Rust

A brief overview

Paolo Baldan

Languages for Concurrency and Distribution

General

- From the official website (http://rust-lang.org):
  - **Performance**: Blazingly fast and memory-efficient (no runtime or garbage collector), it can power performance-critical services, run on embedded devices, and easily integrate with other languages.
  - **Reliability**: Rich type system and **ownership model** guarantee memory-safety and thread-safety, statically!

Reliable systems programming language

Hack without fear!

Some features

- **Memory management**
  - Emphasis on **immutability** and strict **control of mutation**
  - **Ownership, borrowing, lifetimes**: Prevention of buffer overflows and segfaults: **memory safety** enforced by typing
  - Non-blocking garbage collector (syntax driven drops)
- **Types & Functional flavour**
  - Generics and traits (~ Go interfaces, Haskell-like type classes)
  - Pattern matching and algebraic data types (enums)
  - Higher-order functions (closures)

History

- Personal project of **Graydon Hoare** (2006) at Mozilla
- **Mozilla** project since 2009, as a language for programming a new **web browser** (**servo**)
- Rust 1.0 in 2015 (back compatibility, now on)
- Mozilla lays off in 2020 (after covid)
- 2021: Rust Foundation created to guide language's development
- Current version 1.95.0
- Most loved language in Stack Overflow (many times since 2016)

Control vs Safety

- **Control** of **low level details** can be worth
  - Efficiency
  - Flexibility
- ... but **dangerous**
  - Type errors
  - Null pointer, out-of-bound accesses
  - Memory (leaks, use after free, double free, etc.)

Example: C

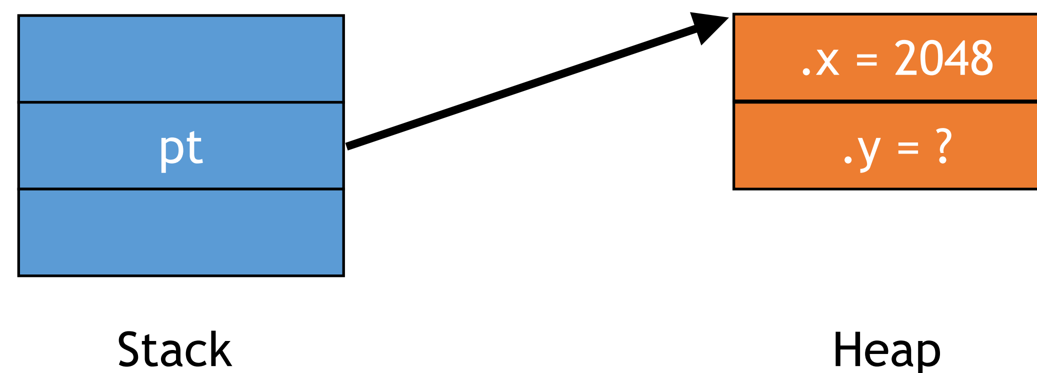
- C offers low-level memory access, with limited cost

```
typedef struct { float x; float y; } Point;

void foo(void) {
    Point *pt = (Point *) malloc(sizeof(Point));
    pt->x = 2048;
    free(pt);
}
```

explicit memory allocation

light reference (addresses)

