PLC 2024, Lecture 16, 14 March 2024

Concurrent programming in Rust

Passing functions

- In Haskell, we can pass functions --- e.g. `twice f x = f (f x)`
- In Java, we typically pass functions indirectly via an interface --- e.g. an object that implements `Comparable` will support a (customized) comparison function `cmp`

Closures

- Unlike Haskell, functions in Rust have internal variables that could capture the state of the context where they are defined
- A **closure** is a function definition with a context

Closures vs functions

- The examples below illustrate the syntacit difference between a function definition and a closure
- A closure is an anonymous function that can be assigned to a variable (last 3 examples below)
- Explicit type declarations are not required if the type can be inferred from context

In [2]:
```
{
    fn  add_one_v1   (x: u32) -> u32 { x + 1 }

    let add_one_v2 = |x: u32| -> u32 { x + 1 };
    println!("add_one_v2(7) is {}",add_one_v2(7));

    let add_one_v3 = |x| { x + 1 };
    println!("add_one_v3(17) is {}",add_one_v3(17));

    let add_one_v4 = |x| x + 1;
    println!("add_one_v4(27) is {}",add_one_v4(27));
}
```

```
add_one_v2(7) is 8
add_one_v3(17) is 18
add_one_v4(27) is 28
```

Out[2]: ()

- The inferred type should be consistent
- In the code below, the invocation of `example_closure` fixes the type of `x` as `String`

In [3]:
```
{
    let example_closure = |x| x;
    let s = example_closure(String::from("hello"));
    println!("{}",s);
}
```

```
hello
```

Out[3]: ()

- Here the type of `x` is (some variety) of integer

In [4]:
```
{
    let example_closure = |x| x;
    let n = example_closure(5);
    println!("{}",n);
}
```

```
5
```

Out[4]: ()

- If we invoke the same closure with two different types, we get an error

In [5]:
```
{
    let example_closure = |x| x;

    let s = example_closure(String::from("hello"));
    println!("{}",s);

    let n = example_closure(5);
    println!("{}",n);

}
```

```
[E0308] Error: mismatched types
   ╭─[command_5:1:1]
   │
 2 │     let example_closure = |x| x;
   │                            ┬
   │                            ╰── note: closure parameter defined here
   │
 7 │     let n = example_closure(5);
   │             ───────────────┬┬
   │                    ╰────────── arguments to this function are incorrect
   │                             ││
   │                             ╰── expected `String`, found integer
   │                              │
   │                              ╰── help: try using a conversion method: `.to_string
()`
───╯
```

Closures and context

- When `cl` is defined, `x` is `8`
- Before `cl` is invoked, `x` is redefined s `88`
- The closure uses the old value that was in its scope when it was defined

In [6]:
```
{
    let x = 8;
    let cl = |y| {x+y};
    let x = 88;
    let s = cl(7);
```

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

15

Out[6]: ()

- Another example
- The function `createclosure` returns a closure. We have to specify the return type. The return type is `FnMut()` which we have not seen --- look up the Rust documentation, this is not the main point of this example!
- Inside the function, we have a local mutable `counter` which is incremented by each call to the closure
- Note that we have to `move` the counter to the closure explicitly, just as we would in a function, for ownership to work correctly

In [7]:
```
fn createclosure() -> impl FnMut() {
    let mut counter = 0;
    let f = move || {counter = counter+1; println!("counter is {}",counter);};
    f
}
```

In [8]:
```
fn main() {
    let mut x = createclosure();
    for _i in 0..10 {
        x();
    }
}
```

In [9]:
```
main()
```

```
counter is 1
counter is 2
counter is 3
counter is 4
counter is 5
counter is 6
counter is 7
counter is 8
counter is 9
counter is 10
```

Out[9]: ()

Exercise: Implement an iterator using closures

- Closures behave like functions in terms of borrowing heap values

Example 1:

