GameSS

EDUCATIONAL MATERIAL IN CYBER SECURITY

MEMORY SAFETY AND CODE VERIFICATION IN RUST

WHO IS THE MATERIAL FOR?

- Students and professionals interested in methods and tools for eradicating memory safety issues.
- Managers, software developers, and security professionals interested in evaluating whether they should use Rust in future projects.
- Students and professionals interested in methods and tools for obtaining functional correctness guarantees on top of memory safety.
- This material is primarily about defensive security, that is, how to guarantee that certain bugs cannot happen

WHO MADE THIS MATERIAL?

Christoph Matheja

Technical University of Denmark

chmat@dtu.dk

www.cmath.eu

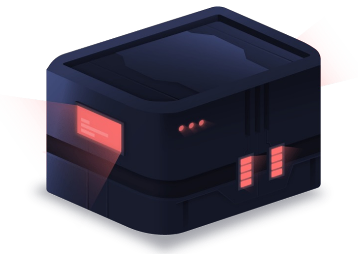
Supplementary material that will be provided alongside these slides:

- Source code for examples, exercises, and challenges
- Video lecture covering the slides and live coding for some examples

WHAT ARE THE MAIN TAKEAWAYS FOR THIS CONTENT?

- There is no security without safety.
- Rust‘s ownership and borrowing system statically guarantee safety by ensuring that references are *either* mutable *or* shared; for exceptions, a synchronization mechanism must enforce safety.
- Flows provide a useful mental model for understanding how the Rust compiler checks memory safety and, in particular, lifetimes.
- Program verification tools, such as Prusti, can provide stronger functional correctness guarantees but require additional annotations.

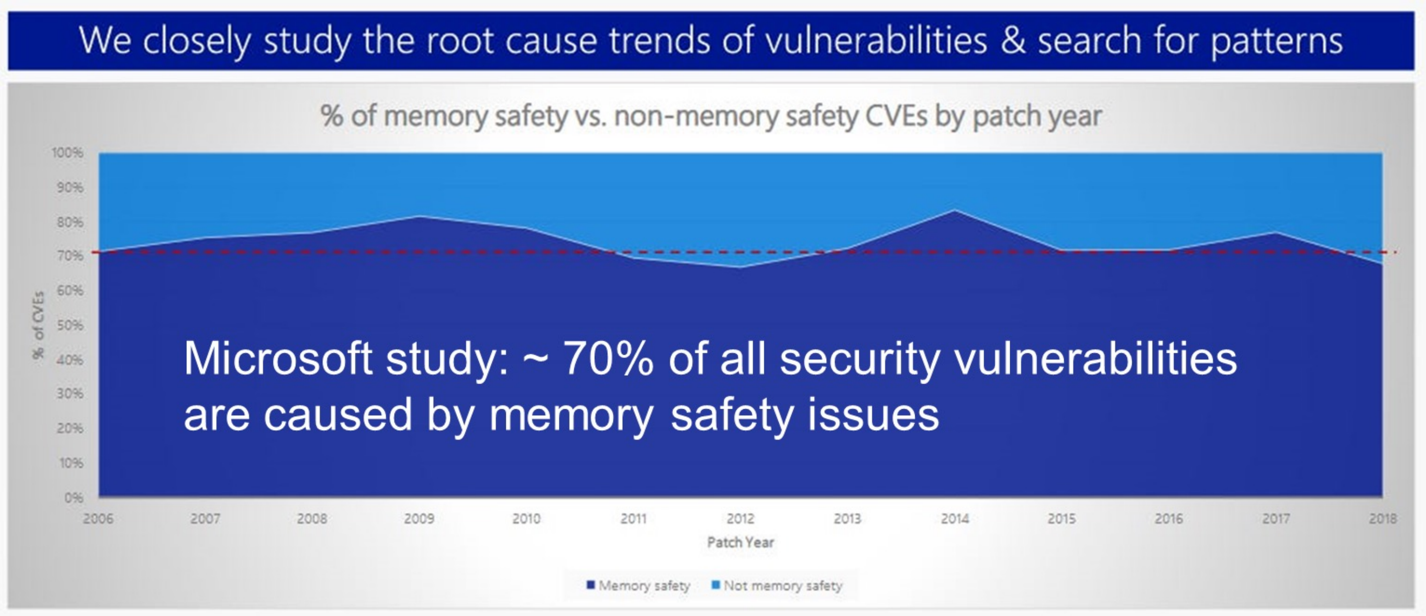
trade-off: writing more annotations ➔ more compile-time guarantees



INTRODUCTION

WHY SHOULD I FOLLOW THIS COURSE?

THERE IS NO SECURITY WITHOUT SAFETY



credits: Matt Miller, Microsoft Security Response Center

Memory safety is the absence of errors related to memory accesses.

THE RUST PROGRAMMING LANGUAGE

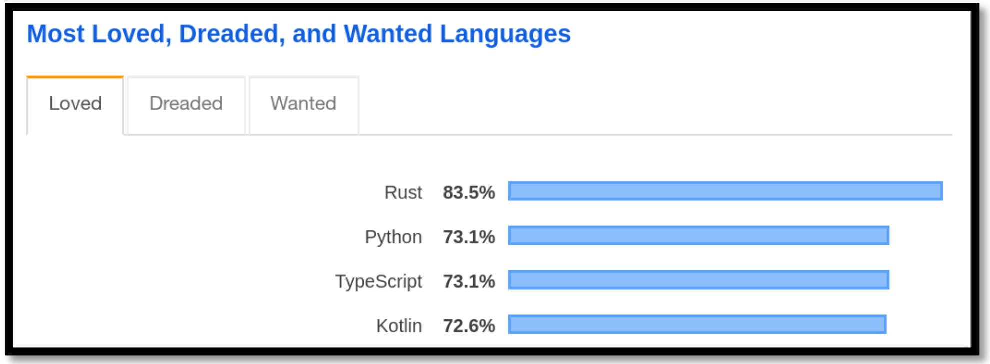
Rust is a modern language aiming at

safe systems programming

“The most beloved programming language since 2016”

“Rust is the industry’s best change at safe systems programming”

– Ryan Levic, Microsoft



credits: Stackoverflow

CHARACTERISTIC FEATURES OF RUST

Memory safety

ownership & borrowing

Performance

memory control, zero-cost abstractions

Ergonomics

trait system

Build environment

cargo, good error reporting

OUR FOCUS

Reasoning about the safety features of Rust code

- *mental models* for memory safety
- functional correctness guarantees

This will help you to write safer and more secure code

even if you never use Rust

But: this is **not** a Rust programming course

- Rust Book
- Rust by Example

AGENDA

1. High-level overview
2. Memory basics
3. Ownership
4. Borrowing
5. The flow model
6. Prusti: guarantees beyond memory safety

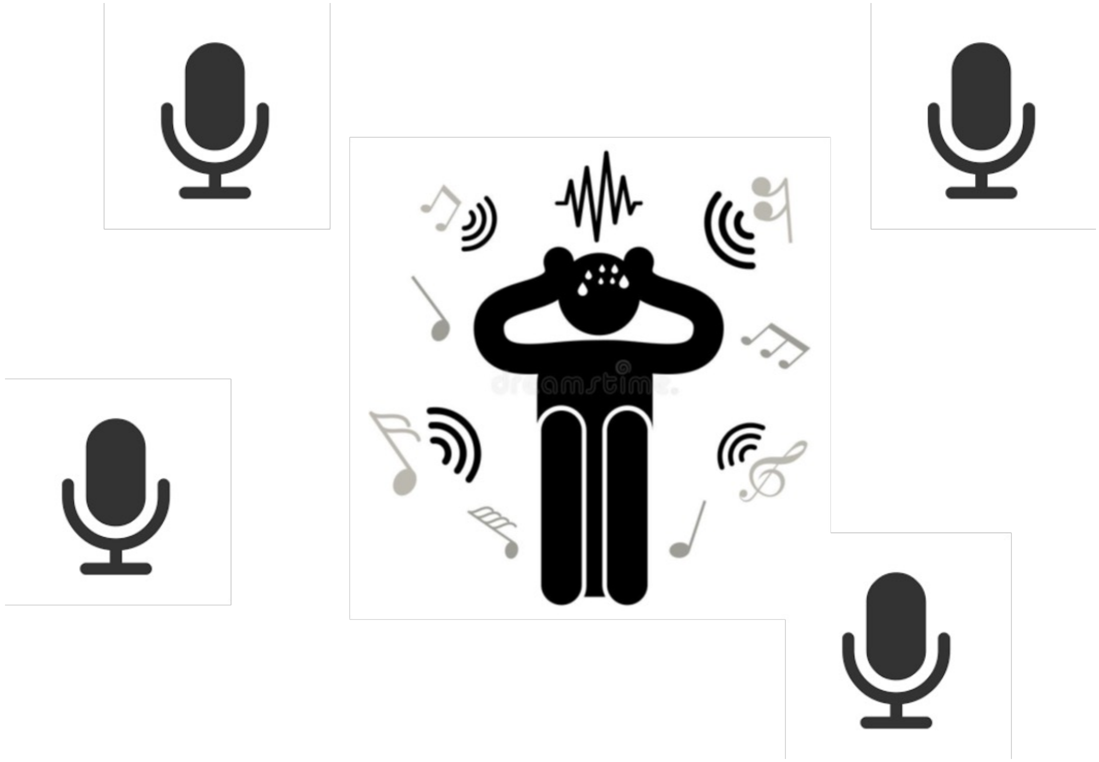
1. HIGH-LEVEL OVERVIEW

A NON-TECHNICAL METAPHOR ILLUSTRATING HOW RUST ENSURES MEMORY SAFETY

METAPHOR: ISSUES WITH VIDEO CONFERENCES

Problem: many participants in a video conference talk at once

➔ **Data race:** multiple agents access the same resource concurrently

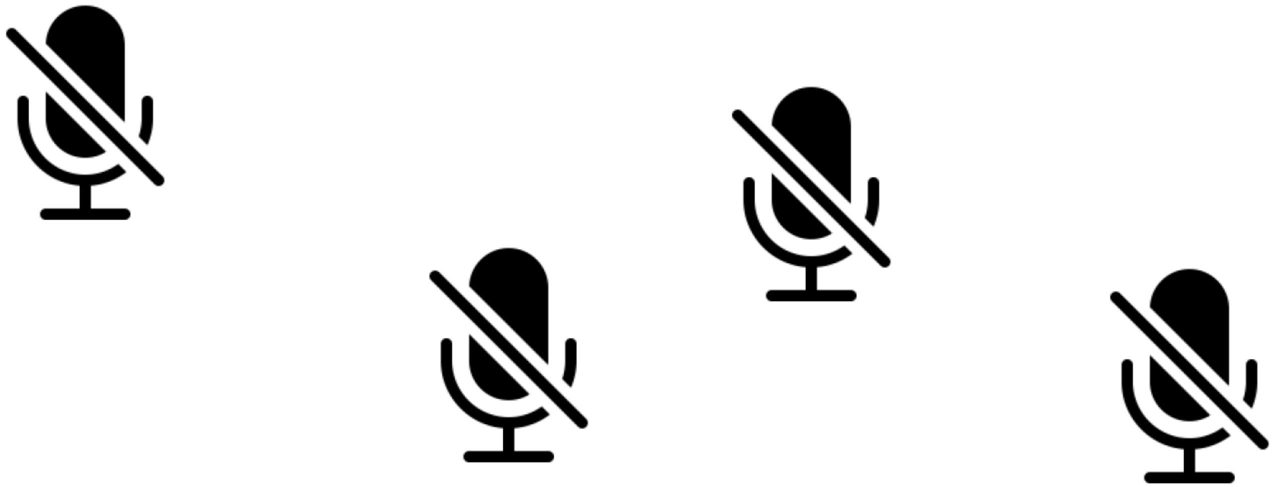


How can we rule out such situations?

