TD-7

Closures, Threads, Async and Concurrency

Pierre Cochard and Tanguy Risset

5TC-RUST 2025

Summary

- Using Threads and Closures to Run Code Simultaneously
- Sharing data safely between threads
  - Using *Shared State* data sets
    - Exclusive access with Mutexes
    - Reference Counted Mutexes
    - Lock-free data sharing with Atomics
  - Using *Message passing* to transfer data between threads
- Asynchronous programming

Using Threads and Closures to Run Code Simultaneously

Closures

- *Anonymous* functions that can be stored in variables, or passed to other processes as arguments.
- They allow *functional-style* programming, behavior customization and can provide *concurrent/parallel* code execution.

```
// Define a closure and store it into a variable:
let my_closure = |x: i32| -> i32 {
    println!("my_closure");
    x*2
};

// Execute the closure like you would normally do with a function:
let y = my_closure(2);
println!("{y}");
```

Borrowed captures

Just like in `C++`, closures can capture the environment it originates in and use external data in its own internal scope. By default, captured data, whether it is mutable or not, **will be borrowed**:

```
let x = 2;

let my_closure = || -> i32 {
    // Use a borrowed (immutable) capture of 'x',
    // and return two times its value:
    x*2
};

println!("{}", my_closure());
```

This time, we modify `x` directly in the closure:

```
let mut x = 2;

// The closure itself also has to be made 'mutable'
// in the case of a mutable borrow:
let mut my_closure_mut = || x *= 2;

my_closure_mut();
println!("{x}");
```

Moved captures

- data can also be **moved** into a closure scope, using the `move` keyword.
- The semantic is the same as for assignment: if the data type implements the `Copy` Trait, the data is captured by value (as shown below for `i32` type), otherwise, data is **moved**.

```
let mut x = 2;

// Capturing 'x' by value. Here, it is made with a simple copy:
let mut my_closure_mut = move || {
    x *= 2;
    println!("x (closure): {x}");
};

my_closure_mut();
println!("x: {x}");
```

Question 1)

Why is the following code invalid? How can we solve the issue?

```
let mut x = vec![31, 47, 27, 16];

let mut my_closure_mut = move || {
    x.push(32);
    println!("{:?}", x);
};

my_closure_mut();
println!("{:?}", x);
```

Passing closures as objects or arguments

One big specificity of closures is that they have a **unique, anonymous type** that **cannot be written out**. This can for instance be demonstrated by running the following piece of code:

```
// Utility function that prints out the type of a variable:
fn print_type_of<T>(_: &T) {
    println!("Type of my closure: {}", std::any::type_name::<T>());
}

fn main() {
    let my_closure = |x: i32| -> i32 {
        x*2
    };

    print_type_of(&my_closure);
}
```

Therefore, passing a closure **as an argument to a function** using a specific type is not possible in Rust. Instead, in order to do that, one would have to use a `trait`. Indeed, all closures implement one or several of the following traits, depending on their nature and properties:

- `FnOnce`: applies to closures that can be called once. **All closures implement this trait**;
- `Fn`: applies to closures that don't move captured values out of their body and that don't mutate captured values, as well as closures that capture nothing from their environment. These closures **can be called more than once** without mutating their environment, which is important in cases such as calling a closure multiple times concurrently.

- `FnMut`: applies to closures that don't move captured values out of their body, but that might mutate the captured values. These closures **can be called more than once**.

