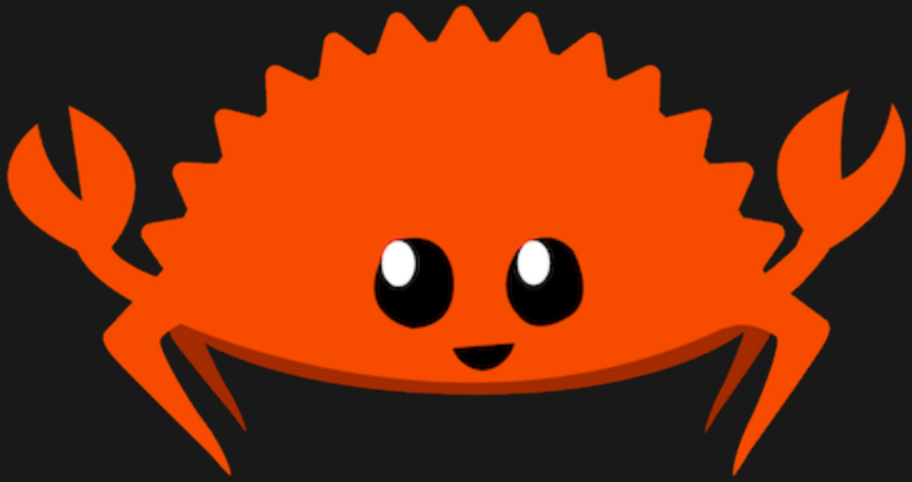
INTRODUCTION À RUST



À PROPOS

https://github.com/KurzF/rust-reveal

OBJECTIFS

- Pourquoi utiliser Rust ?
- Que ce qui rend Rust unique ?

LANGAGE COMPILÉ

- Performant
- Unique à une platforme

MEMORY SAFETY

- Assure la validité des opérations sur la mémoire
- Sans garbage collector

HELLO, WORLD!

CARGO

- Gestion de dépendances
- Et bien plus

Créer une application

```
cargo new projet
```

```
project/
├── src
│   └── main.rs
└── Cargo.toml
```

```rust
fn main() {
    println!("Hello, world!");
}
```

```rust
fn main() {
    println!("Hello, world!");
}
```

```rust
fn main() {
    let name = "Sfeir";
    println!("Hello, {}!", name);
}
```

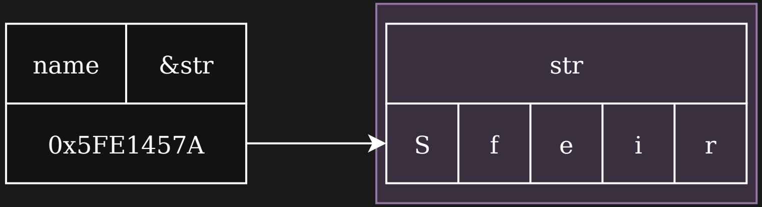
```rust
fn main() {
    let name  = "Sfeir";
    println!("Hello, {}!", name);
}
```

```rust
fn main() {
    let mut name = "World";
    name = "Sfeir";
    println!("Hello, {}!", name);
}
```

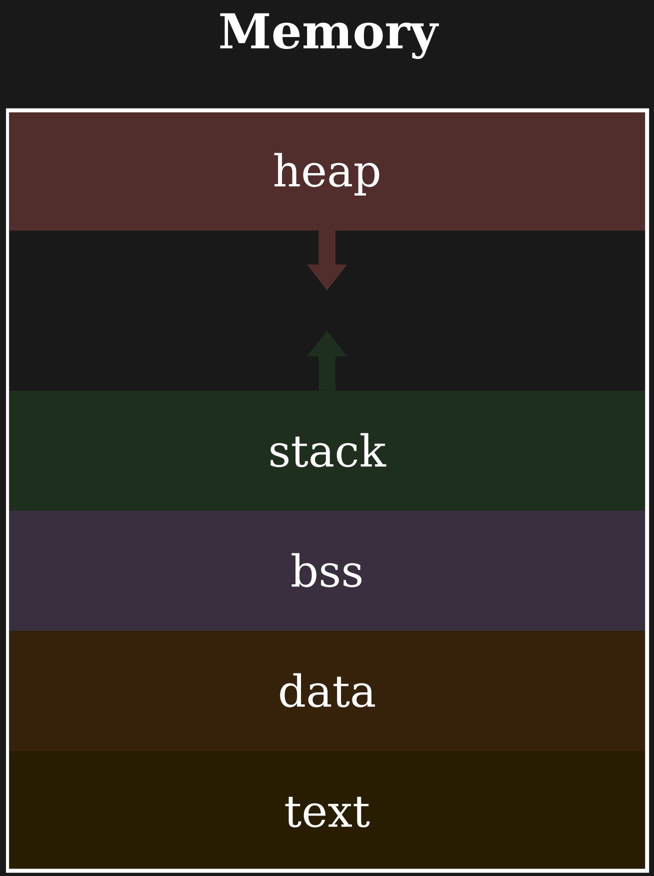
```rust
fn main() {
    let name: &str = "Sfeir";
    println!("Hello, {}!", name);
}
```

