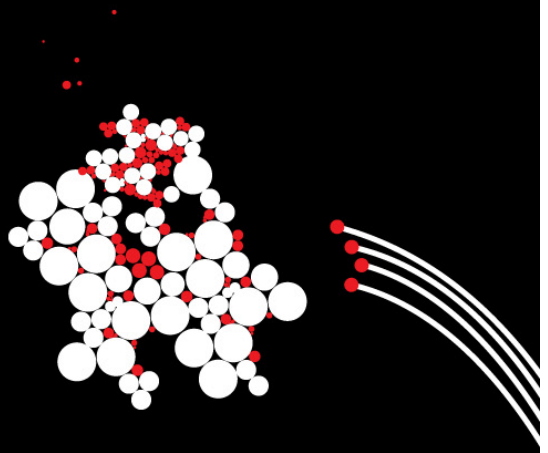
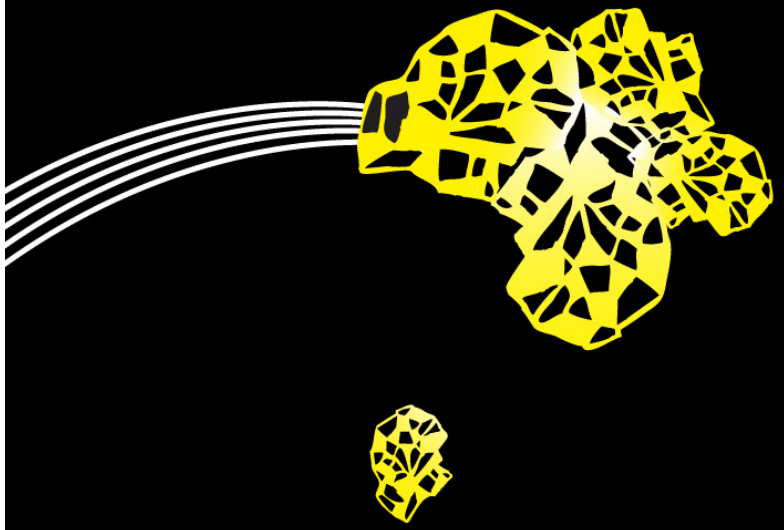
UNIVERSITY OF TWENTE.

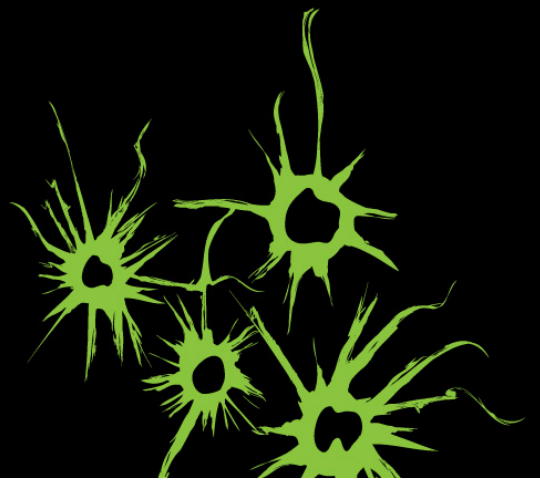


SAFE CONCURRENCY

PROGRAMMING PARADIGMS

3 JUNE 2019





CONCURRENT PROGRAMMING

CONTENTS

2	Tue 30/4	Basics of concurrency, thread safety, testing concurrent systems	Ch. 1-3, 12 (+ SS material)
3	Tue 7/5	Synchronisation	Ch. 4, 5, 13, 14.1-14.4
4	Wed 15/5	Liveness, performance, and fairness	Ch. 10,11
5	Thu 23/5	Homogeneous threading (OpenMP, OpenCL)	Papers
6	Mon 3/6	Safe concurrency (Software Transactional Memory, Rust)	Papers
7	Fri 7/6	Fine-grained concurrency, memory models	Ch. 14.5-6, 15, 16

TODAY'S LECTURE

- Quiz on homogeneous parallelism
- Safe concurrency
  - Software Transactional Memory
  - Rust: safe shared memory concurrency
  - Message passing concurrency (Go, Rust)

Material: papers on Canvas

- Recommended: Paper on Software Transactional Memory
- Two chapters from The Rust language guide (second edition)
- Recommended: Chapter on concurrency from Effective Go

