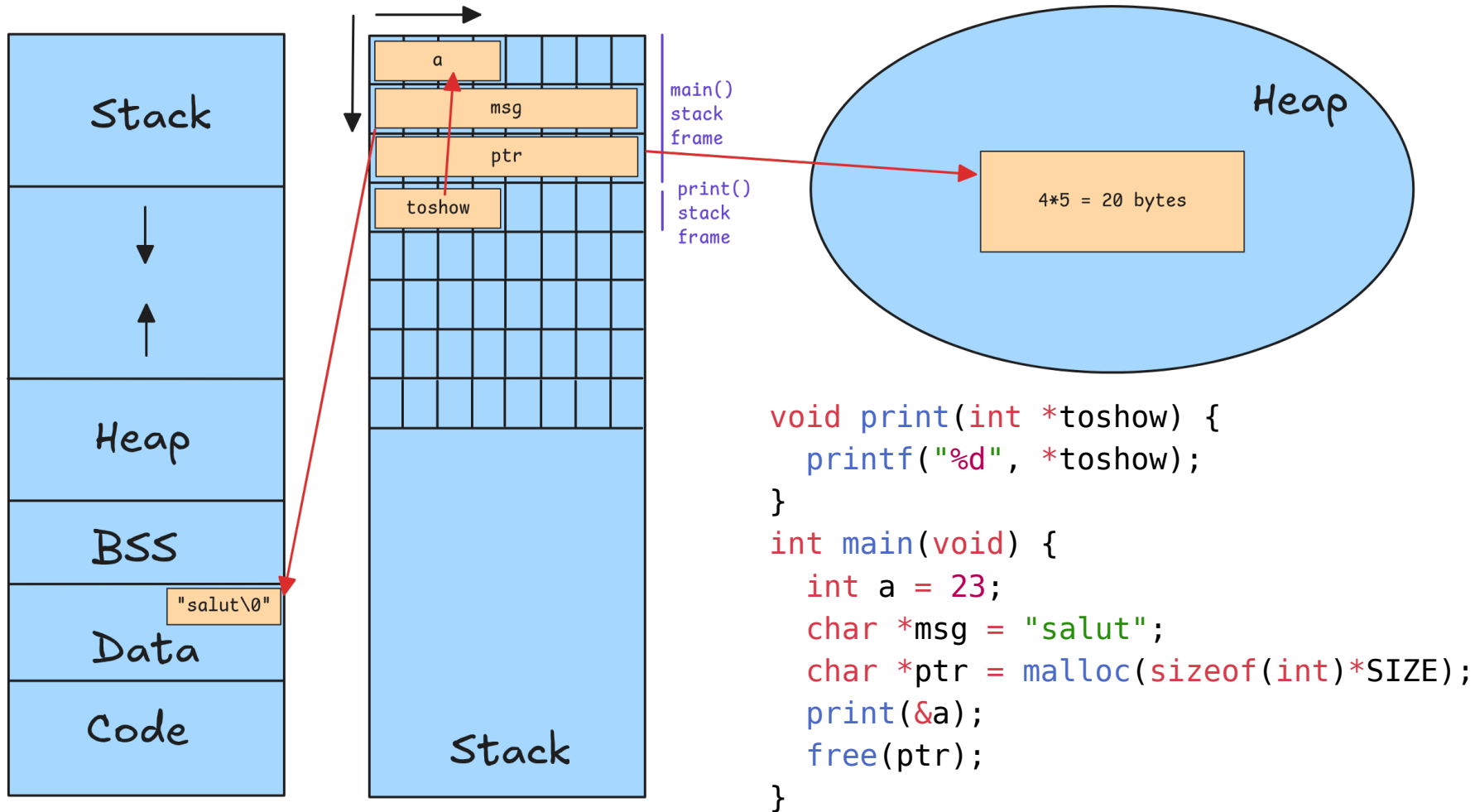
- Multiple processes
- Shared resources
- Critical section