RÉFÉRENCES

- adresse vers une valeur
- toujours valide



RAPPELS SUR LA MÉMOIRE

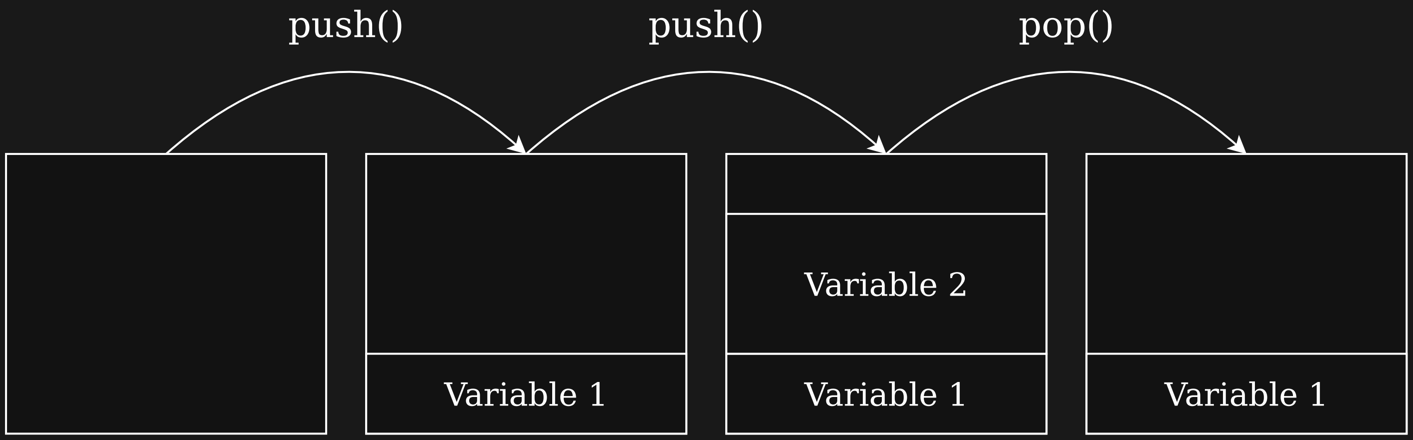


BLOCS STATIQUES

- Text contient les instructions
- Data contient les variables statiques initialisés
- Bss contient les variables statiques non initialisés