QUESTION 1

```
//@ requires a.length == b.length
for (int i = 0; i < a.length; i++) {
    if (i == 0) {
        b[i] = a[i];
    } else {
        b[i] = a[i] + a[i-1]
    }
}
```

Can this loop be parallelised?

a. Yes, no problem
b. Only if you insert synchronisation between reading of a[i] and a[i -1]
c. No, parallelisation would change the behaviour of the loop
d. Yes, but you need to do a code transformation first

QUESTION 2

```
#pragma omp parallel for default(none) shared(a,c) private(i)
for (int i = 0; i < a.length; i++) {
            a[i] = i + 3;
            if (i > 0) {
                c[i] = a[i - 1] + 2*i;
            }
}
```

What happens if the compiler uses this directive for parallelisation

a. Nothing, this code will not be parallellised
b. Program behaviour changes
c. Everything works fine
d. Compiler returns an error message

QUESTION 3

```
for (i=0; i<a.length/100; i++) {
    int sumL = 0;
    for (j = 0; j < 100; j++) {
        sumL += a[j];
    }
    sum += sumL;
}
```

How could we parallelise the outer loop?

a. You cannot parallelise this
b. Execute all loop iterations in parallel
c. Execute all instructions in lockstep
d. Addition to sum needs to be protected in a critical section

QUESTION 4

```
__kernel(__global float* a) {
        int tid = get_global_id(0);
        a[(tid + 1) % get_global_size()] = a[tid];
}
```

What is the problem with this kernel?

a. It only updates a single element in the array
b. It has a data race
c. The array is accessed out of bounds
d. Deadlock

QUESTION 5

```
__kernel(__global float* a) {
        int tid = get_global_id(0);
        int temp = a[tid];
        barrier(CLK_GLOBAL_MEM_FENCE);
        a[(tid + 1) % get_global_size()] = temp;
}
```

What does this kernel do?

a. It copies the first element of the array to the second position
b. It blocks on the barrier
c. It shifts all elements in the array one position to the right
d. It leaves array a unchanged

SOFTWARE TRANSACTIONAL MEMORY

- Optimistic concurrency control
- Analogous to database transactions
- Transaction:
  - Series of reads and writes to shared memory
  - All actions are logged
- After transaction completed:
  - No concurrent updates: commit
  - Otherwise: abort and retry

Origin of ideas:
Hardware Transactional Memory

- Implemented in many different programming languages

Resemblance to lock-free programming (next block)

WHY SOFTWARE TRANSACTIONAL MEMORY

- Advantages:
  - Increased concurrency
  - Non-blocking
  - Reduces number and size of the critical sections
  - Less prone to deadlock and livelock

- Disadvantages:
  - Performance from overhead of logging and commit
  - Actions need to be undo-able
  - Risk of starvation

IMPLEMENTATION: INTUITION

- Start transaction
- Make snapshot of state
- Do reads and writes on snapshot
- CompareAndSwap of initial and current snapshot
- If CompareAndSwap succeeds: commit
- Else: retry

IMPLEMENTATION PSEUDOCODE: START & ABORT

```
Start(code) {
    start := cntr;
    start_pc := first program counter code
}

Abort {
    clear(rd-set);
    clear(wr-set);
    re-execute from start_pc
}
```

dirty array: all entries set to 0

IMPLEMENTATION PSEUDOCODE: WRITE

```
Read(addr) {
    if contains(wr-set, addr)
    then get(wr-set, addr);
    val := read from addr;
    if dirty[hash(addr)] > start
    then Abort;
    add(rd-set, addr);
    return val
}
```

```
Write(addr, val) {
    if dirty[hash(addr)] > start
    then Abort;
    dirty[hash(addr)] := cntr;
    put(wr-set, addr, val)
}
```

IMPLEMENTATION PSEUDOCODE: COMMIT

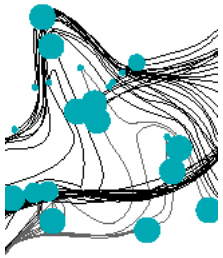
```
Commit {
    if empty(wr-set) then return;
    if (not Validate) then Abort;
    cntr := cntr + 1;
    foreach(addr, val) in wr-set do
        dirty[hash(addr)] := cntr;
        write val to addr;
    cntr := cntr + 1;
}
```

```
Validate {
    if cntr <= start then return true;
    for addr in rd-set or wr-set do
        if dirty[hash(addr)] > start
        then return false
    return true
}
```

TRANSACTION

```
Transaction(code) {
    Start();
    code;  -- using Read and Write operations
    Commit()
}
```

```
Transaction(
        acc1.add(-value);
        acc2.add(value);
)
```



SAFE SHARED STATE CONCURRENCY

WHAT IS RUST?