A closure can then be passed as an argument the same way we do for passing trait-implementing objects, by using the `Fn/FnOnce/FnMut(argument_type) -> return_type`:

```rust
// All of the 'Fn' traits have the format:
// 'Fn(argument-types) -> return types'
fn exec_closure(x: i32, closure: impl Fn(i32) -> i32) -> i32 {
    closure(x)
}

fn main() {
    let c = |x: i32| x + 27;
    let r = exec_closure(31, c);

    println!("{r}");
}
```

The **generic form** also works (and is usually preferrable):

```
fn exec_closure<T>(x: i32, closure: T) -> i32
where
    T: Fn(i32) -> i32
{
    closure(x)
}

fn main() {
    let c = |x: i32| x + 27;
    let r = exec_closure(31, c);

    println!("{r}");
}
```

Question 2)

Using the **generic form**, store the following `chirp_fn` closure as a member of the `struct` `Bird`, with a valid type signature:

```
struct Bird<...> {
    chirp: ...
}

fn main() {
    let chirp_fn = |times: i32| {
        for _ in 0..times {
            println!("chirp!");
        }
    };

    let bird = Bird { chirp: chirp_fn };
    (bird.chirp)(10);
}
```

Spawning Threads using Closures

A `std::thread` **object** in Rust will execute a given *closure* in an *independent* (or *parallel*) context of execution. In the following example, the `std::thread::spawn` call will only return when it's done creating the thread, **but not when the thread has actually finished executing**:

```
// Spawn a new thread which will execute its given closure:
std::thread::spawn(|| {
    println!("Thread 1: chirp!");
});
// At the 'same time', print something from the main thread:
println!("Main thread: chirp chirp!");
```

In this case, the main function returns before the independent thread's `println!` call happens. This is why we can only see the "main thread" print output. Usually, a thread is bound to a local variable, and is waited upon before the parent context of execution finishes. This can be done by calling the `.join()` method on the thread handle:

```
// Bind the thread's "handle" to a variable:
let th = std::thread::spawn(|| {
    println!("Thread 1: chirp!");
});

println!("Main thread: chirp chirp!");

// Wait for 'th' to finish executing, and re-synchronise both threads:
th.join().unwrap();
```

Questions 3) and 4)

The following code prints a **modified value of variable `var` from 3 different threads**, running independently from one another.

- **Question 3**: Is the code safe? Can you guess what will be the resulting output?
- **Question 4**: What would happen if we removed all the `move` keywords from the code?

```rust
use std::thread;

fn main() {
    let mut var = 32;

    let t1 = thread::spawn(move || {
        var += 1;
        println!("Thread 1: reading value {}!", var);
    });

    let t2 = thread::spawn(move || {
        var += 2;
        println!("Thread 2: reading value {}!", var);
    });

    var += 3;
    println!("Main thread: reading value {}", var);

    // Re-synchronise both threads:
    t1.join().unwrap();
    t2.join().unwrap();
}
```

Sharing data safely between threads

Using *Shared State* data sets

Exclusive access with Mutexes

Mutual exclusion, or *mutex* is a mechanism which prevents accessing the same data from multiple threads running at the same time. It relies on a *locking* system in order to do so: a thread must first ask to acquire the mutex's *lock* before being able to access the underlying protected data. The *lock* is a data structure that keeps track of whichever thread has exclusive access to the data. If the *mutex* happens to be locked at the time a thread tries to access the data, it will *stall* until the lock is eventually released, and the data is free to acquire.

```rust
use std::sync::Mutex;

fn main() {
    // Instantiate a new Mutex<i32> instance with the value '32':
    let var: Mutex<i32> = Mutex::new(32);
    {
        // Acquire the mutex's lock (and panic in case of failure):
        let mut v = var.lock().unwrap();
        // Modify the value safely:
        *v += 32;
    }
    // Print the result:
    println!("var = {var:?}");
}
```

Question 5)

In the following example, we want to try to use a `Mutex` to get both threads to use and modify the same value, but the compiler doesn't allow it, what is the underlying issue here?

```rust
use std::thread;
use std::sync::Mutex;

fn main() {
    let mtx = Mutex::new(32);

    let t1 = thread::spawn(move || {
        let mut v = mtx.lock().unwrap();
        *v += 1;
    });

    let t2 = thread::spawn(move || {
        let mut v = mtx.lock().unwrap();
        *v += 2;
    });
    // Re-synchronise both threads:
    t1.join().unwrap();
    t2.join().unwrap();

    println!("Result: {mtx:?}");
}
```

Reference Counted Mutexes

As we could see in our previous example, a `Mutex` in itself is not sufficient to implement viable *thread-safe* data sharing:

- First is the issue of **ownership**, which could be solved using, for instance, a *shared pointer*.
- Second would be the issue of concurrency in accessing this shared pointer from multiple threads simultaneously.

