RUST

Safe low-level programming

Thomas Van Strydonck

November 10, 2022

Comparative Programming Languages

Based on material by **Thomas Winant** and **Jake Goulding**.

Exercises with **Thomas Van Strydonck**.

What is Rust?

Rust is a systems programming language that runs blazingly fast, yet still guarantees memory and thread safety.

Outline

Introduction

Introduction

Control over memory & execution

+ Faster
- More things go wrong

Introduction

Control over memory & execution

+ Faster
- More things go wrong

Automatic memory management

+ Safer
+ Easier
+ More expressive
- Memory & execution overhead

Introduction

Control over memory & execution

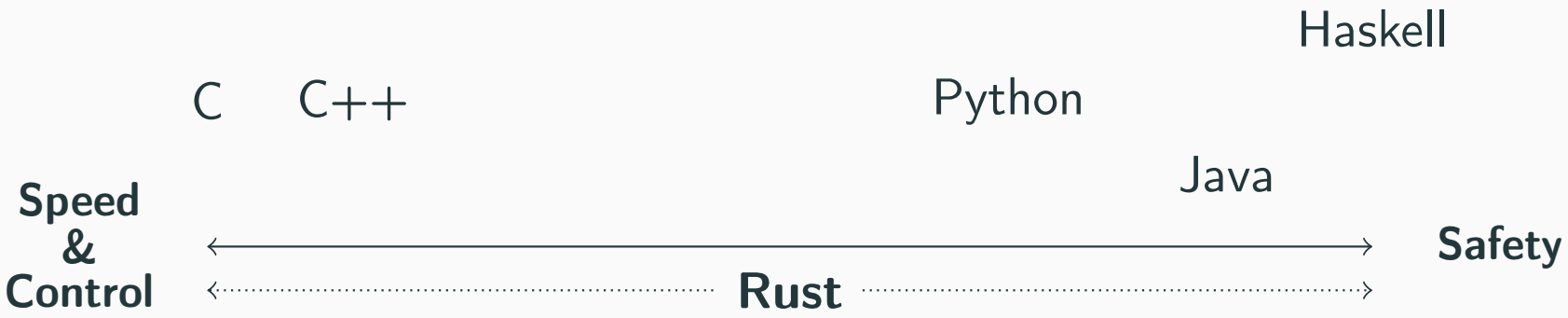
+ Faster
- More things go wrong

Automatic memory management

+ Safer
+ Easier
+ More expressive
- Memory & execution overhead

Unacceptable for: OS, Games, Real-time software

Introduction



Control over memory & execution

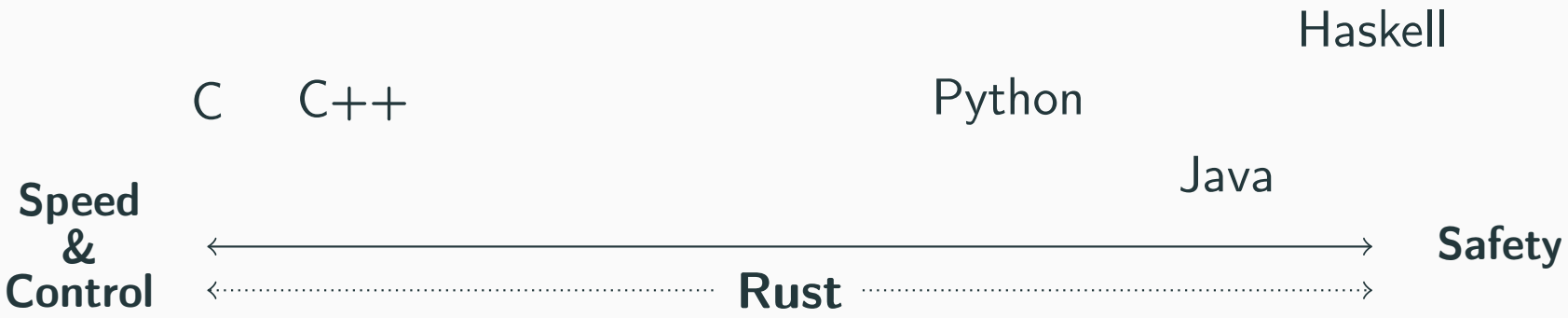
+ Faster
- More things go wrong

Automatic memory management

+ Safer
+ Easier
+ More expressive
- Memory & execution overhead

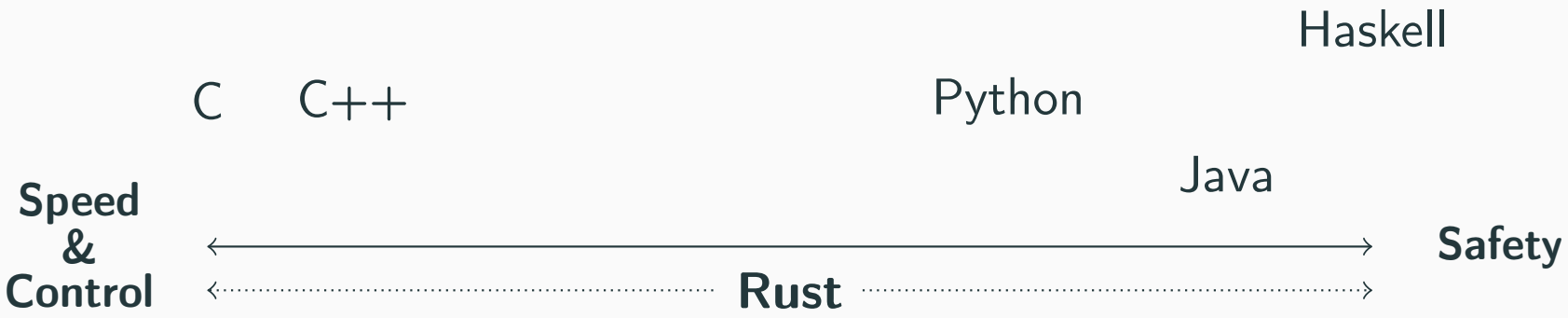
Unacceptable for: OS, Games, Real-time software

Introduction



- Advanced type system for safety and expressivity
- Like Haskell: when code type-checks $\rightarrow$ program does not go wrong

Introduction



- *No segfaults*
- *No double free*
- *No uninitialised memory*
- *No dangling pointers*
- *No null pointers*
- *No race conditions*

Without run-time costs, e.g. no garbage collector!

History

- Mozilla Research project (Graydon Hoare)

History

- Mozilla Research project (Graydon Hoare)

"robust, distributed, and parallel"

History

- Mozilla Research project (Graydon Hoare)
- Rust 1.0 released in May '15

History

- Mozilla Research project (Graydon Hoare)
- Rust 1.0 released in May '15
- Rust Foundation (2021)

History

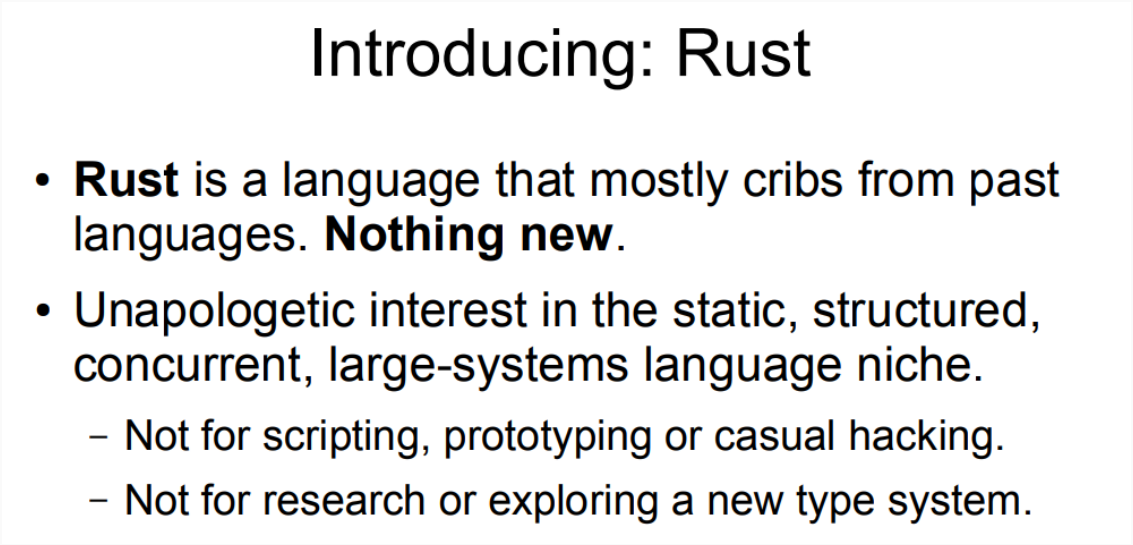
- Mozilla Research project (Graydon Hoare)
- Rust 1.0 released in May '15
- Rust Foundation (2021)
- Open-source: `https://github.com/rust-lang/rust`

History

- Mozilla Research project (Graydon Hoare)
- Rust 1.0 released in May '15
- Rust Foundation (2021)
- Open-source: `https://github.com/rust-lang/rust`
- Latest stable: 1.65

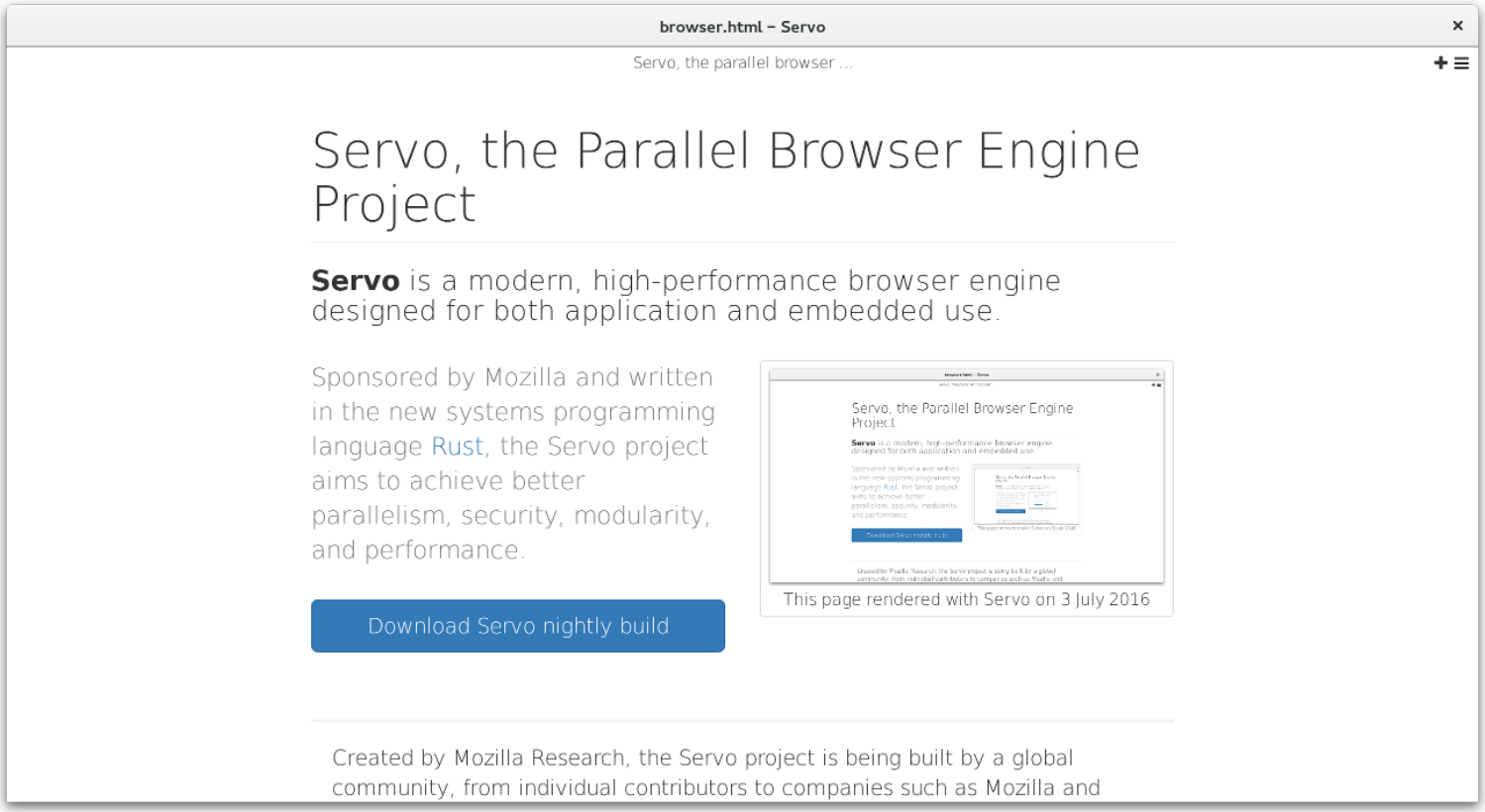
History

- Mozilla Research project (Graydon Hoare)
- Rust 1.0 released in May '15
- Rust Foundation (2021)
- Open-source: `https://github.com/rust-lang/rust`
- Latest stable: 1.65
- Interest from academia



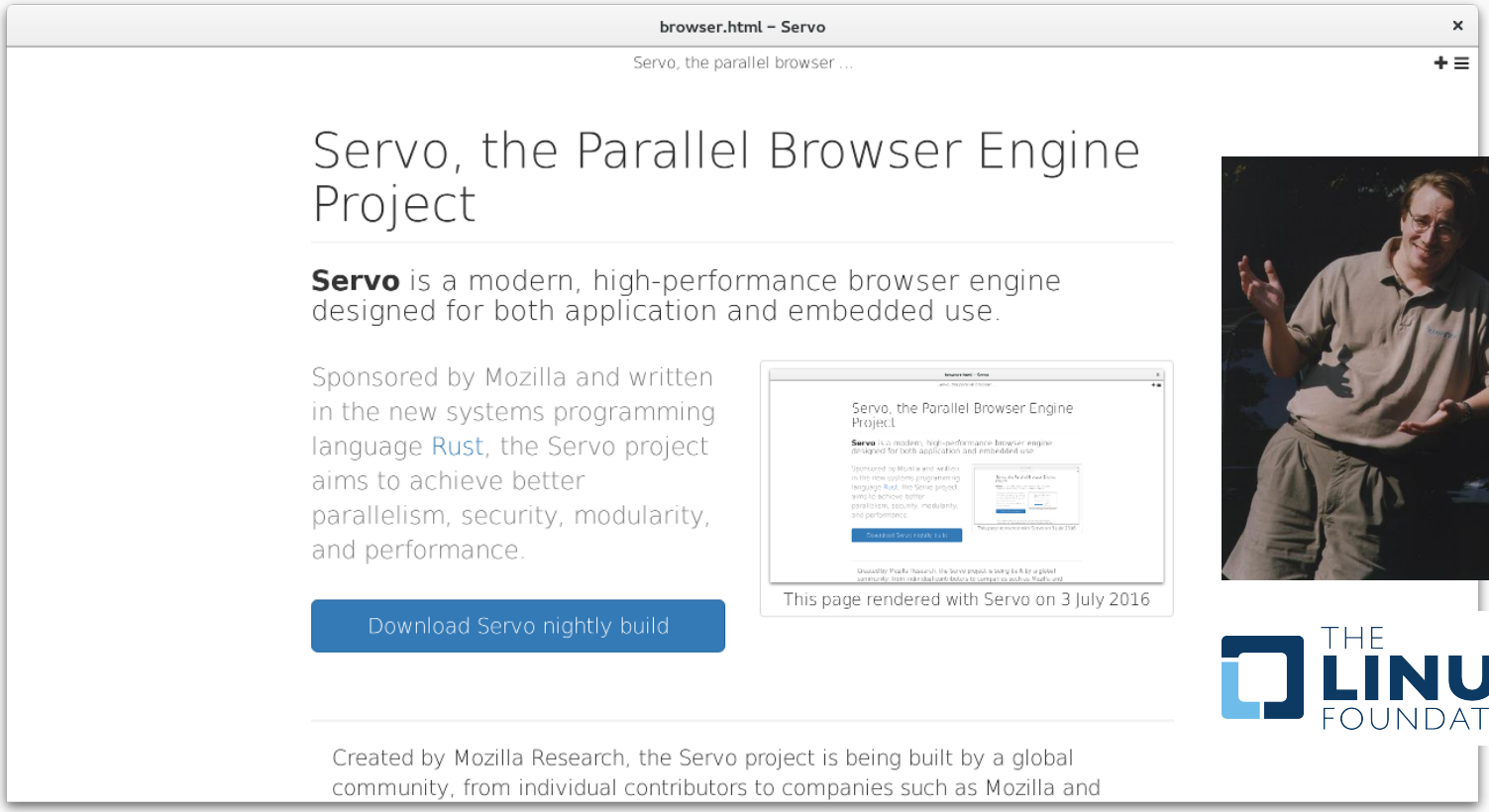
What are people doing with Rust?