Parallel



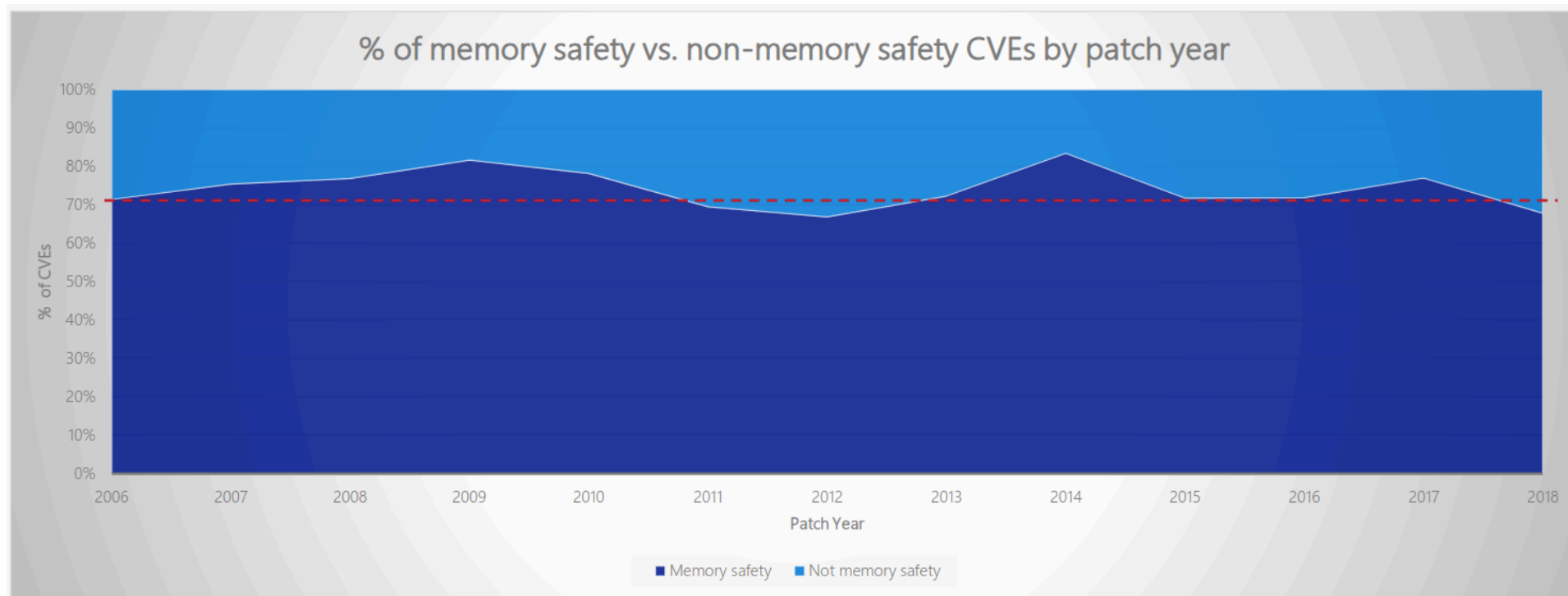


```
void print(int *toshow) {  
    printf("%d", *toshow);  
}  
  
int main(void) {  
    int a = 23;  
    char *msg = "salut";  
    char *ptr = malloc(sizeof(int)*SIZE);  
    print(&a);  
    free(ptr);  
}
```



# Why memory safety is a big deal ?

7/19



“~70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues”. From [Microsoft presentation from 2019](#).

## Rust new paradigms

- Invented at Mozilla, released 1.0 in 2015, most loved programming language Stack-overflow survey from 2016
- Advanced static analysis at compilation time
- In addition to a type and variable, each ressource has an **owner** and a **lifetime**
- Advanced smart pointers, traits and concurrency mecanisms



```
// library.h
```

```
void save_file(float* buffer, char* filename);
```

```
// main.c
```

```
float* buffer = malloc(SIZE * sizeof(float));
```

```
char* filename = "test.txt";
```

```
save_file(buffer, filename);
```

```
free(buffer); // changed ? need to be freed ???
```

- Borrow
- Rules of borrow
  - Only one owner per ressource
  - Only one mutable reference at a time
  - Or several immutable references
  - References must always be valid

```
let mut patrick =  
String::from("scissor");  
let sam = &patrick;  
// cannot borrow `*sam` as mutable,  
// as it is behind a `&` reference  
sam.push_str("s")
```

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

---

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

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

- Each variable has a lifetime
- Compiler determine this to add instructions
- Possible to annotate lifetime