In the Rust programming language, *Atomic Reference Counting* `Arc<T>`, which can be seen as an **atomic shared pointer**, is designed to remedy this very specific problem. It firstly solves the ownership issue by being "*reference-counted*", just like a standard `Rc` object, but also solves the concurrency issue by being *atomic*, meaning that its reference counter is updated as a single indivisible operation. When an atomic operation is executed on an object by a specific thread, no other threads can read or modify the object while the atomic operation is in progress. In other words, other threads will only see the object before or after the operation, there would be no intermediary state.

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

fn main() {
    // We wrap the Mutex in a Atomically Reference-Counted object:
    let arc = Arc::new(Mutex::new(32));

    // We prepare two clones, for moving into the two distinct closures:
    let rc1 = Arc::clone(&arc);
    let rc2 = Arc::clone(&arc);

    let t1 = thread::spawn(move || {
        let mut v = rc1.lock().unwrap();
        *v += 1;
    });

    let t2 = thread::spawn(move || {
        let mut v = rc2.lock().unwrap();
        *v += 2;
    });

    // Re-synchronise both threads:
    t1.join().unwrap();
    t2.join().unwrap();
    println!("Result: {arc:?}");
}
```

Question 6)

If an `Arc` object is sufficient to provide multiple ownership and thread-safe access to data, why do we still need a `Mutex` guarding our data?

Lock-free data sharing with Atomics

Another way of making data *thread-safe* is by directly using *atomic* data structures. The Rust standard library provides a few of them in its `std::sync::atomic` module. The main difference with using *Mutexes* is that atomics are what we call *lock-free*: they rely on special CPU instructions that read and modify a value in a single indivisible step, so a thread never has to wait (or **sleep**) for a lock held by another thread. (Lock-based primitives such as *spinlocks* are themselves built on top of atomics.) In **real-time contexts**, where *thread sleep* is not an option, it is always preferable to use lock-free data structures.

```rust
use std::thread;
use std::sync::Arc;
use std::sync::atomic::AtomicI32;
use std::sync::atomic::Ordering;

fn main() {
    // We replace 'Mutex::new()' with the following:
    let arc = Arc::new(AtomicI32::new(32));
    let rc1 = Arc::clone(&arc);
    let rc2 = Arc::clone(&arc);

    let t1 = thread::spawn(move || {
        // Acquire underlying data as a copy:
        let mut v = rc1.load(Ordering::Acquire);
        // Modify the copy:
        v += 1;
        // Write it back. Warning: 'load' + 'store' is NOT atomic as a whole,
        // another thread may modify the value in between:
        rc1.store(v, Ordering::Release);
    });

    let t2 = thread::spawn(move || {
        // The correct way: a single atomic read-modify-write operation:
        rc2.fetch_add(2, Ordering::AcqRel);
    });
    // Re-synchronise both threads:
    t1.join().unwrap();
    t2.join().unwrap();

    println!("Result: {arc:?}");
}
```

Using *Message passing* to transfer data between threads

Another approach to multiple ownership and thread-safety in Rust would be using an `mpsc::channel` or `mpsc::sync_channel`, which are *asynchronous/synchronous FIFO* queues that store all the updated states of a value in a shared infinite buffer.

An `mpsc::channel()` call will return a tuple containing a handle to a `Sender` object and a `Receiver` object, which are by convention respectively named `tx` and `rx`. These two objects are positioned at each end of a FIFO which tunnels data between the two:

```rust
// Create a 'data channel' between a Sender `tx`, and a Receiver `rx`:
let (tx, rx) = std::sync::mpsc::channel::<i32>();

// Send the values `32` and then `16` through the channel:
tx.send(32).unwrap();
tx.send(16).unwrap();

// Poll the channel, read data if available:
println!("n = {}", rx.recv().unwrap());
println!("n = {}", rx.recv().unwrap());
```

Example of sending from a separate thread:

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

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

    // Do the same thing in a separate thread:
    let th = thread::spawn(move || {
        tx.send(32).unwrap();
        tx.send(16).unwrap();
    });

    th.join().unwrap();

    // Poll the channel, read data if available:
    println!("n = {}", rx.recv().unwrap());
    println!("n = {}", rx.recv().unwrap());
}
```

Or from multiple threads simultaneously:

```
let (tx, rx) = std::sync::mpsc::channel();
let mut vec = Vec::new();

for n in 0..8 {
    // Clone the Sender `tx` for each thread:
    let tx = tx.clone();
    vec.push(std::thread::spawn(move || {
        tx.send(n).unwrap();
    }));
}
for t in vec {
    t.join().unwrap();
}
for _ in 0..8 {
    // Consume the FIFO value-by-value:
    let value = rx.recv().unwrap();
    println!("Received value: {}", value);
}
```

Question 7)

Using an `mpsc::channel`, an `Arc<T>` and a `Mutex` (or an `Atomic`), implement a program which creates and run two independent threads:

- A *producer thread* which continuously counts from 0 to infinity.
- A *consumer thread* which continuously reads and prints the count produced by the producer thread.
- The two threads should run for 5 seconds and then stop.
- **Hint:** you can use `thread::sleep` to pause a thread for a certain amount of time.

Asynchronous programming

While programming with *threads* is a perfectly valid way of implementing concurrent programming, it also has a few disadvantages, such as having to rely on operating system scheduling, as well as sometimes making the code difficult to re-use or modify. To address these issues, programmers came up with a new way of structuring a program in a different set of **tasks** (whether they are independent, concurrent, or sequential), which has been called **asynchronous programming**.

Code within a thread is written in sequential style and the operating system executes them concurrently. With asynchronous programming, **concurrency happens entirely within a program**: the operating system is not involved, making *context switch* faster, and memory overhead also lower. By being natively integrated into a programming language, which is the case with Rust, it also makes control flow more flexible and expressive. For instance, a program using threads, like in the following example:

Futures and await: the Async syntax

- Rust relies on two keywords `async` and `await`, and a concept called ***futures*** to implement *asynchronous programming* within the language.
- A ***future*** is a placeholder data structure for a *future* value (that's not yet ready/accessible when declared and defined).
- It is implemented through the following trait:

```
pub trait Future {
    type Output;

    // Required method:
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>') -> Poll<Self::Output>;
}
```

- The `poll(..)` method is called in attempt to **resolve the future into a final value**.
- It is **non-blocking**: if the value is still not ready when polled, it will be rescheduled in the future.
- A `Poll` result is an enum with two states:
  - `Poll::Pending` : the value is still unresolved.
  - `Poll::Ready(value)` : value is now ready and accessible.

- Fortunately, Rust provides a way to make the syntax less verbose and more readable, using the `.await` suffix keyword:

```
let result = some_future_value.await;
```

- Which would be expanded to:

```
let result = match unsafe {
    Pin::new_unchecked(&mut some_future_value)
}.poll(cx) {
    Poll::Ready(val) => val,
    Poll::Pending => Poll::Pending
};
```

To explain the underlying process into a little more details:

- Future is passed to an **executor**
- Executor calls `poll(future, cx)` with its underlying `Context` instance
- If `Poll::Ready`, future is completed, value is retrieved
- If `Poll::Pending`, executor stops polling
- When external events occur (I/O readiness, timers, etc.), future is **woken up**
- Executor polls future again
- The process is repeated until we get `Poll::Ready`.