METAPHOR: ISSUES WITH VIDEO CONFERENCES

Solution 1: one exclusive speaker

METAPHOR: ISSUES WITH VIDEO CONFERENCES



Solution 2: everyone is muted and only listens

METAPHOR: ISSUES WITH VIDEO CONFERENCES

Solution 3: a moderator assigns speaking rights

METAPHOR: ISSUES WITH VIDEO CONFERENCES

Requirements for data races	**in video conferences**
1. aliasing	many agents use the same channel
2. mutation	and all can speak
3. lack of synchronization	and there is no moderator

Solution: prevent that all three requirements hold at the same time

HOW RUST PREVENTS MEMORY SAFETY ISSUES

Requirements for data races

1. **aliasing**
2. **mutation**
3. **lack of synchronization**

Data races and many memory safety issues can only arise if these three conditions are met

Rust's high-level approach to safety guarantees

- Enforce that there is *either* **aliasing** *or* **mutation**
- Require **synchronization** for exceptions

2. MEMORY BASICS

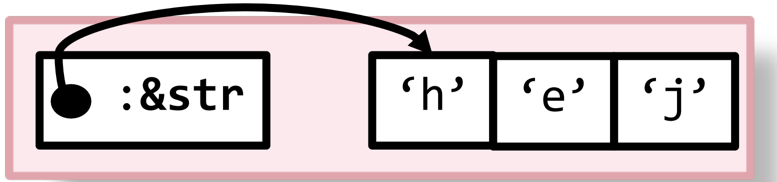
WHAT WE NEED TO TALK ABOUT OWNERSHIP, BORROWING, AND LIFETIMES IN RUST

TERMINOLOGY

- **Value:** a type and an element of that type

```
5:u8, 17:i32, 1.4:f64, “hej”:&str
```

- **Place:** a location holding the address of a value
- **Variable:** a “named slot” for a value
- **Pointer:** a value holding the address of a place

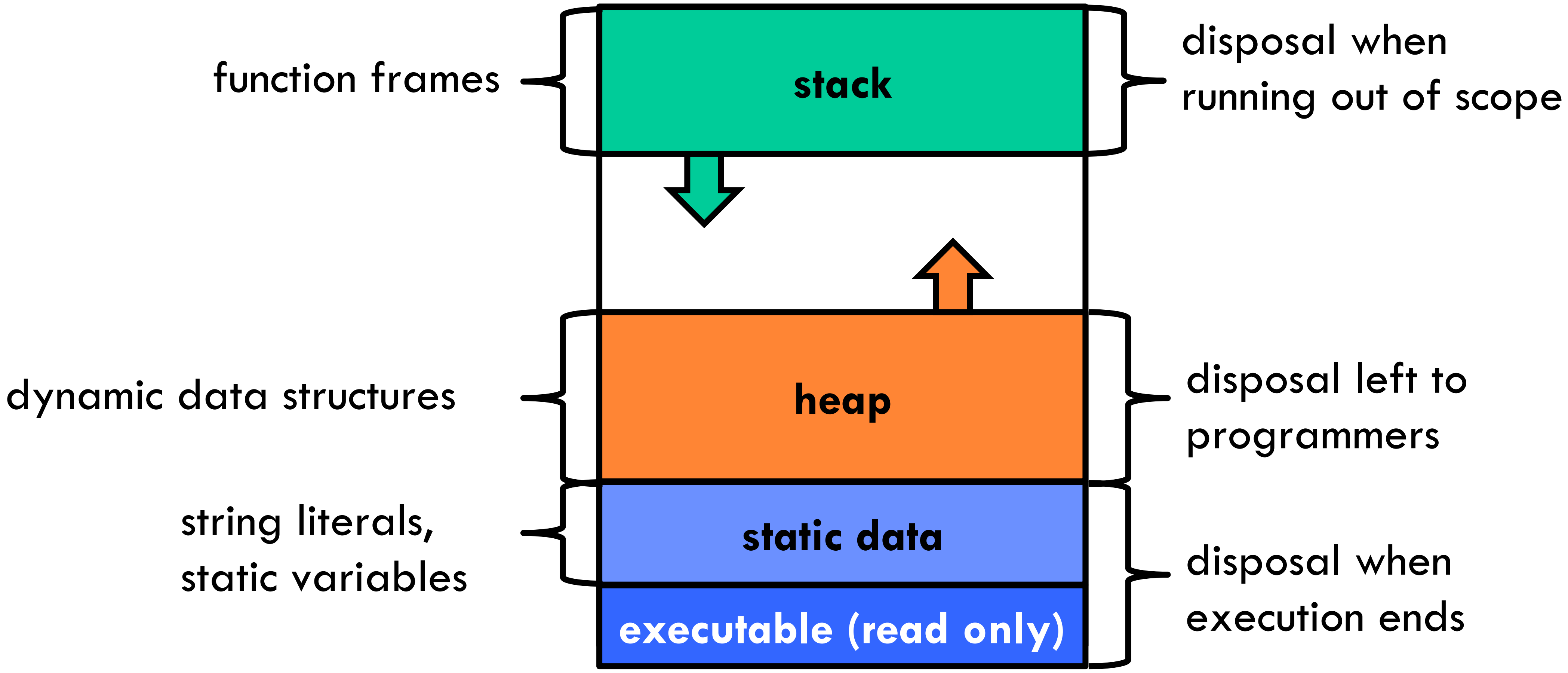


- **Reference:** a pointer with a specific contract
  - here: mutable `&mut T` and read-only `&T`

IDENTIFY ALL VALUES, POINTERS, AND VARIABLES

```
let a = 42;
let b = 43;
let c = &a:
let mut d = &a;
d = &y;
let e = “hello world”;
```

MEMORY LAYOUT



WHERE ARE THE PLACES OF THE FOLLOWING VALUES?

```
let a = 42;
let b = 43;
let c = &a:
let mut d = &a;
d = &y;
let e = “hello world”;
```

```
let tuple = (17, 3.14);

let b = Box::new(tuple);

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

WHERE ARE THE PLACES OF THE FOLLOWING VALUES?

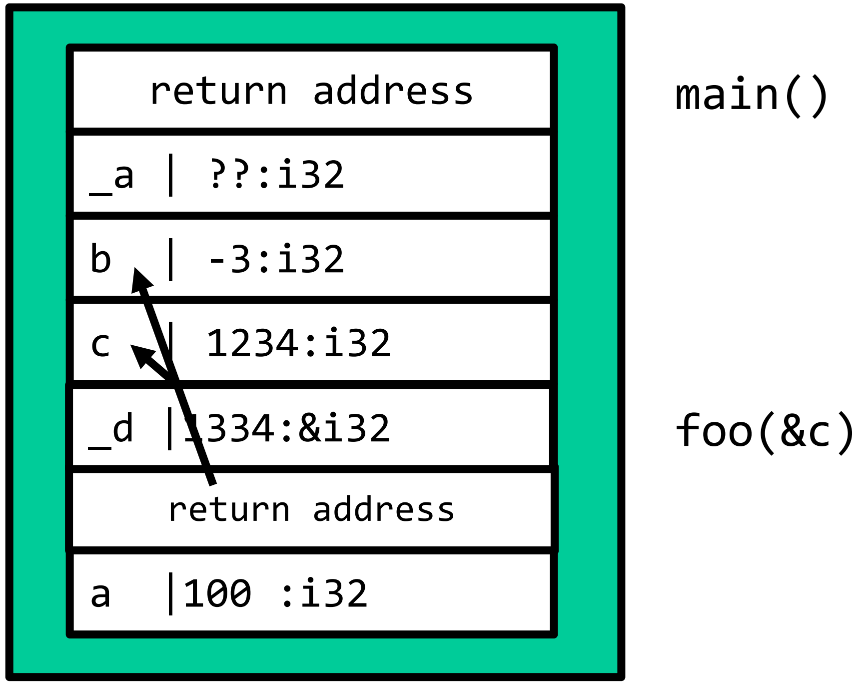
```
let a = 42;
let b = 43;
let c = &a:
let mut d = &a;
d = &y;
let e = “hello world”;
```

```
let tuple = (17, 3.14);

let b = Box::new(tuple);

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

STACK FRAMES

```rust
fn foo(x: &i32) -> i32 {
  let a = 100;
  a + *x
}

fn main() {
  let _a: i32;
  let b = -3;
  let c = 1234;
  // let c = _a;
  let _d = foo(&c);
  let _p = &b;
}
```

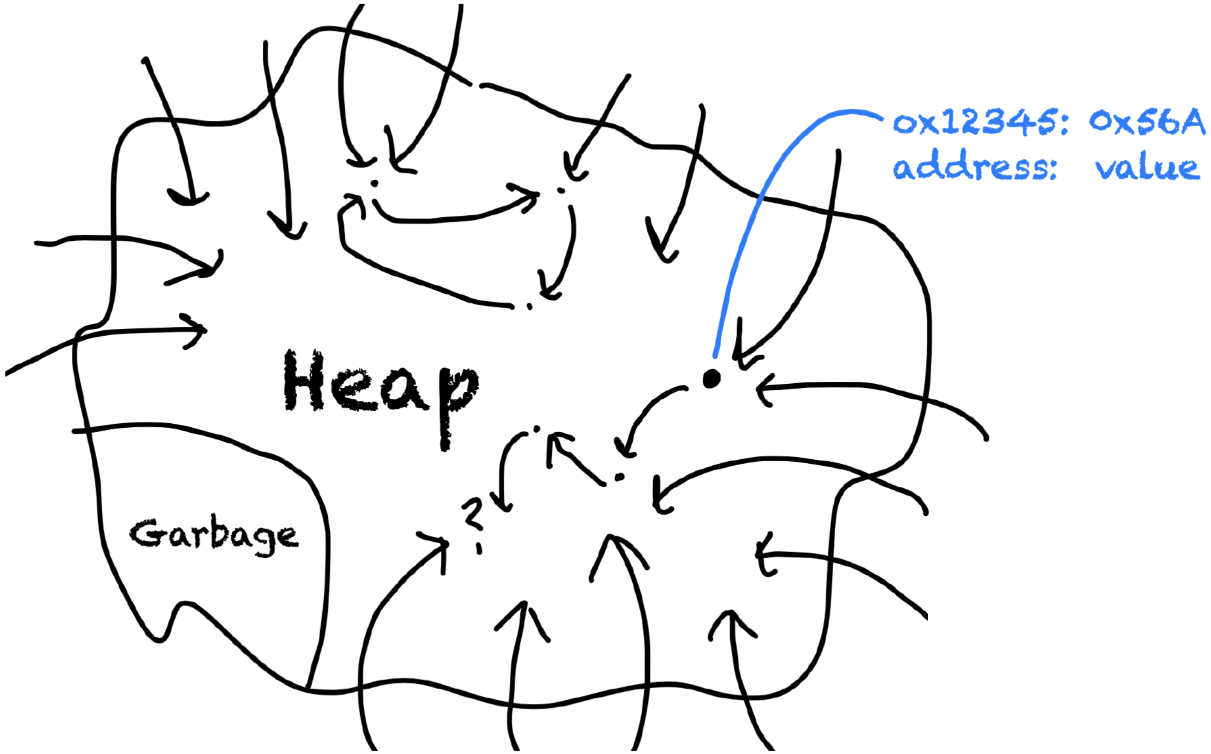
Accessing **uninitialized places** is forbidden



Stack-discipline: automatically drop a frame when it runs out of scope

HEAP

Disposal of heap-allocated values is left to the programmer



Memory error: an attempt to access a place with an *illegal* value

- uninitialized value
- *dangling* pointers to deleted values
- *corrupted value* (due to concurrency)

WHAT COULD POSSIBLY GO WRONG?

```
void foo(Struct* x, Struct* y)
{
    bar(x);
    free(x);
    bar(y);
}
```

C

Potential memory safety issues

- x might point to an uninitialized value
- bar might access the value of x

WHAT *ELSE* COULD POSSIBLY GO WRONG?

```c
void foo(Struct* x, Struct* y)
{
    bar(x);
    free(x);
    bar(y);
}
```

C

Potential memory safety issues

- bar might access the value of x
- x and y might be *aliases*, i.e. point to the same value
- bar(y) attempts to access the value of y, which has previously been deleted via free(x)

➔ **use-after-free bug**

WHAT *ELSE* COULD POSSIBLY GO WRONG? II

```c
void foo(Struct* x, Struct* y)
{
    bar(x);
    free(x);
    bar(y);
}
```

C

Potential memory safety issues

- bar might delete the value of x
- free(x) will attempt to delete the value of x again

➔ **double-free bug**

WHAT *ELSE* COULD POSSIBLY GO WRONG? III

```c
void foo(Struct* x, Struct* y)
{
    bar(x);
    free(x);
    bar(y);
}
```

C

Potential memory safety issues

- If x and y point do different values and bar does not delete anything, then the value of y might never be deleted

➔ **memory leak**

MAIN REASONS FOR MEMORY SAFETY ISSUES IN C

1. Manual disposal of heap locations

2. Mutable aliasing

```c
void foo(Struct* x, Struct* y)
{
    assert(x == y);
    free(x);
    Struct z = *y
}
```

C

➔ What are better memory disposal strategies?

DISPOSAL STRATEGIES FOR HEAP MEMORY

Metaphor: how to keep the office tidy?

