Why I am learning a new programming language - and why you should too!
- part 2 -

"Concurrent programming in Rust"
by example

Dr. Michael O. Distler

Mainz, 12 February 2019

special lecture as part of "Introduction to HPC Programming"
(Dr. Bernd-Peter Otte)

Content

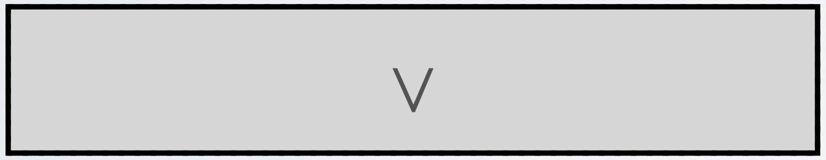
- Ownership and borrowing.
- Traits: Send and Sync.
- Smart pointers: Arc<T> and Mutex<T>.
- Asynchronous communication between threads: mpsc::channel.
- Examples: ping, ring, (dining philosophers problem)

Ownership

```
fn main() {
    let mut v = Vec::new();
    v.push(1);
    v.push(2);
    take(v);
    // ...
}
```

```
fn take(v: Vec<i32>) {
    // ...
}
```

move ownership



Ownership

```
fn main() {
    let mut v = Vec::new();
    v.push(1);
    v.push(2);
    take(v);
    v.push(3);
}
```

```
fn take(v: Vec<i32>) {
    // ...
}
```

error: use of moved variable v

Borrowing

```
fn main() {
    let mut v = Vec::new();
    push(&mut v);
    read(&v);
    // ...
}
```

```
fn push(v: &mut Vec<i32>)
{
    v.push(1);
}

fn read(v: Vec<i32>) {
    // …
}
```

Traits: Copy and Clone

Traits abstract over behavior that types can have in common.

Examples: **Copy** and **Clone**

- Copies happen implicitly, for example as part of an assignment `y = x`. The behavior of **Copy** is not overloadable; it is always a simple bit-wise copy.

Traits: Copy and Clone

Traits abstract over behavior that types can have in common.

Examples: Copy and Clone

- Cloning is an explicit action, x.clone(). The implementation of Clone can provide any type-specific behavior necessary to duplicate values safely. For example, the implementation of Clone for String needs to copy the pointed-to string buffer in the heap. A simple bitwise copy of String values would merely copy the pointer, leading to a double free down the line. For this reason, String is Clone but not Copy.

Traits: Send and Sync

Send and Sync are fundamental to Rust's concurrency story.

- A type is Send if it is safe to send it to another thread.
- A type is Sync if it is safe to share between threads (&T is Send).

Smart pointer

… are data structures that not only act like a pointer but also have additional metadata and capabilities.

Examples:

- Vec<T>
- Box<T> for allocating values on the heap
- Rc<T>, a reference counting type that enables multiple ownership

Smart pointer: Arc<T>

- **Arc<T>**: A thread-safe reference-counting pointer. 'Arc' stands for 'Atomically Reference Counted'.
- The type Arc<T> provides shared ownership of a value of type T, allocated in the heap. Invoking clone on Arc produces a new Arc instance, which points to the same value on the heap as the source Arc, while increasing a reference count. When the last Arc pointer to a given value is destroyed, the pointed-to value is also destroyed.
- Shared references in Rust disallow mutation by default, and Arc is no exception: you cannot generally obtain a mutable reference to something inside an Arc.

Smart pointer: Arc<T>

```rust
// lecture13/src/bin/arc.rs
// cd lecture13; cargo run --bin arc
use std::sync::Arc;
use std::thread;
fn main() {
    let five = Arc::new(5);
    for _ in 0..10 {
        let five = Arc::clone(&five);
        thread::spawn(move || {
            println!("{:?}", five);
        });
    }
}
```

Change the code, so each thread prints it's ID.

Smart pointer: Mutex<T>

- Mutex<T>: A mutual exclusion primitive useful for protecting shared data
- This mutex will block threads waiting for the lock to become available. The mutex can also be statically initialized or created via a new constructor. Each mutex has a type parameter which represents the data that it is protecting. The data can only be accessed through the RAII guards returned from lock and try_lock, which guarantees that the data is only ever accessed when the mutex is locked.

RAII: Resource acquisition is initialization

Communication between threads

```
pub fn channel<T>() -> (Sender<T>, Receiver<T>)
```

- Creates a new asynchronous channel, returning the sender/receiver halves. All data sent on the Sender will become available on the Receiver in the same order as it was sent, and no send will block the calling thread (this channel has an "infinite buffer", unlike sync_channel, which will block after its buffer limit is reached). recv will block until a message is available.

Communication between threads

```
pub fn channel<T>() -> (Sender<T>, Receiver<T>)
```

- The Sender can be cloned to send to the same channel multiple times, but only one Receiver is supported.
- If the Receiver is disconnected while trying to send with the Sender, the send method will return a SendError. Similarly, if the Sender is disconnected while trying to recv, the recv method will return a RecvError.

Communication between threads

```
use std::sync::mpsc::channel;
use std::thread;

let (sender, receiver) = channel();

// Spawn off an expensive computation
thread::spawn(move|| {
    sender.send(expensive_computation()).unwrap();
});

// Do some useful work for awhile

// Let's see what that answer was
println!("{:?}", receiver.recv().unwrap());
```

Smart pointer: Mutex<T>

```
use std::sync::{Arc, Mutex};
use std::thread;
use std::sync::mpsc::channel;

const N: usize = 10;
let data = Arc::new(Mutex::new(0));
let (tx, rx) = channel();
for _ in 0..N {
    let (data, tx) = (Arc::clone(&data), tx.clone());
    thread::spawn(move || {
        let mut data = data.lock().unwrap();
        *data += 1;
        if *data == N {
            tx.send(()).unwrap();
        }
    });
}

rx.recv().unwrap();
```

Change the code, so the value of data is printed

Exercise 3

1. Try to understand ‘ping.rs’
2. Run the program:
   `cargo run --bin ping -- --cycles 100000`
3. Change the transmitted data size.
   Does the transmit time change?
   Why (not)?

Exercise 4

1. Try to understand ‘ring.rs’
2. Run the program:

   `time target/debug/ring --threads 16`
3. Try to understand ‘MPIring.c’
4. Run the program:

   `time mpirun target/debug/MPIring`
5. Do you notice a difference?
6. Double the number of threads in both cases

Further reading and viewing

- The Rust Programming Language
  https://doc.rust-lang.org/stable/book/
- Vorlesung „Programmieren in Rust“, Universität Osnabrück, Wintersemester 2016/17.
  https://github.com/LukasKalbertodt/programmieren-in-rust
- https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/
- https://youtu.be/ecIWPzGEbFc
- https://youtu.be/6f5dt923FmQ

Installing Rust

- rustup: the Rust toolchain installer
  https://github.com/rust-lang-nursery/rustup.rs

```
# curl https://sh.rustup.rs \
    --silent --output rustup-init.sh
# sh rustup-init.sh
```