Async execution contexts

- `await` expressions are only valid within a certain execution context:
  - in `async` blocks or functions, which can be **interrupted** and **resumed**

```
async fn some_fn() {
    let result = some_future.await;
}
```

- Which would be translated to:

```
fn some_fn() -> impl Future<Output = ()> {
    async move {
        let result = some_future.await;
    }
}
```

Async runtimes

- `async/await` keywords and features are fully integrated to Rust.
- But ***Rust does not provide an `async` runtime (executor)!***
- The following does not compile:

```
async fn some_fn() {
    let result = some_future.await;
}

async fn main() {
    some_fn().await;
}
```

- Instead, Rust relies on **3rd-party implementations**:
  - `tokio` : most popular;
  - `smol` : more lightweight;
  - `embassy` : for embedded systems;
  - `glommio` : for I/O-bound workloads.
- For the rest of the TD, we will use `tokio` as our `async` runtime: `cargo add tokio --features full` (see TD6).

```rust
async fn some_fn() { /* async code... */ }

#[tokio::main]
async fn main() {
    some_fn().await;
}
```

Question 8)

- Try to implement the equivalent program with `async` and `await`:

```
use std::thread::{sleep, spawn};
use std::time::Duration;

fn count_to(max: i32) {
    for n in 1..=max {
        println!("{n}");
        sleep(Duration::from_secs(1));
    }
}
fn main() {
    let t1 = spawn(|| { count_to(5) });
    t1.join().unwrap();
    println!("joined!");
}
```

Spawning concurrent tasks

- For parallel tasks running concurrently, `tokio` provides the `spawn()` function, which takes a **future** (typically an `async` block) as argument.
- `async` blocks allow the `move` keyword, like normal closures.

```
async fn main() {
    tokio::spawn(async move {
        my_async_fn("first task").await;
    });
    tokio::spawn(async move {
        my_async_fn("second parallel task").await;
    });
}
```

Question 9)

- Try now to wrap our previous `count_to(5).await` function call inside an `async` block passed to `tokio::spawn`. Notice anything? What is the problem here? How can we fix it?

Question 10)

- Let's now improve our `count_to` function by setting the `sleep` duration as a variable parameter, and add a start offset as well, using for instance a `std::ops::Range` instead of the `i32` parameter `max`:

```
async fn count_to(range: Range<i32>, sleep_dur: Duration) {...}
```

- Implement the modified version of the function, and its usage inside the `main` function.
- Try to also spawn another function call with different parameters. Are the two tasks now running in parallel?

Shared states & Message passing

- State sharing & message passing for `async` is similar to threads
- But is also *runtime-dependent*.
- `tokio` provides a few primitives, such as async `Mutex` or `mpsc` channels (multiple-producer/single-consumer).
- Resources are available here:
  - https://tokio.rs/tokio/tutorial/shared-state
  - https://tokio.rs/tokio/tutorial/channels

Question 11)

- Transform the counter example by using `tokio`, with mutexes or channels, to share the count of one task with another.

Async and networking

- Networking is usually done with `async` in external Rust crates:
- No certainty on the arrival of inputs, and thus on the overall order of events
- `tokio` has in its libraries basic TCP/UDP primitives

```rust
use std::error::Error;
use tokio::{net::{TcpListener, TcpStream}};

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    let addr = "127.0.0.1:8047";
    // Bind a TCP listener to addr:
    let listener = TcpListener::bind(addr).await?;

    let streams = tokio::spawn(async move {
        // Connect to same addr:
        TcpStream::connect(addr)
            .await
            .unwrap();
        println!("Connected!");
    });
    // Wait for a client to connect:
    let (..) = listener.accept().await?;
    streams.await?;
    Ok(())
}
```

Question 12)

- With help from the example above and the `tokio` tutorial and documentation, write a simple *echo* example between a TCP listener and a single stream.

End of TD7