- Closure only reads the vector `list`, so borrowing suffices

In [10]:
```
fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    let only_borrows = || println!("From closure: {:?}", list);
```

```
    println!("Before calling closure: {:?}", list);
    only_borrows();
    println!("After calling closure: {:?}", list);
}
```

In [11]:
```
main()
```

```
Before defining closure: [1, 2, 3]
Before calling closure: [1, 2, 3]
From closure: [1, 2, 3]
After calling closure: [1, 2, 3]
```

Out[11]: ()

Example 2:

- If the closure changes the mutable variable, borrowing is not enough

In [12]:
```
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    let borrows_mutably = || list.push(7);

    borrows_mutably();
    println!("After calling closure: {:?}", list);
}
```

```
[E0596] Error: cannot borrow `borrows_mutably` as mutable, as it is not declared as mu
table
   ╭─[command_12:1:1]
   │
 5 │     let borrows_mutably = || list.push(7);
   │           │                  ──┬─
   │           ╰───────────────────────── help: consider changing this to be mutable: `
mut `
   │                               │
   │                               ╰─── calling `borrows_mutably` requires mutable bi
nding due to mutable borrow of `list`
   │
 7 │     borrows_mutably();
   │     ───────┬───────
   │            ╰──────── cannot borrow as mutable
   │
   │ Note: You can change an existing variable to mutable like: `let mut x = x;`
───╯
```

Example 3:

- If we only update, we can declare the closure to be mutable

In [13]:
```
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    let mut borrows_mutably = || list.push(7);

    borrows_mutably();
```

```
    println!("After calling closure: {:?}", list);
}
```

In [14]:
```
main()
```

```
Before defining closure: [1, 2, 3]
After calling closure: [1, 2, 3, 7]
```

Out[14]: ()

Example 4:

- In the example above, the final `println!` comes after the closure is used, so the mutable reference is no longer needed by the closure and `list` can be borrowed by `println!`
- Adding a `println!` between the definition of the closure and its invocation violates Rust's ownership rules

In [15]:
```
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    let mut borrows_mutably = || list.push(7);
    println!("After defining closure: {:?}", list);

    borrows_mutably();
    println!("After calling closure: {:?}", list);
}
```

```
[E0502] Error: cannot borrow `list` as immutable because it is also borrowed as mutabl
e
   ╭─[command_15:1:1]
   │
 5 │     let mut borrows_mutably = || list.push(7);
   │                               ─┬ ──┬─
   │                                ╰───────── mutable borrow occurs here
   │                                    │
   │                                    ╰───── first borrow occurs due to use of `list`
in closure
 6 │     println!("After defining closure: {:?}", list);
   │                                               ──┬─
   │                                                 ╰─── immutable borrow occurs here
   │
 8 │     borrows_mutably();
   │     ───────┬───────
   │            ╰───────── mutable borrow later used here
───╯
```

Defining threads

- In Java, threads are created using the `Thread` class and calling `start()`, which implicitly invokes `run()` (which must be defined because of the structure of `Thread`)
- In Rust, we *spawn* a thread by passing a closure
- There are functions to sleep etc, as usual

In [16]:
```
use std::thread;
use std::time::Duration;

fn main() {
```

```
    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..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

In [17]:
```
main()
```

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

Out[17]: ()

- Note that the spawned thread prematurely exited when the main function terminated
- We can wait for the thread to end using `join()`
  - The return value of `spawn` is stored in a variable, which is used to invoke `join()`
  - Note: You may have to restart the kernel to see the output show below

In [2]:
```
use std::thread;
use std::time::Duration;

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..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

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

In [3]:
```
main()
```

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
```

Out[3]: ()

- Wherever the `join()` occurs, the concurrent execution blocks
- The example below waits for the spawned thread to complete before executing the main thread

In [4]:

```
use std::thread;
use std::time::Duration;

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));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