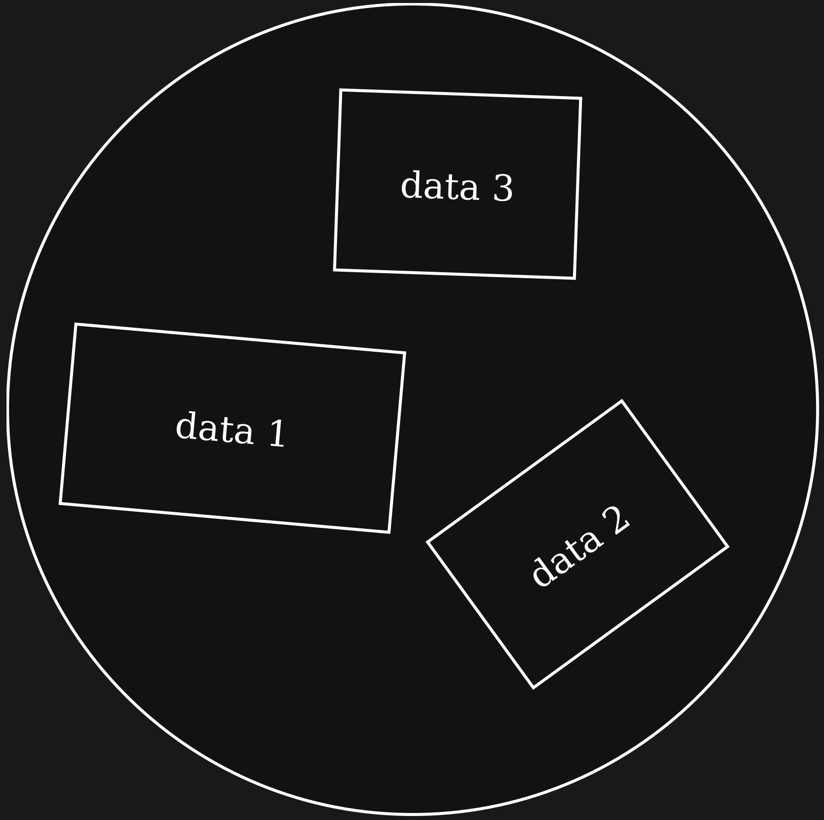
STACK

```
let var1 = 58;
{
    let var2 = 39
} // var2 n'est plus accessible
```



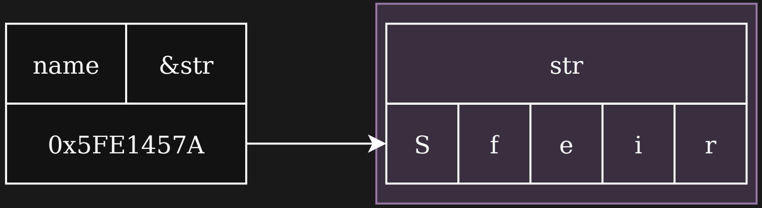
HEAP

- Gestion manuelle
- Allocation lente



SURPRISE!

```
fn main() {
    let name: &str = "Sfeir";
    println!("Hello, {}!", name);
}
```



ET LA MEMORY SAFETY ?

- Ownership
- Borrow checker

OWNERSHIP

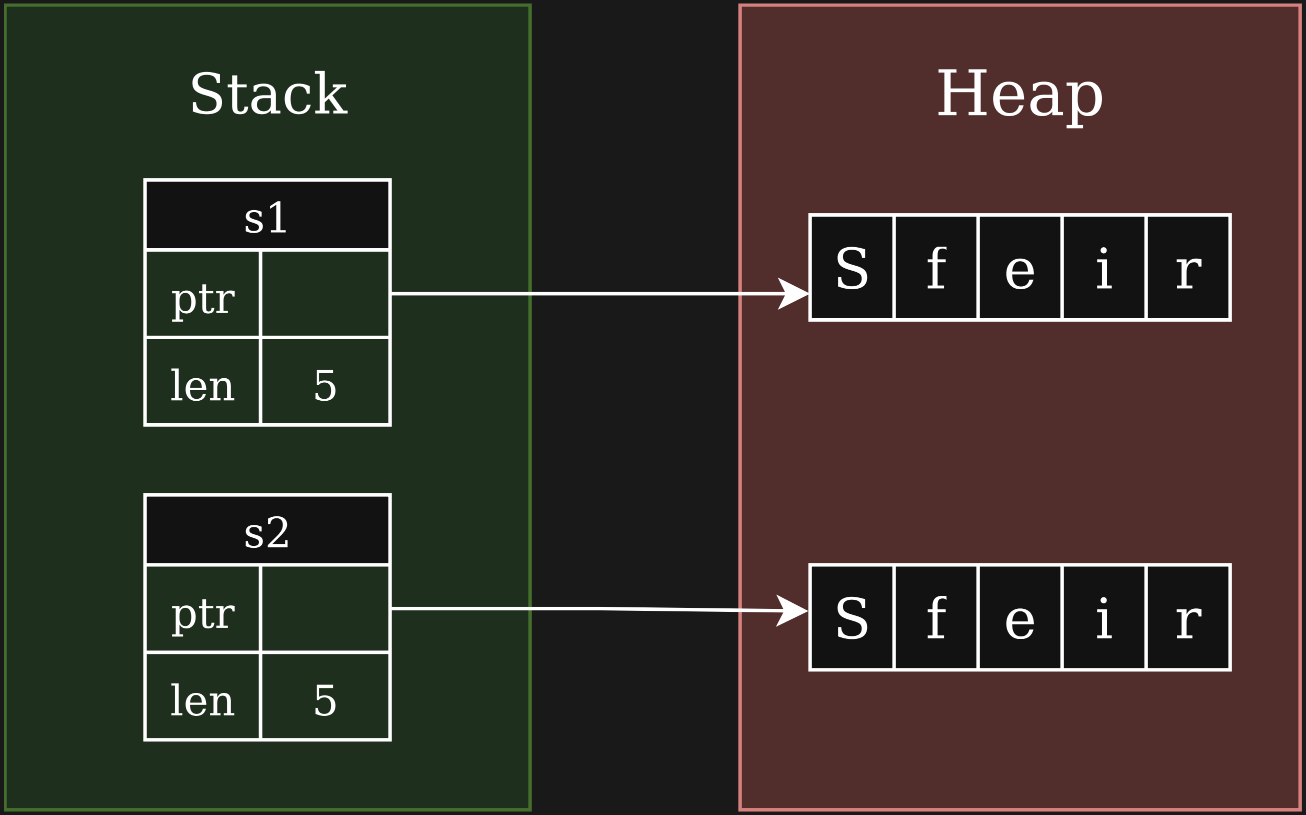
- Chaque valeur a un seul propriétaire
- Si le owner est *out-of-scope* la valeur est **dropped**

```rust
{
    let s: &str = "Hello, world!";
}
```