Initially: *Servo*



What are people doing with Rust?

Initially: *Servo*



THE LINUX FOUNDATION

What are people doing with Rust?

Project Quantum (2017):

- Integrating parts of Servo into Firefox's Gecko
- Stylo (parallel CSS style system), WebRenderer (optimised for GPUs), ...

What are people doing with Rust?

Writing operating systems:

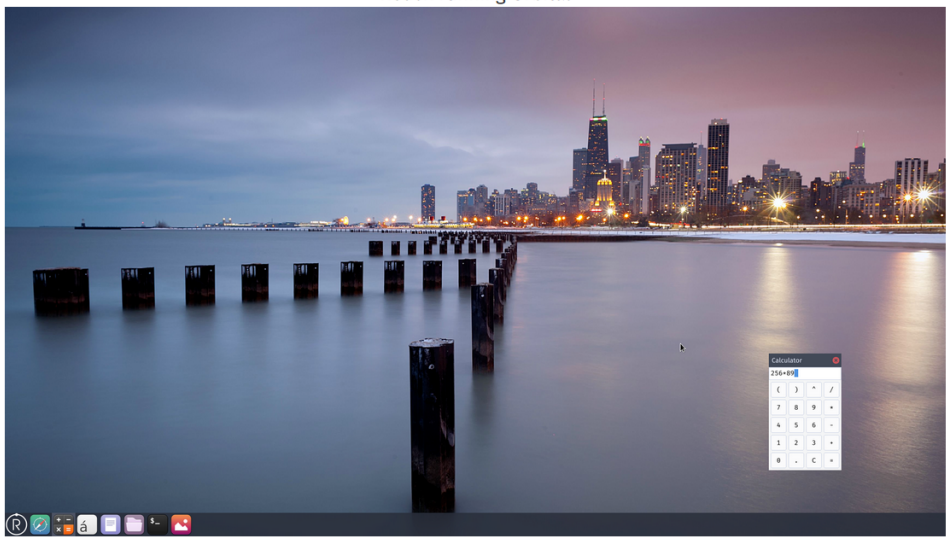
Redox is a Unix-like Operating System written in **Rust**, aiming to bring the innovations of Rust to a modern microkernel and full set of applications.

View Releases

Pull from GitHub

- Implemented in Rust
- Microkernel Design
- Includes optional GUI - Orbital
- Supports Rust Standard Library
- MIT Licensed
- Drivers run in Userspace
- Includes common Unix commands
- Newlib port for C programs

Redox running Orbital



What are people doing with Rust?

Writing operating systems:

Linus Torvalds says Rust is coming to the Linux kernel 'real soon now'

Maintainer lack of familiarity won't be an issue, chief insists, citing his own bafflement when faced with Perl

Thomas Claburn — Thu 23 Jun 2022 // 19:30 UTC

At The Linux Foundation's Open Source Summit in Austin, Texas on Tuesday, Linus Torvalds said he expects support for Rust code in the Linux kernel to be merged soon, possibly with the next release, 5.20.

At least since last December, when a patch added support for Rust as a second language for kernel code, the Linux community has been anticipating this transition, in the hope it leads to greater stability and security.

In a conversation with Dirk Hohndel, chief open source officer at Cardano, Torvalds said the patches to integrate Rust have not yet been merged because there's far more caution among Linux kernel maintainers than there was 30 years ago.

What are people doing with Rust?

Writing operating systems:

Google is developing parts of Android in Rust to improve security

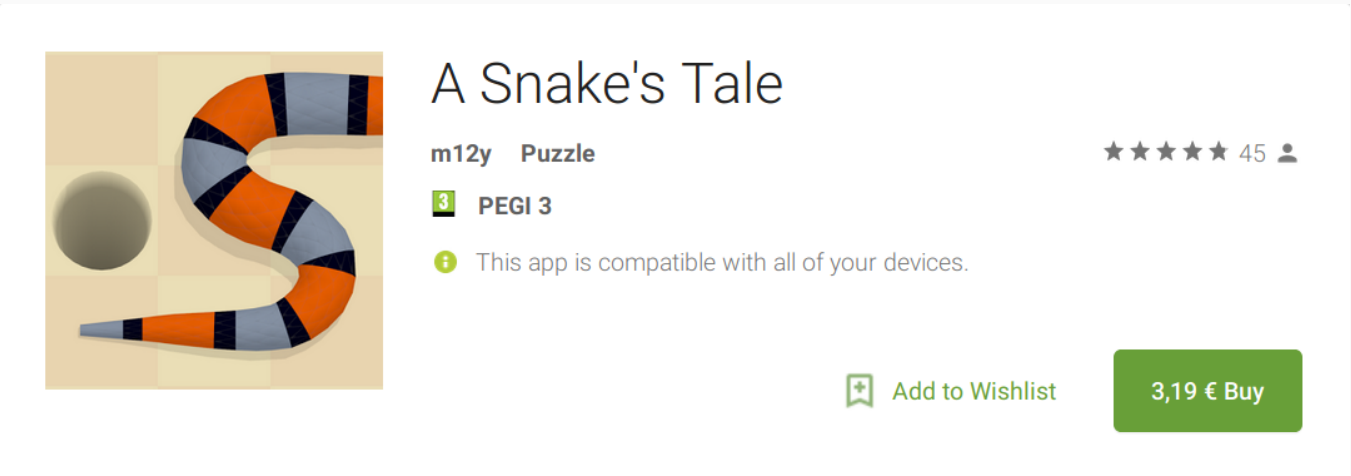
Google is writing and rewriting parts of Android in Rust to improve security of the OS as a whole, vs C and C++. Read on to know more!

BY **AAMIR SIDDIQUI** PUBLISHED APR 06, 2021

What are people doing with Rust?

Developing games:

- `http://arewegameyet.com/`
- `https://www.reddit.com/r/rust_gamedev/`
- `http://www.piston.rs/`
- `http://cityboundsim.com/`
- `https://github.com/cristicbz/rust-doom/`



What are people doing with Rust?

Web development

- `https://github.com/iron/iron`
  concurrent web framework
- `https://github.com/nickel-org/nickel.rs`
  express.js inspired web framework
- `https://github.com/Ogeon/rustful`
  REST-like HTTP framework

Much more

- `https://github.com/uutils/coreutils`
  Rust rewrite of `ls`, `cp`, `rm`, `echo`, `...`
- `https://github.com/BurntSushi/ripgrep`
  Recursive grep that is faster than the original grep
- `https://github.com/rust-unofficial/awesome-rust`
  Extensive list of Rust software

What are people doing with Rust?

Fan favorite: `https://github.com/bnjbvr/rouille`

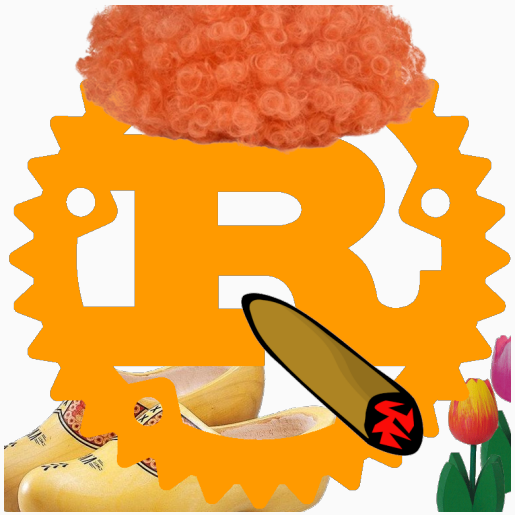
What are people doing with Rust?

Fan favorite: `https://github.com/bnjbvr/rouille`

```
rouille::rouille! {
    utilisons std::collections::Dictionnaire comme Dico;

    convention CléValeur {
        fonction écrire(&soi, clé: Chaine, valeur: Chaine);
        fonction lire(&soi, clé: Chaine) -> PeutÊtre<&Chaine>;
    }

    statique mutable DICTIONNAIRE: PeutÊtre<Dico<Chaine, Chaine>> =

    structure Concrète;

    réalisation CléValeur pour Concrète {
```

What are people doing with Rust?

Fan favorite: `https://github.com/bnjbvr/rouille`



Outline

Basics

C/C++/Java-style syntax with braces and semicolons

Rust

```rust
fn main() {
    // A comment
    println!("Hello World");
}
```

```
>> Hello World
```

Expression-oriented

Rust

```rust
fn abs(x: i32) -> i32 {
    let result: i32 = if x < 0 {
        -x
    } else {
        x
    };
    result
}
```

Basics

Immutable by default

Rust

```rust
fn sum_squared(x: u32, y: u32) -> u32 {
    let sum: u32 = x;
    sum += y;
    sum * sum
}
```

L3 error: cannot assign twice to immutable variable `sum`

Explicit mutability

Rust

```rust
fn sum_numbers(start: u32, end: u32) -> u32 {
    let mut sum: u32 = 0;
    for i in start..end {
        sum += i;
    }
    sum
}
```

Local type inference

Rust

```rust
fn sum_numbers(start: u32, end: u32) -> u32 {
    let mut sum = 0;
    for i in start..end {
        sum += i;
    }
    sum
}
```

Outline

Memory mismanagement

More than 50% of critical security vulnerabilities in large projects (Firefox browser, Linux kernel, Microsoft products, etc.) are due to memory mismanagement!

Memory mismanagement

More than 50% of critical security vulnerabilities in large projects (Firefox browser, Linux kernel, Microsoft products, etc.) are due to memory mismanagement!

- Buffer under-/overflow
- Memory leaks
- Use after free
- Double free
- Dangling pointers

Memory mismanagement

More than 50% of critical security vulnerabilities in large projects (Firefox browser, Linux kernel, Microsoft products, etc.) are due to memory mismanagement!

- Buffer under-/overflow
- Memory leaks
- Use after free
- Double free
- Dangling pointers

Rust's approach: **Ownership + Borrowing**

Memory mismanagement

More than 50% of critical security vulnerabilities in large projects (Firefox browser, Linux kernel, Microsoft products, etc.) are due to memory mismanagement!

- Buffer under-/overflow
- Memory leaks
- Use after free
- Double free
- Dangling pointers

Rust's approach: **Ownership + Borrowing**

- Memory safety: $\neg$ (aliasing $\wedge$ mutation)
- Static: at compile-time $\rightarrow$ no run-time cost

Memory leaks

C

```c
void func()
{
    int *x = malloc(sizeof(int));
    *x = 1;
}
```

Memory leaks

C

```c
void func()
{
    int *x = malloc(sizeof(int));
    *x = 1;
}
```

Memory leak: `x` is never freed

Memory leaks

C

```c
void func()
{
    int *x = malloc(sizeof(int));
    *x = 1;
}
```

Memory leak: `x` is never freed

Rust

```rust
fn func()
{
    let x = Box::new(1);  // +
                          // | Scope of x
                          // |
}                         // -
```

Use after free

C

```c
int *x = malloc(sizeof(int));
*x = 1;
free(x);

int *y = malloc(sizeof(int));
*y = 9;

*x = 2;

printf("x = %d, y = %d\n", *x, *y);
```

Use after free

C

```c
int *x = malloc(sizeof(int));
*x = 1;
free(x);

int *y = malloc(sizeof(int));
*y = 9;

*x = 2;

printf("x = %d, y = %d\n", *x, *y);
```

```
>> x = ?, y = ?
```

Use after free

C

```c
int *x = malloc(sizeof(int));
*x = 1;
free(x);

int *y = malloc(sizeof(int));
*y = 9;

*x = 2;

printf("x = %d, y = %d\n", *x, *y);
```

```
>> x = 2, y = 2
```

L8: use after free $\Rightarrow$ undefined behavior

Use after free

Rust

```
1  {
2      let mut x = Box::new(1); // + Scope of x
3                               // |
4  }                            // -
5
6  let y = Box::new(9);         // + Scope of y
7  *x = 2;                      // |
8  println!("x = {}, y = {}", // |
9           *x, *y);            // |
```

Use after free

Rust

```
1  {
2    let mut x = Box::new(1); // + Scope of x
3                             // |
4  }                          // -
5
6  let y = Box::new(9);       // + Scope of y
7  *x = 2;                    // |
8  println!("x = {}, y = {}", // |
9           *x, *y);          // |
```

```
L7 error: cannot find value `x` in this scope
          (did you mean `y`?)
```

Double free

C

```c
void do_stuff_with(int* x)
{
    printf("x is %d, ", *x);
    free(x);
}
int main()
{
    int* x = malloc(sizeof(int));
    *x = 1;
    do_stuff_with(x);
    do_stuff_with(x);
}
```

Double free

C

```c
void do_stuff_with(int* x)
{
    printf("x is %d, ", *x);
    free(x);
}
int main()
{
    int* x = malloc(sizeof(int));
    *x = 1;
    do_stuff_with(x);
    do_stuff_with(x);
}
```

```
>> x is 1, x is 0,
```

Double free

C

```c
void do_stuff_with(int* x)
{
    printf("x is %d, ", *x);
    free(x);
}
int main()
{
    int* x = malloc(sizeof(int));
    *x = 1;
    do_stuff_with(x);
    do_stuff_with(x);
}
```

```
>> x is 1, x is 0,
Error in ...: double free or corruption ...
```

Double free

Rust

```rust
fn do_stuff_with(x: Box<i32>) {
    println!("x is {}", *x);
}
fn main() {
    let x = Box::new(1);
    do_stuff_with(x);
    do_stuff_with(x);
}
```

Double free

Rust

```
1  fn do_stuff_with(x: Box<i32>) {
2      println!("x is {}", *x);
3  }
4  fn main() {
5      let x = Box::new(1);
6      do_stuff_with(x);
7      do_stuff_with(x);
8  }
```

```
L6 note: `x` value moved here
```

Double free

Rust

```
1  fn do_stuff_with(x: Box<i32>) {
2      println!("x is {}", *x);
3  }
4  fn main() {
5      let x = Box::new(1);
6      do_stuff_with(x);
7      do_stuff_with(x);
8  }
```

L6 note: `x` value moved here

L7 error: use of moved value: `x`

Move semantics

Rust

```rust
fn owner() {
    let mut vec = Vec::new();
    vec.push(1);
    print_length(vec);
    vec.push(2);
}
fn print_length(vec: Vec<i32>) {
    println!("Length: {}", vec.len());
}
```

Move semantics

Rust

```
1  fn owner() {
2      let mut vec = Vec::new();
3      vec.push(1);
4      print_length(vec);
5      vec.push(2);
6  }
7  fn print_length(vec: Vec<i32>) {
8      println!("Length: {}", vec.len());
9  }
```

```
L4 note: `vec` value moved here
L5 error: use of moved value: `vec`
```

Move semantics

Rust

```rust
fn owner() {
    let mut vec = Vec::new();
    vec.push(1);
    print_length(vec);
    vec.push(2);
}
fn print_length(vec: Vec<i32>) {
    println!("Length: {}", vec.len());
}
```

Ownership of `vec` moved from `owner` to `print_length`.