- Compiled language
- Goal: early detection of programming errors
- Type system ensures
  - Memory safety
  - Threads without data races

VARIABLES IN RUST

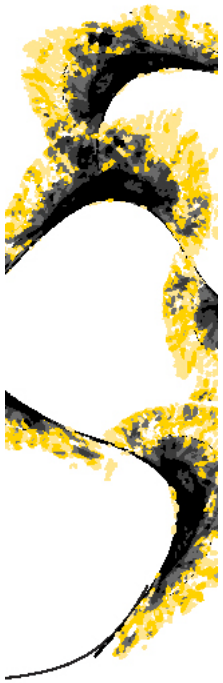
- Immutable by default
  - Guarantees about the value for free
  - Cannot be changed in another part of the code by accident

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

Compiler error: x = 6;
re-assignment of immutable variable

- Variables can be declared to be mutable
  - Explicit mutability intent

Change declaration of x to
let mut x = 5;



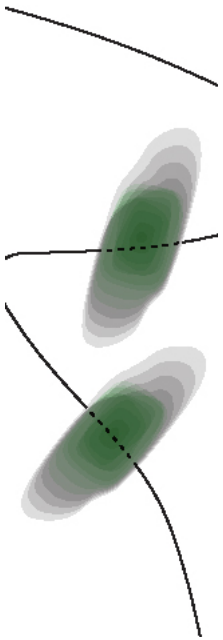
OWNERSHIP

Basic rules

- Each value in Rust has a variable that's called its *owner*
- There can only be one owner at a time
- When the owner goes out of scope, the value will be dropped

```
let  s1 = String::from("hello");
let s2 = s1;
println!("{}", s1);
```

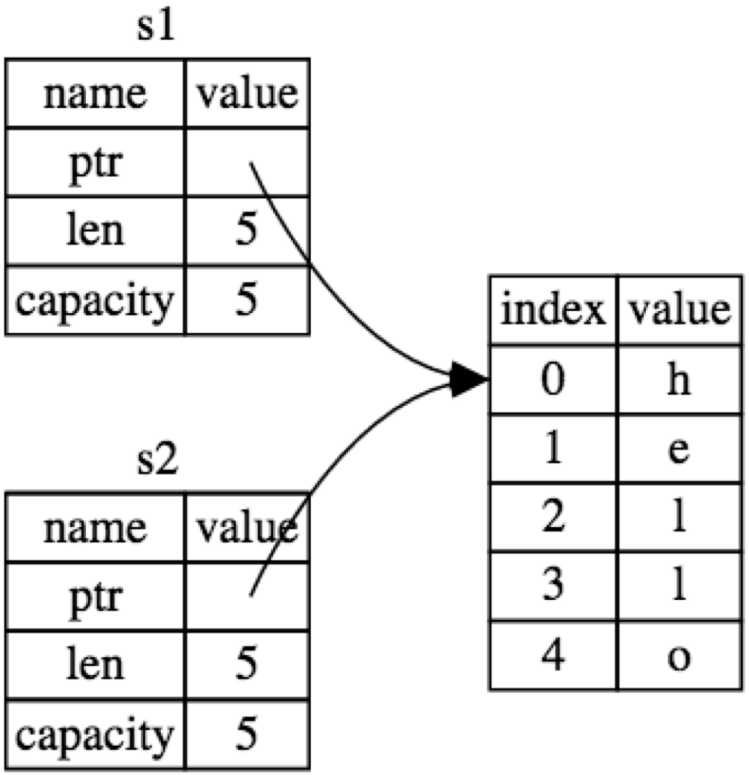
```
error[E0382]: use of moved value: `s1`
 --> src/main.rs:4:27
  |
3 |     let s2 = s1;
  |         -- value moved here
4 |     println!("{}, world!",s1);
  |                           ^^ value used here after move
  |
  = note: move occurs because `s1` has type `std::string::String`,
which does not implement the `Copy` trait
```