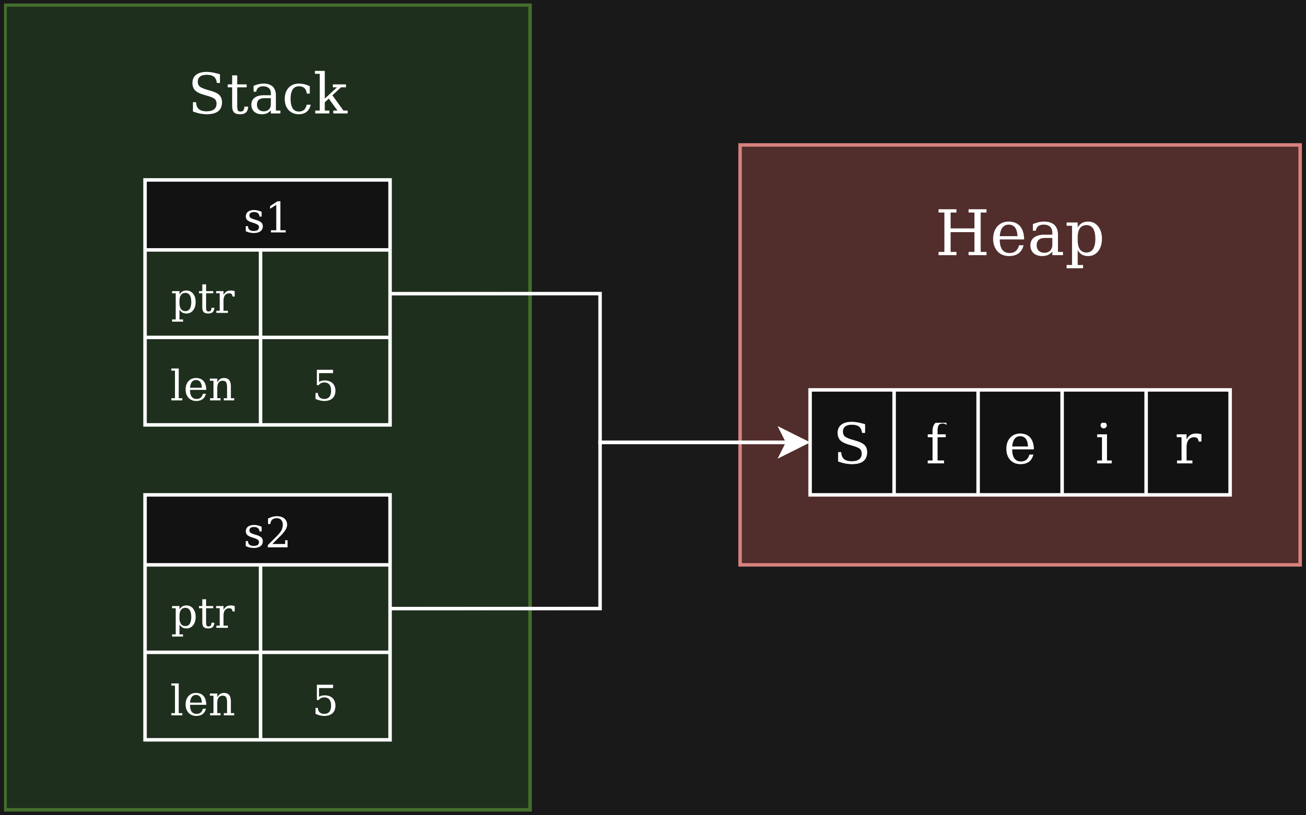
```rust
{
    let s: String = String::from("Hello, world!");
}
```

```rust
{
    let s1: String = String::from("Hello, world!");
    let s2 = s1;
}
```