Move semantics

Rust

```rust
fn owner() {
    let mut vec = Vec::new();
    vec.push(1);
    print_length(vec);
    vec.push(2);
}
fn print_length(vec: Vec<i32>) {
    println!("Length: {}", vec.len());
}
```

Ownership of `vec` moved from `owner` to `print_length`.

What if we don't want the ownership to change?

How not to solve borrowing ownership

Rust

```rust
fn owner() {
    let mut vec = Vec::new();
    vec.push(1);
    vec = print_length(vec);
    vec.push(2);
}
fn print_length(vec: Vec<i32>) -> Vec<i32> {
    println!("Length: {}", vec.len());
    vec
}
```

How not to solve borrowing ownership

Rust

```rust
fn owner() {
    let mut vec = Vec::new();
    vec.push(1);
    vec = print_length(vec);
    vec.push(2);
}
fn print_length(vec: Vec<i32>) -> Vec<i32> {
    println!("Length: {}", vec.len());
    vec
}
```

```
>> Length: 1
```

How not to solve borrowing ownership

Rust

```rust
fn owner() {
    let mut vec = Vec::new();
    vec.push(1);
    vec = print_length(vec);
    vec.push(2);
}
fn print_length(vec: Vec<i32>) -> Vec<i32> {
    println!("Length: {}", vec.len());
    vec
}
```

```
>> Length: 1
```

Better: make **type system** do work for us!

Borrowing: a type-system answer to ownership borrowing

Immutable borrow: `&x`

$\sim$ read-only borrow

Rust

```rust
fn owner() {
    let mut vec = Vec::new();
    vec.push(1);
    print_length(&vec);
    vec.push(2);
}
fn print_length(vec: &Vec<i32>) {
    println!("Length: {}", vec.len());
}
```

Borrowing: a type-system answer to ownership borrowing

Immutable borrow: `&x`

$\sim$ read-only borrow

Rust

```rust
fn owner() {
    let mut vec = Vec::new();
    vec.push(1);
    print_length(&vec);
    vec.push(2);
}
fn print_length(vec: &Vec<i32>) {
    println!("Length: {}", vec.len());
}
```

```
>> Length: 1
```

Borrowing: a type-system answer to ownership borrowing

Mutable borrow: `&mut x`

$\sim$ read-write borrow

Rust

```rust
fn owner() {
    let mut vec = Vec::new();
    vec.push(1);
    push_twice(&mut vec, 2);
    println!("Length: {}", vec.len());
}
fn push_twice(vec: &mut Vec<i32>, x: i32) {
    vec.push(x);
    vec.push(x);
}
```

Borrowing: a type-system answer to ownership borrowing

Mutable borrow: `&mut x`

$\sim$ read-write borrow

Rust

```rust
fn owner() {
    let mut vec = Vec::new();
    vec.push(1);
    push_twice(&mut vec, 2);
    println!("Length: {}", vec.len());
}
fn push_twice(vec: &mut Vec<i32>, x: i32) {
    vec.push(x);
    vec.push(x);
}
```

```
>> Length: 3
```

Dangling pointers

C++

```cpp
#include <vector>
#include <iostream>
using namespace std;
int main(){
    vector<int> vec;
    for (int i = 0; i < 4; i++) {
        vec.push_back(i);
    }
    int *first = &vec[0];
    vec.push_back(4);
    cout << *first << endl;
}
```

Dangling pointers

C++

```cpp
#include <vector>
#include <iostream>
using namespace std;
int main(){
    vector<int> vec;
    for (int i = 0; i < 4; i++) {
        vec.push_back(i);
    }
    int *first = &vec[0];
    vec.push_back(4);
    cout << *first << endl;
}
```

```
>> ?
```

Dangling pointers

C++

```cpp
#include <vector>
#include <iostream>
using namespace std;
int main(){
    vector<int> vec;
    for (int i = 0; i < 4; i++) {
        vec.push_back(i);
    }
    int *first = &vec[0];
    vec.push_back(4);
    cout << *first << endl;
}
```

```
>> 14908464
```

Dangling pointers

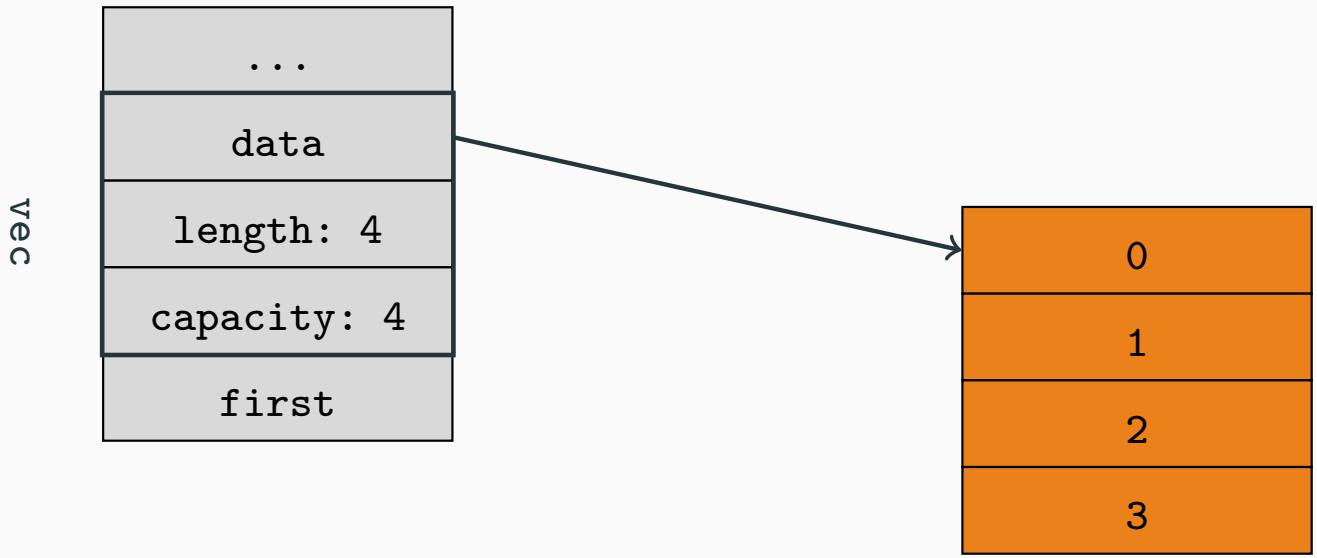
C++

```cpp
#include <vector>
#include <iostream>
using namespace std;
int main(){
    vector<int> vec;
    for (int i = 0; i < 4; i++) {
        vec.push_back(i);
    }
    int *first = &vec[0];
    vec.push_back(4);
    cout << *first << endl;
}
```

```
>> 14908464
```

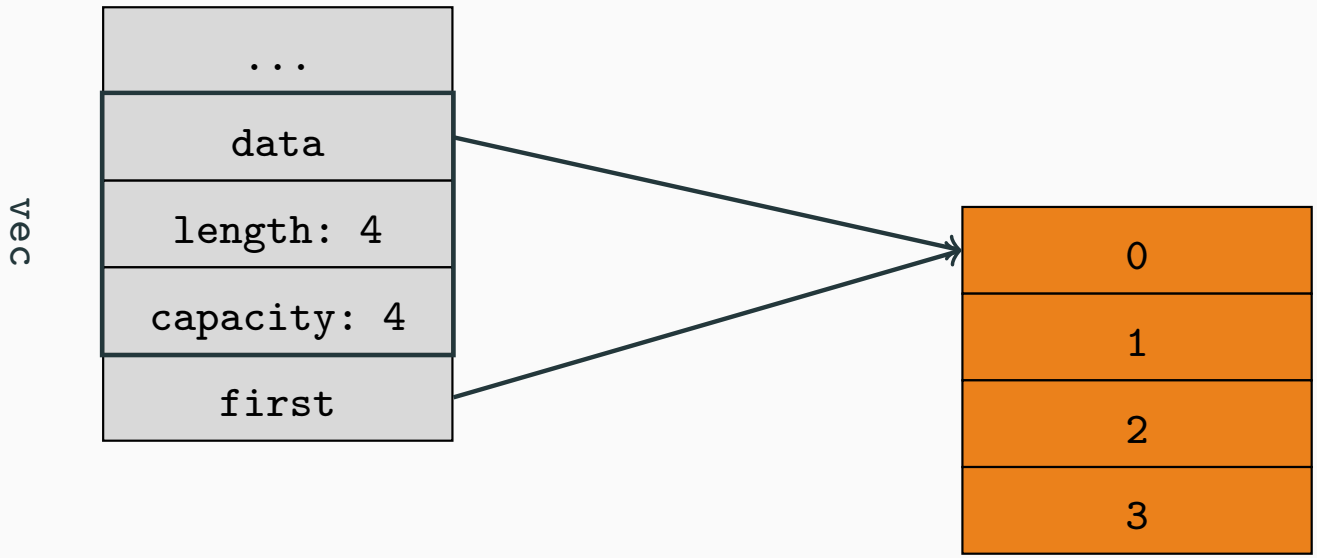
Dangling pointers

```
1  vector<int> vec;
2  for (int i = 0; i < 4; i++) {
3      vec.push_back(i);
4  }
5  int *first = &vec[0];
6  vec.push_back(4);
7  cout << *first << endl;
```



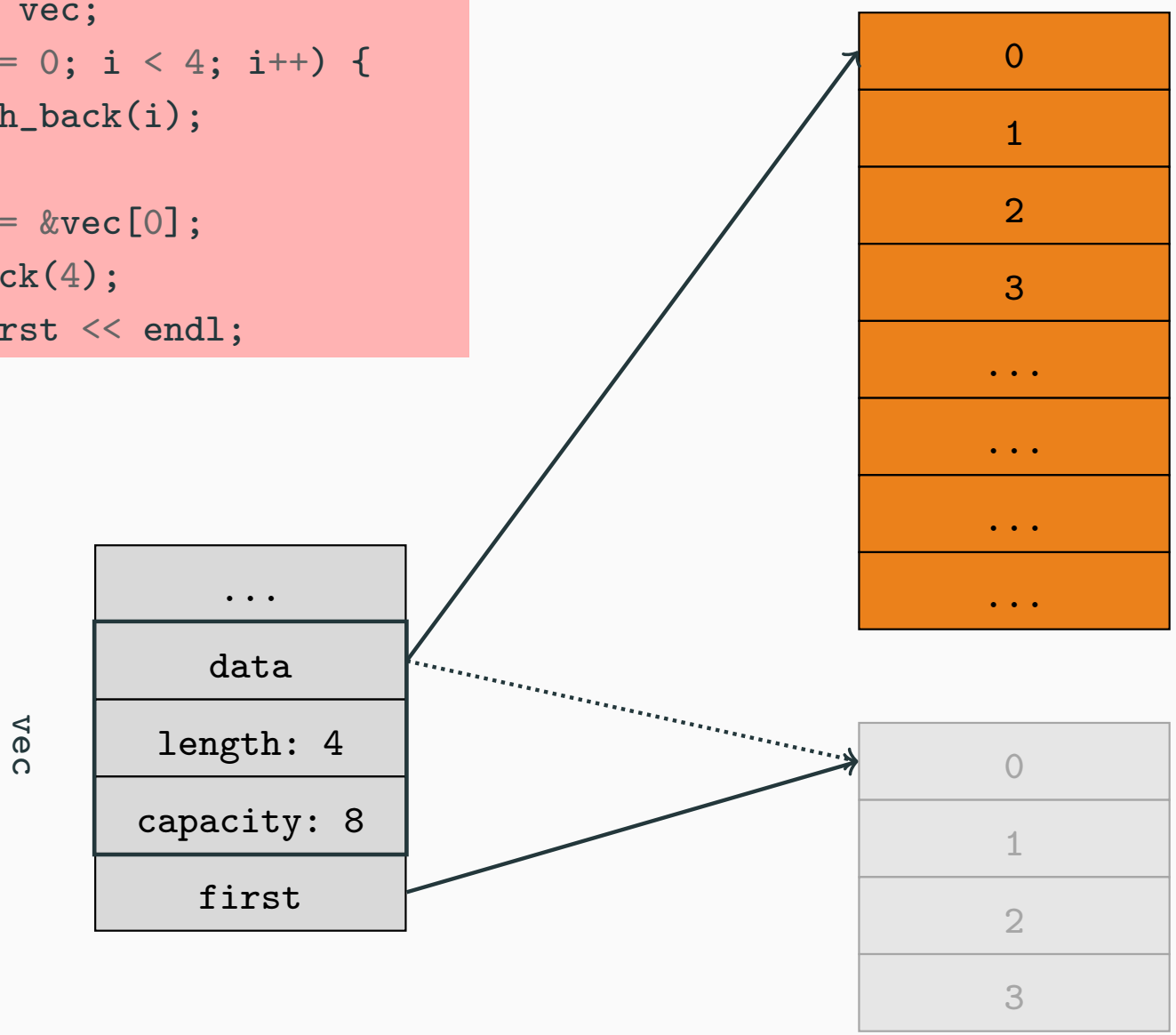
Dangling pointers

```
1  vector<int> vec;
2  for (int i = 0; i < 4; i++) {
3      vec.push_back(i);
4  }
5  int *first = &vec[0];
6  vec.push_back(4);
7  cout << *first << endl;
```



