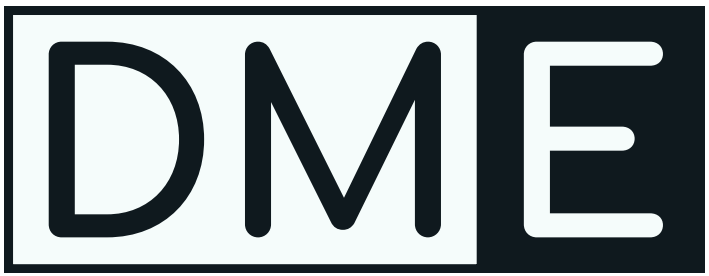



```
cargo new the-delightful-markdown-experience
```

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
```

The project

1. 3 main features
   - Research
   - Preview
   - PDF export
2. Maximum of parallelisation
3. Stability and low memory footprint

TreeSitter

Tauri

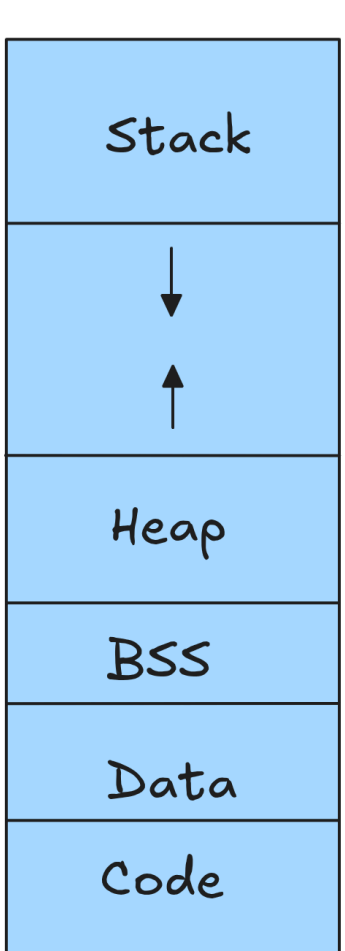
Tree-Sitter generated HTML example

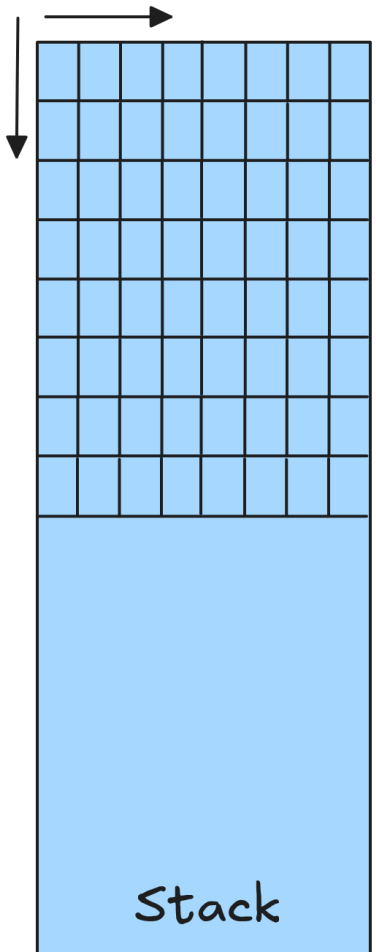
```
<pre>
    <code class="language-c">
        <span class="keyword control repeat">for</span>
        <span class="punctuation bracket">(</span>
        <span class="type">sf_count_t</span>
        <span class="variable">i</span>
        <span class="operator">=</span>
        <span class="constant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

