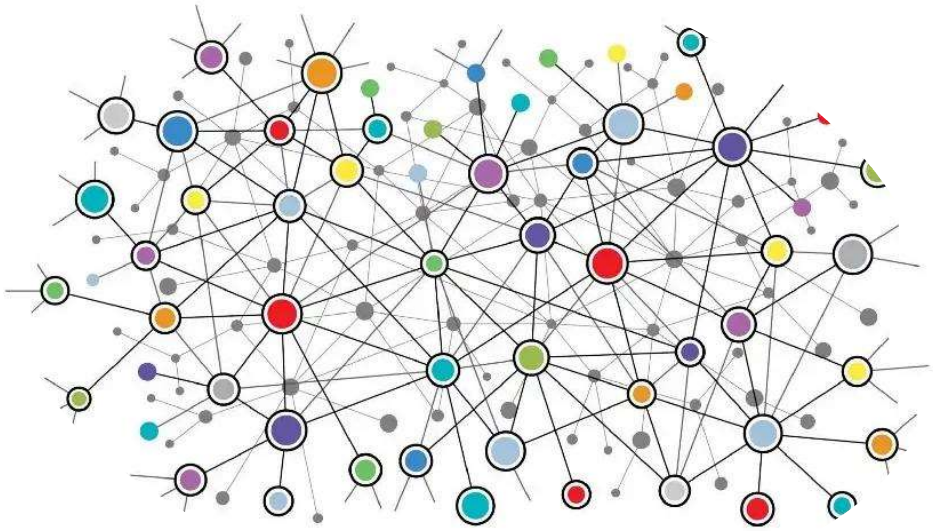


Concurrency

Mutlithreading

- Improved performance, but added complexity
- Must consider:
  - Race conditions
  - Deadlocks
  - Hard to reproduce and fix bugs, etc.

Creating Threads

- The Rust standard library uses OS threads (1:1 threading).

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
{
    Builder::new().spawn(f).expect("failed to spawn thread")
}
```


Creating Threads (cont'd)

```
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));
    }
}
```

Waiting for All Threads to Finish Using join Handles

thread::spawn returns JoinHandle

```
fn main() {
    let handle: JoinHandle<()> = thread::spawn(|| {
        for i in 1..10 { … }
    });

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

    handle.join().expect("Couldn't join on the associated thread");
}
```

waits for its thread to finish

```
fn join(self) -> Result<T> { … }
```


What seems to be a problem?

Because I'm not going to mutate v inside closure, maybe it's ok to just borrow v. But …

```
let v = vec![1, 2, 3];

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

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

- The closure tries to borrow v via reference.
- However, Rust can't tell how long the spawned thread will run, so it doesn't know if the reference to v will always be valid.

```
error[E0373]: closure may outlive the current function, but it borrows `v`,
which is owned by the current function
 --> src/main.rs:6:32
  |
6 |     let handle = thread::spawn(|| {
  |                                ^^ may outlive borrowed value `v`
7 |         println!("Here's a vector: {:?}", v);
  |                                           - `v` is borrowed here
  |
note: function requires argument type to outlive `'static`
 --> src/main.rs:6:18
  |
6 |       let handle = thread::spawn(|| {
  |  __________________^
7 | |         println!("Here's a vector: {:?}", v);
8 | |     });
  | |_____^
help: to force the closure to take ownership of `v` (and any other referenced
variables), use the `move` keyword
  |
6 |     let handle = thread::spawn(move || {
  |                                ++++
```


Using move Closures with Threads

- Adding the `move` keyword before the closure forces the closure to *take ownership of the values* it's using.

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

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

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

Why does this code not compile?

```
fn main() {
    let str = "What's that?";

    let handle = thread::spawn(|| {
        println!("str = {str}");
    });

    handle.join();
}
```


Placing move before closure solves the problem

```
fn main() {
    let str = "What's that?";

    let handle = thread::spawn(move || {
        println!("str = {str}");
    });

    handle.join();
}
```

- But, what if we can guarantee that borrowing threads finish before we use the borrowed variables?

Scoped Threads (as of Rust 1.63)

- The problem is that when using threads we don't know exactly when the closure we give to `thread::spawn` will run.
- Meaning it is hard to determine when variables borrowed by threads will be dropped. That could lead to *use after free bugs*.

"Scoped threads" ensure threads are joined before the Scope exits.

