Safe programming in Rust (TODO: fix title slide)

compscicenter.ru

training@ferrous-systems.com

Ownership

Some Rust

```
use std::io; ❶
use std::fs::File;

fn main() -> Result<(), io::Error> { ❷
    let open_file = File::open("test"); ❸

    let mut file = match open_file { ❹
        Sme(file) => file,
        Err(e) => return io::Error::from(e)
    };

    let mut buffer = String::new(); ❺
    file.read_to_string(&mut buffer)?; ❻
    println!("{}", buffer);

    Ok(()) ❼
}
```

Mutation

- Modern languages often have sematically immutable data.
- In Rust, mutation must be declared.

Mutation

```
fn main() {
    let answer = 42;
    answer = 30;
}

error[E0384]: cannot assign twice to immutable variable `answer`
 --> scratch.rs:3:5
  |
2 |     let answer = 42;
  |         ------
  |         |
  |         first assignment to `answer`
  |         help: make this binding mutable: `mut answer`
3 |     answer = 30;
  |     ^^^^^^^^^^^ cannot assign twice to immutable variable

error: aborting due to previous error
```

Working example

```rust
fn main() {
    let mut answer = 42;
    answer = 30;
}
```

Ownership

- Ownership is fundamental to Rust
- It is the basis for memory and resource management in Rust

Rules

- Every value has exactly one owner
- Ownership can be passed on, both to functions and other types
- The owner is responsible for removing the data from memory
- The owner has all powers over the data and can mutate it

Rules

These rules:

- are fundamental to Rusts type system
- are enforced at compile time

Example

```
#[derive(Debug)]
struct Dot {
    x: i32,
    y: i32
}

fn main() {
    let dot = Dot { x: 1, y: 2
}; ❶
    pacman(dot);
}

fn pacman(dot: Dot) { ❷
    println!("Eating {:?}",
dot);
    ❸
}
```

❶ Stack allocation

❷ Bare type names indicate ownership passing

❸ Deallocation point (automatically inserted)

Example

```
#[derive(Debug)]
struct Dot {
    x: i32,
    y: i32
}

fn main() {
    let dot = Dot { x: 1, y: 2
};
    pacman(dot);
    pacman(dot); ❶
}
```

❶ Illegal. TODO: insert error message

Oops!

In Rust-Lingo, this is called `consuming`. `pacman` consumes `dot`.

The value cannot be used anymore.

Background

When calling `pacman` with `dot`, the value is "moved" into the arguments of `pacman`. At that moment, ownership passes to `pacman`. `main` is not owner of the data anymore and thus not allowed to access or manipulate them.

Detour: What does that save us from?

```
use std::fs::File;

fn main() {
    let file =
File::open("test").unwrap();

    use_file(file);
    use_file(file); ❷
}

fn use_file(f: File) {
    // File drops here
    ❶
}
```

❶ Dropping a file handle closes it

❷ The second call to use_file would access a closed file

Making illegal state irrepresentable

Rust `File` handles are always open and the type system can enforce that.

Similar modelling is possible for other types that can be in multiple states.

Coming back: Plain Data

But our `Dot` is plain data, and this is inconvenient.

Working with moves: explicit clone

We can create a second copy of the data!

Example

```
#[derive(Debug, Clone)] ❶
struct Dot {
    x: i32,
    y: i32
}

fn main() {
    let dot = Dot { x: 1, y: 2
};
    pacman(dot.clone()); ❷
    pacman(dot);
}

fn pacman(dot: Dot) {
    println!("Eating {:?}",
dot);
}
```

❶ The `Clone` derive autogenerates cloning code

❷ `clone()` must be called before the value is moved.

This semantically creates **2** owned values of `Dot`.

Cloning

Cloning is a general operation that - depending on the complexity of the data at hand - can be costly.

Working with moves: Copy

But this is still inconvenient!

Copy to the rescue!

```
#[derive(Debug, Clone, Copy)]
❶
struct Dot {
    x: i32, ❷
    y: i32
}

fn main() {
    let dot = Dot { x: 1, y: 2
};
    pacman(dot); ❸
    pacman(dot);
}

fn pacman(dot: Dot) {
    println!("Eating {:?}",
dot);
}
```

❶ Copy types must always be Clone

❷ Copy can only be derived if all fields are Copy

❸ move is replaced by a copy

This semantically creates 3 owned values of Dot.

About Copy

Copy is meant for data that can be quickly copied in memory (using memcopy) and are allowed to be copied (e.g.: not File pointers).

About Copy

Values that are copy follow the standard ownership rules, but they are copied when ownership is passed on.

Warning

The terminology around moves is similar, but not the same to the one used in C++, which is why you should always use Rust-Terminology: Ownership, passing on ownership and consumption.

TODO: use fancy asciidoc warnings

Strategy

Rust does not assume, it makes you establish guarantees. It cannot eassily figure out if a value is allowed to be `Copy` or not - so it lets you establish guarantees.

TODO: use fancy asciidoc infobubbles

Small quiz

`drop` is the function that forces dropping a value immediately.

What does implementation look like?

```rust
use std::fs::File;

fn main() {
    let mut file = File::open("test").unwrap();
    let buffer = read_from(&mut file); //read_from is a standin, it doesn't exist
    drop(file);
    // do something long
}
```

Solution

```
#[inline]
fn drop<T>(_: T) { ❶
  // take ownership, drop out
of scope
}
```

❶ Functions in Rust can be generic, this one takes any type