In [5]:

```
main()
```

```
hi number 1 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
hi number 1 from the main thread!
hi number 2 from the main thread!
hi number 3 from the main thread!
hi number 4 from the main thread!
```

Out[5]: ()

- We have to be careful about lifetimes, as with normal functions

In [6]:
```rust
use std::thread;

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

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

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

```
[E0373] Error: closure may outlive the current function, but it borrows `v`, which is owned by the current function
   ╭─[command_6:1:1]
   │
 6 │ ╭─▶     let handle = thread::spawn(|| {
   │ │                                  ┬┬
   │ │                                  ╰── help: to force the closure to take ownership of `v` (and any other referenced variables), use the `move` keyword: `move `
   │ │                                   │
   │ │                                   ╰── may outlive borrowed value `v`
 7 │ │           println!("Here's a vector: {:?}", v);
   │ │                                             ┬
   │ │                                             ╰── `v` is borrowed here
 8 │ ├─▶     });
   │ │
   │ ╰───────────── note: function requires argument type to outlive `'static`
───╯
```

- For instance, the main thread could have "unset" the value of `v` using `drop(v)`

```rust
use std::thread;

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

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

    drop(v); // oh no!

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

- One solution is to `move` the vector to the closure

In [7]:
```rust
use std::thread;

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

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

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

In [8]:
```
main()
```

```
Here's a vector: [1, 2, 3]
```

Out[8]:
```
()
```

Coordinating threads

Message passing

- *"Do not communicate by sharing variables, instead share variables by communicating"*
- Send values via a channel
- By convention, *producer* sends messages on the channel and *consumer* receives them
- `mpsc` stands for *multiple producer, single consumer*
  - Many threads can write to the same channel, only one thread can read it

- Creating a channel returns a pair, handles to transmit (`tx` , below) and receive (`rx` , below)
- In this example, the spawned thread sends on `tx` , the main thread receives on `rx`

In [9]:
```
use std::sync::mpsc;
use std::thread;

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

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

In [10]:
```
main()
```

```
Got: hi
```

Out[10]:
```
()
```

- Sending a value `move` s it to the receiver
- In the example below, the spawned thread cannot refer to `val` after sending it to the main thread

In [11]:
```
use std::sync::mpsc;
use std::thread;

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

    thread::spawn(move || {
        let val = String::from("hi");
```

```
        tx.send(val).unwrap();
        println!("Sent: {}", val);
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

```
[E0382] Error: borrow of moved value: `val`
    ╭─[command_11:1:1]
    │
  8 │         let val = String::from("hi");
    │             ─┬─
    │              ╰─── move occurs because `val` has type `String`, which does not im
plement the `Copy` trait
  9 │         tx.send(val).unwrap();
    │                 ─┬─│
    │                  ╰──── value moved here
    │                    │
    │                    ╰── help: consider cloning the value if the performance cost i
s acceptable: `.clone()`
 10 │         println!("Sent: {}", val);
    │                              ─┬─
    │                               ╰─── value borrowed here after move
────╯
```

- It is permissible to print `val` before sending it

In [12]:
```
use std::sync::mpsc;
use std::thread;

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

    thread::spawn(move || {
        let val = String::from("hi");
        println!("Going to send: {}", val);
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

In [13]:
```
main()
```

```
Going to send: hi
Got: hi
```

Out[13]: ()

- A channel can have multiple senders (producers)
- Here we clone `tx` and pass `tx` to first spawned thread and `tx1` to second spawned thread
- The contents are received as some arbitrary interleaving

In [14]:
```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;
```

```
fn main() {

    let (tx, rx) = mpsc::channel();

    let tx1 = tx.clone();

    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));
        }
    });

    thread::spawn(move || {
        let vals = vec![
            String::from("more"),
            String::from("messages"),
            String::from("for"),
            String::from("you"),
        ];

        for val in vals {
            tx1.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

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

In [15]:
```
main()
```