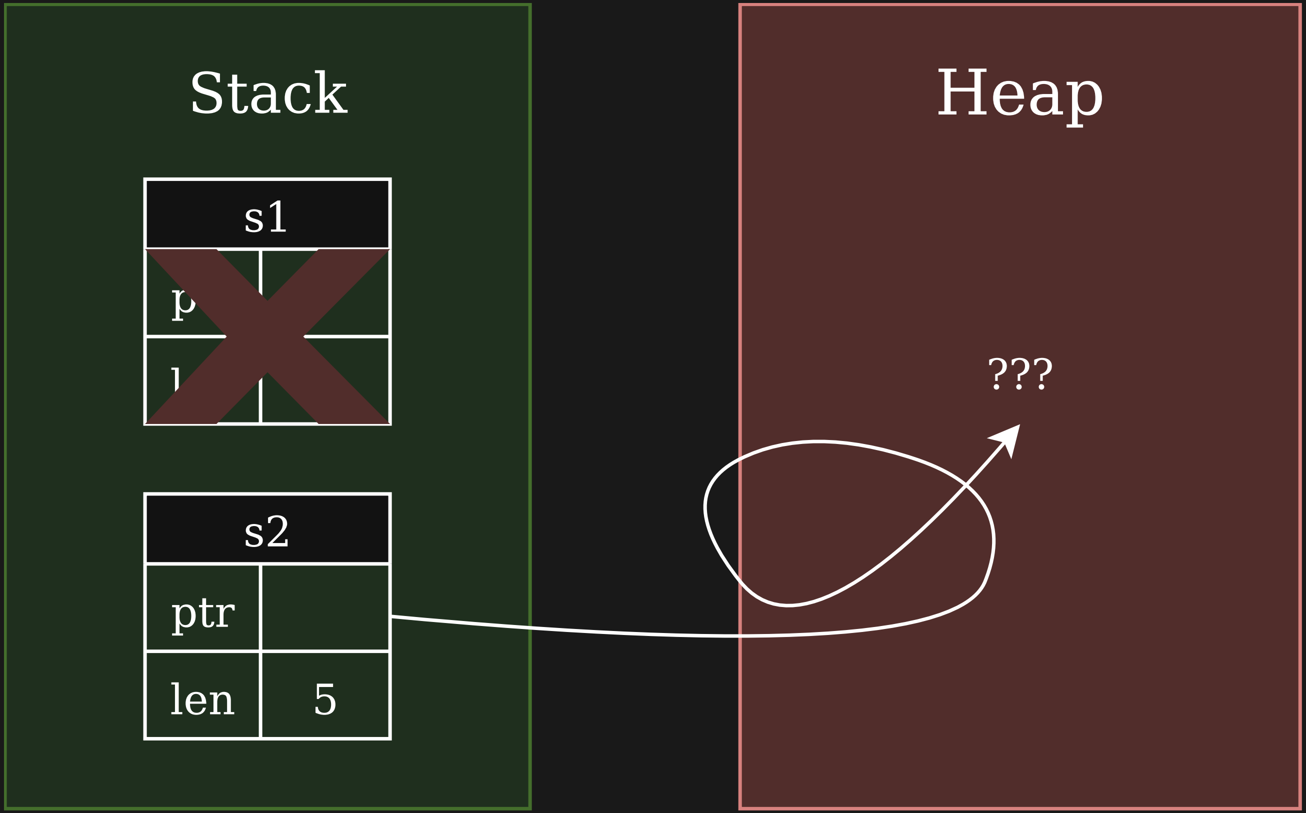
```rust
{
    let s1: String = String::from("Hello, world!");
    let s2 = s1;
}
```



```rust
{
    let s1: String = String::from("Hello, world!");
    let s2 = s1;
}
```