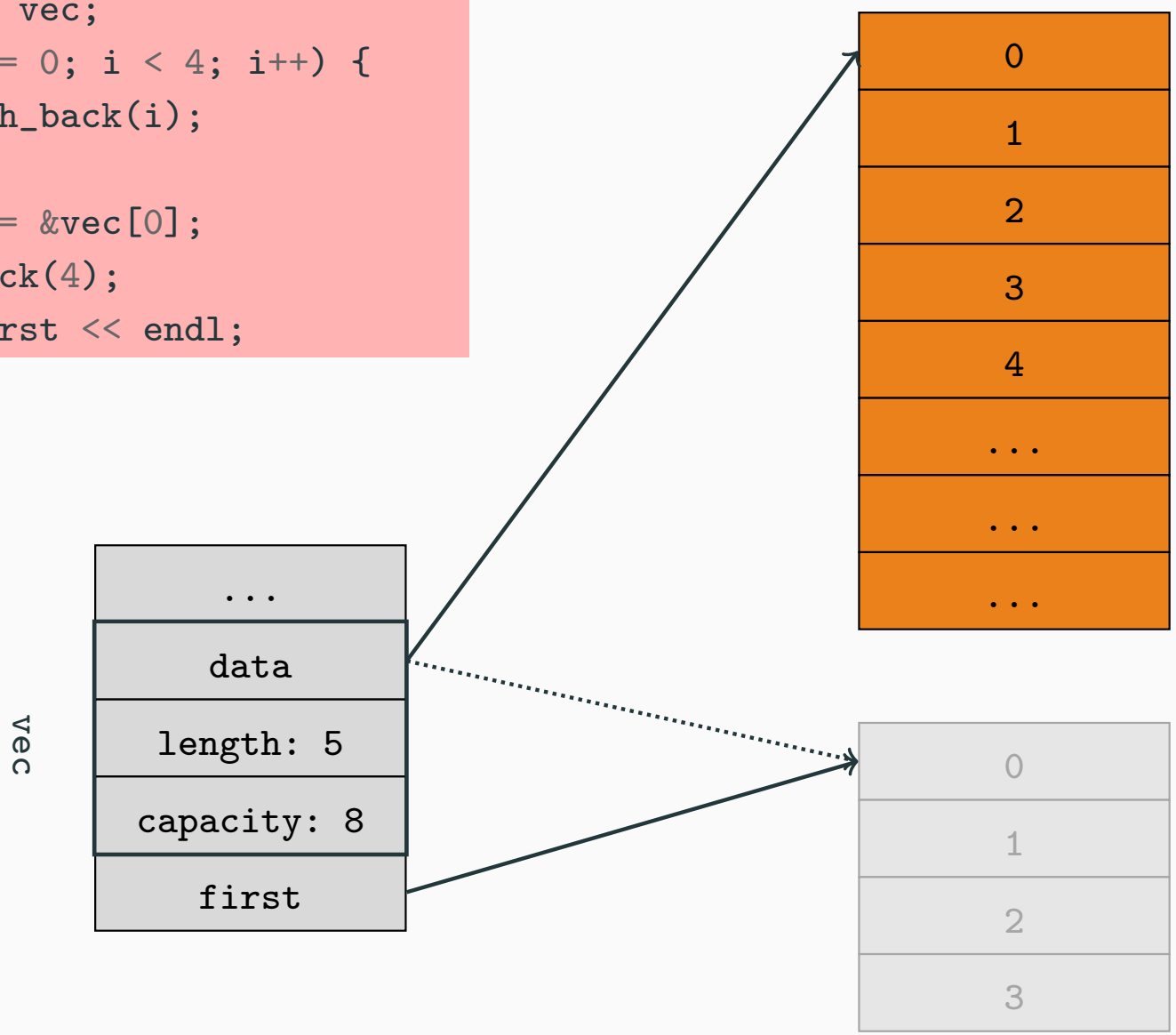
Dangling pointers

```
1  vector<int> vec;
2  for (int i = 0; i < 4; i++) {
3      vec.push_back(i);
4  }
5  int *first = &vec[0];
6  vec.push_back(4);
7  cout << *first << endl;
```



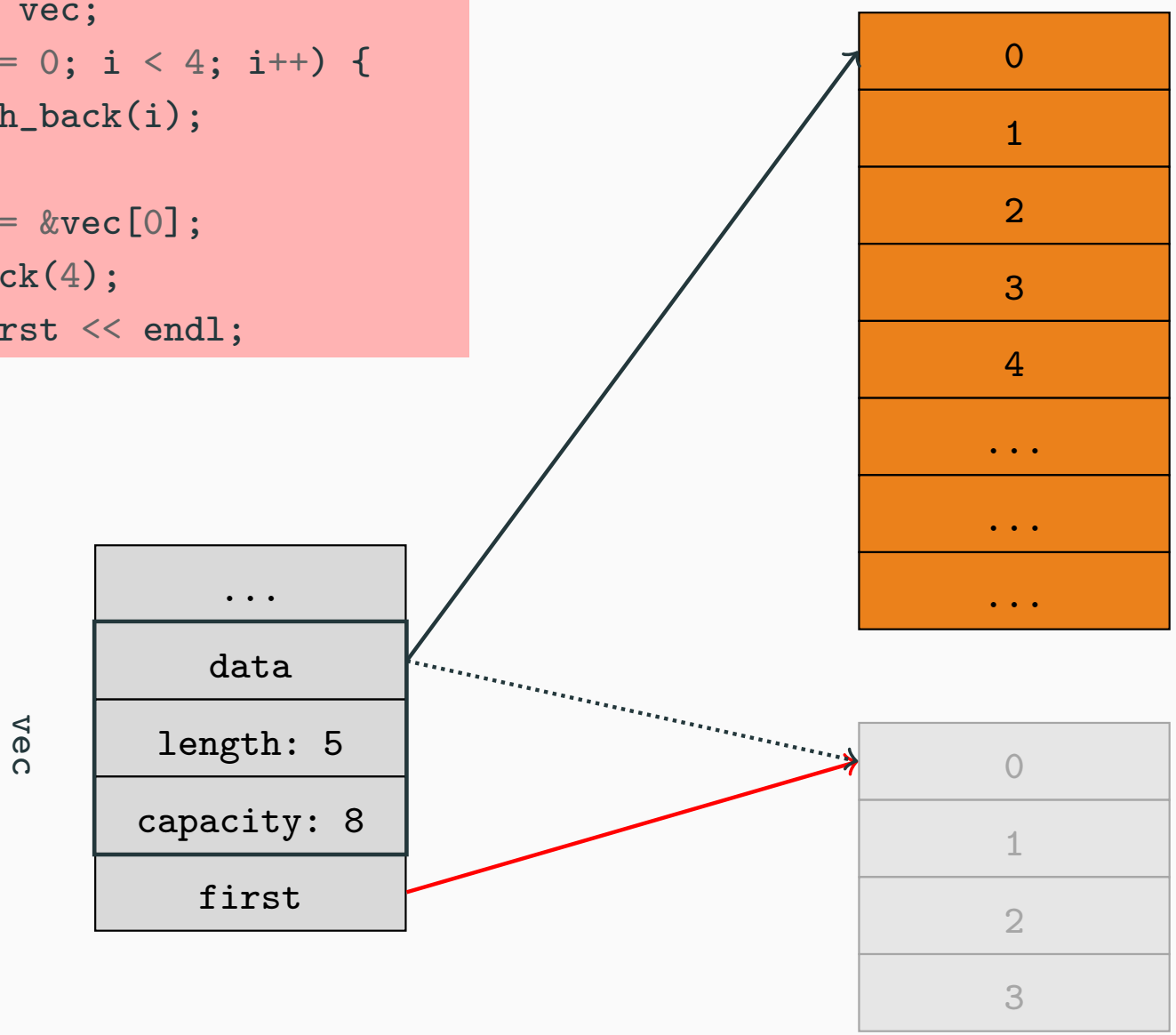
Dangling pointers

```
1  vector<int> vec;
2  for (int i = 0; i < 4; i++) {
3      vec.push_back(i);
4  }
5  int *first = &vec[0];
6  vec.push_back(4);
7  cout << *first << endl;
```



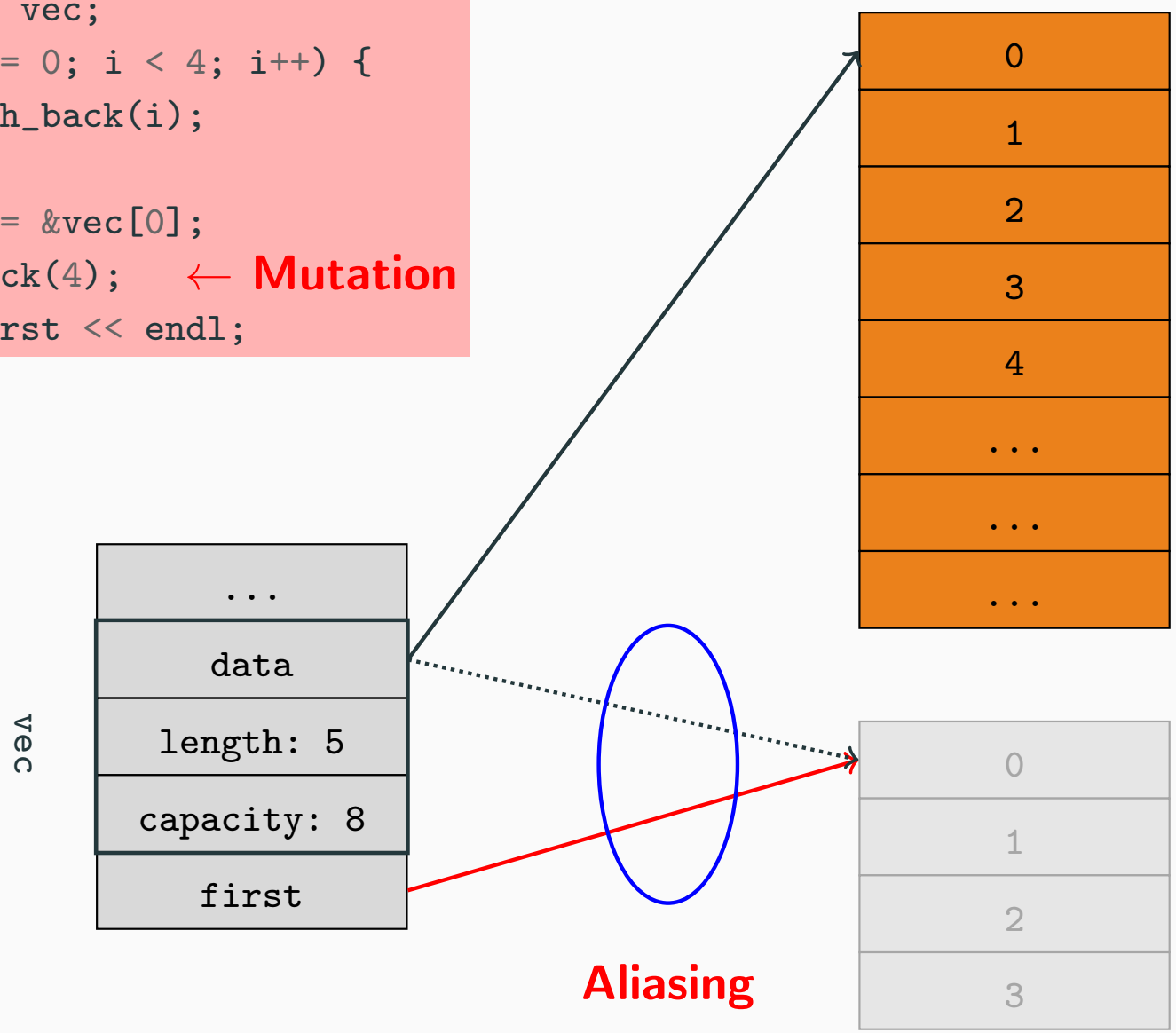
Dangling pointers

```
1  vector<int> vec;
2  for (int i = 0; i < 4; i++) {
3      vec.push_back(i);
4  }
5  int *first = &vec[0];
6  vec.push_back(4);
7  cout << *first << endl;
```



Dangling pointers

```
1  vector<int> vec;
2  for (int i = 0; i < 4; i++) {
3      vec.push_back(i);
4  }
5  int *first = &vec[0];
6  vec.push_back(4);    ← Mutation
7  cout << *first << endl;
```



Dangling pointers

Rust

```rust
let mut vec = Vec::new();
for i in 0..4 {
    vec.push(i);
}
let first = &vec[0];
vec.push(4);
println!("{}", *first);
```

Dangling pointers

Rust

```
1  let mut vec = Vec::new();
2  for i in 0..4 {
3      vec.push(i);
4  }
5  let first = &vec[0];
6  vec.push(4);
7  println!("{}", *first);
```

```
L5 info: immutable borrow occurs here
L6 error: cannot borrow `vec` as mutable
   because it is also borrowed as immutable
```

Ownership & Borrowing – Video

`ownership-borrowing.mp4` (10:03)

Fragment of "The Rust Programming Language", Alex Crichton, Google Tech Talk, June 6 '15.

Buffer over-/underflow

Rust

```rust
fn func() {
    let array = [1, 2 ,3];
    let i = 100;
    println!("{}", array[i]);
}
```

Buffer over-/underflow

Rust

```rust
fn func() {
    let array = [1, 2 ,3];
    let i = 100;
    println!("{}", array[i]);
}
```

```
L4 error: this operation will panic at runtime
```

Buffer over-/underflow

Rust

```rust
fn func(i : usize) {
    let array = [1, 2 ,3];
    println!("{}", array[i]);
}
fn main(){
    func(100);
}
```

Buffer over-/underflow

Rust

```rust
fn func(i : usize) {
    let array = [1, 2 ,3];
    println!("{}", array[i]);
}
fn main(){
    func(100);
}
```

```
L3 runtime error: thread panicked at 'index out of bounds'
```

- Dynamic bounds checking (weak static checks)
- Often optimized away during compilation

Null pointer dereference

C

```c
void foo() {
  device_t *dev = NULL;
  // ...
  int id = dev->id;
  if (dev) {
    // ...
  }
}
```

Null pointer dereference

C

```c
void foo() {
  device_t *dev = NULL;
  // ...
  int id = dev->id;
  if (dev) {
    // ...
  }
}
```

```
L4 runtime error: segfault (undefined behavior)
```

Null pointer dereference

C

```c
void foo() {
  device_t *dev = NULL;
  // ...
  int id = dev->id;
  if (dev) {
    // ...
  }
}
```

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. [...] This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

— Tony Hoare, 2009

Null pointer dereference

Rust does not have `null` ⇒ `Option` instead

Rust

```rust
fn foo() {
    let dev: Option<Device> = None;
    // ...
    let id = dev.id;
}
```