explicit deallocation



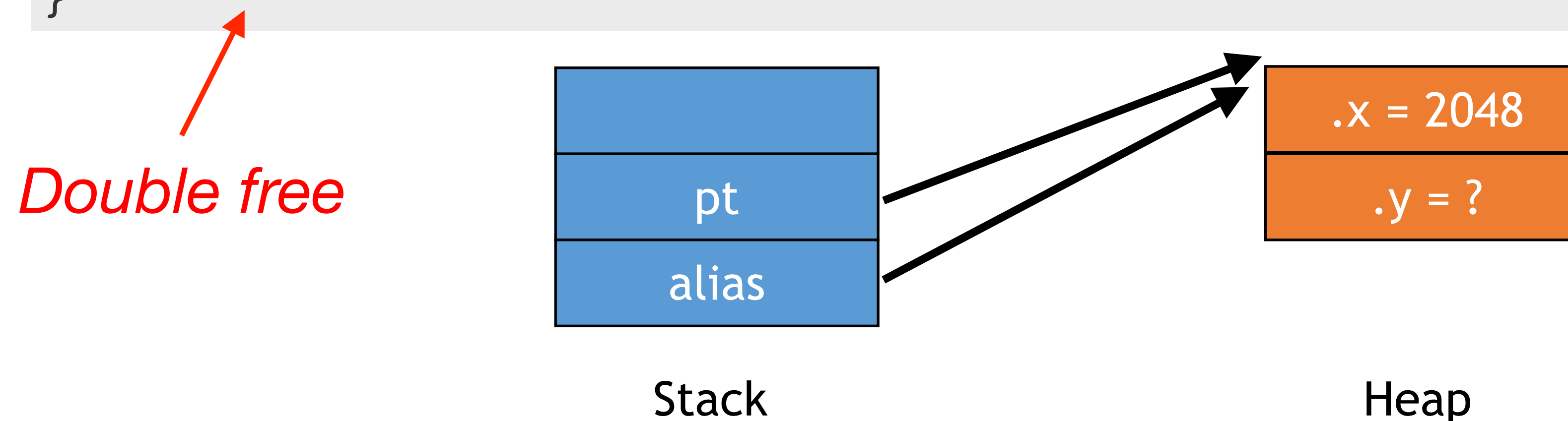
Example: C

- Freedom means responsibility

```
typedef struct Point { float x; float x; } Point;

void foo1(void) {
    Point *pt = (Point *) malloc(sizeof(Point));
    Point *alias = pt;
    free(pt);
    float a = alias->x;
    free(alias);
}
```

Uncontrolled sharing

Dangling references



- Even worse in a concurrent setting

A solution: restrict freedom

- Restrict access to memory (see, e.g., Go or Java)
  - No direct allocation or free
  - Periodic garbage collection
- But ...
  - Need of tracking references/escape analysis
  - GC can make responsiveness unpredictable
  - Virtual machine or runtime included
  - Larger code, reduced efficiency

The Rust approach

- Don't prevent direct access
- but force the programmer to be disciplined and declare what (s)he does!
- Memory-safety can be checked statically with an advanced type-system
- ... and thus minimal overhead at runtime (fast!)

Basics of types

- **Strict typing discipline** (as in Go)

- Primitive types

  `bool`
  `char` (4-byte unicode)
  `i8/i16/i32/i64/i128/isize`
  `u8/u16/u32/u64/u126/usize`
  `f32/f64`

- Note:
  - Separate bool type
  - Sized numeric types (no automatic coercion/promotion)

Arrays, Vectors, Slices

- **Arrays**: sequences of elements of fixed size

```
let nums = [1,2,3,4,5,6,7,8];          // [i32; 8]
```

stored with their length `[T; N]`

- **Vectors**: dynamic or "growable" arrays, implemented as the standard library `type Vec<T>`

```
let nums = vec![1,2,3,4,5,6,7,8];     // Vec<i32>
```

- **Slice**: A slice is a reference to (or "view" into) an array

```
let some = &nums[2:5];                  // &[i32]
```

Some code

arrays.rs

- Run-time (and some compile-time) checks

```
fn main() {
    let nums = vec![1,2,3,4,5,6,7,8];
    for x in 0..10 {
        println!("{}", nums[x]);
    }
}
```

```
1
2
...
8
thread 'main' panicked at arrays.rs:4:24:
index out of bounds: the len is 8 but the index is 8
```

Clippy-driver static analysis

```
warning: the loop variable `x` is only used to index `nums`
 --> arrays.rs:3:14
  |
3 |     for x in 0..10 {
  |              ^^^^^
  |
  ...
help: consider using an iterator
  |
3 |     for <item> in nums.iter().take(10) {
  |         ~~~~~~    ~~~~~~~~~~~~~~~~~~~~

warning: useless use of `vec!`
 --> arrays.rs:2:16
  |
2 |     let nums = vec![1,2,3,4,5,6,7,8];
  |                ^^^^^^^^^^^^^^^^^^^^^ help: you can use an array
directly: `[1,2,3,4,5,6,7,8]`
  |
  ...
```

