Rust del 2

Agenda

1. Runde rundt bordet
2. Rust-foredrag 1: Eigarskap og låning
3. Rustlings og/eller CodeWars
4. Rust-foredrag 2: Fleirtråding i Rust
   . Meir progging
5. Oppsummering

Runde rundt bordet

- Har du gjort noko kult sidan sist?
- Har du nokon datatriks å dele?
- Treng ikkje å vere lågnivå

Rust-foredrag 1: Eigarskap og låning

Ved Ole-Magnus

Rustlings

- Lær Rust ved å fikse feil i oppgåvene!
- Vi anbefalar at ein **går saman to og to**, men det går også an å gjere oppgåvene åleine.

Rustlings er ein slags interaktiv tutorial
rustlings.cool

Om du er ferdig med rustlings, foreslår vi heller CodeWars
codewars.com

Rust-foredrag 2: Fleirtråding i Rust

Formål: Bedre forstå kvifor kompilatoren klagar så gæli når ein prøvar å bruke fleire trådar

Og forhåpentlegvis kva ein kan gjere med det

Fleirtråding i Rust 101

```
fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..10 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap()
}
```

Kvifor vil dette gi kompilatorfeil?

```rust
fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

Kvifor vil dette gi kompilatorfeil?

```rust
fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    drop(v); // Frigjere minne, men vil det skje før eller etter at lambdaen har brukt v?

    handle.join().unwrap();
}
```

Løysing

```rust
fn main() {
    let v = vec![1, 2, 3];

    // move er eit lykelord som gjer at lambdaen tek eigarskap over v
    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    // Kan dermed ikkje bruke v her

    handle.join().unwrap();
}
```

Kommunikasjon på tvers av trådar

Kan kommunisere på tvers av trådar, eller dele data, på minst to måtar (som er skildra i The Rust Book):

1. Message passing
2. Shared state

Message passing

```rust
fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

Message passing

```
fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
            // Dette går faktisk ikkje, fordi send tek eigarskap av val
            println!("val is {}", val);
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

Message passing vs shared state

- Message passing/channels medfører at berre ein part har eigarskap over data på same tid
- Message passing er minst komplekst og derfor ofte å anbefale
- Shared state gjer at fleire kan ha eigarskap samtidig. Skummelt! 😱
- Men shared state kan av og til vere den modellen som passar best for eit gitt problem
    - Døme: Ein global cache

Kva trengs for shared state?

1. Mutex

```
fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m = {:?}", m);
}
```

Forsøk 1

```rust
fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

Søren!

```
$ cargo run
   Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0382]: use of moved value: `counter`
  --> src/main.rs:9:36
   |
5  |     let counter = Mutex::new(0);
   |         ------- move occurs because `counter` has type `Mutex<i32>`, which does not implement the `Copy` trait
...
9  |         let handle = thread::spawn(move || {
   |                                    ^^^^^^^ value moved into closure here, in previous iteration of loop
10 |             let mut num = counter.lock().unwrap();
   |                           ------- use occurs due to use in closure

For more information about this error, try `rustc --explain E0382`.
error: could not compile `shared-state` due to previous error
```

Rc

Rc<T>, the Reference Counted Smart Pointer

In the majority of cases, ownership is clear: you know exactly which variable owns a given value. However, there are cases when a single value might have multiple owners. For example, in graph data structures, multiple edges might point to the same node, and that node is conceptually owned by all of the edges that point to it. A node shouldn't be cleaned up unless it doesn't have any edges pointing to it and so has no owners.

You have to enable multiple ownership explicitly by using the Rust type `Rc<T>`, which is an abbreviation for *reference counting*. The `Rc<T>` type keeps track of the number of references to a value to determine whether or not the value is still in use. If there are zero references to a value, the value can be cleaned up without any references becoming invalid.

Rc og clone

Module std::rc 1.0.0 · source · [−]

[redacted] 'Rc' stands for 'Reference Counted'.

The type `Rc<T>` provides shared ownership of a value of type `T`, allocated in the heap. Invoking `clone` on `Rc` produces a new pointer to the same allocation in the heap. When the last `Rc` pointer to a given allocation is destroyed, the value stored in that allocation (often referred to as "inner value") is also dropped.

Clone

Trait std::clone::Clone

1.0.0 · source · [-]

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

    // Provided method
    fn clone_from(&mut self, source: &Self) { ... }
}
```

[-] A common trait for the ability to explicitly duplicate an object.

Differs from `Copy` in that `Copy` is implicit and an inexpensive bit-wise copy, while `Clone` is always explicit and may or may not be expensive. In order to enforce these characteristics, Rust does not allow you to reimplement `Copy`, but you may reimplement `Clone` and run arbitrary code.

Since `Clone` is more general than `Copy`, you can automatically make anything `Copy` be `Clone` as well.

Forsøk 2

```rust
fn main() {
    let counter = Rc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

Doh!

```
$ cargo run
   Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0277]: `Rc<Mutex<i32>>` cannot be sent between threads safely
  --> src/main.rs:11:36
   |
11 |           let handle = thread::spawn(move || {
   |                        ------------- ^------
   |                        |             |
   |  ______________________|_____________within this `[closure@src/main.rs:11:36: 11:43]`
   | |                      |
   | |                      required by a bound introduced by this call
12 | |             let mut num = counter.lock().unwrap();
13 | |
14 | |             *num += 1;
15 | |         });
   | |_________^ `Rc<Mutex<i32>>` cannot be sent between threads safely
   |
   = help: within `[closure@src/main.rs:11:36: 11:43]`, the trait `Send` is not implemented for `Rc<Mutex<i32>>`
note: required because it's used within this closure
  --> src/main.rs:11:36
   |
11 |         let handle = thread::spawn(move || {
   |                                    ^^^^^^^
note: required by a bound in `spawn`
  --> /rustc/d5a82bbd26e1ad8b7401f6a718a9c57c96905483/library/std/src/thread/mod.rs:704:8
   |
   = note: required by this bound in `spawn`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `shared-state` due to previous error
```

Send

Trait std::marker::Send

1.0.0 [−] [src]

```
pub unsafe auto trait Send { }
```

[−] Types that can be transferred across thread boundaries.

This trait is automatically implemented when the compiler determines it's appropriate.

An example of a non-`Send` type is the reference-counting pointer `rc::Rc`. If two threads attempt to clone `Rc`s that point to the same reference-counted value, they might try to update the reference count at the same time, which is undefined behavior because `Rc` doesn't use atomic operations. Its cousin `sync::Arc` does use atomic operations (incurring some overhead) and thus is `Send`.

Altså: Arc implementerer både Clone og Send

Forsøk 3

```rust
fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

Oppsummering av shared state

- Mutex var ikkje nok fordi den ikkje implementerer Copy eller Clone, som gjer at den berre kan ha maksimalt éin eigar
- Kunne ikkje bruke Rc, fordi den ikkje implementerer Send, ergo kan den ikkje sendast mellom trådar
- Løysinga er Arc i kombinasjon med Mutex.
- Mutex sørger for låsing, Arc sørger for at fleire kan eige og bruke Mutexen "samtidig"
- I tillegg: Rust-kompilatoren brukar traits flittig for å sikre minnetryggleik

No: Meir Rustlings/CodeWars!