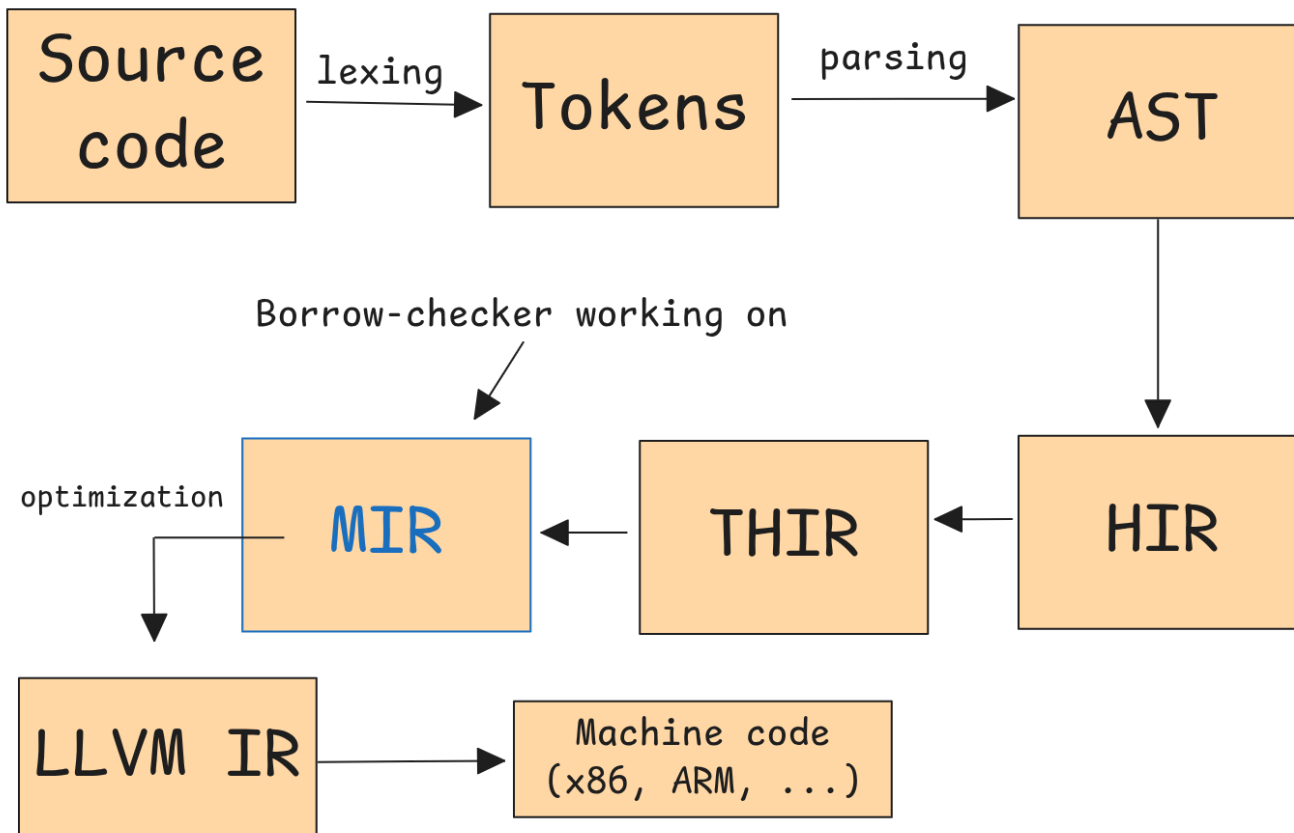
```
fn longest(x: &str, y: &str) -> &str {  
    if x.len() > y.len() {  
        x  
    } else {  
        y  
    }  
}
```

Error: missing lifetime specifier, this function's return type contains a borrowed value, but the signature does not say whether it is borrowed from x or y

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
    if x.len() > y.len() {  
        x  
    } else {  
        y  
    }  
}
```

- No manual management of memory
- No garbage collector
- Use lifetime and ownership to ensure no memory leak

What are the steps of the Rust compiler ?



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

## MIR Simplified

```
fn main() -> () {  
    let mut _0: ();  
    let _1: std::boxed::Box<i32>;  
    scope 1 {  
        debug m => _1;  
        scope 2 {  
            debug _a => _2;  
        }  
    }  
    bb1: {  
        _2 = move _1;  
        drop(_2) -> [return: bb2, unwind continue];  
    }  
}
```

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



In Java, OUPS...

```
public class WebServer {
    ArrayList<User> users;

    @POST
    public Response createUser() {
        users.add(new User("John"));
        // ...
    }
}
```

In Rust

```
struct Server {
    users: Rc<Vec<String>>,
}

impl Server {
    fn start(&self) {
        thread::spawn(move || {
            println!("{:?}", self.users);
        });
    }
}
```

Error: Rc<Vec<String>> cannot be shared between threads safely within Server, required for &Server to implement std::marker::Send

In C++, OUPS...

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

```
struct MegaCounter {  
    some_counter: Mutex<i32>,  
}  
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() }  
}  
fn main() {  
    let counter = Arc::new(MegaCounter::new());  
    for i in 0..10 {  
        let arc = counter.clone();  
        thread::spawn(move || {  
            arc.increment(i);  
        }); } }
```

## Borrow checker enforced rules

- Only one mutable reference at a time
- Or several immutable references
- References must always be valid

## Why Rust ?

- Combining no garbage collector and no manual memory management
- Dynamic memory allocation with hidden free / drop
- Minimal overhead at runtime
- Whole package of memory safety issues removed
- Data-races fixed, easier multi-threading