Some code

- Checking bound at each access adds some overhead
- Mostly avoided in idiomatic Rust, using iterators

```
fn main() {
    let nums = vec![1,2,3,4,5,6,7,8];
    for num in nums.iter() {
        println!("{}", num);
    }
}
```

Strict numeric types

- No automatic casting

```
let x:i8 = 1;
let y:i16 = 1;

if x==y  {
       ...
}
```

```
  | if x==y  {
  |        ^ expected i8, found i16
```

- No overflow

```
let mut x : u8 = 0xFF;
x =x+1;
```

```
thread 'main' panicked at …
attempt to add with overflow'
```

Ownership, borrow and lifetime

Ownership

- Each value has a **owner** (variable)
- There can only be **one owner at a time.**
- When the owner goes out of scope, the value is **dropped**.

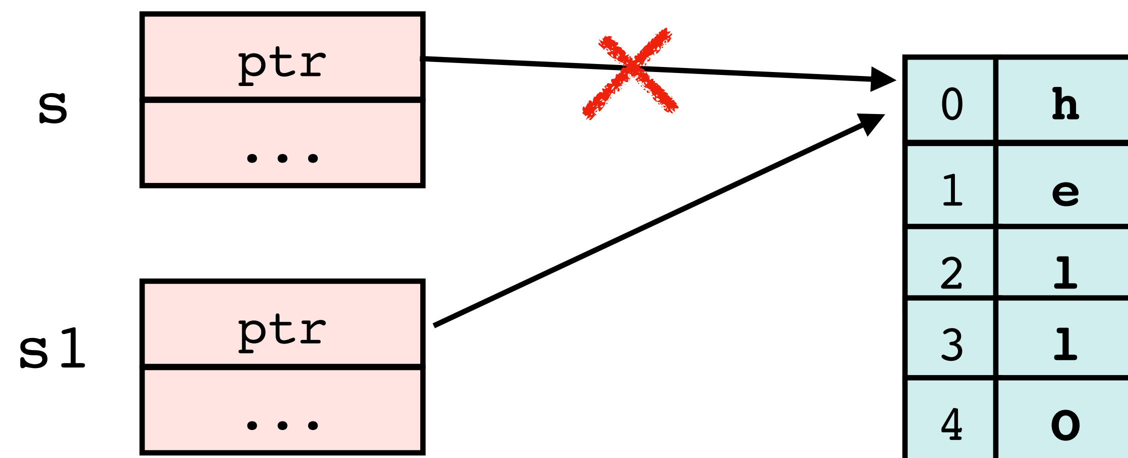
```
{                         // s not valid here (not yet declared)
  let s = String::from("hello"); // s valid from this point on
  ... // use s
  ...
                          // scope is over, s no longer valid
  drop(s)                 // automatically inserted
}
```

Ownership: Move

- There can only be **one owner at a time.** Assignements **move** ownership!

```
{
  let s = String::from("hello");
  let s1 = s;                    // ownerships moved from s to s1
  println!("{}", s1);
  println!("{}", s);
}
```

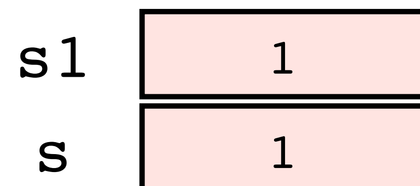
```
error[E0382]: use of moved value: `s`
```



Copy trait

- Values of types implementing the **copy trait** are not moved but cloned

```
{
  let s = 1
  let s1 = s;                    // copied (not moved) to s1
  println!("{}", s1);
  println!("{}", s);
}
```



Move & Functions

- Passing parameters moves ownership (as assignements)

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

fn calculate_length(s: String) -> usize {
    let length = s.len();
    length
}
```

Move & Functions

- Also returning values moves ownership