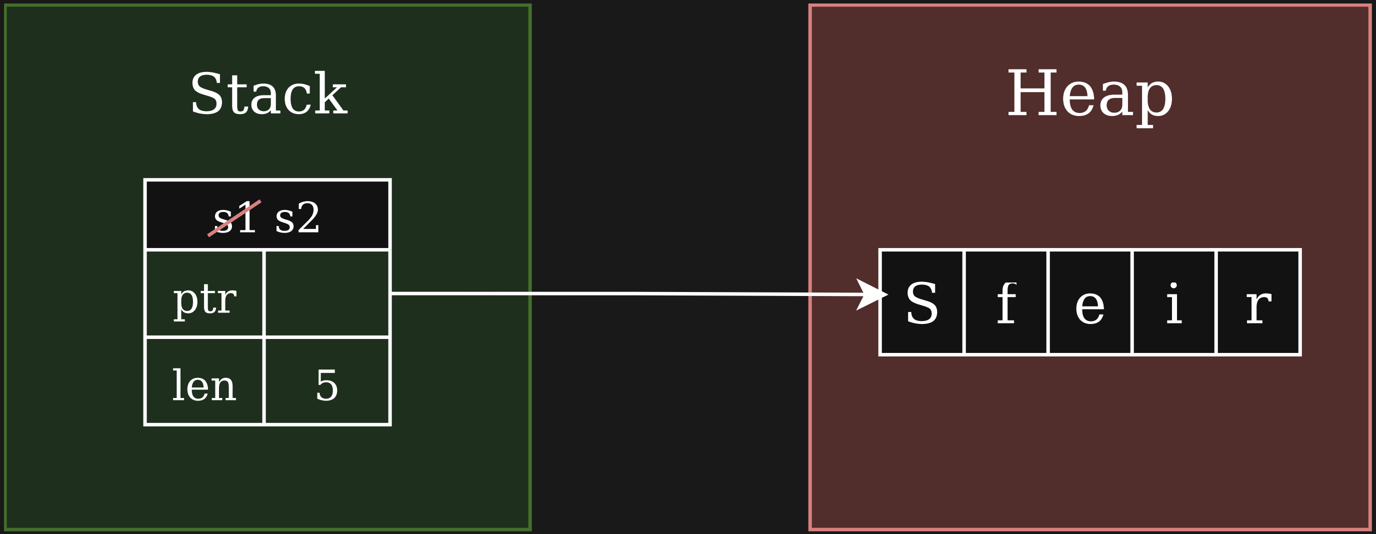


```rust
{
    let s1: String = String::from("Hello, world!");
    let s2 = s1;
    drop(s1);
    println!("{}", s2);
}
```



MOVE

```
{
    let s1: String = String::from("Hello, world!");
    let s2 = s1;
    drop(s1);
    println!("{}", s2);
}
```