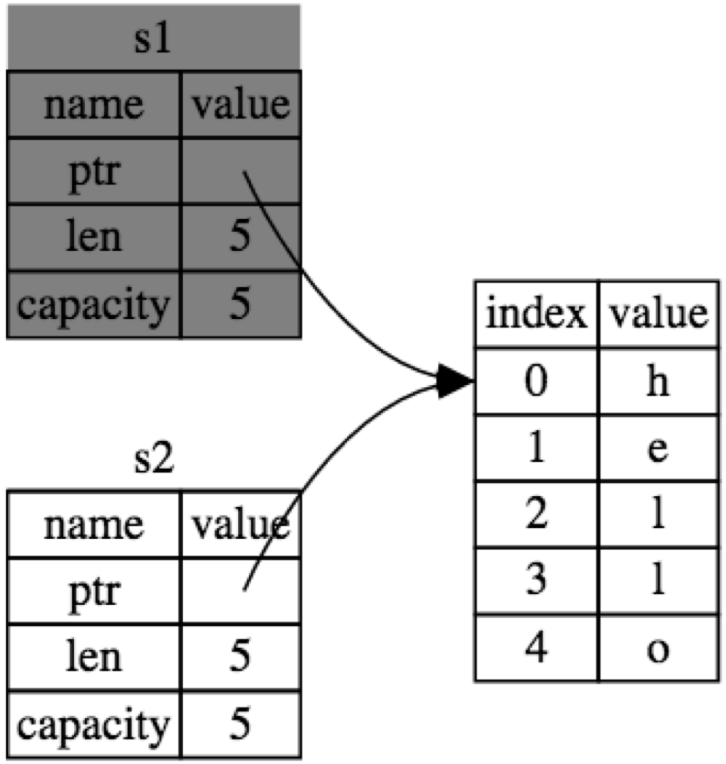
COPY VS. MOVE SEMANTICS

Advantage: easy memory clean up

Copy semantics

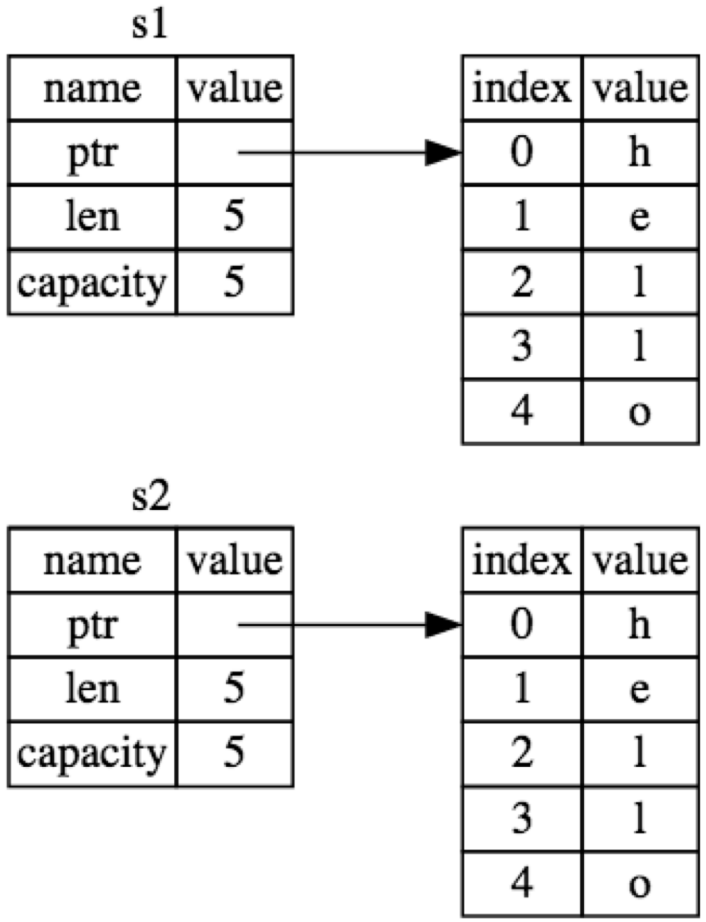


Rust's move semantics



CLONE: AVOIDING MOVE

```
let  s1 = String::from("hello");
let  s2 = s1.clone();
println!("s1 = {}, s2 = {}", s1, s2);
```



COPYABLE TYPES

- Primitive types
- Arrays
- Tuples of copyable types
- …

Vector:
Non-copyable/mutable version of sequences

```
let x = 5;
let y = x;
println!("x = {}, y = {}", x, y);
```