```
fn gives_ownership() -> String {
    let some_string = String::from("hello");
    some_string   // ownership is moved to the caller
}
```

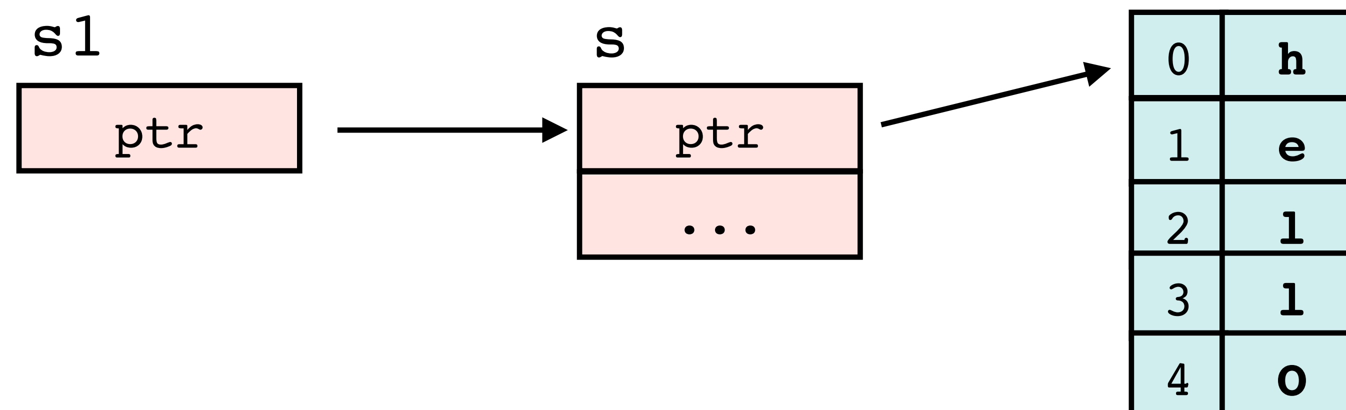
- If one needs the value after calling a function ...

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

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len();
    (s, length)
}
```

References & Borrowing

- Variable values can be (temporarily) **borrowed**, via a **reference &**

```
fn main() {
    let s = String::from("hello");
    let len = calculate_length(&s);  // value borrowed
    println!("The length of '{}' is {}.", s, len);
}

fn calculate_length(s1: &String) -> usize {
    s1.len()
}
```



A step aside: Mutability

- Variables are **immutable** by default

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

```
error[E0384]: cannot assign twice to immutable variable `x`
```

- To be mutable they must be declared as such

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

```
The value of x is: 5
The value of x is: 6
```

Shadowing

- Variable can be redeclared (shadowed)

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

```
The value of x is: 5
The value of x is: 6
```

- It is a completely new variable

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

```
The value of x is: 5
The value of x is: hello
```

Back to Borrowing

- A value borrowed as immutable cannot be mutated

```
fn main() {
    let x = 1;
    inc(&x);
    println!("{}", x)
}

fn inc(x : &i8) {
    *x += *x + 1;
}
```

```
error[E0594]: cannot assign to `*x`, which is behind a `&`
reference

(A non-mutable value was assigned a value.)
```

Back to Borrowing/1

- Types must match also concerning mutability

```
fn main() {
    let x = 1;
    inc(&x);
    println!("{}", x)
}

fn inc(x : &mut i8) {
    *x += *x + 1;
}
```

```
error[E0308]: mismatched types

(types differ in mutability.)
```

Back to Borrowing/2

- An immutable value cannot be borrowed as mutable

```
fn main() {
    let x = 1;
    inc(&mut x);
    println!("{}", x)
}

fn inc(x : &mut i8) {
    *x += *x + 1;
}
```

```
error[E0596]: cannot borrow `x` as mutable, as it is not
declared as mutable
```

Back to Borrowing/3

- It works, finally!

```
fn main() {
    let mut x = 1;
    inc(&mut x);
    println!("{}", x)
}

fn inc(x : &mut i8) {
    *x += *x + 1;
}
```

```
2
```

Borrowing, with care

- **Many** can borrow a value as **non-mutable** at the same time, but if it is borrowed as **mutable** this must be the **unique** borrowing

```
fn main() {
    let s = String::from("hello");
    let borrow1 = &s;
    let borrow2 = &s;
    println!("borrowed {} and {}", borrow1, borrow2);
}
```

OK!

```
fn main() {
    let mut s = String::from("hello");
    let borrow1 = &mut s;
    let borrow2 = &mut s;
    println!("borrowed {} and {}", borrow1, borrow2);
}
```

NO!