MANUAL DISPOSAL

- examples: C, C++
- very efficient
- no safety guarantees

➔ "control first"

GARBAGE COLLECTOR

- examples: Java, C#
- ensures safety at runtime
- expensive

➔ "safety first"

OWNERSHIP SYSTEM

- examples: Rust
- safety at compile time
- efficient

➔ both: "clean desk policy"

3. OWNERSHIP

HOW RUST ACHIEVES MEMORY SAFETY (FOR CODE WITHOUT POINTERS)

OWNERSHIP RULES – PART 1

1. For every value there is a unique place, called its owner
2. A value is disposed (or “dropped”) when its owner leaves scope
3. Variables own their values

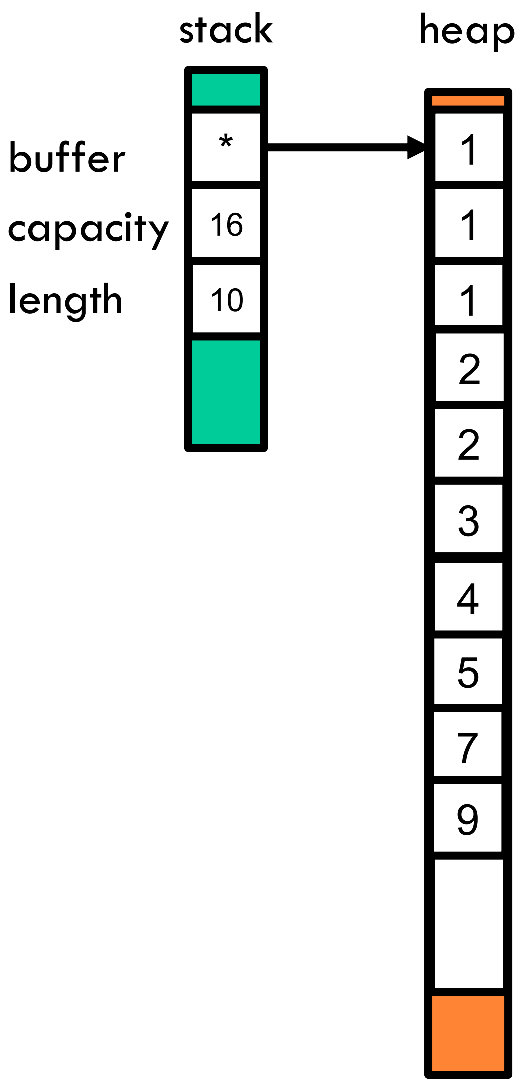
EXAMPLE

```
fn main() {

  let mut v = vec![1,1,1];

  for i in 3..10 {
    let next = v[i-3] + v[i-2];
    v.push(next);
  }
  println!("P(1..10) = {:?}", v);

}
```

allocate new vector with owner v

manipulate vector

v runs out of scope ➔ drop vector



OWNERSHIP RULES – PART 2

1. For every value there is a unique place, called its owner
2. A value is disposed (or “dropped”) when its owner leaves scope
3. Variables own their values
4. Composite types (structs, tuples, vectors, ...) own their elements

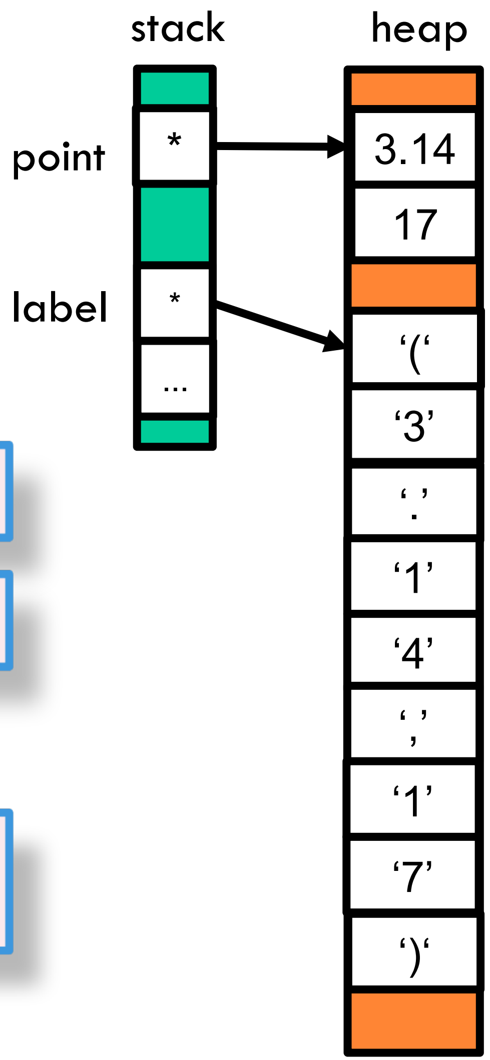
EXAMPLE

```
fn main() {
    let point = Box::new( (3.14, 17) );

    let label = format!(“{:?}”, point);
    assert_eq!(label, “(3.14, 17)”);
}
    // label and point are out of scope
    // drop owned string and tuple
    // drop 3.14:f32, 17:i32
    //      and characters of string
```

allocate tuple with owner point

allocate string with owner label

drop label, point, and their owned values



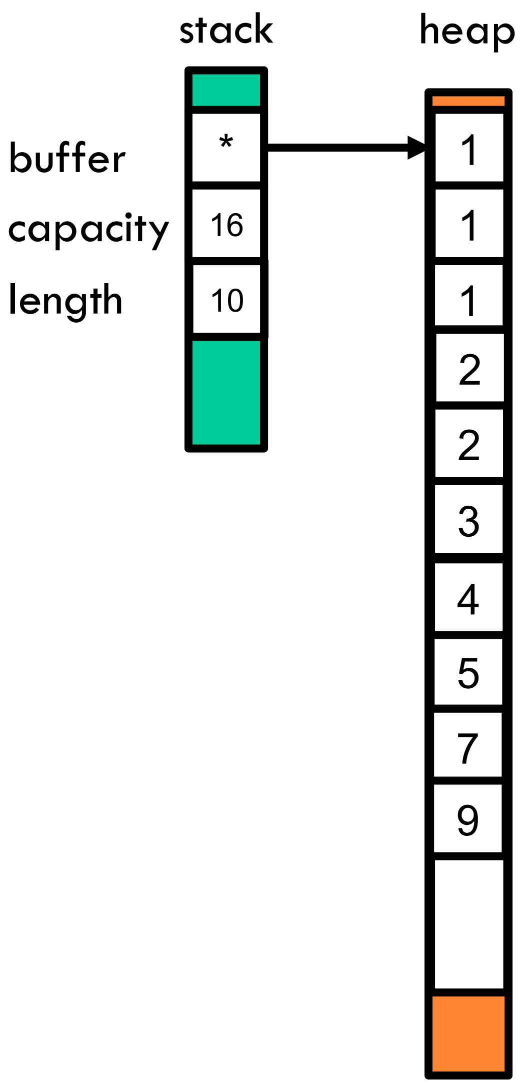
EXAMPLE

```
fn main() {

  let mut v = vec![1,1,1];

  for i in 3..10 {
    let next = v[i-3] + v[i-2];
    v.push(next);
  }
  println!("P(1..10) = {:?}", v);

}
```

allocate new vector with owner v

manipulate vector

v runs out of scope ➔ drop vector

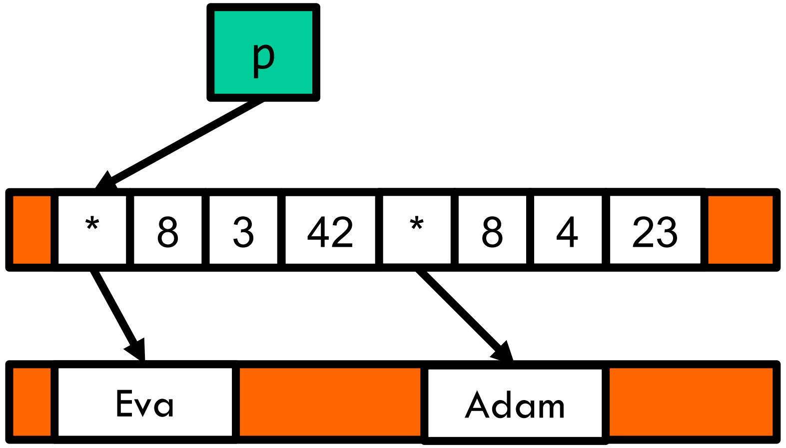


LIMITATIONS

- Memory consists of ownership trees with variables at the root
- All values are dropped when leaving a function's scope

  ➔ Move ownership to a new owner

```
struct Person { name: String, age: i32 }

let mut p = Vec::new();
p.push(Person{ name: “Eva”, age: 42});
p.push(Person{ name: “Adam”, age: 23});

// ...
```



OWNERSHIP RULES – PART 3

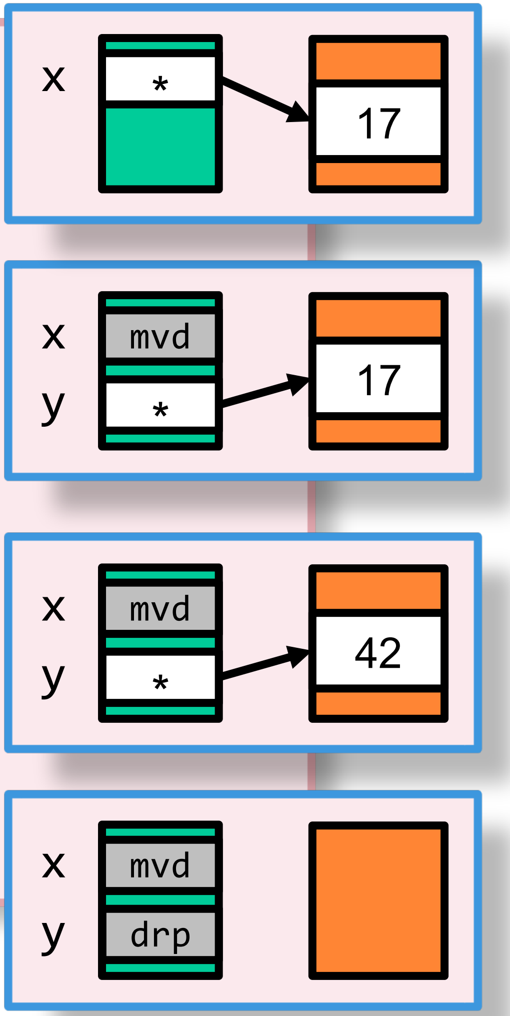
1. For every value there is a unique place, called its owner
2. A value is disposed (or “dropped”) when its owner leaves scope
3. Variables own their values
4. Composite types (structs, tuples, vectors, …) own their elements
5. Ownership can be moved to a new owner

   ➔ the old owner becomes an uninitialized place

   ➔ accessing the old owner is forbidden until it is initialized again

EXAMPLE

```
fn main() {

  let mut x = Box::new(17);


  let mut y = x;

  // fails: let z = x

  *y = 42;

  assert!(*y == 42);

}
```



Operations that move

- assignments

```
let x = Box::new(17)
```

- passing values to a function

```
foo(Person { age: 32, ... })
```

- returning values from a function

```
fn bar(n: String) -> Person {
  Person { age: 32, name: n }
}
```

MENTAL MODELS FOR UNDERSTANDING OWNERSHIP

- **Low-level model:** “what’s actually happening”
  - Variables are places that hold possibly illegal bytes
  - Ownership rules guide how long a variable is accessible

- **High-level model:** “how we can reason about ownership”
  - A variables exists as long as there is a capability flow to it
  - and parallel flows do not conflict each other

CAPABILITY FLOWS

Idea: annotate programs with flows for each owner

- ▶ Taking ownership of a place starts a new flow (color indicates the owner)
- ◀ Moving a place stops the flow
- ⬇ Accessing a place adds a flow from the last access to the current access
- ⬇ mutable flow for values that can be modified (keyword “mut”)
- ⇩ immutable flow for values that cannot be modified

EXAMPLE

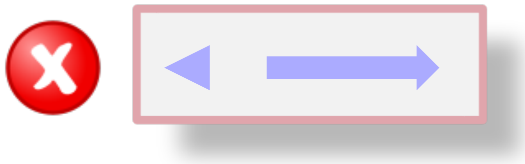
```
fn main() {
  let mut x = Box::new(17);

  let mut y = x;

  let z = x;

  *y = 42;

  assert!(*y == 42);
}
```

There are two (mutable) flows

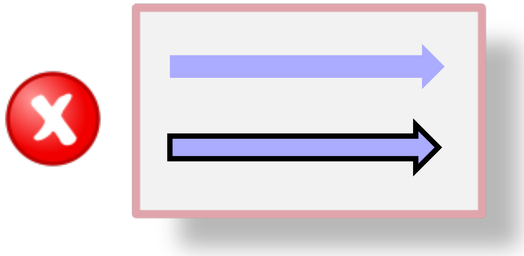
RUST'S FLOW-SENSITIVE ANALYSIS FOR OWNERSHIP

Checking ownership: check that all flows are compatible

1. No access after move: no flow from an end-of-flow marker ◀ to a place



2. Parallel flows for the same place (same color) must be immutable

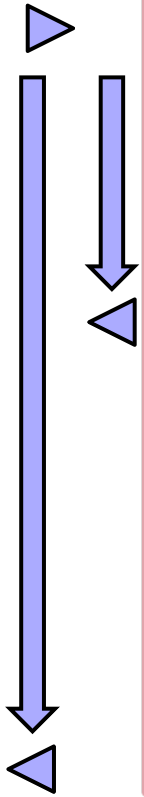


EXAMPLE

```
fn main() {
  let mut x = Box::new(17);

  let mut y = x;

  let z = x;

  *y = 42;

  assert!(*y == 42);
}
```

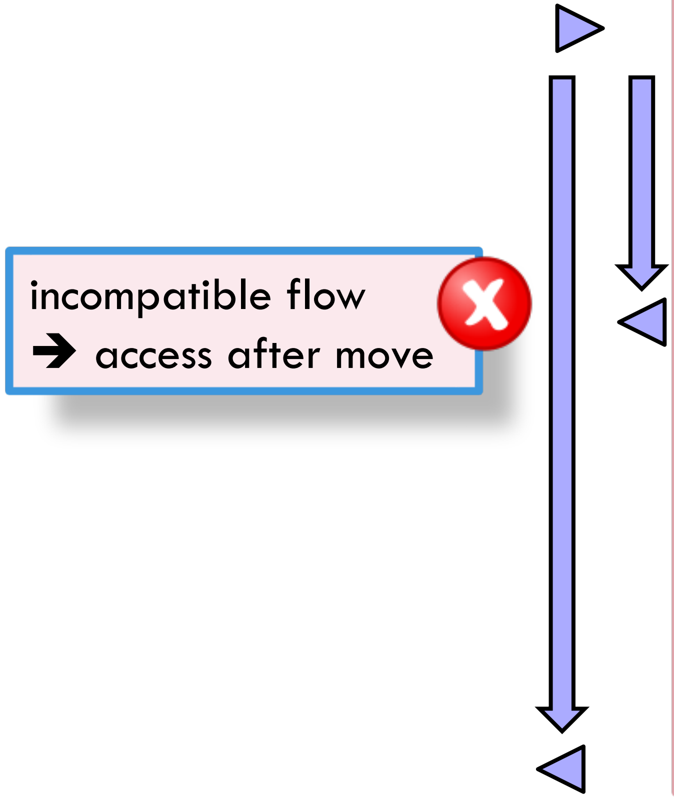
incompatible flow
➔ access after move

ARE ALL FLOWS COMPATIBLE?