FUNCTIONS, ARGUMENTS AND OWNERSHIP

```
let s = String::from("hello");      // s comes into scope
takes_ownership(s);                 // s's value moves into the function
                                    // s no longer valid here
let x = 5;                          // x comes into scope
makes_copy(x)                       // i32 is copyable, thus okay to still use x
x = 6;
```

```
fn takes_ownership(some_string : String)
{
        println!("{}", some_string);
}
```

```
fn makes_copy(some_int : i32) {
        println!("{}", some_int);
}
```

- In both functions, argument comes into scope
- takes_argument drops some_string at end of body
- makes_copy: some_integer out of scope, nothing else

RETURN VALUES AND OWNERSHIP

```
let s1 = gives_ownership();          // moves return value into s1
let s2 = String::from("hello");      // s2 into scope
let s3 = takes_and_gives_back(s2);   // s2 moved into takes_and_gives_back
                                     // return value moves into s3
```

```
fn gives_ownership() -> String {
    let some_string = String::from("hello");
    some_string;
}
```

Both functions: return value moves to caller

Commonly used: borrowing

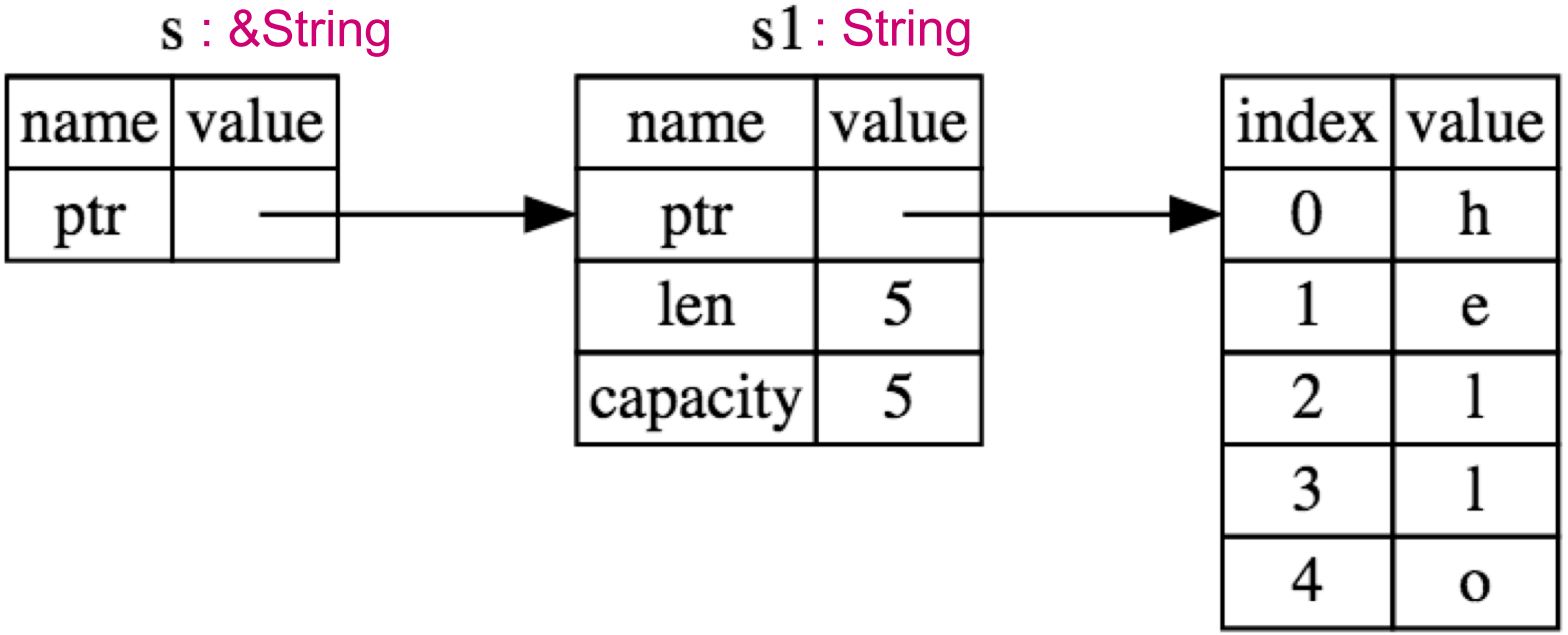
```
fn takes_and_gives_back(a_string : String) -> String {
    a_string
}
```

BORROWING

```
let s1 = String::from("hello")
let len = calculate_length(&s1);
println!("The length of '{}' is {}.", s1, len);
```

```
fn calculate_length(s: &String) -> usize {
    s.len
}
```