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
```

Borrowing, with care

- **Many** can borrow a value as **non-mutable** at the same time, but if it is borrowed as **mutable** this must be the **unique** borrowing

```
fn main() {
    let s = String::from("hello");
    let borrow1 = &s;
    let borrow2 = &s;
    println!("borrowed {} and {}", borrow1, borrow2);
}
```

OK!

```
fn main() {
    let mut s = String::from("hello");
    let borrow1 = &mut s;
    let borrow2 = &s;
    println!("borrowed {} and {}", borrow1, borrow2);
}
```

NO!

```
error[E0502]: cannot borrow `s` as immutable because it is also
borrowed as mutable
```

Conflicts, sequentially

- Conflicts also in sequential computation

```
fn main() {
    let mut buf = vec![1, 2, 3, 4];
    for i in buf.iter() {
            buf.push(*i);
    }
}
```

NO!

```
error[E0502]: cannot borrow `buf` as mutable because it is also
borrowed as immutable
```

Dangling refs?

- Dangling references are prevented statically

```
fn main() {
    let s = String::from("world!");
    let new = add_hello(&s);
    println!("{} after add is {}", s, new);
}

fn add_hello(s : &String) -> &String {
    let mut s1 = String::from("hello ");
    s1.push_str(s);
    &s1
}
```

NO!

```
error[E0515]: cannot return reference to local variable `s1`
```

Dangling refs?

- The function must **move** the value to the caller

```
fn main() {
    let s = String::from("world!");
    let new = add_hello(&s);
    println!("{} after add is {}", s, new);
}

fn add_hello(s : &String) -> String {
    let mut s1 = String::from("hello ");
    s1.push_str(s);
    s1
}
```

```
world! after add is hello world!
```

Borrowing: sum up

- One can borrow **immutable references** from **mutable values**, but not the other way round
- At any given time, you can have either **one mutable reference** or **any number of immutable references**.
    - Multiple readers, single writer
- **References** are required to be always **valid**.
    - The **lifetime** of a borrowed reference should end before the lifetime of the owner of the value

Lifetime

- A construct used by the compiler (specifically, by its borrow checker) for ensuring all borrows are valid.

- A variable's lifetime begins when it is created and ends when it is destroyed (out of scope).

```
{
    let r;                // ---------+-- 'a
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
    println!("r: {}", r); //          |
}                         // ---------+
```

NO!

```
error[E0597]: `x` does not live long enough
```

Lifetime

longest.rs

- Lifetime is normally inferred

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}                            expected named lifetime parameter
```

- ... but in some situations it needs to be specified.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Back to C example

```c
typedef struct { float x; float x; } Point;

void foo(void) {
    Point *pt = (Point *) malloc(sizeof(Point));
    Point *alias = pt;
    free(pt);
    float a = alias.x;
    free(alias);
}
```

```rust
struct Point {x : f32, y : f32}

fn main() {
    let pt = Point{x:2.0, y:3.0};
    let alias = pt;
    println!("original point {:?}\nand alias {:?}", pt, alias);
}
```

```
error[E0382]: borrow of moved value: `pt`
```

Back to C example/1

```c
typedef struct Point { float x; float x; } Point;

void foo(void) {
    Point *pt = (Point *) malloc(sizeof(struct Point));
    Point *alias = pt;
    free(pt);
    float a = alias.x;
    free(alias);
}
```

```rust
struct Point {x : f32, y : f32}

fn main() {
    let pt = Point{x:2.0, y:3.0};
    let alias = &pt;
    println!("original point {:?}\nand alias {:?}", pt, alias);
}
```

```
original point Point { x: 2.0, y: 3.0 }
and alias Point { x: 2.0, y: 3.0 }
```