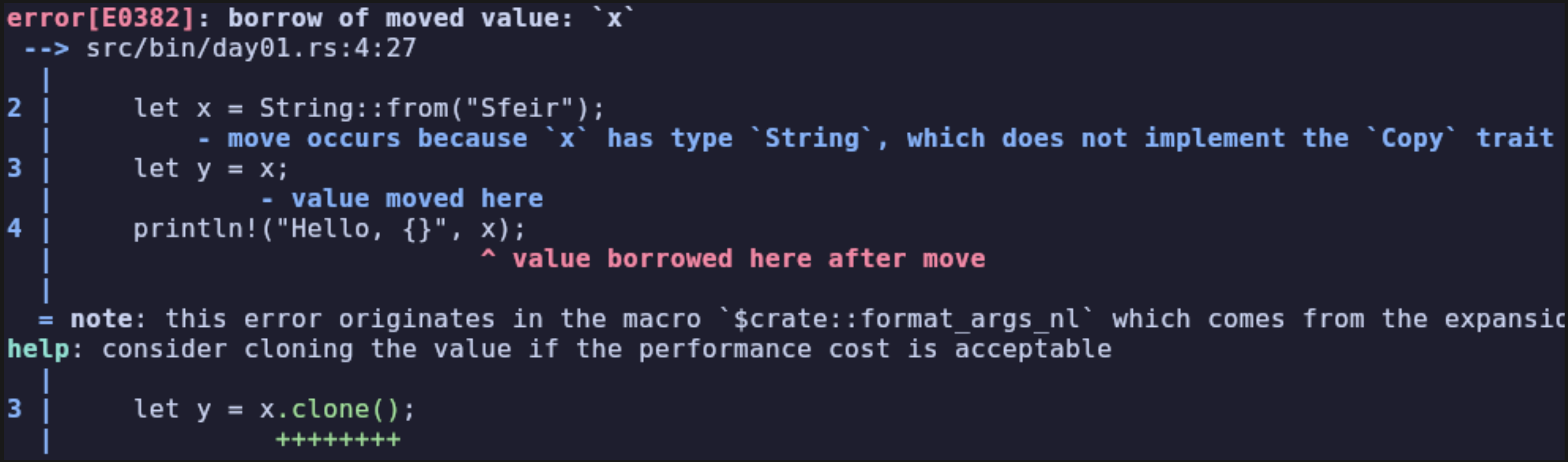
SURPRISE 2

```rust
let x = String::from("Sfeir");
let y = x;

println!("Hello, {}", x);
```

```rust
let x = String::from("Sfeir");
let y = x;

println!("Hello, {}", x);
```



```
error[E0382]: borrow of moved value: `x`
 --> src/bin/day01.rs:4:27
  |
2 |     let x = String::from("Sfeir");
  |         - move occurs because `x` has type `String`, which does not implement the `Copy` trait
3 |     let y = x;
  |             - value moved here
4 |     println!("Hello, {}", x);
  |                           ^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansi
help: consider cloning the value if the performance cost is acceptable
  |
3 |     let y = x.clone();
  |              ++++++++
```

BORROW

```rust
let x = String::from("Sfeir");
let y = &x;

println!("Hello, {}", x);
```

DANGLING ?

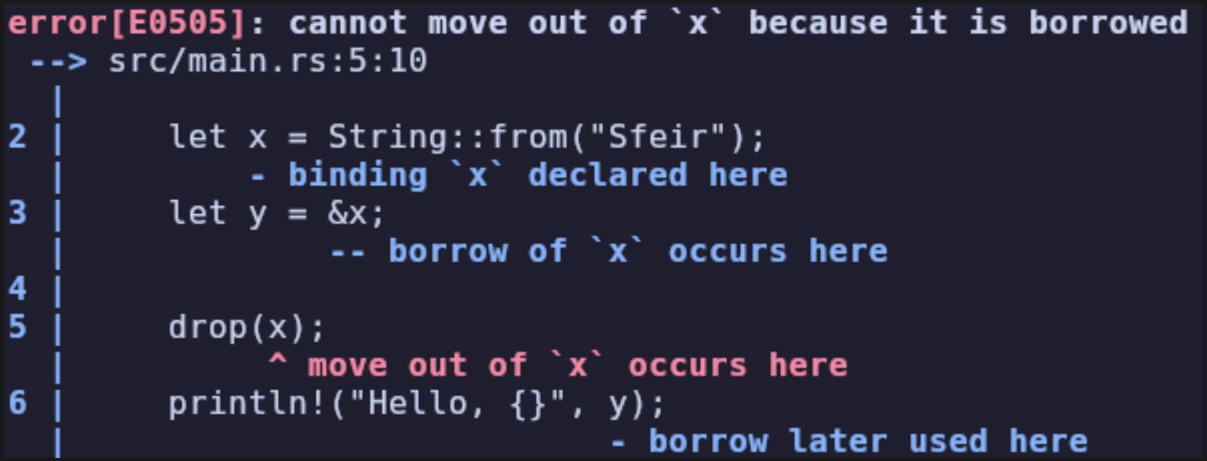
```
let x = String::from("Sfeir");
let y = &x;
drop(x);

println!("Hello, {}", y);
```