Null pointer dereference

Rust does not have `null` ⇒ `Option` instead → type-level safety

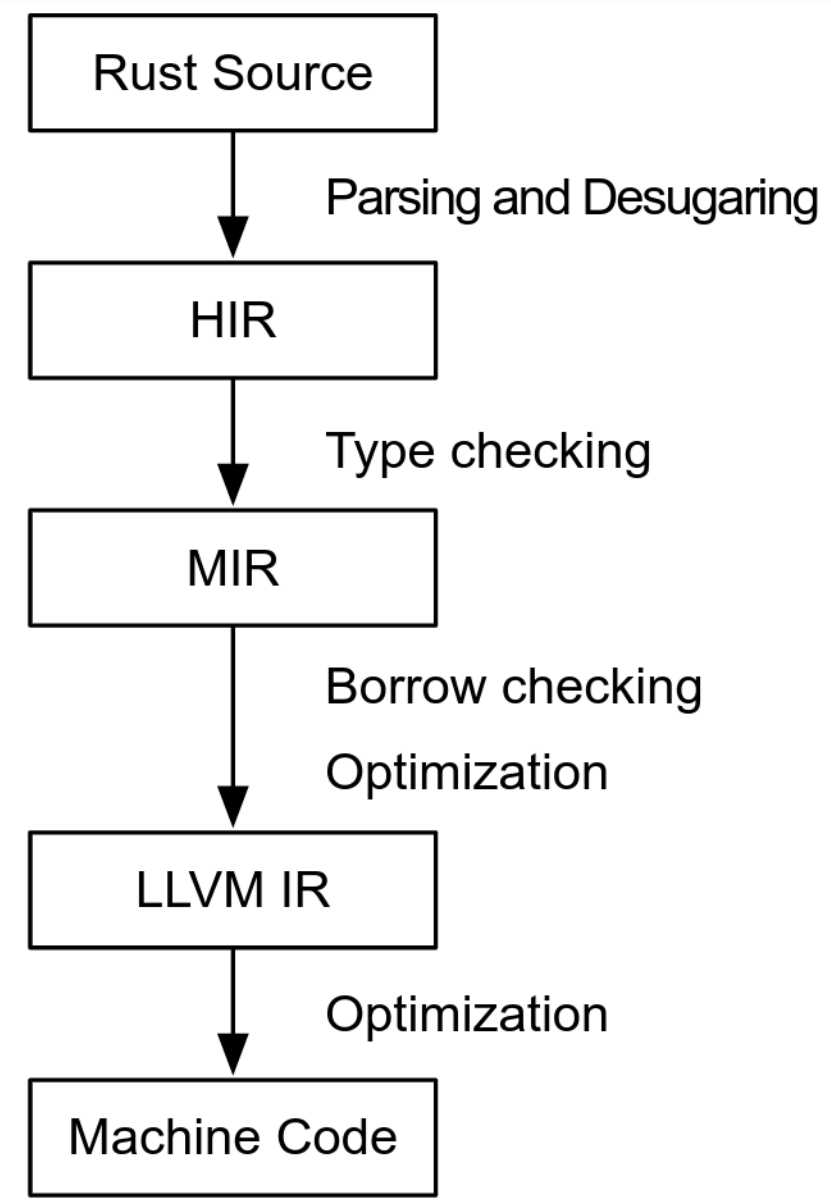
Rust

```rust
fn foo() {
    let dev: Option<Device> = None;
    // ...
    let id = dev.id;
}
```

```
L4 error: no field `id` on type std::option::Option`
```

Outline

Compilation



Features

Functional programming languages research (ML/Haskell)

Features

Functional programming languages research (ML/Haskell)

- Favour immutability
- Algebraic data types & pattern matching
- Traits ($\approx$ Haskell's type classes)
- Closures
- Error handling without exceptions
- Iterators

Features

Functional programming languages research (ML/Haskell)

- Favour immutability
- Algebraic data types & pattern matching
- Traits ($\approx$ Haskell's type classes)
- Closures
- Error handling without exceptions
- Iterators

Zero-cost abstractions

Features

Functional programming languages research (ML/Haskell)

- Favour immutability
- Algebraic data types & pattern matching
- Traits ($\approx$ Haskell's type classes)
- Closures
- Error handling without exceptions
- Iterators

Zero-cost abstractions

"What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better." — Bjarne Stroustrup

Algebraic Data Types

Rust

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

Algebraic Data Types

Rust

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

Rust

```rust
enum Shape {
    Circle,
    Rectangle,
}
```

Algebraic Data Types

Rust

```rust
enum Shape {
    Circle {
        origin: Point,
        radius: f32,
    },
    Rectangle {
        top_left: Point,
        bot_right: Point,
    },
}
```

Pattern matching

Rust

```rust
use std::f32::consts::PI;

fn shape_area(shape: &Shape) -> f32 {
    match shape {
        Shape::Circle { radius: r, .. } => PI * r * r,
    }
}
```

Pattern matching

Rust

```rust
use std::f32::consts::PI;

fn shape_area(shape: &Shape) -> f32 {
    match shape {
        Shape::Circle { radius: r, .. } => PI * r * r,
    }
}
```

```
L4 error: non-exhaustive patterns:
          `&Rectangle { .. }` not covered
```

Pattern matching

Rust

```rust
use std::f32::consts::PI;

fn shape_area(shape: &Shape) -> f32 {
    match shape {
        Shape::Circle { radius: r, .. } => PI * r * r,
        Shape::Rectangle { top_left, bot_right } =>
            ((bot_right.x - top_left.x)
            * (bot_right.y - top_left.y)).abs()
    }
}
```

Pattern matching

Rust

```rust
let circle: Shape = Shape::Circle {
    origin: Point { x: 0.0, y: 0.0 },
    radius: 1.0,
};
println!("Area: {}", shape_area(&circle));
```

Pattern matching

Rust

```rust
let circle: Shape = Shape::Circle {
    origin: Point { x: 0.0, y: 0.0 },
    radius: 1.0,
};
println!("Area: {}", shape_area(&circle));
```

```
>> Area: 3.1415927
```

Traits

Collection of methods for `Self` with optional default implementations

Rust

```rust
trait HasArea {
    fn area(&self) -> f32;

    fn integer_area(&self) -> i32 {
        self.area() as i32
    }
}
```

Traits

`impl .. for ..` : trait implementations

Rust

```rust
struct Circle {
    origin: Point,
    radius: f32,
}
impl HasArea for Circle {
    fn area(&self) -> f32 {
        PI * self.radius * self.radius
    }
}
```

Traits

Rust

```rust
let circle: Circle = Circle {
    origin: Point { x: 0.0, y: 0.0 },
    radius: 1.0,
};
println!("Area: {}", circle.area());
```

```
>> Area: 3.1415927
```

Traits

`impl .. :` static & non-static (i.e. instance-) methods

Rust

```rust
impl Circle {
    fn new(radius: f32) -> Circle {
        Circle {
            origin: Point { x: 0.0, y: 0.0 },
            radius: radius,
        }
    }
    fn get_radius(&self) -> f32 {
        self.radius
    }
    fn set_radius(&mut self, radius: f32) {
        self.radius = radius;
    }
}
```

Traits

Rust

```rust
let mut circle = Circle::new(10.0);
circle.set_radius(11.0);
println!("Radius: {}", circle.get_radius());
```

Traits

Rust

```rust
let mut circle = Circle::new(10.0);
circle.set_radius(11.0);
println!("Radius: {}", circle.get_radius());
```

```
>> Radius: 11
```

Traits

Rust

```rust
let mut circle = Circle::new(10.0);
circle.set_radius(11.0);
println!("Radius: {}", circle.get_radius());
```

```
>> Radius: 11
```

→ *Object-oriented programming with Rust*

Is Rust an Object-Oriented Programming Language?

`https://doc.rust-lang.org/book/ch17-00-oop.html`

Traits

Rust

```rust
fn total_area<S: HasArea>(shapes: &Vec<S>) -> f32 {
    let mut total_area = 0.0;
    for s in shapes {
        total_area += s.area();
    }
    total_area
}
```

Trait bounds on generics

<S: HasArea> refers to a generic type S that must implement the trait HasArea.

Closures

Anonymous functions that *close over* free variables

Closures

Anonymous functions that *close over* free variables

JavaScript

```
let x = 3;
let addX = y => x + y;
console.log(addX(5));
```

Rust

```
let x = 3;
let add_x = |y| x + y;
println!("{}", add_x(5));
```

Closures

Anonymous functions that *close over* free variables

JavaScript

```
let x = 3;
let addX = y => x + y;
console.log(addX(5));
```

Rust

```
let x = 3;
let add_x = |y| x + y;
println!("{}", add_x(5));
```

```
>> 8
```

Error handling

- Exceptions: run-time cost and complexity
- C-style error codes: easily forgotten & confusing: is `0` bad or not?

Error handling

- Exceptions: run-time cost and complexity
- C-style error codes: easily forgotten & confusing: is `0` bad or not?

Rust's approach: `Option<T>` and `Result<T, E>`

Cheap and explicit error-handling

Error handling

Rust

```rust
enum Option<T> {
    None,
    Some(T),
}
```

Error handling

Rust

```rust
enum Option<T> {
    None,
    Some(T),
}
```

Rust

```rust
fn div(x: f32, y: f32) -> Option<f32> {
    if y == 0.0 {
        None
    } else {
        Some(x / y)
    }
}
```

Error handling

Rust

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Error handling

Rust

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Rust

```rust
fn int_to_byte(input: i32) -> Result<u8, String> {
    if input < 0 {
        Err(String::from("Negative number"))
    } else if input > std::u8::MAX as i32 {
        Err(String::from("Too large"))
    } else {
        Ok(input as u8)
    }
}
```

Iterators

Efficient way to traverse a sequence

- Iterators
- Iterator adaptors
- Consuming adaptors

Iterators

Efficient way to traverse a sequence

- Iterators
- Iterator adaptors
- Consuming adaptors

Rust

```rust
let result: Vec<i32> = (0..5)
    .map(|x| x * x)
    .filter(|x| x % 2 == 0)
    .collect();
```

Iterators

Efficient way to traverse a sequence

- Iterators
- Iterator adaptors
- Consuming adaptors

Rust

```rust
let result: Vec<i32> = (0..5)
    .map(|x| x * x)
    .filter(|x| x % 2 == 0)
    .collect();
```

```
>> [0, 4, 16]
```

Iterators

Rust

```rust
fn total_area<S: HasArea>(shapes: &Vec<S>) -> f32 {
    let mut total_area = 0.0;
    for s in shapes {
        total_area += s.area();
    }
    total_area
}
```

Iterators

Rust

```rust
fn total_area<S: HasArea>(shapes: &Vec<S>) -> f32 {
    let mut total_area = 0.0;
    for s in shapes {
        total_area += s.area();
    }
    total_area
}
```

Rust

```rust
fn total_area2<S: HasArea>(shapes: &Vec<S>) -> f32 {
    shapes.iter()
          .map(|s| s.area())
          .sum()
}
```

Iterators

Rust

```rust
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Iterators

Rust

```rust
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Iterator API Documentation

Iterators

Rust

```rust
struct FinalCountdown { current: u32 }
```

Iterators

Rust

```rust
struct FinalCountdown { current: u32 }
```

Rust

```rust
impl Iterator for FinalCountdown {
    type Item = u32;

    fn next(&mut self) -> Option<u32> {
        if self.current > 0 {
            let result = self.current;
            self.current -= 1;
            Some(result)
        } else {
            None
        }
    }
}
```

Iterators

Rust

```rust
for i in (FinalCountdown { current: 5 }) {
    println!("{}", i);
}
```

Iterators

Rust

```rust
for i in (FinalCountdown { current: 5 }) {
    println!("{}", i);
}
```

```
5
4
3
2
1
```

Iterators

Rust

```rust
for i in (FinalCountdown { current: 5 }) {
    println!("{}", i);
}
```

Rust

```rust
match IntoIterator::into_iter(FinalCountdown {current : 5}) {
    mut iter => loop {
        let next;
        match iter.next() {
            Some(val) => next = val,
            None => break,
        };
        let i = next;
        let () = { println!("{}",i); };},
};
```

Unsafe

`unsafe`: "*Trust me compiler, I know what I'm doing*"

Unsafe

`unsafe`: "*Trust me compiler, I know what I'm doing*"

- Call unsafe functions
- Mutate static (global) variables: data races
- Dereference raw pointers

Unsafe

`unsafe`: "*Trust me compiler, I know what I'm doing*"

- Call unsafe functions
- Mutate static (global) variables: data races
- Dereference raw pointers

Rust

```rust
unsafe fn vec_first<T>(vec: &Vec<T>) -> &T {
    vec.get_unchecked(0)
}
```

Unsafe

`unsafe`: "*Trust me compiler, I know what I'm doing*"

- Call unsafe functions
- Mutate static (global) variables: data races
- Dereference raw pointers

Rust

```
1  unsafe fn vec_first<T>(vec: &Vec<T>) -> &T {
2      vec.get_unchecked(0)
3  }
```

Rust

```
1  let vec = vec![1, 2, 3];
2  let first = unsafe { vec_first(&vec) };
```

Much more

- Data race protection mechanism by ownership/borrowing
- Macros (`println!`, `vec!`)
- Foreign Function Interface (FFI)
- Lifetimes
- Built-in support for testing & benchmarking
- Newtypes
- Flexible module system
- ...

Outline

Can we trust Rust's claims?

Well-typed programs should not go wrong!

- Avoid certain illegal behavior
- Rust: memory safe, no data races

Can we trust Rust's claims?

Well-typed programs should not go wrong!

- Avoid certain illegal behavior
- Rust: memory safe, no data races

Given

- Rust's type system

To prove:

Can we trust Rust's claims?

Well-typed programs should not go wrong!

- Avoid certain illegal behavior
- Rust: memory safe, no data races

Given

- Rust's type system

To prove:

Can we trust Rust's claims?

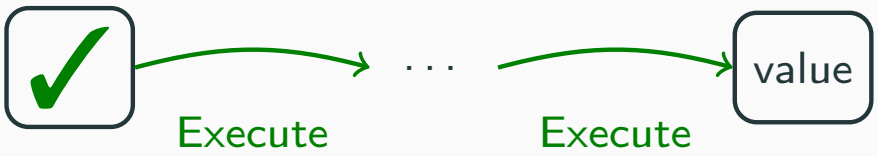
Well-typed programs should not go wrong!

- Avoid certain illegal behavior
- Rust: memory safe, no data races

Given

- Rust's type system

To prove:



Can we trust Rust's claims?

Well-typed programs should not go wrong!

- Avoid certain illegal behavior
- Rust: memory safe, no data races

Given

- Rust's type system

To prove:

Can we trust Rust's claims?

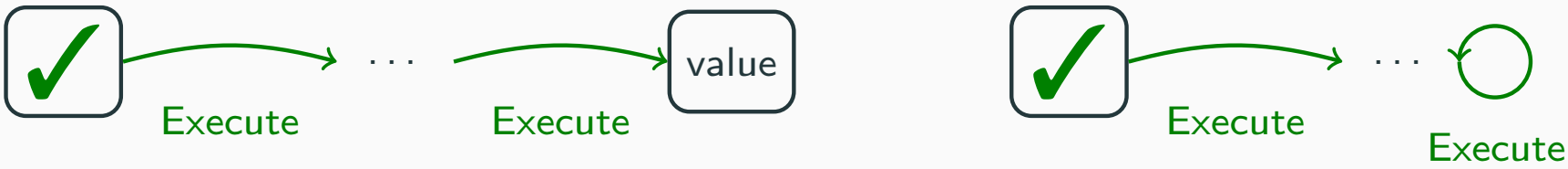
Well-typed programs should not go wrong!