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

if y > 0 {

  f(x); // x is moved to f

} else {

  println!(“blabla”);

}

f(x); // x is moved to f
```

ARE ALL FLOWS COMPATIBLE? NO!




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

if y > 0 {

  f(x); // x is moved to f

} else {

  println!("blabla");

}

f(x); // x is moved to f
```

If a place has been moved in one branch of a control flow statement and has not **definitely** been given a new value, it is uninitialized after the statement.

ARE ALL FLOWS COMPATIBLE?

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

while y > 0 {

  foo(x); // x is moved to foo

  y = y - 1;


}
```

ARE ALL FLOWS COMPATIBLE? NO!

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

while y > 0 {

  foo(x); // x is moved to foo

  y = y - 1;


}
```

flow across
loop iterations!

incompatible flow
➔ access after move

ARE ALL FLOWS COMPATIBLE?

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

while y > 0 {

   foo(x); // x is moved to foo

   y = y - 1;

   x = bar() // move to x
}
```

ARE ALL FLOWS COMPATIBLE? YES!

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

while y > 0 {

  foo(x); // x is moved to foo

  y = y - 1;

  x = bar() // move to x
}
```

OK: x is re-initialized in the loop ➔ new flow

IN WHAT LINES CAN WE DETECT INCOMPATIBLE FLOWS?

```
fn notify(v: Vec<Person>)
    -> Vec<Person> {
  for i in &v { println!("{}", i); }
  v
}
```

```
fn P() -> mut Vec<Person> { vec![
  Person { name: "Adam", age: 27 },
  Person { name: "Eva", age: 42 },
  Person { name: "Chris", age: 32 },
]}
```

```
1 fn main() {
2   let mut ids = P();
3   ids.push(Person { ... });
4   notify(ids);
5 }
```

```
6  fn main() {
7    let mut ids = P();
8    ids = notify(ids);
9    ids.push(Person { ... });
10 }
```

```
11 fn main() {
12   let mut ids = P();
13   notify(ids);
14   notify(ids);
15 }
```

```
16 fn main() {
17   let mut ids = P();
18   let x = ids[0];
19   ids = notify(ids);
20   let y = x;
21 }
```

IN WHAT LINES CAN WE DETECT INCOMPATIBLE FLOWS?

```
fn notify(v: Vec<Person>)
    -> Vec<Person> {
  for i in &v { println!("{}", i); }
  v
}
```

```
fn P() -> mut Vec<Person> { vec![
  Person { name: "Adam", age: 27 },
  Person { name: "Eva", age: 42 },
  Person { name: "Chris", age: 32 },
]}
```

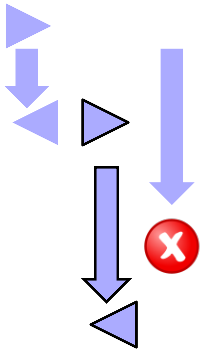
```
1 fn main() {
2   let mut ids = P();
3   ids.push(Person { ... });
4   notify(ids);
5 }
```

```
6  fn main() {
7    let mut ids = P();
8    ids = notify(ids);
9    ids.push(Person { ... });
10 }
```

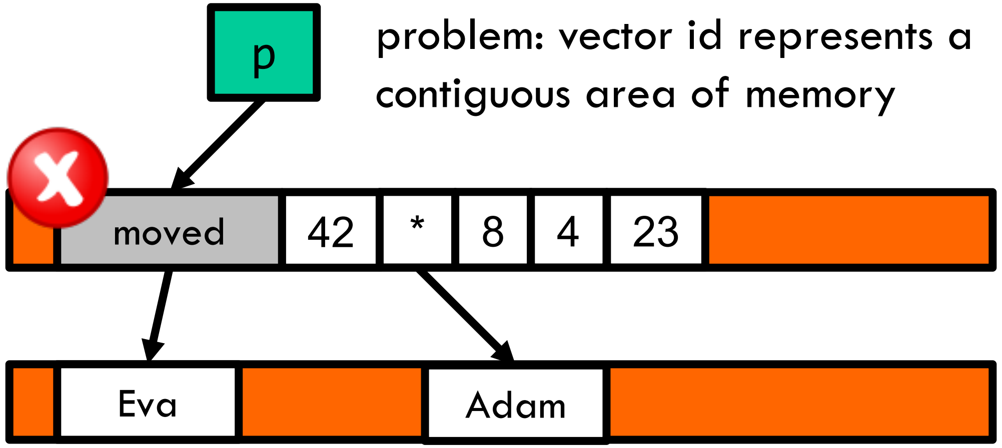
```
11 fn main() {
12   let mut ids = P();
13   notify(ids);
14   notify(ids);
15 }
```

```
16 fn main() {
17   let mut ids = P();
18   let x = ids[0];
19   ids = notify(ids);
20   let y = x;
21 }
```

MOVES AND DATA STRUCTURES



```
fn main() {
  let mut ids = P();
  let x = ids[0];
  ids = notify(ids);
  let y = x;
}
```



problem: vector id represents a contiguous area of memory

One cannot move values out of data structures that do not permit holes in their representation, e.g. vectors or arrays

MOVING IN AND OUT OF VECTORS

Approach 1: Only move the last value and resize the vector

```
let x = ids.pop().expect("vector empty");
```

Approach 2: Swap a value with the last value before move

```
let x = ids.swap_remove(0);
```

Approach 3: Swap in another value for the one we take out

```
let x = std::mem::replace(&mut ids[0], Person { ... });
```

Approach 4: Create a new copy of an element instead of moving it

```
let x = ids[0].clone();
```

EXCEPTION: COPY TYPES

- Main advantage of ownership: safe & efficient disposal of resources

```
f32, f64, char, bool, usized, u8, i8, i32, ...
```

- No advantage for simple types that only manage their own bits

  ➔ These types are always copied bitwise instead of moved

```
let mut ids = vec![1,2,3];
let x = ids[0]; // x is a copy of ids[0]
notify(ids); // nothing has been moved out of ids
```

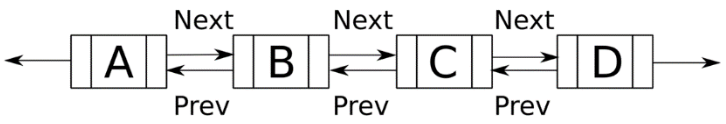
- Custom copy types may only consist of other copy types

```
#[derive(Copy, Clone)]
struct Point { name: i32, age: i32 }
```

EXCEPTION: REFERENCE COUNTING

- Ownership enforces tree data structures

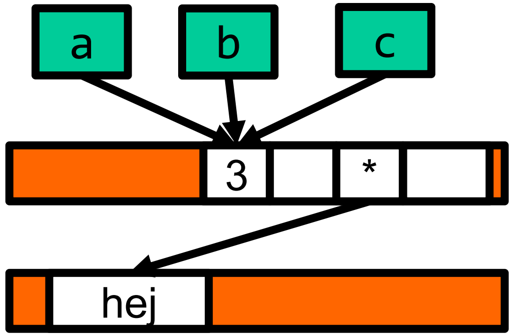
➔ It may be unclear who should own a resource



Who should own A,B,C,D?

- Escape hatch: reference-counted pointers (Rc)
  - Clone only increments reference count
  - Resources are dropped once count falls to zero
  - Safety: Rc pointers can only be immutable

```
let a = Rc::new(“hej”.to_string());
let b = a.clone();
let c = a.clone();
```



4. REFERENCES

HOW RUST CHECKS MEMORY SAFETY FOR CODE WITH POINTERS

WHAT COULD POSSIBLY GO WRONG?

```
use std:collections::HashMap;

// table of authors and their books
type Table = HashMap<String, Vec<String>>;

fn print(table: Table) {
  for (author, books) in table {
    println!(“works by {}”, author);
    for book in books {
      println!(“    {}”, book);
    }
  }
}
```

We move a table into print(…) but never give it back

➔ printing a table also destroys it!

WHY BORROWING?

Pointer: value holding the address of a place

```
fn average_age(a: Person, b: Person)
  -> (Person, Person, f32)
{
  (a, b, (a.age + b.age) / 2)
}
```

```
// so far: owning pointers
let x = Box::new(Person {...});
// drop x => drop person on the heap
```

```
let alice = Person { ..., age: 42 };
let bob = Person {..., age: 23 };


let (alice, bob, avg) =
    average_age(alice, bob);
```

unergonomic solution: move values back to their original owner

```
let (alice, ???, avg) =
average_age(alice, alice)
```

error: can move only once!

REFERENCES (OR *BORROWS*)

References: pointers with a specific contract that temporarily borrow ownership.

Shared references &T

- Create as many as you want
- Read-only
- Copy type

- On creation: stop mutable flow
  ➔ owner cannot modify its value
- All dropped: mutable flow returns to owner

- Compiler may assume that the pointed to value does not change until the reference is dropped

Mutable references &mut

- Create one with exclusive access for each place
- Read and write
- not a copy type

- On creation: stop all flows from borrowed place
  ➔ owner cannot access its value
- All dropped: flow returns to owner

- Compiler may assume exclusive access to the immediate location pointed to (accessible via *)

PRINTING A TABLE WITHOUT DESTROYING IT

```
use std:collections::HashMap;

// table of authors and their books
type Table = HashMap<String, Vec<String>>;

fn print(table: &Table) {
  // implicitly dereferenced to *table
  for (author, books) in table {
    println!("works by {}", author);
    for book in books { // implicitly uses &Table
      println!("    {}", book);
    }
  }
}
```

use shared reference

```
print(&table);
table.insert("Nichols".to_string(),
             vec!["Rust Book".to_string()]);
```

ownership returns to callee

WHAT COULD POSSIBLY GO WRONG?

```
fn caching(input: &i32, sum: &mut i32) {

  // *input is the value input points to
  *sum = *input + *input;

  // can this fail?
  assert_eq!(*sum,  2 * (*input));

}
```

The assertion never fails. The compiler is in its rights to read the value behind a shared reference only once.

In particular, we know that input and sum point to different values!

DO THESE PROGRAMS BEHAVE THE SAME?

```rust
fn fun(input: &i32, sum: &mut i32) {
  if *input == 1 {
    *sum = 2
  }
  if *input != 1 {
    *sum = 1
  }
}
```

```rust
fn fun(input: &i32, sum: &mut i32) {
  if *input == 1 {
    *sum = 2
  } else {
    *sum = 1
  }
}
```

Yes, the compiler can exploit that sum and input do not alias

➔ modifying the value of sum does not affect the value of input

WHICH STATEMENTS ARE LEGAL?

```
let mut a = 42;
let b = 23;
let mut y = &a;
a = 19;
*x = 17;
*y = &b;
a = b;
let z = &mut y;
*z = &a;
**z = b;
```

WHICH STATEMENTS ARE LEGAL?

```
let mut a = 42;
let b = 23;
let mut y = &a;
a = 19; //illegal: a is borrowed
*y = 17; //illegal: y is a mutable shared borrow
y = &b; //ok: y is mutable
a = b; //ok: a is not borrowed
let z = &mut y; //ok: mutable borrow of shared b.
*z = &a; //ok
**z = b; //illegal: write through shared borrow
```

TAKING OWNERSHIP VS. MUTABLE REFERENCES

- Mutable references are not responsible for dropping resources
- Otherwise, having a mutable reference is almost identical to owning a value
- Exception: moving values behind mutable references

```
fn moves(x: &mut Box<i32>) {
  let y = *x;
}
```

Bad: when the flow returns to the owner, we might attempt to drop a value twice!

```
fn moves(x: &mut Box<i32>) {
  let y = std::mem::take(x);

  let mut z = Box::new(17);
  std::mem::swap(x, &mut z);
}
```

Ok: leave another value in place

Ok: swap mut. refs without owning them

5. THE FLOW MODEL FOR REFERENCES

A MENTAL MODEL FOR CHECKING MEMORY SAFETY USING BORROWS AND LIFETIMES

HOW RUST VALIDATES REFERENCES

- Rust's borrow checker ensures that references are safe
  - No reference is used after it has been dropped
  - Shared references are read-only
  - Mutable references give exclusive access
- Analysis matches our mental model of capability flows
  - Check that the flow of every reference we access does not conflict with parallel flows
  - Moves and borrows create new and may block ▲ other flows
  - Flow of reference ends: unblock ▼ flow of borrowed-from place
- Rust assigns a name to flows and calls them lifetimes

Lifetime constraint: a variable's lifetime must contain the lifetime of its borrows.