REFERENCES IN MEMORY



BORROWING PREVENT UPDATES

```
let s = String::from("hello");
change(&s);
```

```
fn change(some_string : &String) {
    some_string.push_str(", world");
}
```

```
error: cannot borrow immutable borrowed content `*some_string` as mutable
 --> error.rs:8:5
  |
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^
```

MUTABLE REFERENCES

```
let mut s = String::from("hello");
change(&mut s);
```

```
fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

But this solution has restrictions:
Only one mutable reference to a particular piece of data in a particular scope

RESTRICTION ON MUTABLE REFERENCES

```
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;
```

Restricting aliasing

This guarantees data race freedom and avoids dangling pointers!

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> borrow_twice.rs:5:19
  |
4 |     let r1 = &mut s;
  |                   - first mutable borrow occurs here
5 |     let r2 = &mut s;
  |                   ^ second mutable borrow occurs here
6 | }
  | - first borrow ends here
```

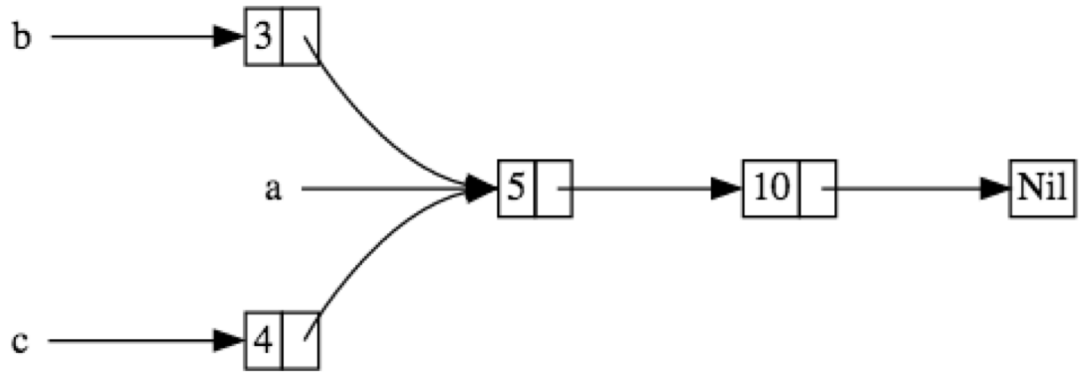
RC TRAIT: MULTIPLE OWNERS

- Data on the heap that can be read by multiple users
- Rc: reference count – keep track of number of users reading it
- If a : Rc<T> then a.clone() increases reference count
- If a goes out of scope, reference count is decreased

RC EXAMPLE

```
enum List {
        Cons(i32, Rc<List>,
        Nil
}
```

```
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil))))); // count = 1
    let b = Cons(3, a.clone());                                // count = 2
    let c = Cons(4, a.clone());                                // count = 3
}                                  // count = 0 (all out of scope)
```



CONCURRENCY IN RUST

Fearless concurrency

- Many common concurrency errors detected during compilation
- Threads
  - Default physical thread – logical thread: 1-1
  - But other options possible
- Message passing concurrency
- Shared state concurrency

THREAD'S SPAWN AND JOIN

```
use std::thread;
fn main() {
    let handle = thread::spawn(|| {
        for i in 1 .. 10 {
            println!("hi number {} from the spawned thread", i);
        }
    });
    for i in 1 .. 5 {
        println!("hi number {} from the main thread", i);
    }
    handle.join();
}
```

Closure notation

Threads can return a value, this is returned by join

MOVE CLOSURE

```
use std::thread;
fn main() {
    let v = vec![1, 2, 3];
    let handle = thread::spawn(move || {
        println!("here's a vector: {:?}", v);
    });
    handle.join();
}
```

- Thread needs to own v
- move keyword transfers ownership

Shared mutable state is the root of all evil. Most languages attempt to deal with this problem through the 'mutable' part, but Rust deals with it by solving the 'shared' part.

SHARED MEMORY CONCURRENCY

```
fn main() {
    let mut data = vec![1, 2, 3];
    for i in 0 .. 3 {
        thread::spawn(move || {
            data[0] += i;
        });
    }
}
```

```
8:17 error: capture of moved value: `data`
        data[0] += i;
        ^~~~
```

- Data moved to thread 0
- Is not available anymore when other threads are created

SHARED MEMORY CONCURRENCY

```
fn main() {
    let mut data = Rc::new(vec![1, 2, 3]);
    for i in 0 .. 3 {
        let data_ref = data.clone();
        thread::spawn(move || {
            data_ref[0] += i;
        });
    }
}
```

- Clone data
- Keep reference count

Rc is not threadsafe!