Back to C example/2

```
typedef struct Point { float x; float x; } Point;

void foo(void) {
    Point *pt = (Point *) malloc(sizeof(struct Point));
    Point *alias = pt;
    free(pt);
    float a = alias.x;
    free(alias);
}
```

```
struct Point {x : f32, y : f32}

fn main() {
    let pt = Point{x:2.0 y:3.0};
    let alias = &pt;
    drop(pt)
    println!("alias {:?}", alias);
}
```

```
error[E0505]: cannot move out of `pt` because it is borrowed
```

Object orientation

Methods

- Struct (and enum) can implement methods

methods.rs

```
#[derive(Debug)]
struct Point { x : f32, y : f32}

impl Point {
    fn module (&self) -> f32 {
       (self.x*self.x + self.y*self.y).sqrt()
    }

    fn traslate(&mut self, dx: f32, dy:f32) {
        self.x = self.x + dx;
        self.x = self.y + dy;
    }
}

fn main() {
    let mut pt = Point{x:2.0, y:3.0};
    pt.traslate(1.0,1.0);
    println!("module {}", pt.module());
}
```

Traits

`methods-traits.rs`

- Similar to interfaces

```
pub trait Module {
    fn module(&self) -> f32;
}
```

- Generic functions working on any type implementing the trait **`Module`**

```
fn print_module<T:Module>(v:T){
   println!("module {}", v.module())
}
```

Traits: implementing

- Implementing a trait

```
struct Point {x : f32, y : f32}

impl Point {
    fn traslate(&mut self, dx: f32, dy:f32) {
        self.x = self.x + dx;
        self.x = self.y + dy;
    }
}

impl Module for Point {
    fn module (&self) -> f32 {
      (self.x*self.x + self.y*self.y).sqrt()
    }
}
```

Traits

- Implementing a trait (also basic types can do)

```
impl Module for f32 {
    fn module (&self) -> f32 {
        self.abs()
    }
}
```

- Using ...

```
fn main() {
    let mut pt = Point{x:2.0, y:3.0};
    pt.traslate(1.0,1.0);
    print_module(pt);

    let f:f32 = -2.0;
    print_module(f);
}
```

Default implementation

- A default implementation can be specified, types implementing the trait can "inherit" it

methods-traits-default.rs

```
pub trait Module {
    fn module(&self) -> f32;

    fn print_module (&self) {
        println!("module {}", self.module())
    }
}
```

```
fn main() {
    let mut pt = Point{x:2.0, y:3.0};
    pt.traslate(1.0,1.0);
    pt.print_module();

    let f:f32 = -2.0;
    f.print_module();
}
```

Closures

Closures

- Or **unnamed functions** (lambda expr) à la Rust

inc.rs

```
fn main() {
    let inc =           |i: i32| -> i32 { i + 1 };

    let inc_infer  = |i      |            i + 1   ;

    let val = 1;

    println!("inc:       {}", inc(val));
    println!("inc_infer: {}", inc_infer(val));
}
```

```
inc:       2
inc_infer: 2
```

Closures and move

- Parameters moved as in normal functions

inc-string.rs

```
fn main() {
    let inc =           |i: String| -> String { i + " more" };
    let inc_infer  =    |i        |           { i + " more" };

    let val = String::from("My string and");

    println!("inc:       {}", inc(val));
    println!("inc_infer: {}", inc_infer(val));
```

```
error[E0382]: use of moved value: `val`
```

Closures & Borrow

- Closures can capture the env: **borrow (immutably)**

closure.rs

```
fn main() {
    let val: i32 = 1;
    let inc =  || -> i32 { val + 1 };
    println!("out: {}", inc());
    println!("val: {}", val);
}
```

```
out: 2
val: 1
```