EXAMPLE I

Lifetime constraint: a variable's lifetime must contain the lifetime of its borrows.

```
let r;
{

  let x = 1;

  r = &x;

}

assert_eq!(*r, 1);
```

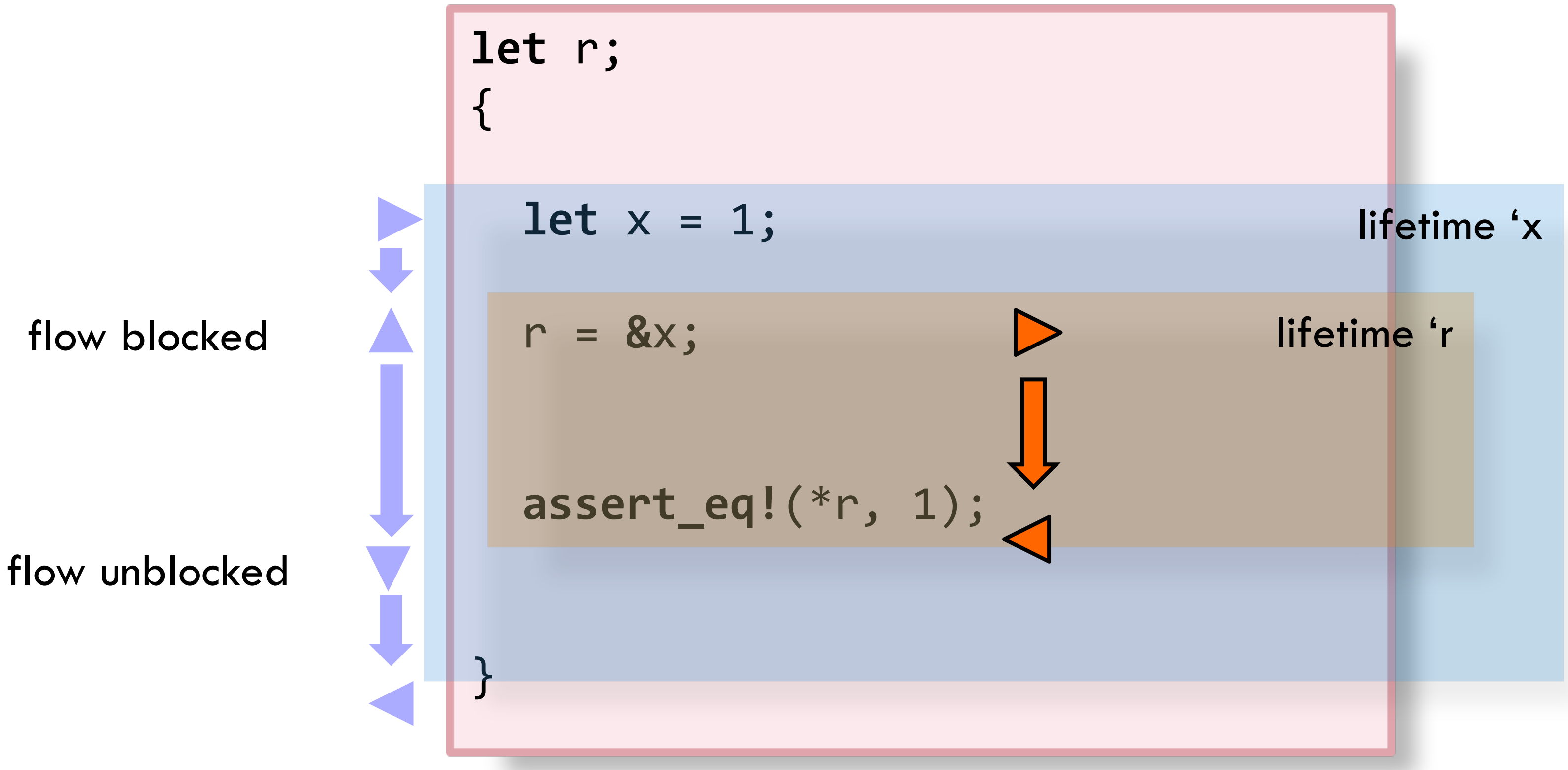
lifetime 'x

lifetime 'r

conflicting flows ➔ reject

EXAMPLE II

Lifetime constraint: a variable's lifetime must contain the lifetime of its borrows.



no conflicting flows ➔ accept

EXAMPLE III

Lifetime constraint: a variable's lifetime must contain the lifetime of its borrows.

```
let mut x = Box::new(42);
let r = &x;
if rand() > 0.5 {
  *x = 84;
} else {
  println!("{}", r);
}
println!("{}", x);
```

lifetime 'a

r is not needed in this branch
➔ expire and unblock x

lifetime 'a

r is needed in this branch
➔ x still blocked

no conflicting flows ➔ accept
even though lifetime appears to have "holes"

EXAMPLE IV

Lifetime constraint: a variable's lifetime must contain the lifetime of its borrows.

```
let mut x = Box::new(42);
let r = &x;
if rand() > 0.5 {

  *x = 84;
} else {
  println!("{}", r);
}
println!("{}", r);
```

lifetime 'a

What happens if we replace x by r in the last line?

conflicting flows ➔ reject

EXAMPLE V

Lifetime constraint: a variable's lifetime must contain the lifetime of its borrows.

```
let mut x = Box::new(42);
let mut z = &x;
for i in 0..100 {
  println!("{}", z);
  x = Box::new(i);
  z = &x;
}
println!("{}", z);
```

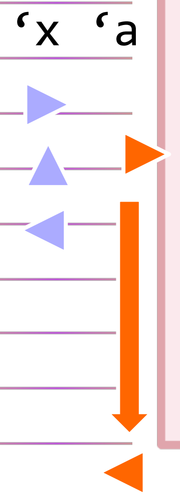
lifetime 'z

lifetime 'z

no conflicting flows ➔ accept

LIFETIMES IN CUSTOM TYPES

```
struct S {
  r: &i32
}

// ...

let s;
{
  let x = 10;
  s = S { r : &x };
}

// bad: reads from dropped x
assert_eq!(*s.r, 10);
```




```
error: missing lifetime specifier
   |
   | r: &i32
   |    ^ expected lifetime parameter
```

How does the borrow checker validate the lifetimes of references inside of structs?
➔ **lifetime annotations**

```
struct S {
  r: &'static i32
}
```

r can only refer to values that live until program termination

```
struct S<'a> {
  r: &'a i32
}
```

each instance of S gets a new lifetime constrained by usage

EXPLICIT LIFETIME PARAMETERS

Explicit lifetime parameters reveal whether there are non-static references and how their lifetimes are related

```
struct S<'a> {
  r: &'a i32
}

struct D {
  s: S<'static>
}
```

Restrictive:
D can only borrow values that live for the entire program

```
struct S<'a> {
  r: &'a i32
}

struct D<'a> {
  s: S<'a>
}
```

Permissive:
D can borrow any values, including those in local scope

read <'a> as
for any lifetime 'a

LIFETIMES OF FUNCTIONS

```
fn g<'a>(p: &'a i32) { ... }


let x = 10;
g(&x) // ok: x flows into the call
```

read as: any lifetime that contains g works for 'a

```
fn g(p: &'static i32) { ... }


let x = 10;
g(&x) // fail: &x does not
      //       live until termination
```

read as: parameter must live until termination

result must live at least as long as input

```
fn parse<'a>(input: &'a [u8]) -> Record<'a> { ... }
```

"whatever (non-static) references the returned record contains, they must point into the input buffer"

Rust can often infer lifetimes for functions automatically

EXAMPLE

result must live at least as long as values

add flow from values to result

```
fn min<'a>(values: &'a [i32]) -> &'a i32 {
  let mut s = &v[0];
  for r in &v[1..] {
    if *r < *s {
      s = r;
    }
  }
  s
}
```

```
let s;
{
  let values = [7, 4, 1, 0, 1, 4, 7];
  s = min(&values)
}
assert_eq!(*s, 0);
```

conflicting flow from values to s

error: `values` does not live long enough

EXAMPLE CONTINUED

result must live at least as long as values

add flow from values to result

```
fn min<'a>(values: &'a [i32]) -> &'a i32 {
  let mut s = &v[0];
  for r in &v[1..] {
    if *r < *s {
      s = r;
    }
  }
  s
}
```

```
let s;
{
  let values = [7, 4, 1, 0, 1, 4, 7];
  s = min(&values);
  assert_eq!(*s, 0);
}
```

no conflicting flows

error: `values` does not live long enough

MULTIPLE LIFETIME PARAMETERS

One lifetime is sufficient unless a method returns a subset of a type's references

```
struct StrSplit<'s, 'p> {
  delimiter: &'p str,
  document: &'s str,
}

impl<'s, 'p> Iterator for StrSplit<'s, 'p> {
  type Item = &'s str;
  fn next(&self) -> Option<Self::Item> { ... }
}

fn str_before(x: &str, c: char) -> Option<&str> {
  StrSplit {
    document: x, delimiter: &c.to_string()
  }.next()
}
```

we get a reference into the original document

for 's = 'p, result would be constrained by the document *and* a local variable
➔ not possible

CHECKING LIFETIME PARAMETERS

Lifetimes parameters are types that interact with the borrow checker

A type's variance describes which types can be used in its place

- Covariance
- Invariance
- Contravariance

'static subtype of 'a

read: outlives

COVARIANT LIFETIMES

’static subtype of ’a

read: outlives

- Allow subtypes instead of the actual type
- T subtype S implies C<T> subtype C<S>
- &’a T is covariant in ’a and T

```
fn foo(x: &Vec<&’a str>) { ... }
let y: &Vec<&’static str> = ...;
foo(y) // ok
```

INVARIANT LIFETIMES

'static subtype of 'a

read: outlives

- Allow only the exact type
- &mut T is invariant

```
fn foo(x: &Vec<&'a str>) { ... }
let y: &Vec<&'static str> = ...;
foo(y) // ok
```

CONTRAVARIANT LIFETIMES

'static subtype of 'a

read: outlives

- Allow supertypes instead of the actual type
- T subtype S implies Fn(S) subtype Fn(T)
- Fn(T) is contravariant in T

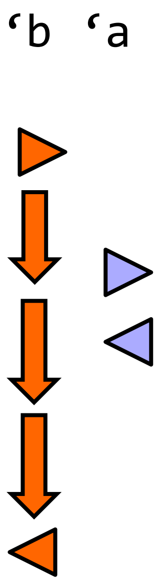
```
// &'static str outlives &'a str
fn f(&'static str) // admits only 'static
fn g(&'a str) // admits any lifetime 'a
```

LIFETIME PUZZLE

- Should the Rust compiler accept this?
- Are both lifetime parameters needed?

```
struct MutString<‘a, ‘b> {
  s: &’a mut &’b str
}

fn main() {
  let mut s = “hello”;
  *MutString { s: &mut s }.s = “world”;
  println!(“{}”, s);
}
```

PUZZLE SOLUTION




```
struct MutString<‘a, ‘b> {
  s: &’a mut &’b str
}

fn main() {

  let mut s = “hello”;

  *MutString { s: &mut s }.s = “world”;

  println!(“{}”, s);

}
```

‘b = ‘static, lifetime of “hello”

‘a: lifetime of &mut s

covariance: ’static str shortened to ‘a str such that print can borrow from s

No conflicting flows ➔ accept

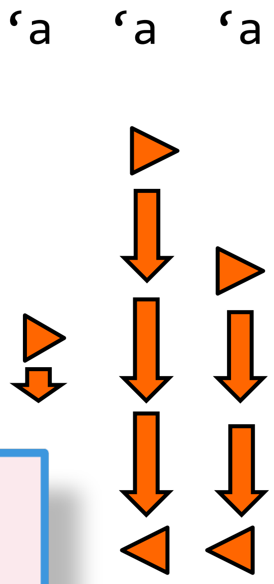
PUZZLE SOLUTION II

```
struct MutString<’a> {
  s: &’a mut &’a str
}

fn main() {

  let mut s = “hello”;

  *MutString { s: &mut s }.s = “world”;

  println!(“{}”, s);

}
```

’a = ‘static, lifetime of “hello”

’a: lifetime of &mut s

&’static mut str is invariant and cannot be shortened



conflicting flows: attempt to borrow while there is a mutable reference
➔ reject

One lifetime is insufficient

EXCEPTION: INTERIOR MUTABILITY

- Some types allow sharing *and* mutation
- Those types maintain the abstraction

  "exclusive read-write access XOR shared read-only access"

- These types are safe but rely on external safety mechanisms (e.g. locks)
- Two main kinds of interior mutability
  - Mutex, RefCell: get a mutable reference through a shared reference
  - Cell, sync::Atomic: replace an immutable value

EXAMPLE: MUTEX

```rust
fn critical(mutex: &Mutex<Data>) {
  // get mutable reference
  // block read access from others
  let mut data = mutex.lock();

  data.payload = 23;

  // drop data => drop exclusive access
  //           => release lock
}
```

EXAMPLE: CELL

```
struct Robot { count: Cell<u32>, ... }
impl Robot {
  fn add_error(&self) {
    let n = count.get();
    self.count.set(n+1); // why ok?
  }
  fn has_errors(&self) -> bool {
    self.count.get() > 0
  }
}
```

WRAP-UP: RUST'S MEMORY SAFETY GUARANTEES

Rust enforces aliasing XOR mutation
and requires synchronization for exceptions

Components

- Ownership system: for every value a unique owner is in charge of disposal
- Borrow checker: references are only used when they are valid
- Reference contracts: exclusive write-access XOR shared read-only access

6. PRUSTI

OBTAINING GUARANTEES BEYOND MEMORY SAFETY

WHAT COULD POSSIBLY GO WRONG?