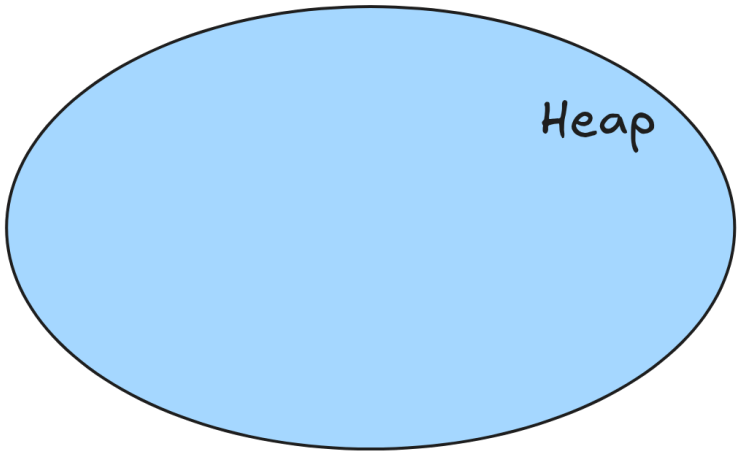
Basics of concurrency

- Multiple process
- Shared ressource
- Critical section

Basics of memory management

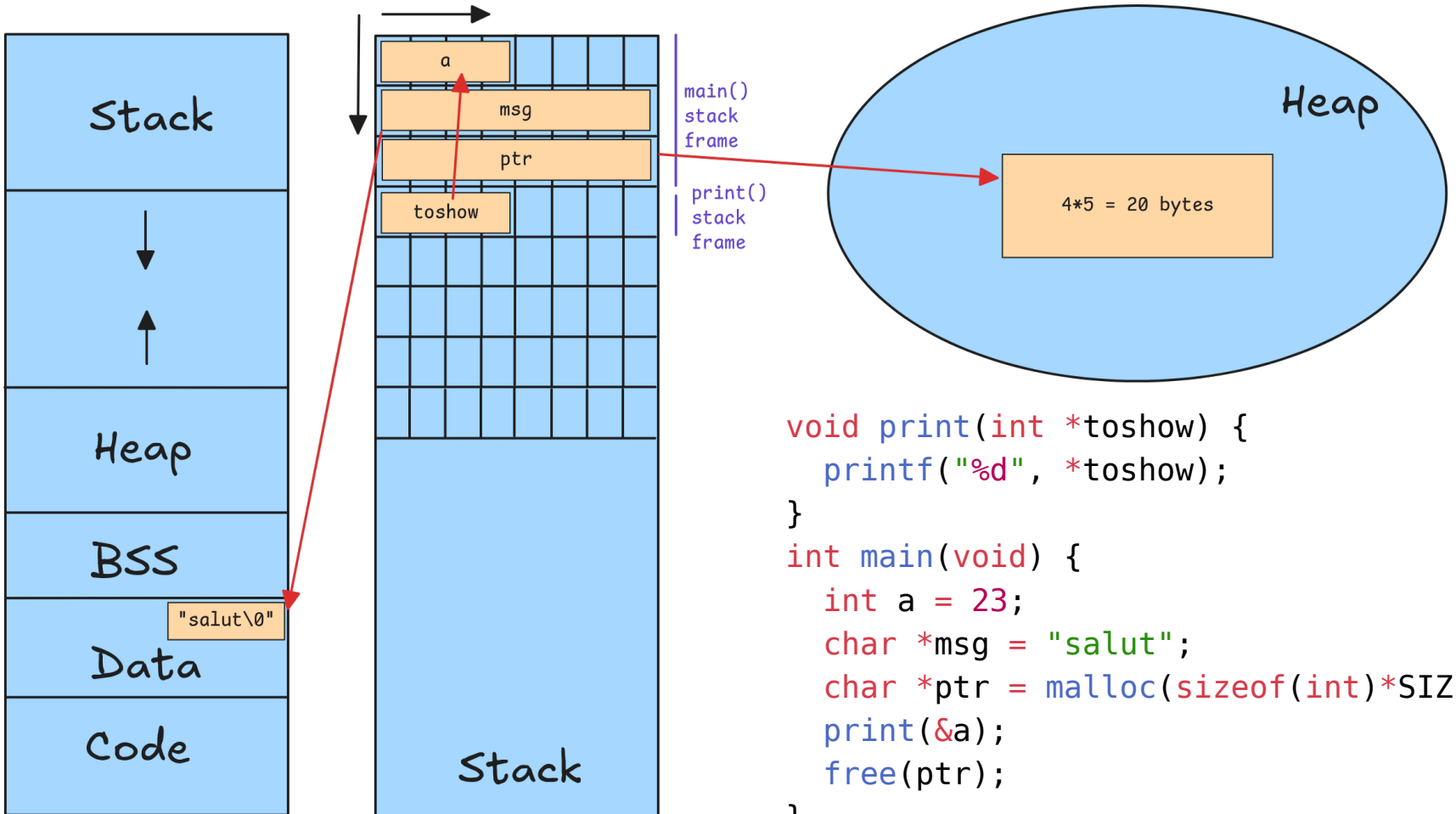






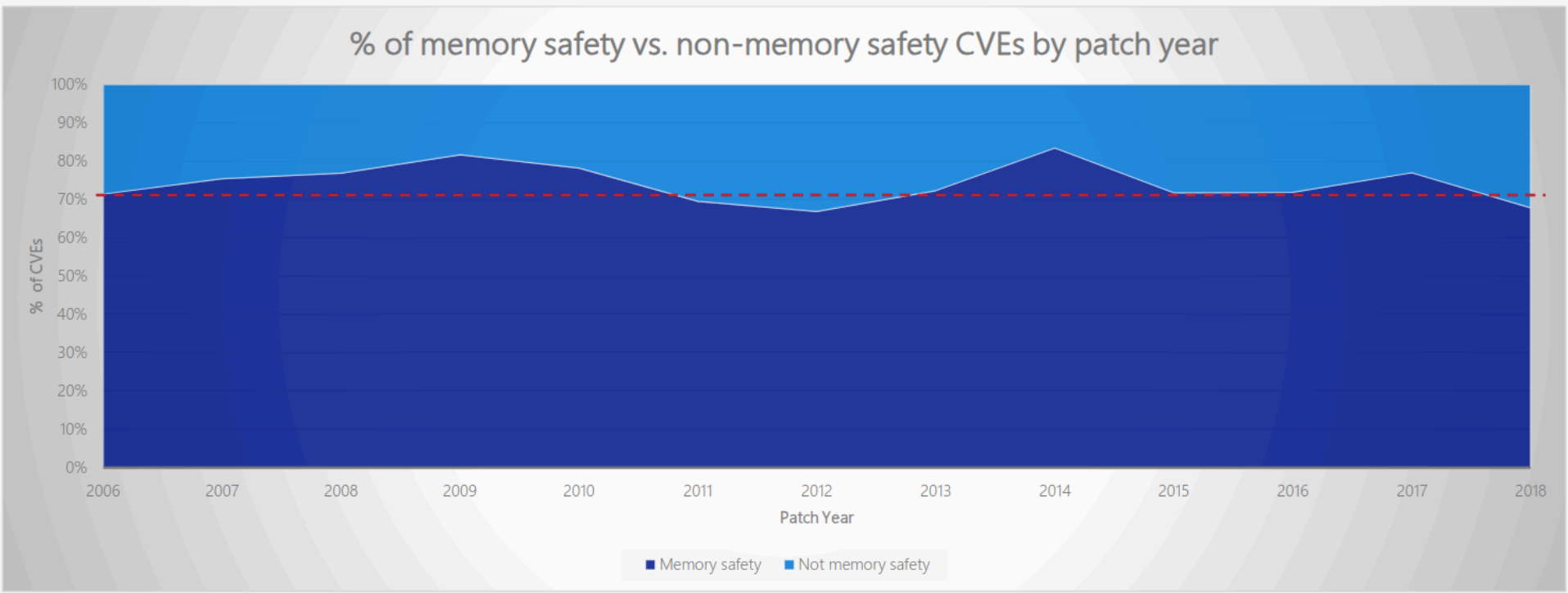

```c
void print(int *toshow) {
  printf("%d", *toshow);
}
int main(void) {
  int a = 23;
  char *msg = "salut";
  char *ptr = malloc(sizeof(int)*SIZE);
  print(&a);
  free(ptr);
}
```

Basics of memory management




```c
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 ?



"~70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues". From Microsoft presentation from 2019.

What's the solution ?

Rust new paradigms

- 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

Lifetimes and Ownership


```
// 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 ???
```

```
fn save_file(buffer: &[f32], filename: &str) {}

fn main() {
    let buffer = [10.2, 3.2, 5.2];
    let filename = "test.txt";
    save_file(&buffer, filename);
}
```

Ownership

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

Lifetimes

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

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

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

Memory management

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

Who knows how a compilator transform source code in machine language ?

Memory management

Code rust

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

HIR

```
[prelude_import]
use ::std::prelude::rust_2015::*;
#[macro_use]
extern crate std;

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

Memory management

MIR :

```
fn main() -> () {
    let mut _0: ();
    let _1: std::boxed::Box<i32>;
    scope 1 {
        debug _m => _1;
    }
    bb0: {
        _1 = Box::<i32>::new(const 2_i32) -> [return: bb1, unwind continue];
    }
    bb1: {
        drop(_1) -> [return: bb2, unwind continue];
    }
    bb2: { return; }
}
```

Concurrency - thread-safe data structures

In Java, OUPS…

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

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

In Rust

```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;
    }
};
```

In Rust

```
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);
        });
    }
}
```

Recap of the magic

Borrow checker enforced rules

- Only one mutable reference at a time
- Or several immutables references
- References must always be valid
- Dynamic memory allocation with hidden free / drop

Combining no garbage collector and no manual memory management

- Minimal overhead at runtime
- Whole package of memory safety issues removed
- Data-races fixed, easier multi-threading