Scoped Thread (cont'd)

- To use scoped threads we first need to create a `Scope` for the scoped threads to operate in.

```
use std::thread::scope;

fn main() {
    let str = "What's that?";

    thread::scope(|s: &Scope| {
        s.spawn(|| {
            println!("str = {str}");
        });
    });
}
```

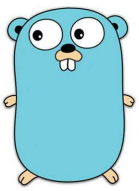
Scoped threads are guaranteed to be joined before the Scope exits

```
fn guarantee_to_join_before_scope_exits() {
    let mut i = 0;

    thread::scope(|s| {
        s.spawn(|| {
            thread::sleep(Duration::from_millis(1000));
            i += 1;
        });
    }); // thread is joined here

    println!("i = {i}"); // it's safe to use variable i
}
```


Shared Mutable State

"A widely accepted method of communication is by inspection and updating of a common store … However, this can create severe problems in the construction of correct programs and it may lead to expense and … unreliability (e.g., glitches) …"

– Tony Hoare, CSP 1978

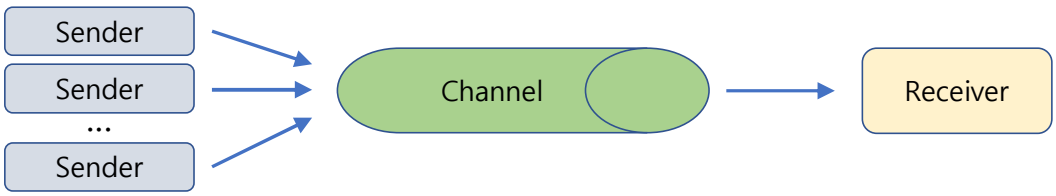


"Don't communicate by sharing memory;
instead,
share memory by communicating"

mpsc Channels

- Threads can communicate with each other via *channels*.

- Multiple producers and a single consumer can be associated with a channel (mpsc model).



Creating Channels

- Create a new channel using the `mpsc::channel` function.
  - **mpsc** stands for multiple producer, single consumer.
- A channel is said to be *closed* if either the transmitter or receiver half is dropped.

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

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

transmitter

receiver

Single Producer Single Receiver

```rust
use std::sync::mpsc::{ self, Sender, Receiver };
use std::thread;

fn main() {
    let (tx, rx): (Sender<String>, Receiver<String>) = mpsc::channel();

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

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



```
error[E0382]: borrow of moved value: `val`
  --> src/main.rs:10:31
   |
8  |         let val = String::from("hi");
   |             --- move occurs because `val` has type `String`, which does not implement the
`Copy` trait
9  |         tx.send(val).unwrap();
   |                 --- value moved here
10 |         println!("val is {}", val);
   |                               ^^^ value borrowed here after move
   |
```

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

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

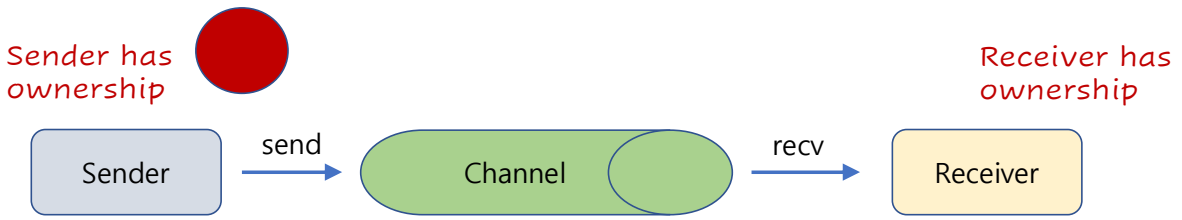
val was already moved ☹

Channels and Ownership Transference

- The ownership rules play a vital role in message sending because they help you write safe, concurrent code.
- The `send` function takes ownership of its parameter, and when the value is moved, the receiver takes ownership of it.
- This stops us from accidentally using the value again after sending it; the ownership system checks that everything is okay.

Channels and Ownership Transference

- The ownership rules play a vital role in message sending because they help you write safe, concurrent code.



- This stops us from accidentally using the value again after sending it; the ownership system checks that everything is okay.

Creating Multiple Producers by Cloning the Transmitter

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

let tx1 = tx.clone();

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

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

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

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

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


Channel vs. sync_channel

- The `recv` method for both channel types block until a message is received.
- In Channel, calls to `send` do not block the sending thread.

```
let (sender, receiver) = mpsc::channel();
sender.send(2); // Does not block, even if no receiver handles value
```

- A sender created with `sync_channel` may block. When we call `sync_channel` we specify a queue size. If the queue is full a call to `send` will block until space on the queue is available.

```
use std::sync::mpsc::{ self, Sender, Receiver };

let (sender, receiver): (Sender<i32>, Receiver<i32>) = mpsc::channel();
let (sender, receiver) = mpsc::sync_channel::<String>(1); // queue size
```

Shared State Concurrency

- While channels are similar to single ownership, shared memory concurrency is like multiple ownership.
- Smart pointers made multiple ownership possible.
- But, multiple ownership can add complexity because these different owners need managing.
- Management of mutexes can be incredibly tricky to get right.
- Rust's type system and ownership rules greatly assist in getting this management correct.

Data Races

A data race is defined to occur when two distinct threads access the same memory location, where

- at least one of them is a write, and

- there is no synchronization mechanism that enforces an ordering on the accesses.

Using Mutexes to Allow Access to Shared Data

- A mutual exclusion primitive useful for protecting shared data
- Unlike C++, the value we want to protect is literally inside the mutex.
  - The **mutex** is described as **guarding the data** it holds via the locking system.
- The data can only be accessed through `MutexGuard` returned from `lock` and `try_lock`.