Variation: atomic reference count

```
13:9: 13:22 error: the trait bound `alloc::rc::Rc<collections::vec::Vec<i32>> : core::marker::Send`
            is not satisfied
...
13:9: 13:22 note: `alloc::rc::Rc<collections::vec::Vec<i32>>`
            cannot be sent between threads safely
```

SHARED MEMORY CONCURRENCY

```
fn main() {
    let mut data = Arc::new(vec![1, 2, 3]);
    for i in 0 .. 3 {
        let data = data.clone();
        thread::spawn(move || {
            data[0] += i;
        });
    }
}
```

Atomic reference count (Arc) is thread safe

Multiple references to same location: location is read-only
Solution: Mutex

```
<anon>:11:24 error: cannot borrow immutable borrowed content as mutable
<anon>:11                       data[0] += i;
                                ^~~~
```

SHARED MEMORY CONCURRENCY

```
fn main() {
    let mut data = Arc::new(Mutex::new(vec![1, 2, 3]));
    for i in 0 .. 3 {
        let data = data.clone();
        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            data[0] += i;
        });
    }
}
```

- Mutex
- Call lock on the data
- Unlocking when the variable goes out of scope

SOME OBSERVATIONS

- Mutex is part of the type of a shared variable
  - Makes it explicit which data is protected by the lock
- Multiple variables protected by the same lock need to be combined in a struct
- Passing locks to multiple threads (or keeping a copy in the main thread) requires cloning of the lock

MESSAGE PASSING CONCURRENCY

Do not communicate by sharing memory;
instead, share memory by communicating

MESSAGE PASSING CONCURRENCY

- Avoid all problems caused by shared memory concurrency
- Pass shared values through channels
- Only one thread/process/go-routine/… has access to a value at any given time
- Communication is the synchronizer

- Example: Unix pipelines

- Supported by wide variety of languages: Go, Rust, Scala, Erlang, …
- Theoretical model: Communicating Sequential Processes (CSP)

CHANNELS

- Communication via sends and receives
- Receive: blocking operation
- Send: different options
  - Synchronous communication – blocking
  - Asynchronous communication – non-blocking

- Bounded channel:
  - Maximum number of messages pending in channels
  - Send blocks when maximum is reached (until a message is read)

SIMPLE SYNCHRONISATION VIA CHANNELS

```
channel<int> c;
```

Process 1

```
doSomething();
c.send("done");
```

Process 2

```
doSomethingElse();
c.receive(_);
continue();
```

SEMAPHORE USING BOUNDED CHANNEL

channel<int, MaxOutstanding> sem

channel<Task> queue

Process Server

queue.receive(t);

sem.send(1);

create Execute(t)

Process Execute(t)

// Execute t

sem.receive(1);

Ensures at most MaxOutstanding Execute processes are in progress

CHANNELS OF CHANNELS

Framework for remote procedure calls in Go

```
type Request struct {
    args        []int
    f           func([]int) int
    resultChan  chan int
}
```

Client

```
func sum(a []int) (s int) {
    for _, v := range a {
        s += v
    }
    return
}

request := &Request{[]int{3, 4, 5}, sum, make(chan int)}
// Send request
clientRequests <- request
// Wait for response.
fmt.Printf("answer: %d\n", <-request.resultChan)
```

Create request

Server

```
func handle(queue chan *Request) {
    for req := range queue {
        req.resultChan <- req.f(req.args)
    }
}
```

- Apply function
- Send result on channel

MESSAGE PASSING IN RUST: INTERACTION WITH OWNERSHIP

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

fn main() {
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
        println!("val is {}", val);
    });
    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Sending a value transfers ownership

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

CONCLUSIONS

- Functional programming
  - Natural parallel execution
  - Par construct: potential parallel execution
  - Transactional programming
- Safe shared memory concurrency
  - Ownership ensures traceability of pointers
  - Memory safety: always at most 1 point in the program where a memory location may be changed
  - Ownership guarantees also help to ensure thread safety
- Message passing concurrency
  - Variables are not shared, values are passed over channels
  - Communication is the synchronisation