- Avoid certain illegal behavior
- Rust: memory safe, no data races

Given

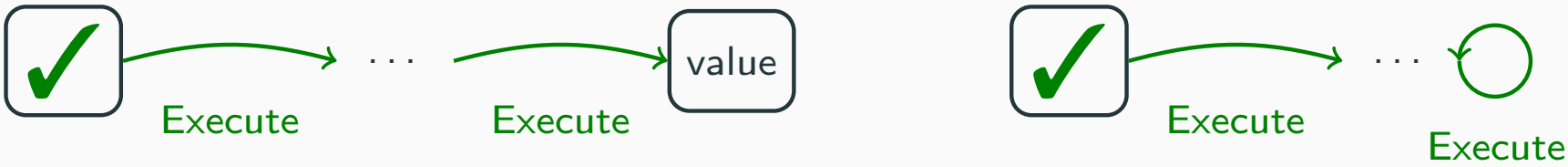
- Rust's type system

To prove:



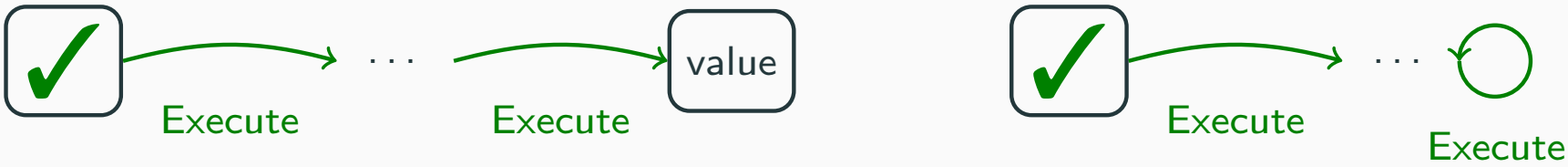
Classic type safety

To prove:



Classic type safety

To prove:

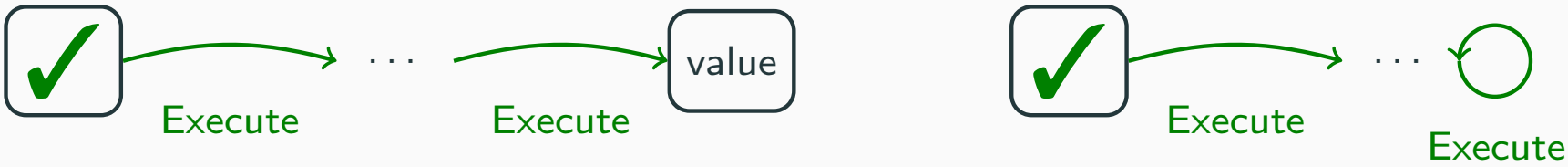


Classic (syntactic) type safety:

- **Progress**: well-typed code is either a value or can take a step
- **Preservation**: well-typed code stays well-typed after taking a step

Classic type safety

To prove:

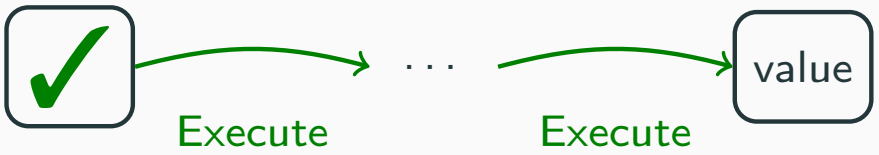


Classic (syntactic) type safety:

- **Progress**: well-typed code is either a value or can take a step
- **Preservation**: well-typed code stays well-typed after taking a step

Classic type safety

To prove:

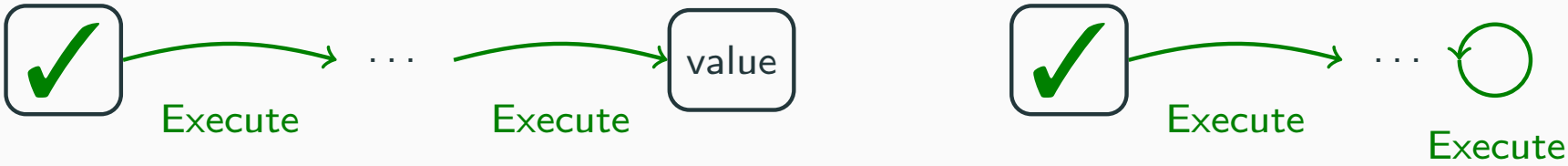


Classic (syntactic) type safety:

- **Progress**: well-typed code is either a value or can take a step
- **Preservation**: well-typed code stays well-typed after taking a step

Classic type safety

To prove:

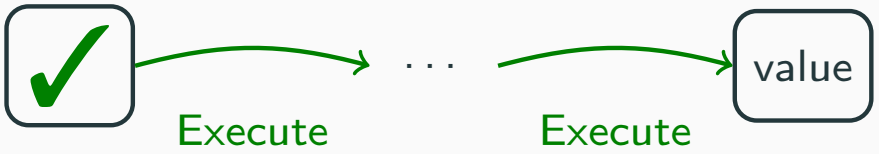


Classic (syntactic) type safety:

- **Progress**: well-typed code is either a value or can take a step
- **Preservation**: well-typed code stays well-typed after taking a step

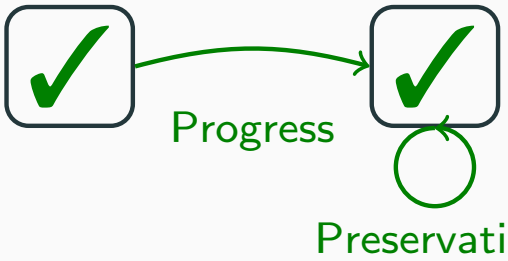
Classic type safety

To prove:



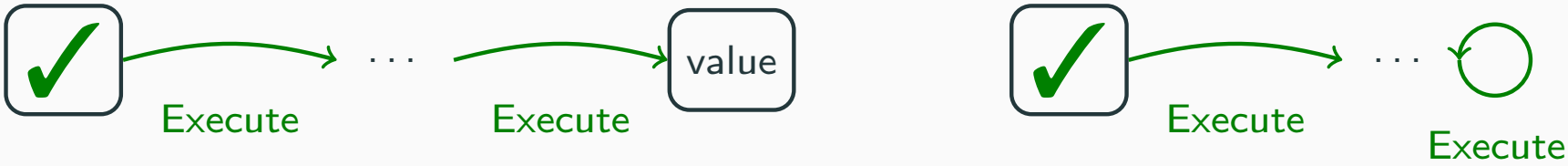
Classic (syntactic) type safety:

- **Progress**: well-typed code is either a value or can take a step
- **Preservation**: well-typed code stays well-typed after taking a step



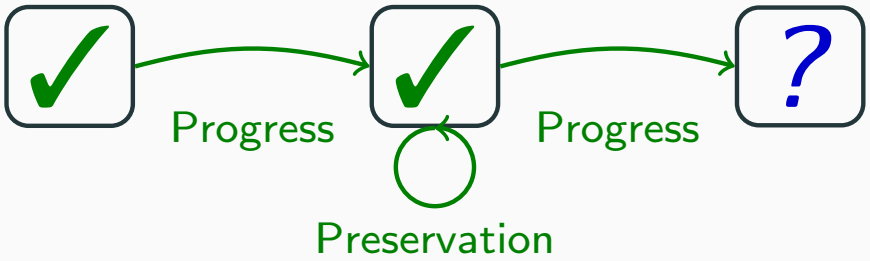
Classic type safety

To prove:



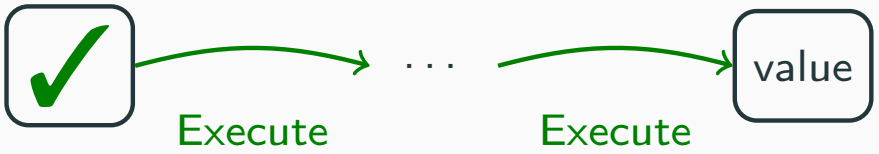
Classic (syntactic) type safety:

- **Progress**: well-typed code is either a value or can take a step
- **Preservation**: well-typed code stays well-typed after taking a step



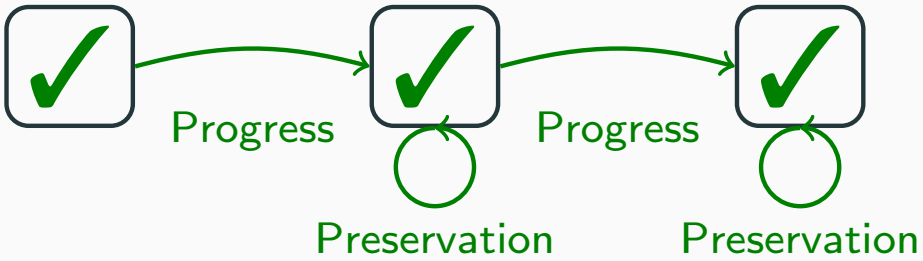
Classic type safety

To prove:



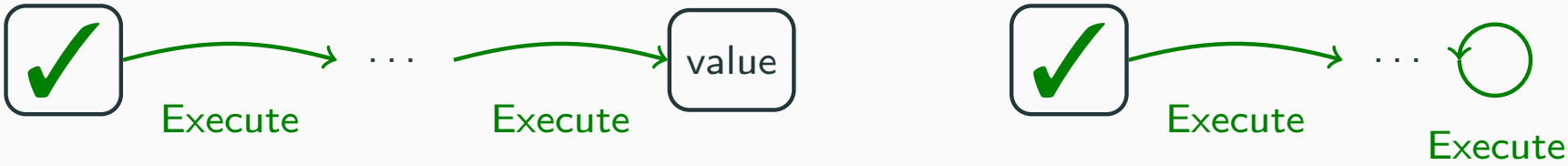
Classic (syntactic) type safety:

- **Progress**: well-typed code is either a value or can take a step
- **Preservation**: well-typed code stays well-typed after taking a step



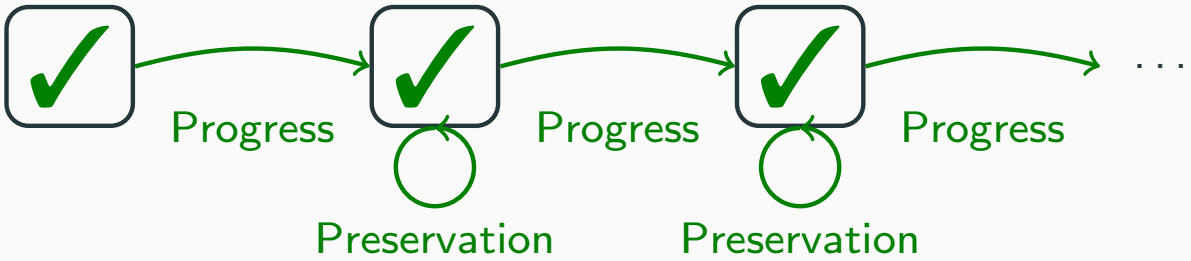
Classic type safety

To prove:



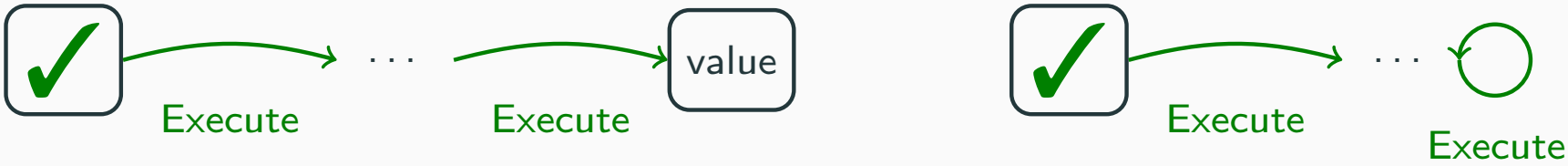
Classic (syntactic) type safety:

- **Progress**: well-typed code is either a value or can take a step
- **Preservation**: well-typed code stays well-typed after taking a step



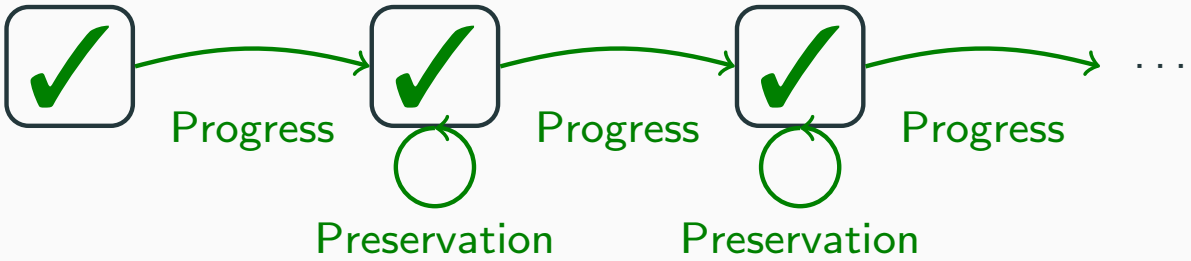
Classic type safety

To prove:



Classic (syntactic) type safety:

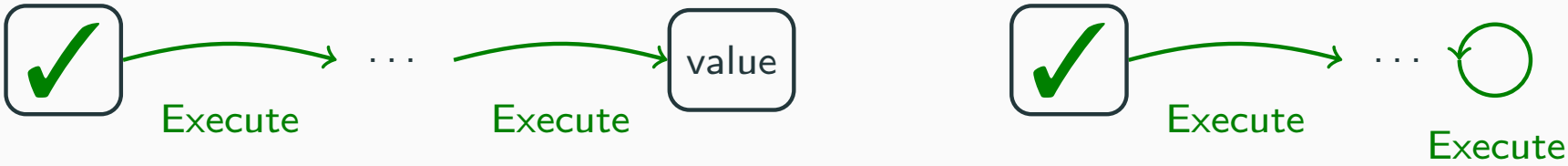
- **Progress**: well-typed code is either a value or can take a step
- **Preservation**: well-typed code stays well-typed after taking a step



So, we just prove type safety and Rust happily ever after?

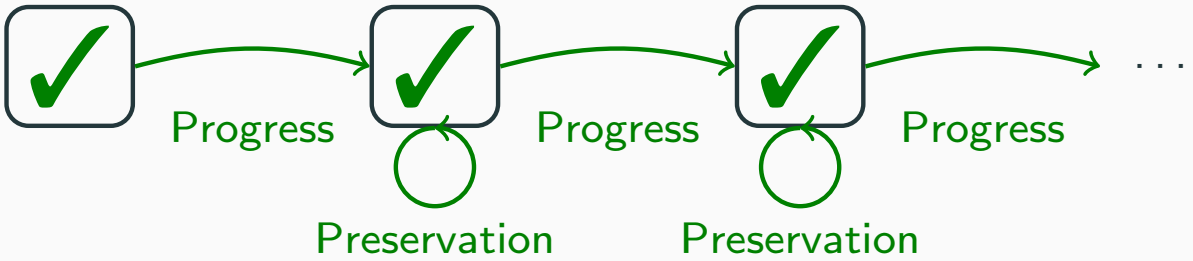
Classic type safety

To prove:



Classic (syntactic) type safety:

- **Progress**: well-typed code is either a value or can take a step
- **Preservation**: well-typed code stays well-typed after taking a step



So, we just prove type safety and Rust happily ever after? **Not so fast!**

Can we trust Rust's claims? (2)

Well-typed programs do not go wrong?

- Rust: memory safe, no data races
- 'No' runtime errors

Given

- Rust's type system

Can we trust Rust's claims? (2)

Well-typed programs do not go wrong?

