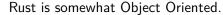
## Rust: memory safety and systems programming

Manos Pitsidianakis

Software Engineering, NTUA, 2017-18

Rust is a systems programming language, like C, inspired by C and Ocaml.



► Trait generics instead of inheritance

- Traits instead of classes/interfaces

Types can implement Traits (what Rust calls interfaces+mixins).

that implement a certain Trait).

We can abstract over many types by using Trait Objects (structs

```
pub trait Draw {
    fn draw(&self);
pub struct Screen {
    pub components: Vec<Box<Draw>>,
impl Screen {
   pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
```

Figure 1: A simple trait definition

```
pub struct Screen<T: Draw> {
    pub components: Vec<T>,
impl<T> Screen<T>
    where T: Draw {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
```

Figure 2: Alternative definition using trait generics

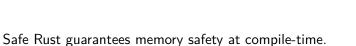
Rust is safe unlike C, and also unsafe like C. This is because Rust is memory safe by default, but we can turn off memory safety and use it (somewhat) like C.

Undefined behaviour is also very limited. Examples of memory safety:

- buffer overflow
- null pointer dereference
- use after free
- use of uninitialized memory
- illegal free

Unsafe Rust is code wrapped in an unsafe $\{\}$ block. Its main difference from safe Rust is the ability to dereference raw pointers and calling external functions.

Wrapping unsafe code in special sections of our code enables easier
checking for program correctness; only unsafe blocks must be
checked for memory safety. This makes review very easy.



A compiled safe Rust program will not segfault (unless there's an

implementation error in the language itself).

The concept of lifetimes is the most interesting and confusing concept about Rust

▶ All values have one owner (reference) at a time.

called.

▶ When the owner goes out of scope, the value's destructor is

To transfer values between functions, a value can either be	
► moved, or	

borrowed as a reference (a kind of smart pointer)

fn capitalize(data: &mut [char]) {

// do something

Figure 3:

references.

To ensure memory safety at any time the following holds: For each value there's one mutable reference XOR unlimited immutable

Borrow checker checks these constraints during compilation.	It's the
first and hardest obstacle for newcomers to Rust. Not even	unsafe

Rust can turn off the borrow checker.

```
fn take(v : Vec<i32>) {
 // ownership of the vector now transferred
 // to v in this scope
let v = vec![1, 2, 3]; // take ownership
take(v);
                // moved ownership
println!("v[0] is {}", v[0]);
// error: use of moved value: `v`
// println!("v[0] is {}", v[0]);
//
```

Figure 4:

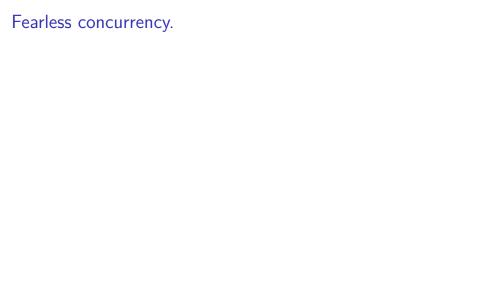
```
// note we're taking a reference, &Vec<i32>
// instead of Vec<i32>
fn take(v : &Vec<i32>) {
  // no need to deallocate the vector after
 // we go out of scope here
let v = vec![1, 2, 3]; // take ownership
// notice we're passing a reference, &v,
// instead of v
take(&v); // borrow ownership
// so now we have the ownership back
// we can use v again!
println!("v[0] is {}", v[0]);
// v[0] is 1
```

```
fn take(v : &Vec<i32>) {
   v.push(5);
}
let v = vec![];
take(&v);
// cannot borrow immutable borrowed content `*v` as mutable
// v.push(5);
// ^
```

Figure 6:

```
// v is a mutable reference
fn take(v : &mut Vec<i32>) {
 v.push(5);
// v has to be mutable too in order for you
// to borrow a mutable ref
let mut v = vec![];
take(&mut v);
println!("v[0] is {}", v[0]);
// v[0] is 5
```

Figure 7:



"Do not communicate by sharin communicating."	g memory;	instead,	share mem	nory by

# Message passing

```
// Suppose chan: Channel<Vec<i32>>
let mut vec = Vec::new();
// do some computation
send(&chan, vec);
print vec(&vec);
//Error: use of moved value `vec`
```

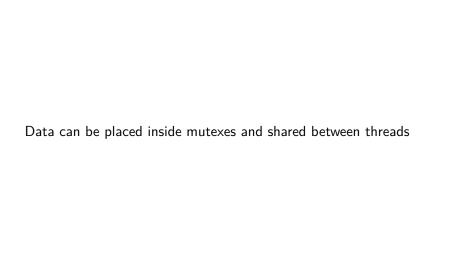
Figure 8:

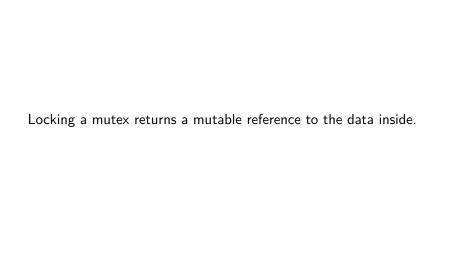
### Locks - Shared memory

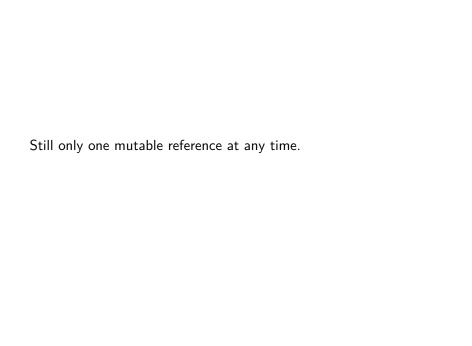
Necessary evil for systems programming



Reminder: Borrow checker doesn't allow more than one mutable reference at any time.









memory leak by repeatedly allocating objects in an infinite loop

### What Rust doesn't prevent

deadlocks



integer overflows (but it can panic or wrap around if we want)

### What Rust doesn't prevent

logic errors, etc

#### References

https://doc.rust-lang.org/book/