Task: write a Rust program that returns the absolute value of an integer (type: i32) x

```
fn abs(x:i32) -> i32 {
  if x >= 0 {
    x
  } else {
   -x
  }
}
```

i32: 32-bit integers in two's complement!

```
i32::MIN    is    -2_147_483_648i32
i32::MAX    is     2_147_483_647i32
abs(i32::MIN) == ???
```

This is a safe Rust program

But: it's also logically wrong!

➔ Rust does **not** guarantee functional correctness

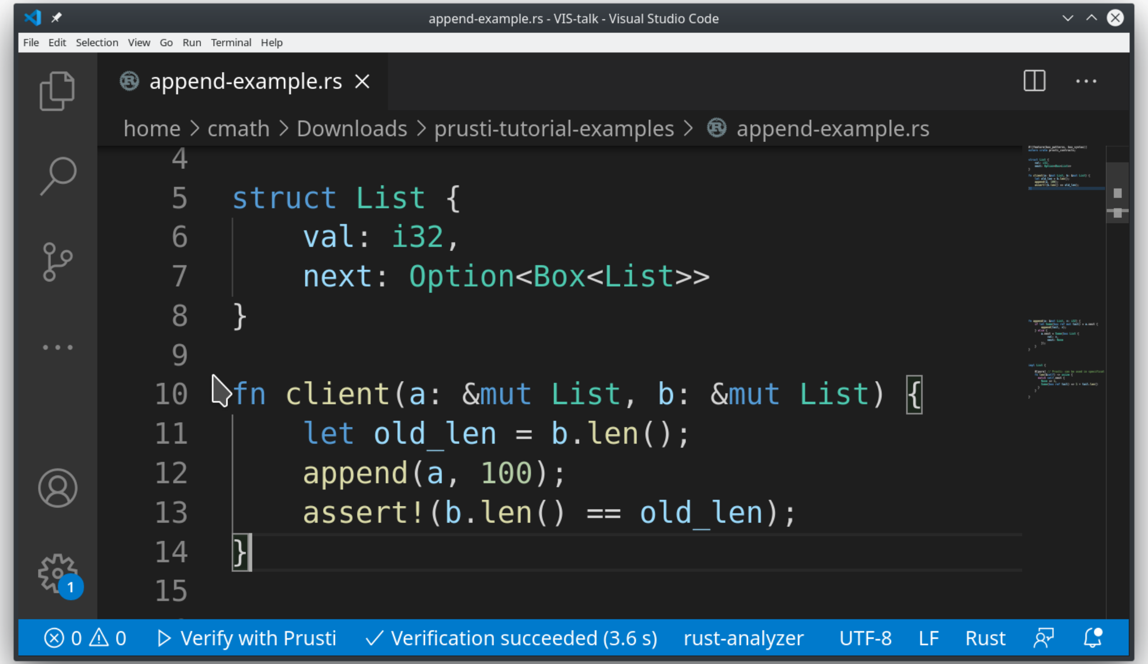
BEYOND MEMORY SAFETY

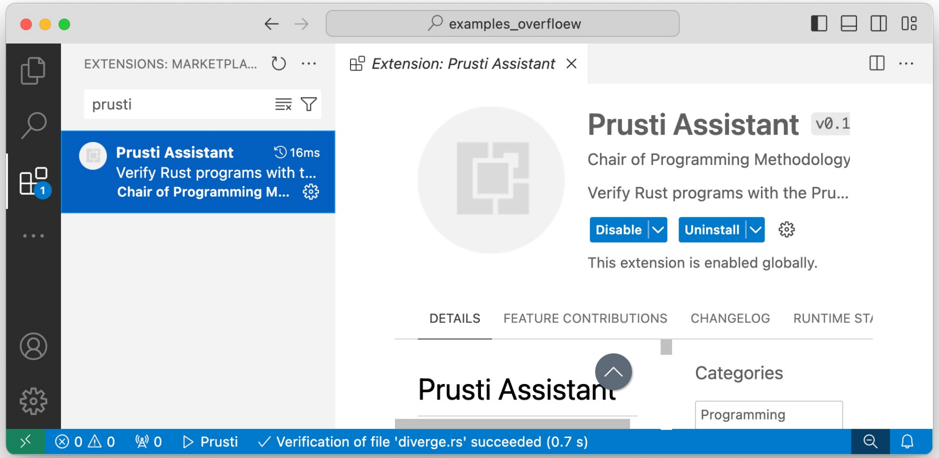
- Rust comes with compile-time safety guarantees
  - no uninitialized values, no dangling pointers, no data races
  - no double-free, null pointer, or use-after-free bugs
  - prevents many (but not all) memory leaks
- Memory safety is enforced by checking privileges and obligations
  - Ownership, borrowing, lifetimes
  - The Rust compiler requires annotations to check safety

➔ Can we trade writing more annotations for stronger correctness guarantees to also avoid logical security flaws?

THE PRUSTI VERIFIER

- Tool for checking functional correctness of Rust functions
- Implemented as a compiler plugin
- Checks may require contract annotations written in a subset of Rust
- Open-source VSCode plugin
  - Can be installed via marketplace
  - Search for “Prusti Assistant”
  - Needs Java runtime

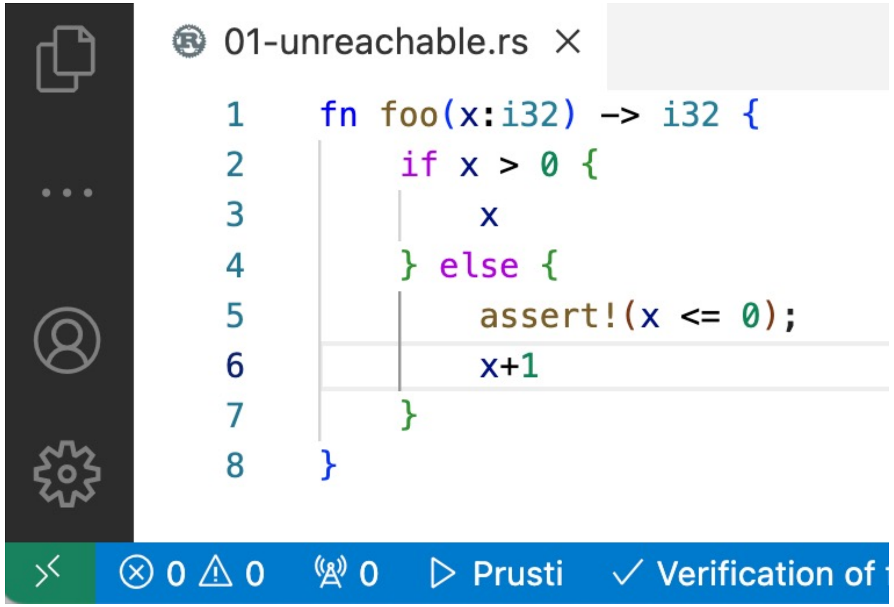


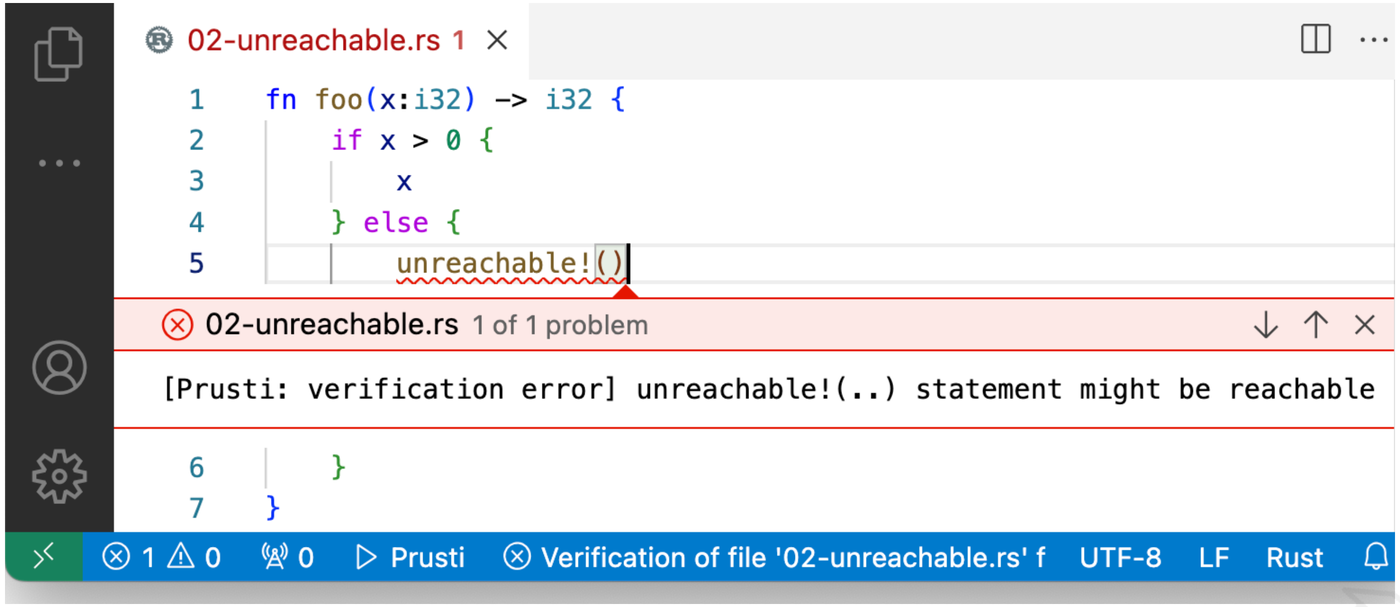


ABSOLUTE VALUE REVISITED

ASSERTIONS

- Prusti checks that no Rust assertion fails





- Conservative approach: compilation fails if correctness cannot be proven

➔ Requires annotations about inputs and outputs of functions

CONTRACTS

- Constrain inputs and results of functions
  - requires keyword constrains inputs
  - ensures keyword constrains outputs
  - Constraints must be side-effect-free, terminating Rust expressions
- Function implementors
  - **Privilege:** assume inputs comply with contract
  - **Obligation:** results must comply with contract
- Function clients
  - **Privilege:** assume results comply with contract
  - **Obligation:** inputs must comply with contract

Precondition:
all 32-bit integers but the smallest one are ok

```
#[requires(x != i32::MIN)]
#[ensures(result >= 0)]
#[ensures(result*result == x * x)]
fn abs(x:i32) -> i32
```

Postcondition:
the function's result will be the absolute value of x

MEANING OF CONTRACTS

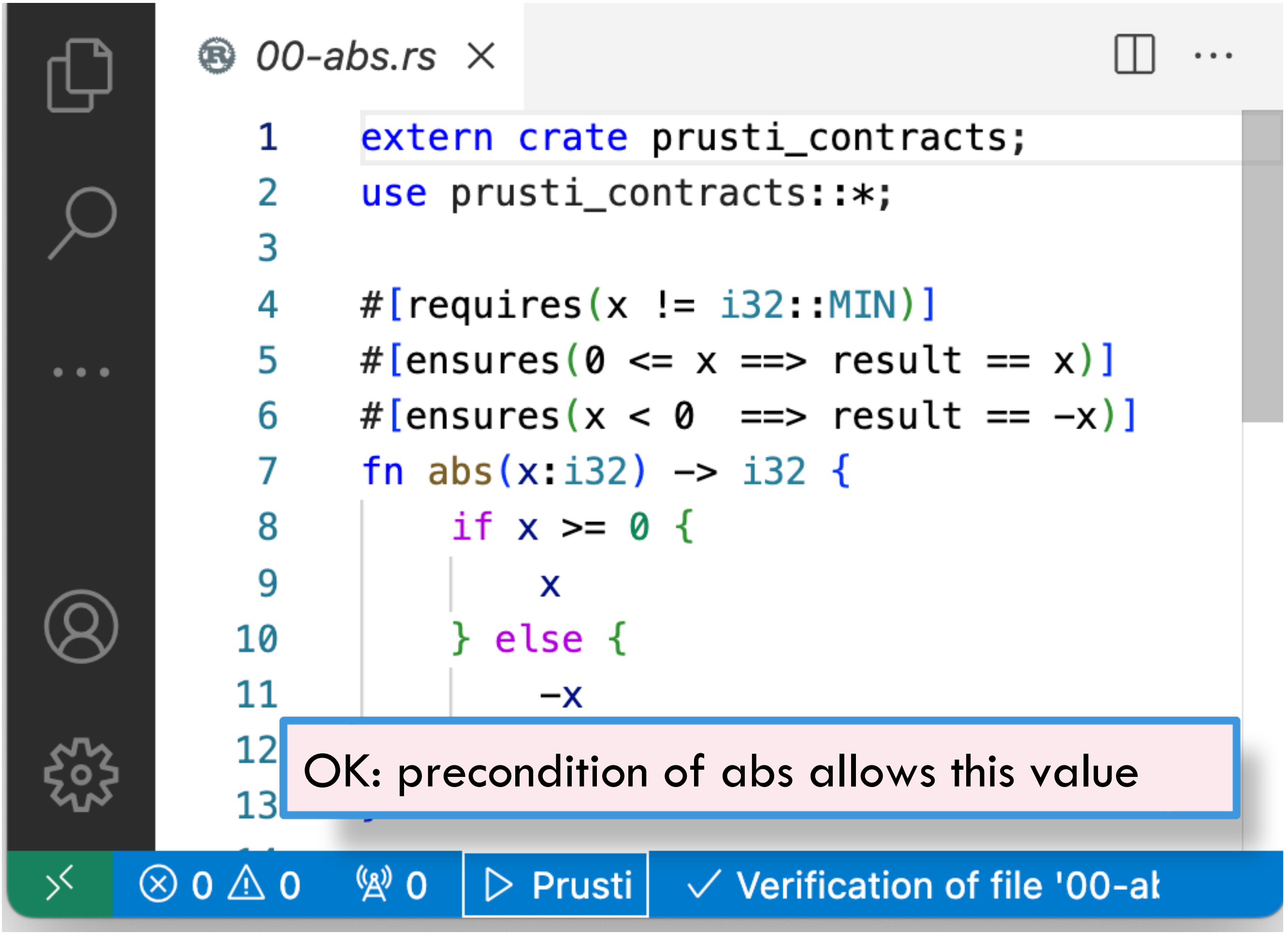
Precondition (before call):
all 32-bit integers but the smallest one are ok

Postcondition (after call):
the function's result will be the absolute value of x

```
#[requires(x != i32::MIN)]
#[ensures(result >= 0)]
#[ensures(result*result == x * x)]
fn abs(x:i32) -> i32
```

Whenever we execute a function whose parameters satisfy the precondition *and execution terminates*, then *no run-time error occurs* (e.g. an assertion failure) during execution *and* the postcondition holds upon termination.