- Rust: memory safe, no data races
- 'No' runtime errors

Given

- Rust's type system
- `Unsafe` **construct**: encapsulated in libraries

Can we trust Rust's claims? (2)

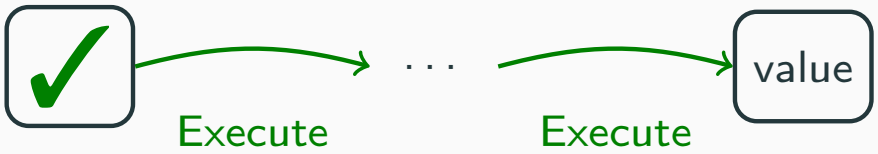
Well-typed programs do not go wrong?

- Rust: memory safe, no data races
- 'No' runtime errors

Given

- Rust's type system
- `Unsafe` **construct**: encapsulated in libraries

To prove:



Can we trust Rust's claims? (2)

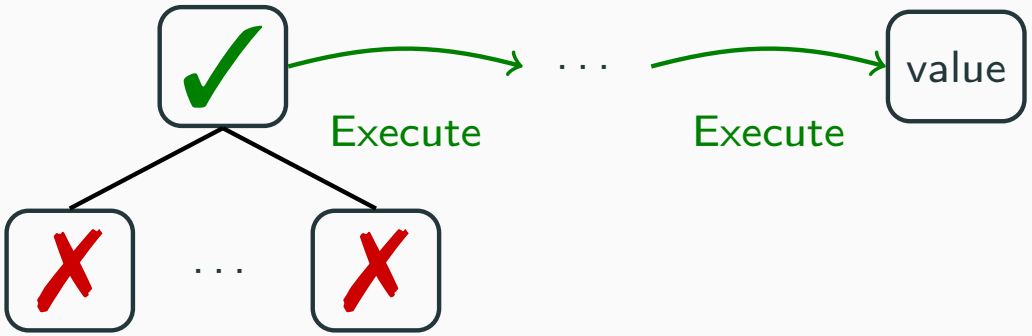
Well-typed programs do not go wrong?

- Rust: memory safe, no data races
- 'No' runtime errors

Given

- Rust's type system
- `Unsafe` **construct**: encapsulated in libraries

To prove:



Can we trust Rust's claims? (2)

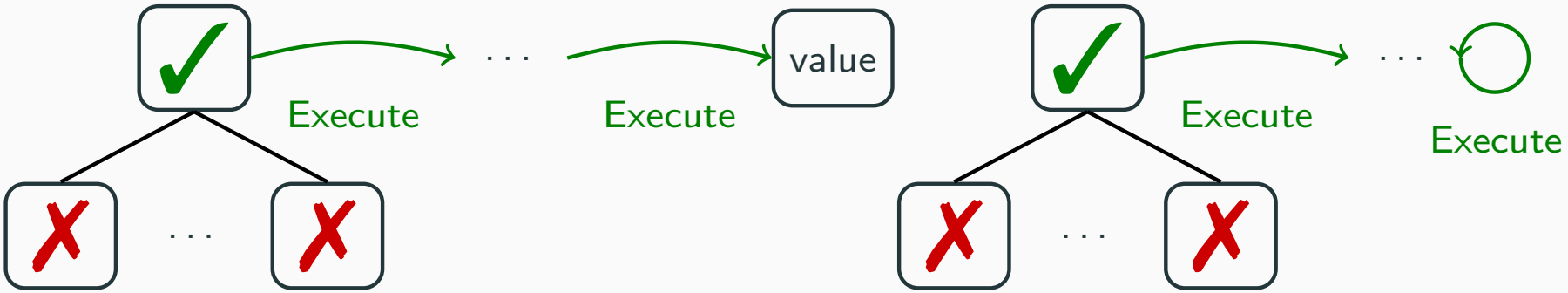
Well-typed programs do not go wrong?

- Rust: memory safe, no data races
- 'No' runtime errors

Given

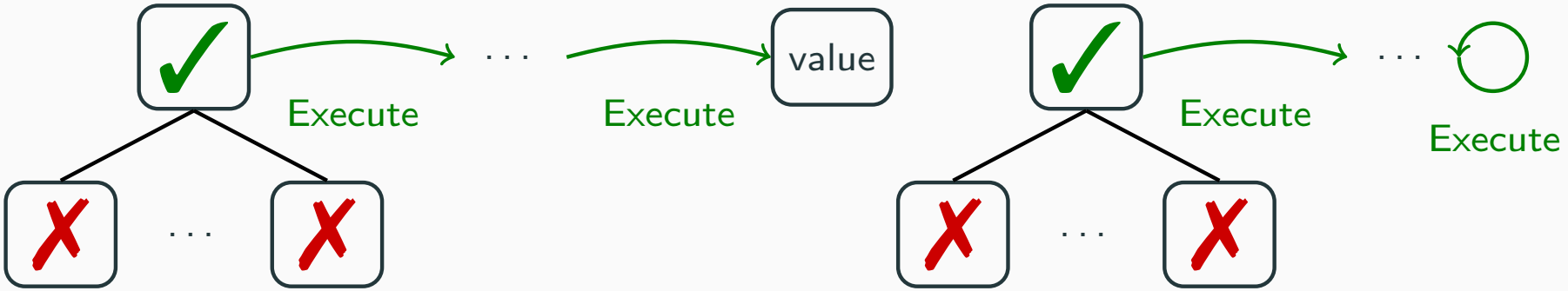
- Rust's type system
- `Unsafe` **construct**: encapsulated in libraries

To prove:



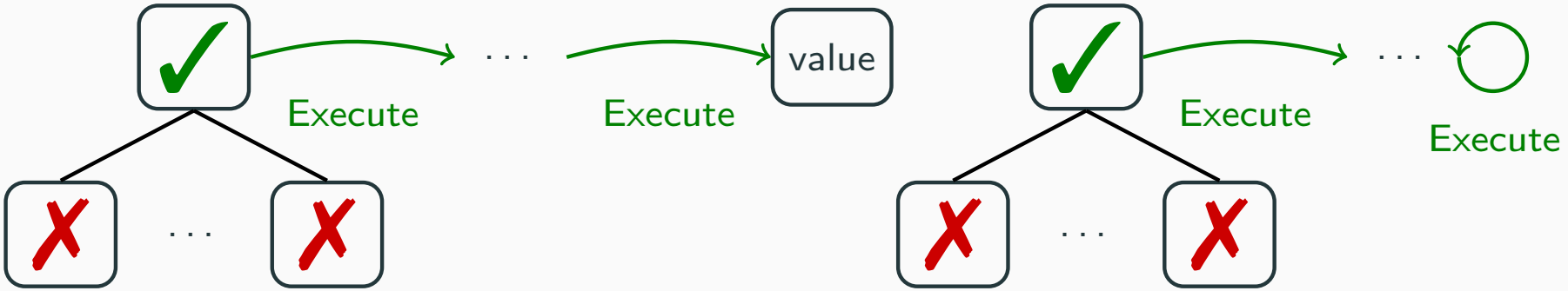
Unsafe: modular type safety

To prove:



Unsafe: modular type safety

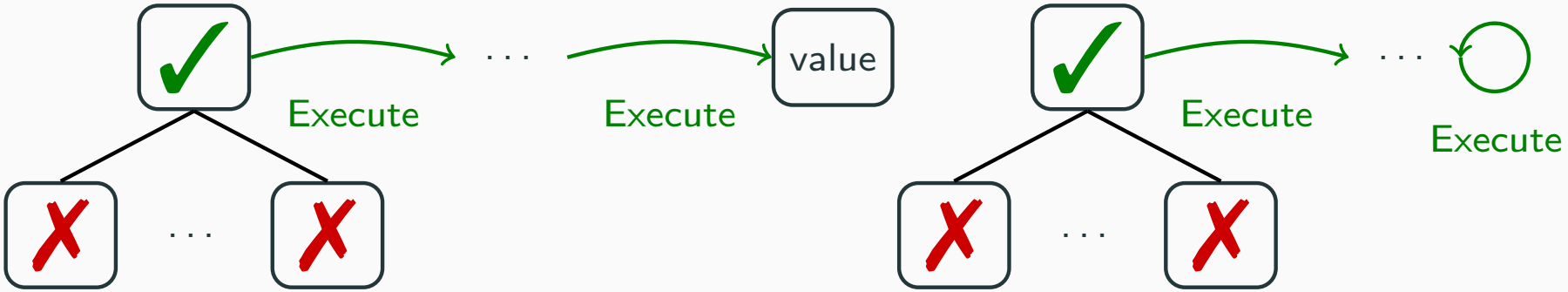
To prove:



Problem: unsafe blocks are **not** well-typed...

Unsafe: modular type safety

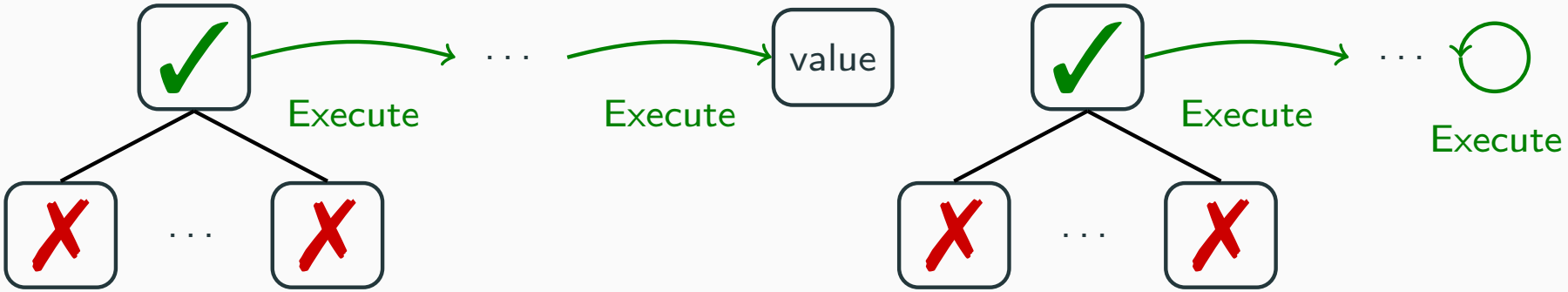
To prove:



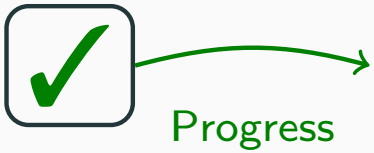
Problem: unsafe blocks are **not** well-typed...

Unsafe: modular type safety

To prove:

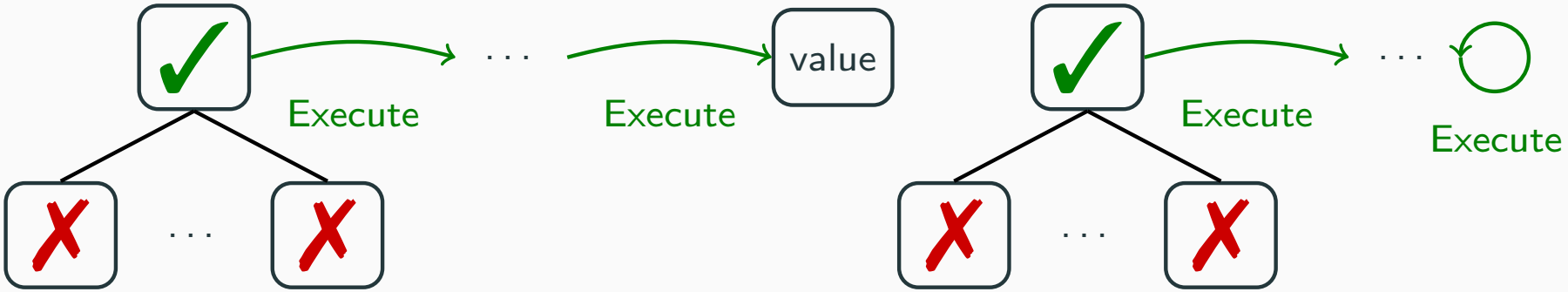


Problem: unsafe blocks are **not** well-typed...



Unsafe: modular type safety

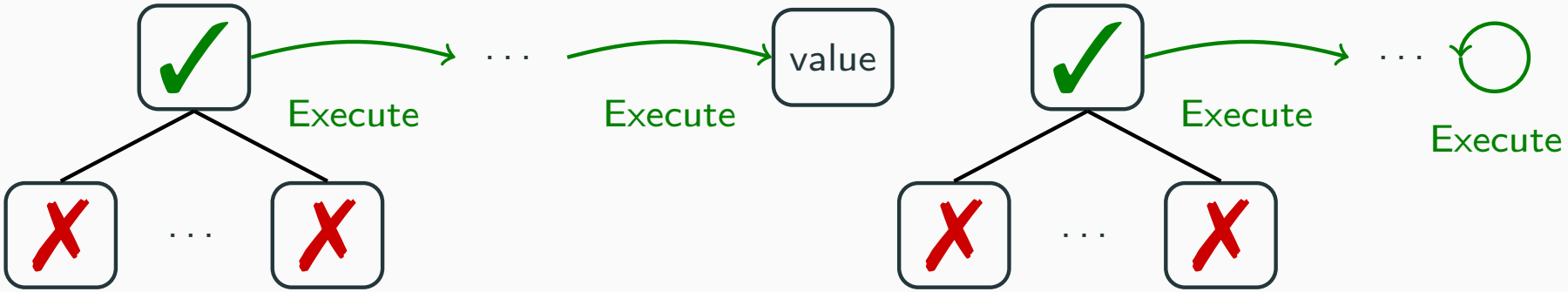
To prove:



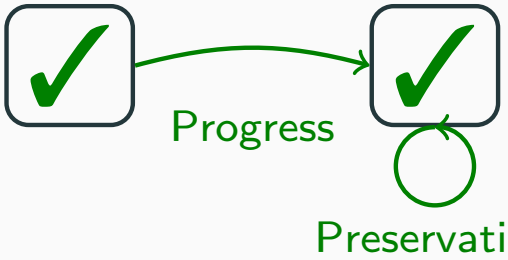
Problem: unsafe blocks are **not** well-typed...

Unsafe: modular type safety

To prove:

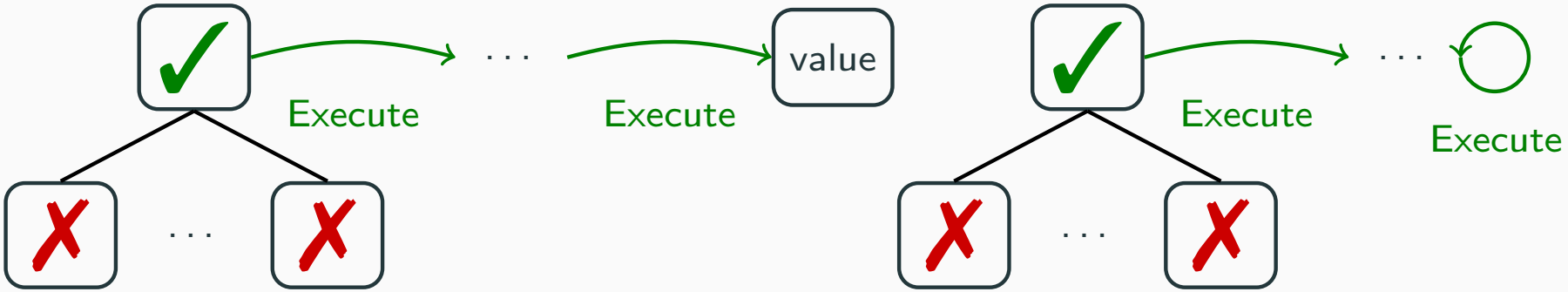


Problem: unsafe blocks are **not** well-typed...