BORROW CHECKER

```rust
let x = String::from("Sfeir");
let y = &x;
drop(x);

println!("Hello, {}", y);
```



```
error[E0505]: cannot move out of `x` because it is borrowed
 --> src/main.rs:5:10
  |
2 |     let x = String::from("Sfeir");
  |         - binding `x` declared here
3 |     let y = &x;
  |             -- borrow of `x` occurs here
4 |
5 |     drop(x);
  |          ^ move out of `x` occurs here
6 |     println!("Hello, {}", y);
  |                           - borrow later used here
```

BORROW CHECKER

- Une référence est toujours valide
- Une valeur a une seule référence mutable **ou** plusieurs références immutables

SURPRISE 3

```rust
let s1 = String::from("Sfeir");
let s2 = &s1;
let s3 = &s1;
```

SURPRISE 3

```rust
let mut s1 = String::from("Sfeir");
let s2 = &mut s1;
```

SURPRISE 3

```rust
let mut s1 = String::from("Sfeir");
{
    let s2 = &mut s1;
}
let s3 = &mut s1;
```

FÉLICITATIONS

STRUCT

```
struct Person {
    name: String,
    role: Role
}
```

```rust
struct Person {
    pub name: String,
    pub(crate) role: Role,
}
```

```rust
struct Person {
    pub name: String,
    pub(crate) role: Role,
}
```

```rust
let speaker = Person {
    name: String::from("Fabien"),
    role: Role::DevFullstack,
};
```

```rust
impl Person
    pub fn manager() -> Person {
        Self {
name: String::from("Joshua Liaud"),
role: Role::DevFront, //Angular
        }
    }
}
```

```rust
let em = Person::manager();
```

```rust
impl Person {
    fn welcome(&self) {
        println!(
"Bonjour, je m'appelle {} et je suis {}",
self.name, self.role
        );
    }
}
```

ENUMÉRATION

```
enum Role {
    DevFront,
    DevBack,
    DevFullstack,
    DevOps,
}
```

ENUMÉRATION

```
enum Role {
    DevFront(TechFront),
    DevBack(TechBack),
    DevFullstack {
        front: TechFront,
        back: TechBack
    },
    DevOps,
}
```

```
Person {
    name: String::from("Joshua Liaud"),
    role: Role::DevFront(TechFront::Angular),
}
```

```rust
impl Role {
    fn fullstack_java() -> Self {
        Self::DevFullstack {
front: TechFront::GWT,
back: TechBack::Java
        }
    }
}
```

PATTERN MATCHING

```
fn use_java(&self) -> bool {
    match self {
        DevFront(front) => front == TechFront::GWT,
        DevBack(back) => back == TechBack::Java,
        DevFullstack { front, back } => {
front == TechFront::GWT || back == TechBack::Java,
        },
        _ => false
    }
}
```

TRAITS

```rust
pub trait Clone: Sized {
    fn clone(&self) -> Self;

    fn clone_from(&mut self, source: &Self) {
        *self = source.clone()
    }
}
```

```rust
pub trait Clone: Sized {
    fn clone(&self) -> Self;

    fn clone_from(&mut self, source: &Self) {
        *self = source.clone()
    }
}
```

```rust
impl Clone for Person {
    fn clone(&self) -> Self {
        Self {
name: self.name.clone(),
role: self.role.clone()
        }
    }
}
```

```rust
#[derive(Debug, Clone)]
pub struct Person {
    name: String,
    role: Role
}
```

GÉNÉRIQUE

OPTION

```rust
pub enum Option<T> {
    None,
    Some(T),
}
```

```rust
match opt {
    None => println!()
    Some(v) => println!()
}
```

RESULT

```rust
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

```rust
match res {
    Ok(v) => println!("Tout va bien: {}", v),
    Err(e) => println!("Erreur: {}", e),
}
```

TESTS UNITAIRES

```rust
#[cfg(test)]
mod tests {

    #[test]
    fn add() {
        assert_eq!(1 + 2, 3);
    }
}
```

TESTS UNITAIRES

```rust
#[cfg(test)]
mod tests {

    #[test]
    fn add() {
        assert_eq!(1 + 2, 3);
    }
}
```

CONCLUSION

RESOURCES

- The Rust book
- The Rust book with quiz
- Rustlings
- Rust by example

DES QUESTIONS ?

MERCI POUR VOTRE ATTENTION