ABSOLUTE VALUE WITH CONTRACT



Needed for using contract annotations

Precondition (“requires”):
assume x is not the smallest integer

Postcondition (“ensures”):
Prusti proves that the returned result is the absolute value of x

A CLIENT OF ABS

00-abs.rs 1

```
fn client() {
    let a = abs(i32::MIN + 1);

    assert!(0 <= a);
    assert!(a == i32::MAX);

    let b = abs(i32::MIN);
```

00-abs.rs 1 of 1 problem

```
[Prusti: verification error] precondition might not hold.

00-abs.rs(4, 12): the failing assertion is here
```

1 0 0 Prusti Verification of file '00-abs.rs' fai Rust

OK: precondition of abs allows this value

assertions are known to hold due to postcondition of abs

ERROR: precondition of abs does not allow passing the smallest integer!

MODULAR CONTRACT VERIFICATION

- Prusti proves that every function meets its contract
  - Default: pre- and postcondition are true
- Modular verification
  - To check calls, Prusti relies solely on the called function's contract
  - Pros: Implementation changes ➔ clients do not have to be re-checked
  - Cons: Possible false negatives if we do not write sufficiently strong contracts

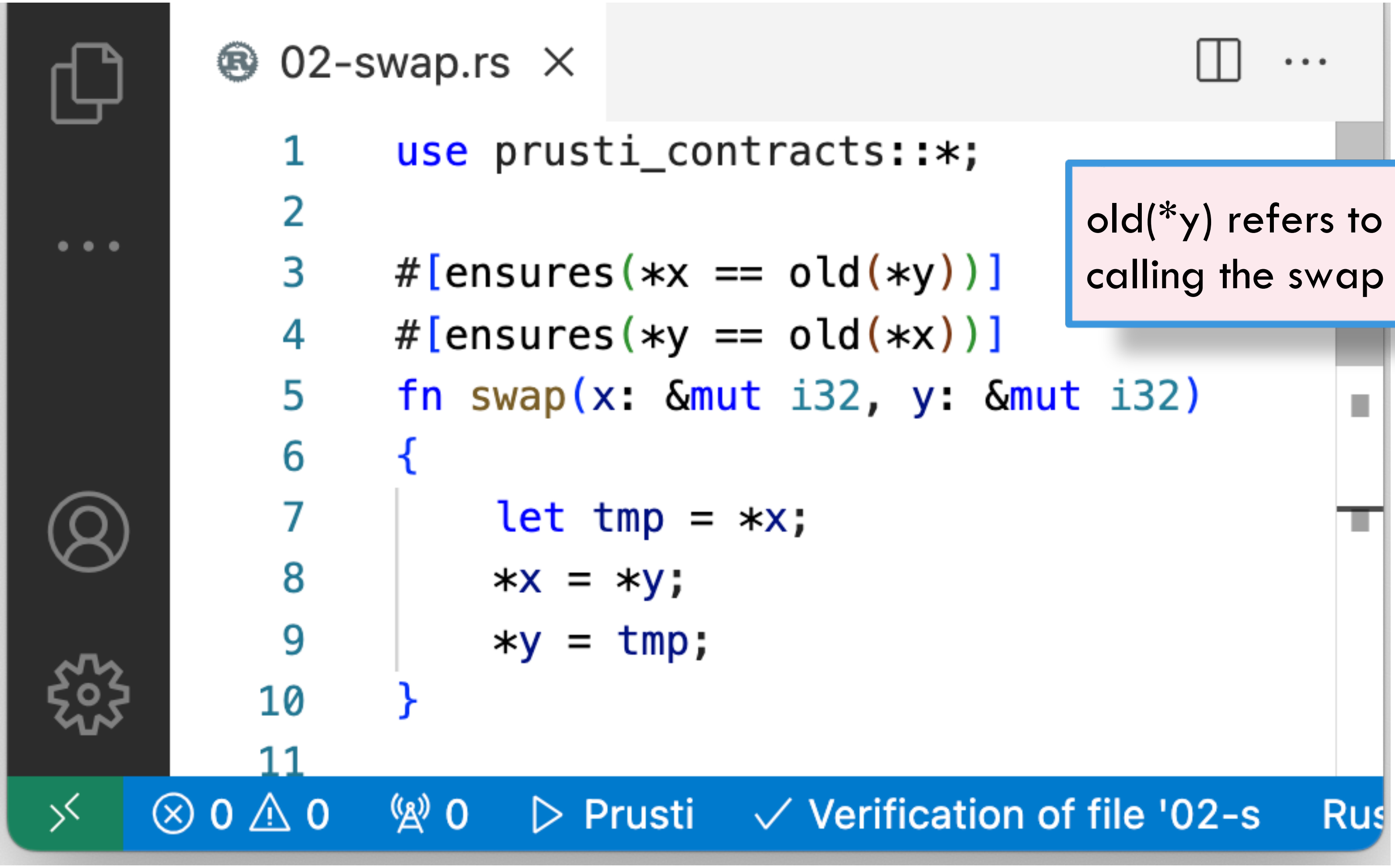
```
#[requires(x != i32::MIN)]
#[ensures(result >= 0)]
#[ensures(result*result == x * x)]
fn abs(x:i32) -> i32 {
    x * sign(x)
}
```

✔

```
#[requires(x != i32::MIN)]
#[ensures(result >= 0)]
#[ensures(result*result == x * x)]
fn abs(x:i32) -> i32 {
    if x >= 0 { x } else { -x }
}
```

✔

EXAMPLE: SWAP BY REFERENCE



old(*y) refers to the value y points to *before* calling the swap function

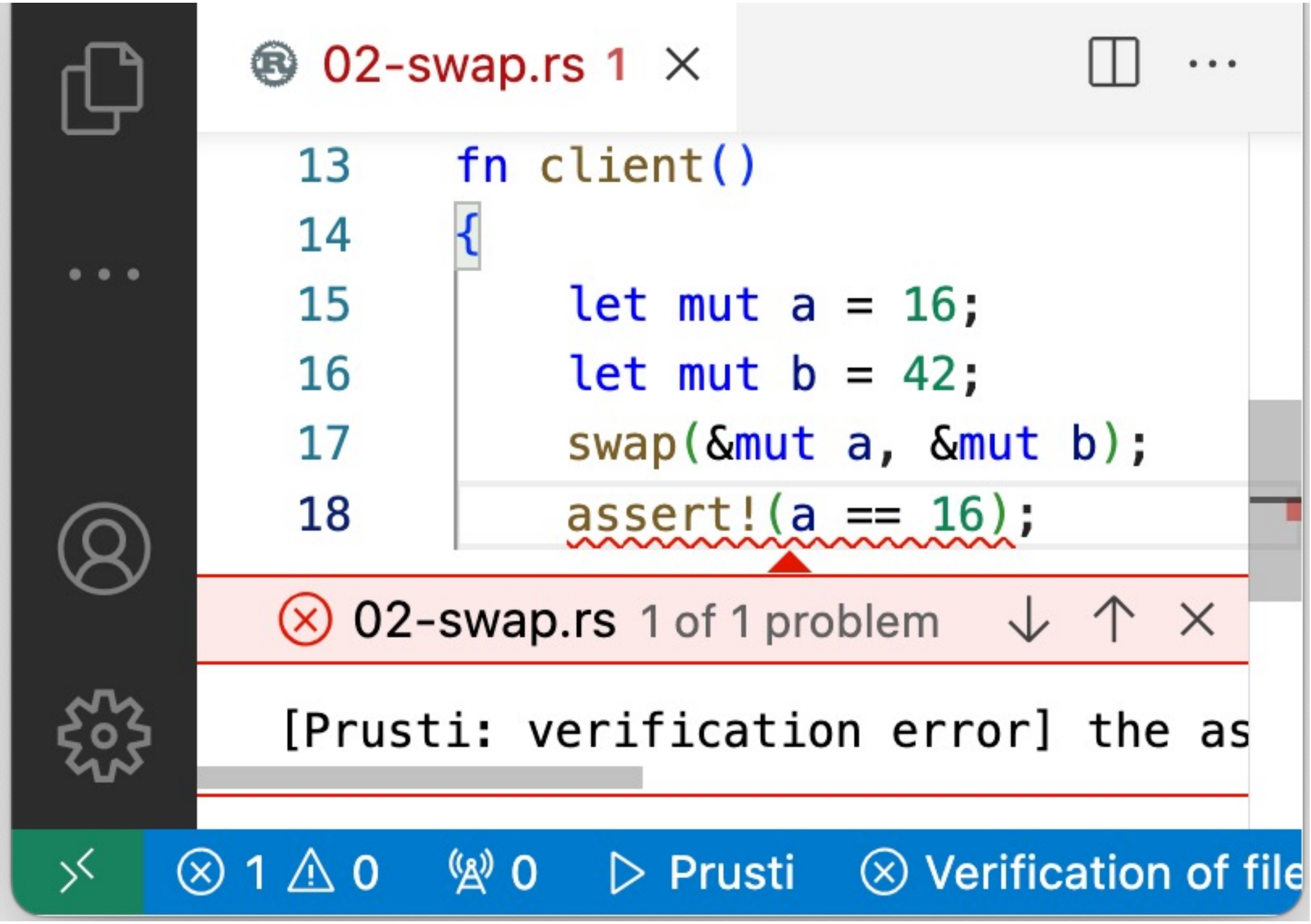
EXAMPLE: CLIENT OF SWAP

02-swap.rs

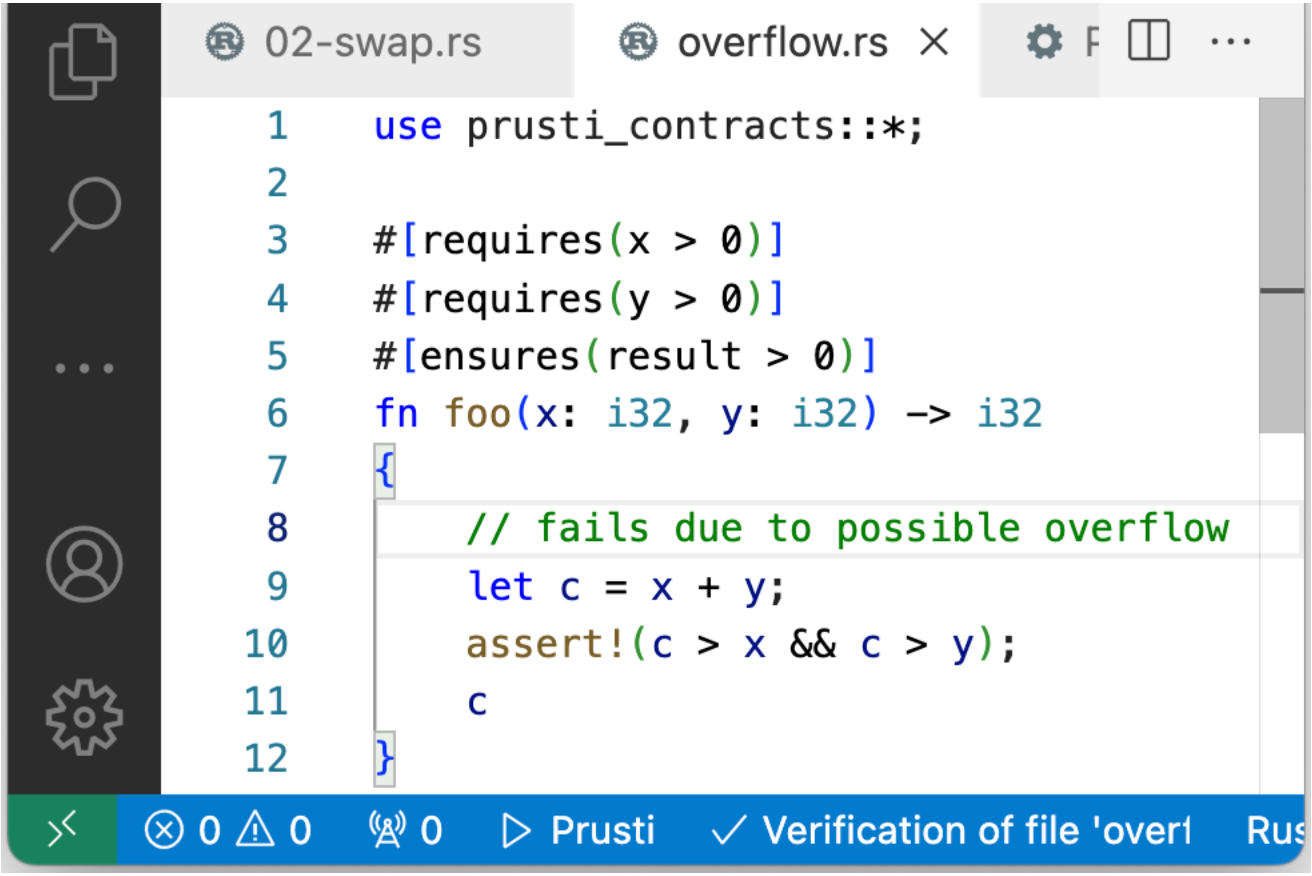
```rust
fn client()
{
    let mut a = 16;
    let mut b = 42;
    swap(&mut a, &mut b);
    assert!(a == 42 && b == 16)
}
```