- Closures can capture the env: **mutably borrow**

mut-closure.rs

```
fn main() {
    let mut val: i32 = 1;
    let mut inc_mut  = ||   { val = val + 1; val };
    println!("out: {} {}", inc_mut(), inc_mut());
}
```

```
out: 2 3
```

Closures & Move

- Closures can take ownership with **move**

```
fn main() {
    let x = String::from("Hello");

    let equal_to_x = move |z| z == x;

    println!("can't use x here: {:?}", x);

    let y = String::from("Hello");
    println!("Result: {}", equal_to_x(y));
}
```

```
error[E0382]: borrow of moved value: `x`
```

Concurrency

(Fearless?) Concurrency

- Ownership is a key tool also to address concurrency problems
- Lower level concepts than in other languages, but better isolation guarantees
- Many concurrency errors become compile-time errors
- Threads are system threads: no green threads in order to limit the runtime at minimum

Concurrency

- Threads running closures with ...
- Shared state concurrency (sync)
- Message passing concurrency (channels)

Concurrency

- Threads can be spawned running closures

thread.rs

```
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));
        }
    });

    thread::sleep(Duration::from_millis(10));
    handle.join().unwrap();
}
```

The way to sharing

- Sharing via closure "capture" does not work

```
fn main() {
    let shared = String::from("Hello");

    let handle = thread::spawn(|| {
        println!("{} from thread", shared);
    });

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

```
error[E0373]: closure may outlive the current function, but
it borrows `shared`, which is owned by the current function
```

Moving values to threads

- Not really sharing ...

```
fn main() {
    let shared = String::from("Hello");

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

    println!("{} from main", shared);

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

```
error[E0382]: borrow of moved value: `shared`
```

Smart pointers: Rc

- Value with **multiple owners**
- Keeps track of the number of references. If none, value can be cleaned up (GC in disguise)

```
fn main() {
    let shared = Rc::new(String::from("Hello"));

    let shared_clone = Rc::clone(&shared);
    let handle = thread::spawn(move || {
        println!("{} from thread", *shared_clone);
    });

    println!("{} from main", *shared);

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

```
error[E0277]: `std::rc::Rc<String>` cannot be sent between
threads safely
```

Sync library for shared states

- Familiar synchronisation mechanisms, with Rust flavour
  - **Arc**: Atomically Reference Counted pointer (rc pointer safe for threads)
  - **Barrier**: common sync point for threads
  - **Condvar**: block a thread waiting for an event
  - **Mutex**, RwLock: Mutual Exclusion mechanisms (possibly with multiple readers)
  - **Once**: Thread-safe, one-time initialization of a global variable.
  - ...

Atomic rc for sharing

arc.rs

```
fn main() {
    let shared = Arc::new(String::from("Hello"));

    let shared_clone = Arc::clone(&shared);

    let handle = thread::spawn(move || {
            println!("{} from thread", *shared_clone);
        });
    }

    println!("{} from main", *shared);

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

```
Hello from main
Hello from thread
```

Atomic rc for sharing

arc.rs

```
fn main() {
    let shared = Arc::new(String::from("Hello"));
    let handle;

    {
        let shared = Arc::clone(&shared);

        handle = thread::spawn(move || {
            println!("{} from thread", shared);
        });
    }

    println!("{} from main", shared);

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

Automatic dereferencing

```
Hello from main
Hello from thread
```

Modifying?

```
fn main() {
    let shared = Arc::new(String::from("Hello"));
    let handle;

    {
        let shared = Arc::clone(&shared);

        handle = thread::spawn(move || {
            shared.push_str("  from thread");
            println!("{}", shared);
        });
    }

    shared.push_str("  from main");
    println!("{}", shared);

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

```
error[E0596]: cannot borrow data in an `Arc` as mutable
```

Mutex for modifying

mutex.rs

```
fn main() {
    let shared = Arc::new(Mutex::new(String::from("Hello")));
    let handle;

    {
        let shared = Arc::clone(&shared);

        handle = thread::spawn(move || {
            // the lock lives until the end of the scope
            let mut msg = shared.lock().unwrap();
            msg.push_str("  from thread");
            println!("{}", msg);
        });
    }

    {  // the lock lives until the end of the scope
       let mut msg = shared.lock().unwrap();
       msg.push_str("  from main");
       println!("{}", msg);
    }

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