```
Got: hi
Got: more
Got: messages
Got: from
Got: for
Got: the
Got: you
Got: thread
```

Out[15]: ()

- We cannot clone the receive handle

In [16]:
```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
```

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

    let tx1 = tx.clone();
    let rx1 = rx.clone();
}
```

```
[E0599] Error: no method named `clone` found for struct `std::sync::mpsc::Receiver` in the current scope
    ╭─[command_16:1:1]
    │
 10 │     let rx1 = rx.clone();
    │                  ──┬──
    │                    ╰──── method not found in `Receiver<_>`
────╯
```

Shared variables

- This is the "normal" way to communicate in Java etc
- Recall that we have to have a mechanism to avoid race conditions
- Rust provides `Mutex` for this
  - To share a variable "safely", wrap it a `Mutex`
  - Each `Mutex` is equipped with a lock
  - To access the variable, need to acquire the lock -- wait if it is not available
  - There is no `unlock()`! The lock is automatically released when the lock goes out of scope
  - Avoid typical pitfalls with forgetting to unlock, unlocking something that is not locked etc

In [17]:

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

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

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

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

In [18]:

```
main()
```

```
m = Mutex { data: 6, poisoned: false, .. }
```

Out[18]: ()

- Note that printing a `Mutex` gives extra information

- In the example above, the `lock()` was in an inner block
- In the example below, the lock is released when `main()` exits
- When we print `m`, it is still reported as `locked`

In [19]:

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

    let mut num = m.lock().unwrap();
```

```
    *num = 6;

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

In [20]:
```
main()
```

```
m = Mutex { data: <locked>, poisoned: false, .. }
```

Out[20]: ()

- How can we share a `Mutex` across threads?
- Ownership problem: can have only own owner for a `Mutex`

In [21]:
```
use std::sync::Mutex;
use std::thread;

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

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

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

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

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

```
[E0596] Error: cannot borrow `num` as mutable, as it is not declared as mutable
    ╭─[command_21:1:1]
    │
 10 │             let num = counter.lock().unwrap();
    │                 │
    │                 ╰── help: consider changing this to be mutable: `mut `
    │
 12 │             *num += 1;
    │              ─┬─
    │               ╰──── cannot borrow as mutable
    │
    │ Note: You can change an existing variable to mutable like: `let mut x = x;`
────╯
```

```
[E0382] Error: borrow of moved value: `counter`
    ╭─[command_21:1:1]
    │
  5 │     let counter = Mutex::new(0);
    │         ───┬───
    │            ╰───── move occurs because `counter` has type `Mutex<i32>`, which doe
s not implement the `Copy` trait
    │
  9 │         let handle = thread::spawn(move || {
    │                                    ───┬───
    │                                       ╰───── value moved into closure here, in p
revious iteration of loop
    │
 21 │     println!("Result: {}", *counter.lock().unwrap());
    │                             ───┬───
    │                                ╰───── value borrowed here after move
────╯
```

Reference counting

- Main motivation for single ownership is to avoid problems when heap storage is released
- If `l1` and `l2` both refer to the same list and we "drop" `l2`, the value `l1` becomes undefined
- One way to deal with this is **reference counting**
  - When we assign a variable to point to a chunk of heap storage, set reference count to one
  - When we add a new reference to same storage, increment reference count
  - When we "drop" a reference, decrement reference count
  - Release storage only when reference count becomes 0

- Rust allows us to explicitly use reference counting
- Simplest version in concurrent programming context is to wrap the value in `Arc`
  - `Arc` stands for *Atomic reference counter*
  - Combines reference counting with atomic updates, making the contents safe to share across threads
- Below, we wrap clone `Mutex` within an `Arc` and create cloned `Arc` references within each thread

In [22]:

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

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());
}
```

In [23]:
```
main()
```

```
Result: 10
```

Out[23]: ()

Race conditions

- Rust is designed to *prohibit* race conditions in normal code
- Ownership, lifetimes etc ensure this