```
pub struct Mutex<T: ?Sized> {
    inner: sys::Mutex,
    poison: poison::Flag,
    data: UnsafeCell<T>,
}
```

```
pub struct MutexGuard<'a, T: ?Sized + 'a> {
    lock: &'a Mutex<T>,
    poison: poison::Guard,
}
```


The API of Mutex<T>

```
// Create a mutex that holds an i32.
let m: Mutext<i32> = Mutex::new(5);

{
    // To access data inside mutex, use `lock` to acquire the lock.
    // This call will block the current thread until it has the lock.
    // The call to lock returns a smart pointer called `MutexGuard`, wrapped
    // in a `LockResult` that we handled with the call to `unwrap`.
    let mut num: MutexGuard<i32> = m.lock().unwrap();

    // `MutexGuard` smart pointer implements `Deref` to point at our inner data;
    *num = 6;
} // The smart pointer also has a `Drop` implementation that releases the lock
  // automatically when a `MutexGuard` goes out of scope

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

}
```

C++ vs. Rust

```cpp
int cplus_mutex() {
  int counter = 0;

  std::mutex mtx;
  std::vector<std::thread> handles;

  for (int j = 0; j < 10; ++j) {
    handles.push_back(
      std::thread([&mtx, &counter]() {
        std::scoped_lock lk(mtx);
        counter += 1;
    }));
  }

  for (auto &handle : handles) {
    handle.join();
  }

  std::cout << counter;
}
```

protect codes

protect data

```rust
fn rust_mutex() {
  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!("{}", *counter.lock().unwrap());
}
```

What's wrong?

value moved into closure here, in previous iteration of loop

```rust
// Wrong ver. 1
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);
}
```

Rc<T> cannot be used across thread boundaries since reference counting operation is not atomic.

```rust
// Wrong ver. 2
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);
}
```

Arc<Mutex<T>> for Multithreading

- `std::sync::Arc<T>` is an *atomically reference counted smart pointer* type, i.e., thread-safe equivalent of `Rc<T>`.
- `std::sync::Mutex<T>` provides *interior mutability,* as the `Cell`/`RefCell` do.
- Use `Rc<RefCell<T>>` in sequential setting
- Use `Arc<Mutex<T>>` in concurrent setting

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

fn share_mutex_between_threads() {
  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!("{}", *counter.lock().unwrap());
}
```


Extensible Concurrency with the Sync and Send Traits

- Rust language has very few concurrency features.
- Almost every concurrency feature we've talked about so far has been *part of the standard library*, not the language.
- You can write your own concurrency features or use those written by others, if necessary.
- However, two concurrency concepts are embedded in the language: the `std::marker::{Sync, Send}` traits.
- Implementing `Send` and `Sync` manually is unsafe.

Trait std::marker::Send

- Types that can be transferred across thread boundaries.
- Send types
  - most primitive types,
  - Arc
  - Cell, Mutex
- Non-Send types
  - Rc, MutexGuard

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


Allowing Transference of Ownership Between Threads with Send

- The Send marker trait indicates that ***ownership*** of values of the type implementing Send ***can be transferred*** between threads.
- Almost every Rust type is Send, but there are some exceptions, including Rc<T>.

```
impl<T: ?Sized> !Send for Rc<T> {}
impl<T: ?Sized> !Send for MutexGuard<'_, T> {}
```

- Almost all primitive types are Send aside from raw pointers.

```
impl<T: ?Sized> !Send for *const T {}
impl<T: ?Sized> !Send for *mut T {}
```

- Any type composed entirely of Send types is automatically marked as Send as well.

Trait std::marker::Sync

- Types for which it is safe to share references between threads.

  A type T is Sync if and only if &T is Send.

- Sync types
  - Primitive types like u8, f64 etc. and simple aggregate types like tuples, enums, and structs containing them
  - &T, &mut T, Box<T>, Vec<T> and most other collection types (if T is Sync)
  - Generic parameters need to be Sync for their container to be Sync
  - Arc, Mutex and RwLock, and atomic data types
- Non-Sync types
  - Rc, Cell, RefCell

Allowing Access from Multiple Threads with Sync

- The `Sync` indicates that it is safe for the type implementing `Sync` to be referenced from multiple threads.
- In other words, any type `T` is `Sync` if `&T` (an immutable reference to `T`) is `Send`, meaning the reference can be sent safely to another thread.
- Similar to `Send`, primitive types are `Sync`, and types composed entirely of types that are `Sync` are also `Sync`.
- `Rc<T>`, `RefCell<T>`, and the family of related `Cell<T>` types are not `Sync`.
- `Mutex<T>` is `Sync`.

Summary

- The Rust standard library provides channels for message passing and smart pointer types, such as `Mutex<T>` and `Arc<T>`, that are safe to use in concurrent contexts.
- The type system and the borrow checker ensure that the code using these solutions won't end up with data races or invalid references.
- Once you get your code to compile, you can rest assured that it will happily run on multiple threads without the kinds of hard-to-track-down bugs common in other languages.
- Concurrent programming is no longer a concept to be afraid of: go forth and **make your programs concurrent, fearlessly!**