```
Hello  from main
Hello  from main  from thread
```

Message passing

- Message passing through **channels**

- **Multiple-producer single-consumer** (mpsc::channel)

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

- tx (transmitter end), rx (receiver end)

```
tx.send(val)
```

```
rx.recv()
```

to be unwrapped

Example

channel.rs

```
fn main() {
    let msg = String::from("Hello");

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

    let handle = thread::spawn(move || {
            let msg = rx.recv().unwrap();
            println!("{} from thread", msg);
        });

    tx.send(msg).unwrap();

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

```
Hello from thread
```

Send move ownership

- Sent values ownership is transferred to receiver

```
fn main() {
    let msg = String::from("Hello");

    let (tx, rx) = mpsc::channel();
    let handle = thread::spawn(move || {
            let msg = rx.recv().unwrap();
            println!("{} from thread", msg);
        });

    tx.send(msg).unwrap();

    println!("{} from main", msg);

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

```
error[E0382]: borrow of moved value: `msg`
```

Send move ownership

- Need to get it back to reuse it

rendez-vouz.rs

```
fn main() {
    let msg = String::from("Hello");
    let (tx1, rx1) = mpsc::channel();
    let (tx2, rx2) = mpsc::channel();

    thread::spawn(move || {
            let msg = rx1.recv().unwrap();
            println!("{} from thread", msg);
            tx2.send(msg).unwrap();
        });

    tx1.send(msg).unwrap();
    let msg = rx2.recv().unwrap();

    println!("{} from main", msg);
}
```

```
Hello from thread
Hello from main
```

Multiple senders

- By cloning the send side ...

multiple-senders.rs

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

    let tx_a = mpsc::Sender::clone(&tx);

    thread::spawn(move || {
            let msg = String::from("Hello from Thread");
            tx.send(msg).unwrap();
        });

    thread::spawn(move || {
            let msg = String::from("Hello from Thread A");
            tx_a.send(msg).unwrap();
        });

    println!("{}", rx.recv().unwrap());
    println!("{}", rx.recv().unwrap());
}
```

```
Hello from Thread A
Hello from Thread
```

Multiple senders

- By cloning the send side ...

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

    {
     let tx = mpsc::Sender::clone(&tx);

     thread::spawn(move || {
            let msg = String::from("Hello from Thread");
           tx.send(msg).unwrap();
     });
    }

    thread::spawn(move || {
        let msg = String::from("Hello from Thread A");
        tx.send(msg).unwrap();
    });

    for msg in rx {
        println!("{}", msg);    }

    println!("Channel closed! Main thread exiting cleanly.");
}
```

multiple-senders1.rs

```
Hello from Thread A
Hello from Thread
Channel closed! …
```

And so on ...

- **Crossbeam**: scoped threads, for easier sharing
- **Rayon**, parallel iteration
- **Thread** pools, executing tasks in parallel on a set of workers
- **Futures,** for asynchronous computations
- ...

Wrap Up

Rust is good

- **Efficient** code (thanks to the extremely light runtime controls)
- Strongly typed (really **powerful type system**)
- **Borrow checker** ensures **safe memory** usage (no use after free, double free, dangling refs)
- Rigorous sharing discipline for absence of **data races**

... but

- Somehow coding requires an additional conceptual effort (is this a plus or minus?), steep learning curve
- Slow compile (compilation does more)
- Simple high level construct of other languages missing (but you have libraries)

New vs old?

Rust bases its safety on

- **Linear types**
  [Linear Logic, Jean Yves Girard, 1987]

$$A \to B, A \vdash A \wedge B$$

- **Region-based memory management**

  "Tofte, M. and Talpin, J.-P. Implementing the call-by-value lambda-calculus using a stack of regions. POPL1994.

Nothing can be included in Rust that isn't at least a decade old.
— Graydon Hoare, Rust founder.