0 0 0 Prusti ✓ Verification of file '02-s Rus

EXAMPLE: FAULTY CLIENT OF SWAP



IGNORING OVERFLOWS



We sometimes do not care about overflows for a given contract

To disable overflow checks, add a file Prusti.toml with

check_overflows=false

From now on, we disable overflow checks to focus on other features

PURE FUNCTIONS

- Pre- and postcondition can contain arbitrary Rust code as long as it is pure
  - i.e. specifications must have no side effects
- Functions marked with the annotation #[pure]
  - can be called in pre-and postconditions
  - are checked to have no side effects
  - are *not* modular, i.e. their implementation is inspected during contract verification

```
#[pure]
#[requires(x != i32::MIN)]
fn abs(x:i32) -> i32 {
    if x >= 0 { x } else { -x }
}
```

```
#[requires(y != i32::MIN)]
#[requires(abs(y) > 5)]
fn client(y:i32) -> i32 {
    y*y + 5
}
```

EXAMPLE

pure function length allows referring to list length in specifications

pure2.rs

```rust
struct Node {
    elem: i32,
    next: List,
}

enum List {
    Empty,
    More(Box<Node>),
}

impl List {
    #[pure]
    #[ensures(result >= 0)]
    fn len(&self) -> usize {
        match self {
            List::Empty => 0,
            List::More(box node) => 1 + node.next.len(),
        }
    }
// ...
```

⊗ 0 ⚠ 0 0 ▷ Prusti ✓ Verification of file 'pure.rs' succe LF Rust

EXAMPLE

Postcondition: zipping two lists into one yields a list whose length is equal to the sum of the two input lists

pure2.rs

```rust
#[ensures(result.len() == self.len() + that.len())]
pub fn zip(&self, that: &List) -> List {
    match self {
        List::Empty => that.cloneList(),
        List::More(box node) => {
            let new_node = Box::new(Node {
                elem: node.elem,
                next: that.zip(&node.next),
            });
            List::More(new_node)
        }
    }
}
```

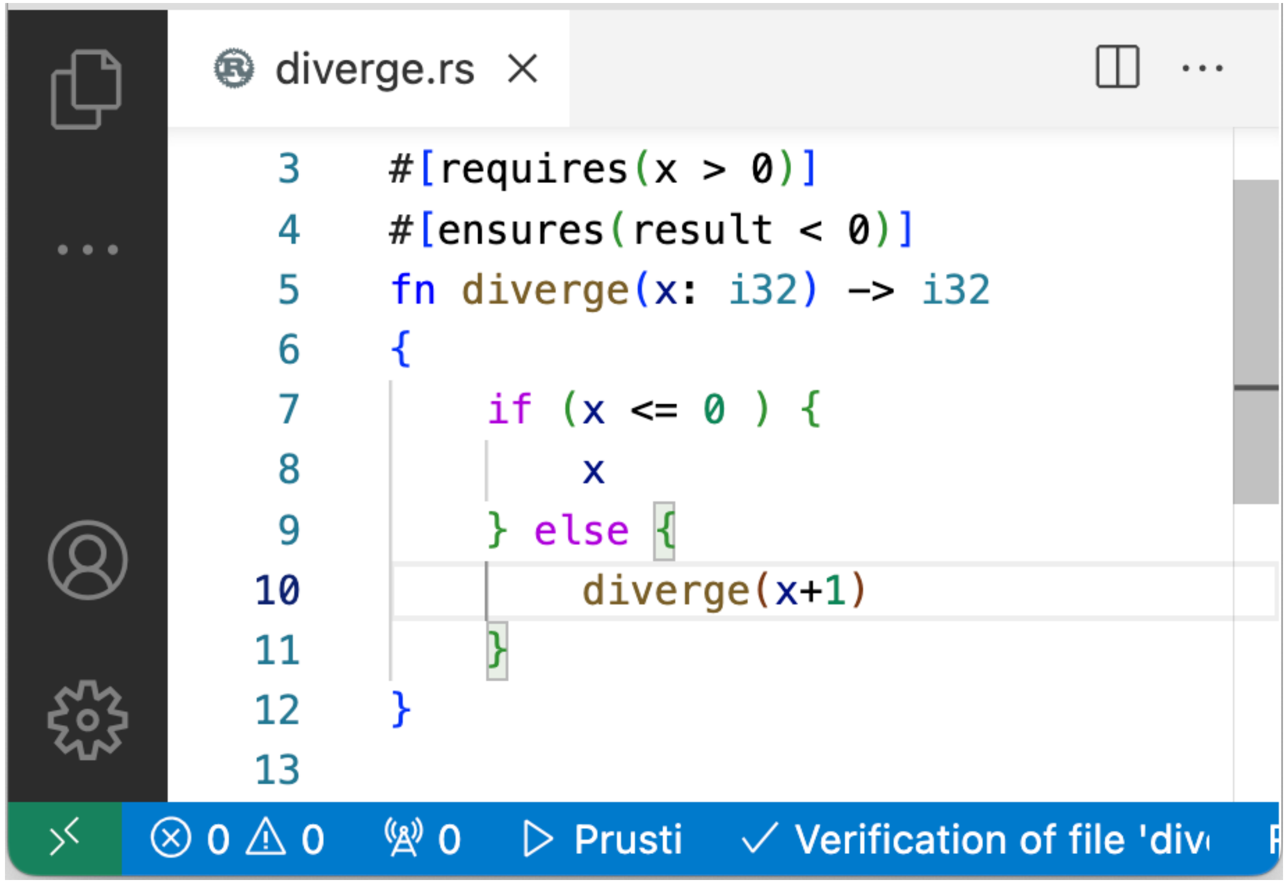
⊗ 0 ⚠ 0 (Ꭿ) 0 ▷ Prusti ✓ Verification of file 'pure.rs' succe LF Rust

TERMINATION

Prusti verifies contracts for partial correctness

➔ Non-terminating executions (via loops or recursion) are allowed

➔ Termination can be shown separately (e.g. with ranking functions)



TRUSTED FUNCTIONS

- Some code cannot be checked at compile time
- Examples: unsupported features, foreign code, unsafe Rust, libraries
- Pragmatic workaround: mark such functions as #[trusted]
  - Prusti uses the contracts of #[trusted] functions
  - Prusti does not check the implementation of #[trusted] functions

➔All results are only valid if trusted functions really adhere their contract

➔Put unverifiable code into trusted wrappers and check them by other means

EXAMPLE

A wrapper for an unsafe function from the standard library

```
#[trusted]
#[requires(src.is_empty())]
#[ensures(dest.is_empty())]
#[ensures(old(dest.len()) == result.len())]
fn replace(dest: &mut Link, src: Link) -> Link {

      // library function that cannot be verified
      // because it needs unsafe Rust code
      mem::replace(dest, src)

}
```

LOOP INVARIANTS

- To verify loops, Prusti needs invariants

➔ Property that holds whenever reaches the annotation body_invariant!(…)

05-sum.rs

```rust
use prusti_contracts::*;

#[requires(n >= 0)]
#[ensures(result == n * (n + 1) / 2)]
fn sum(n: i32) -> i32
{
    let mut res = 0;
    let mut i = 0;
    while i <= n {
        body_invariant!(res == (i - 1) * i / 2);
        res = res + i;
        i = i + 1;
    }
    res
}
```

⊗ 0 ⚠ 0 (ꙩ) 0 ▷ Prusti ✓ Verification of file 'diverge.rs' Rust

05-sum.rs

```rust

fn client() {
    let r = sum(5);
    assert!(r == 1 + 2 + 3 + 4 + 5);
}
```

SUMMARY: PRUSTI SPECIFICATIONS

- Specifications: pure fragment of Rust's Boolean expressions
- #[requires(B)]: B must hold right before a function call
- #[ensures(B)]: B must hold after a function call
  - old(x) refers to the value of x at the beginning of a function
  - result refers to a function's returned value
- #[pure] marks a function as usable in specifications
  - Needs to be free of side effects
- #[trusted] lets Prusti ignore checking a function's implementation
- In implementations: body_invariant!(B), assert!(B), unreachable!()

EXERCISE

Consider the following Rust implementation of Bank accounts. Add annotations such that Prusti can prove that no money is illegally redirected from an account.

```
use prusti_contracts::*;

struct Account {
    bal: u32,
}

impl Account {

  // # TODO
  fn balance(&self) -> u32 {
    self.bal
  }
```

```
// # TODO
fn deposit(&mut self, amount: u32) {
  self.bal = self.bal + amount;
}

// # TODO
fn withdraw(&mut self, amount: u32) {
  self.bal = self.bal - amount;
}

// # TODO
fn transfer(&mut self,
    other: &mut Account, amount: u32) {
      self.withdraw(amount);
      other.deposit(amount);
  }
}

fn main() {}
```

SOLUTION

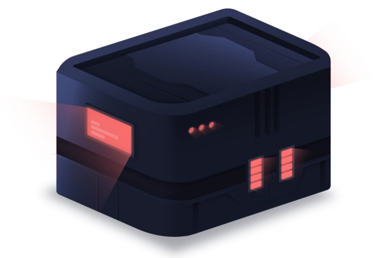
```
#[pure]
fn balance(&self) -> u32 {
  self.bal
}

#[ensures(self.balance() == old(self.balance()) + amount)]
fn deposit(&mut self, amount: u32) {
  self.bal = self.bal + amount;
}

#[requires(amount <= self.balance())]
#[ensures(self.balance() == old(self.balance()) - amount)]
fn withdraw(&mut self, amount: u32) {
  self.bal = self.bal - amount;
}

#[requires(amount <= self.balance())]
#[ensures(self.balance() == old(self.balance()) - amount)]
#[ensures(other.balance() == old(other.balance()) + amount)]
fn transfer(&mut self, other: &mut Account, amount: u32) {
      self.withdraw(amount);
      other.deposit(amount);
}
```

SUMMARY

WHAT ARE THE MAIN TAKEAWAYS FOR THIS CONTENT?

- There is no security without safety.
- Rust's ownership and borrowing system statically guarantee safety by ensuring that references are *either* mutable *or* shared; for exceptions, a synchronization mechanism must enforce safety.
- Flows provide a useful mental model for understanding how the Rust compiler checks memory safety and, in particular, lifetimes.
- Program verification tools, such as Prusti, can provide stronger functional correctness guarantees but require additional annotations.

trade-off: writing more annotations ➔ more compile-time guarantees



FURTHER READING

- The Rust programming language
- Gjengset, J. Rust for Rustaceans: Idiomatic Programming for Experienced Developers. No Starch Press, 2021.
- www.prusti.org

The Prusti Project: Formal Verification for Rust

Vytautas Astrauskas[1], Aurel Bílý[1], Jonáš Fiala[1], Zachary Grannan[2], Christoph Matheja[3], Peter Müller[1], Federico Poli[1], and Alexander J. Summers[2]

[1] Department of Computer Science, ETH Zurich, Switzerland
[2] University of British Columbia, Canada
[3] Technical University of Denmark

WHO IS BEHIND GameSS

Partners behind the project

SDU

DTU

IT UNIVERSITY OF CPH

AALBORG UNIVERSITET

Happy 42

Collaborators

ERHVERVSAKADEMI AARHUS

KEA KØBENHAVNS ERHVERVSAKADEMI

Dania ERHVERVSAKADEMI

UCN

UCL Erhvervsakademi og Professionshøjskole

Supported by

Uddannelses- og Forskningsministeriet

Ministry of Higher Education and Science Denmark

CHALLENGES

CHALLENGES

Challenges will be similar to the examples and exercises on the slides:

1. Use the flow model to identify memory safety issues in Rust code.
    - To capture the flag, one has to provide a unique solution consisting of a flow annotation for every line of source code and a judgment of whether there is a conflict.
    - We will have three challenges of this form covering ownership, borrows, and lifetimes
2. Provide Prusti annotations at the marked places of a program to verify a functional correctness property, similar to the Bank account.
    - We will have three challenges, including recursive and loopy code
3. Use Prusti to implement a proven-correct program
    - We fix the function signatures and Prusti annotations and ask participants to write Rust implementations that satisfy the given contracts.

COMMENTS ON CHALLENGES

A fourth challenge would ask participants to write a proven-correct Rust code by themselves. While this would be the most challenging and arguably most intriguing task, we cannot guarantee that we can automatically provide a flag for all correct solutions.

We thus opted to fix either the code or the annotations to simplify checking whether a solution is correct.