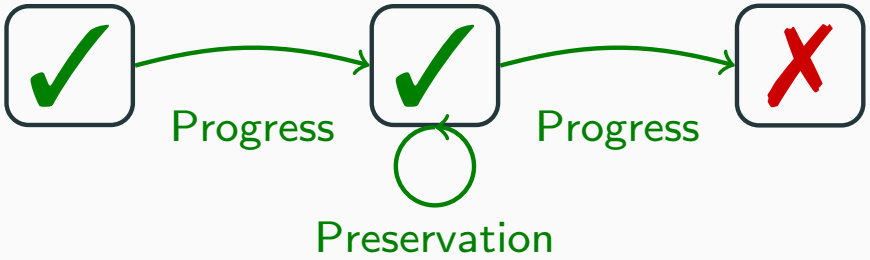


Unsafe: modular type safety

To prove:

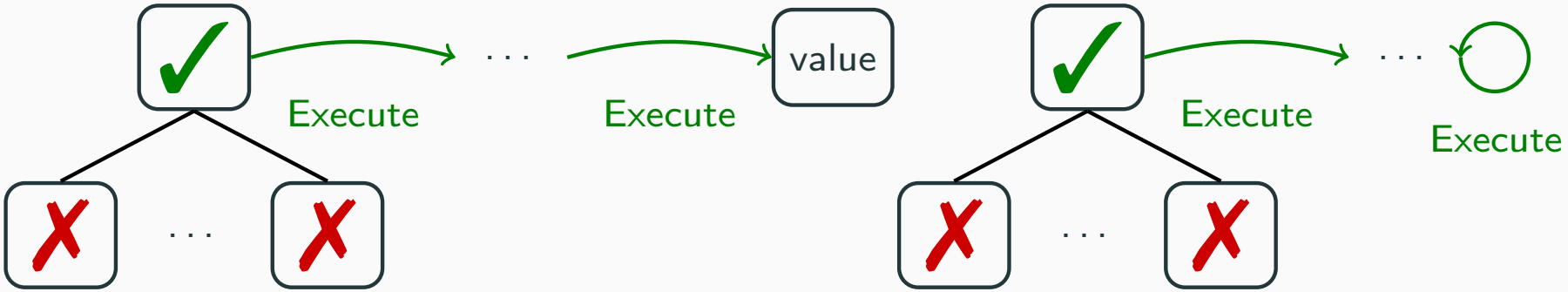


Problem: unsafe blocks are **not** well-typed...

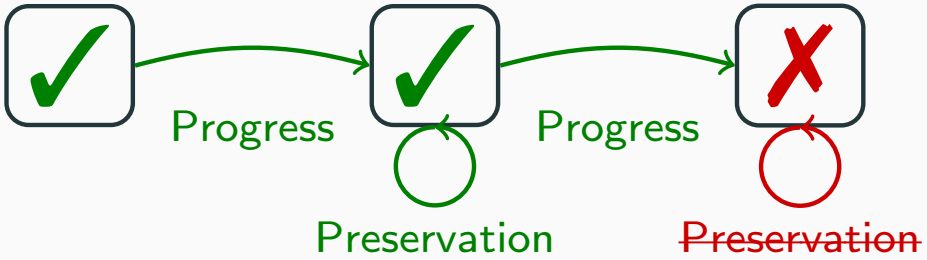


Unsafe: modular type safety

To prove:

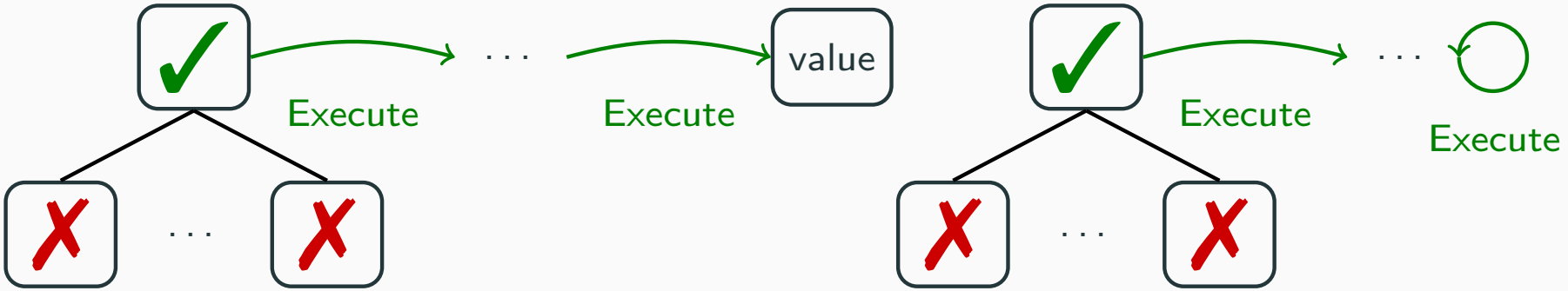


Problem: unsafe blocks are **not** well-typed...

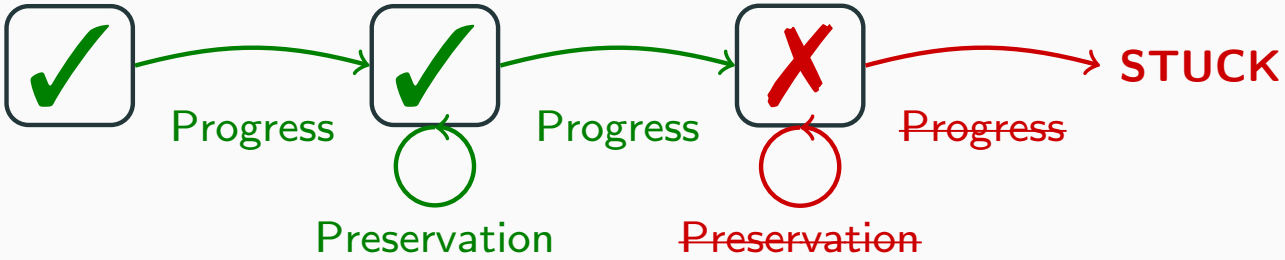


Unsafe: modular type safety

To prove:

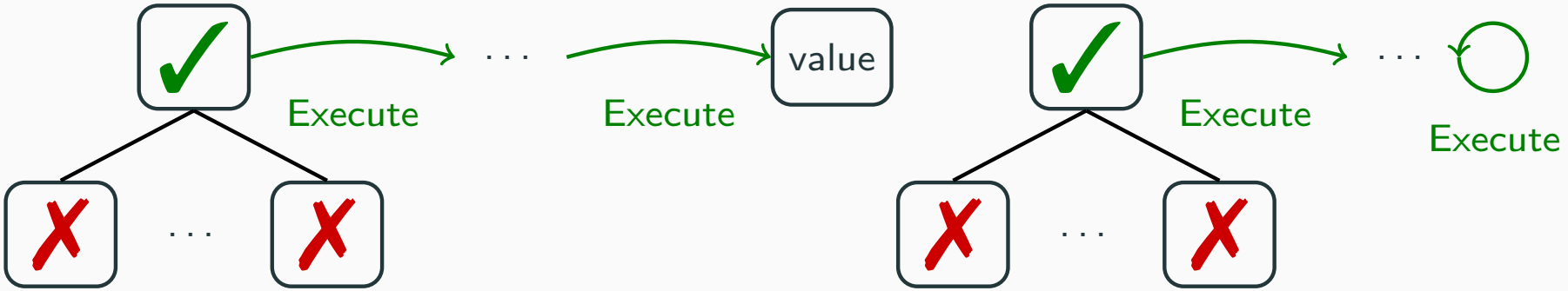


Problem: unsafe blocks are **not** well-typed...



Unsafe: modular type safety

To prove:



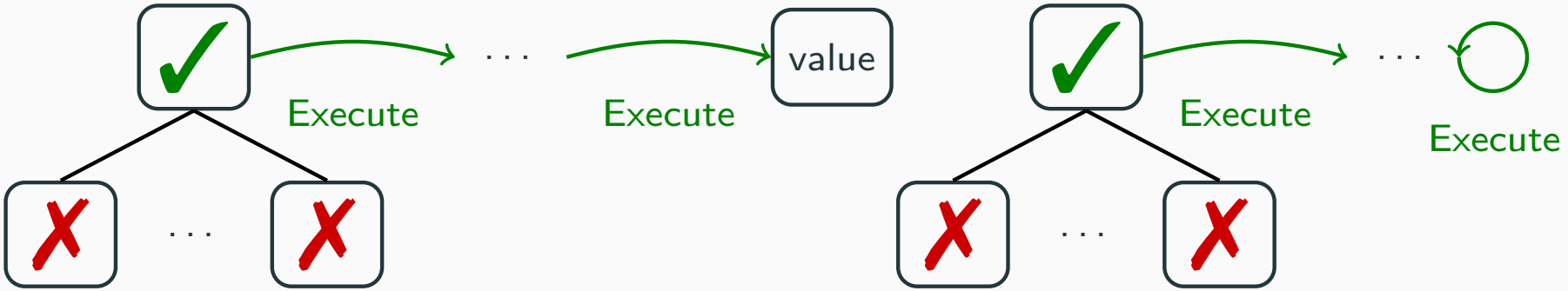
Problem: unsafe blocks are **not** well-typed...

Solution: prove that unsafe libraries **behave like** safe code at interfaces

$\Rightarrow$ **semantic** type safety

Unsafe: modular type safety

To prove:



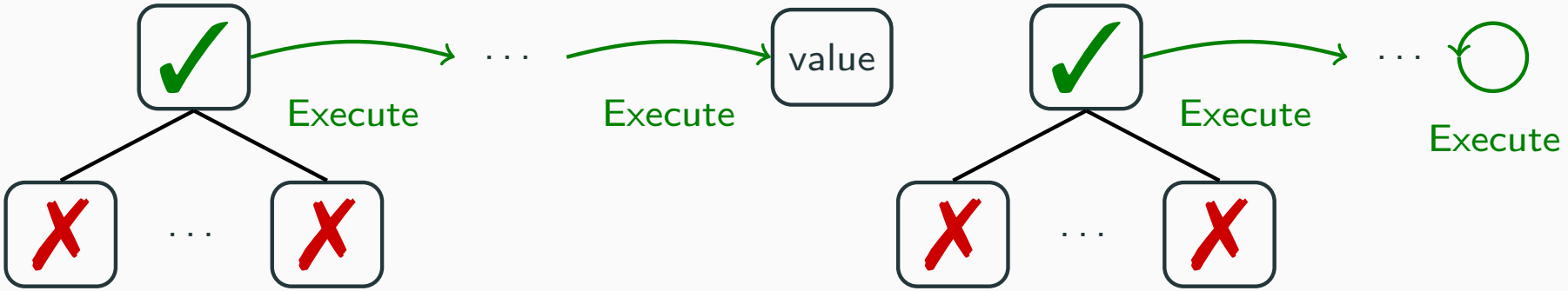
Problem: unsafe blocks are **not** well-typed...

Solution: prove that unsafe libraries **behave like** safe code at interfaces

$\Rightarrow$ **semantic** type safety

Unsafe: modular type safety

To prove:



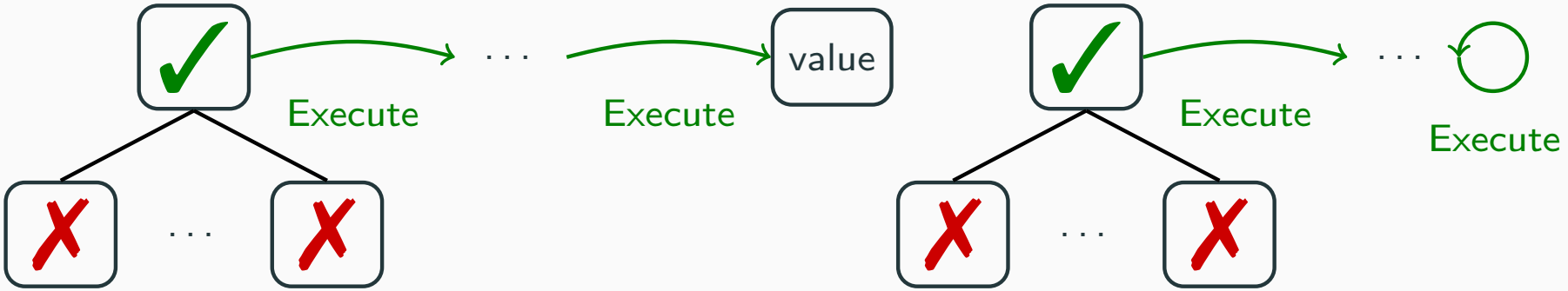
Problem: unsafe blocks are **not** well-typed...

Solution: prove that unsafe libraries **behave like** safe code at interfaces

$\Rightarrow$ **semantic** type safety

Unsafe: modular type safety

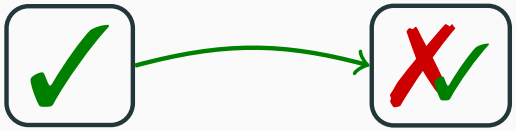
To prove:



Problem: unsafe blocks are **not** well-typed...

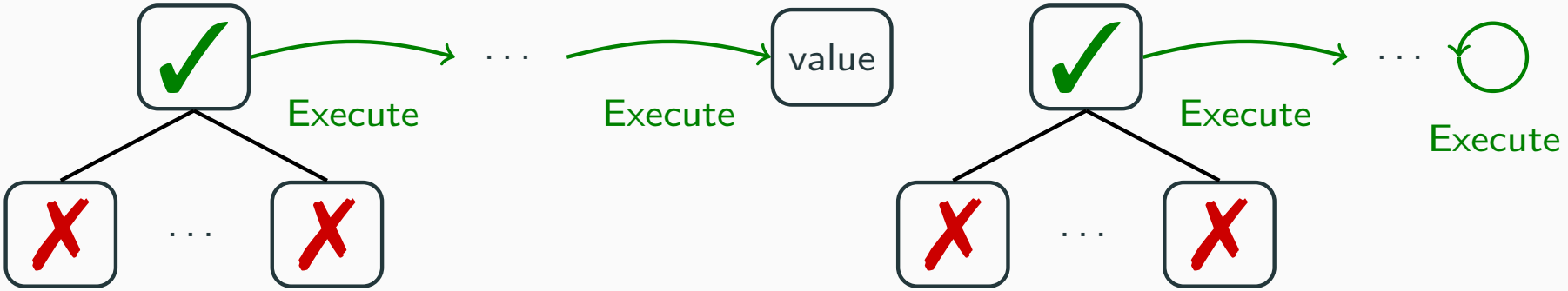
Solution: prove that unsafe libraries **behave like** safe code at interfaces

$\Rightarrow$ **semantic** type safety



Unsafe: modular type safety

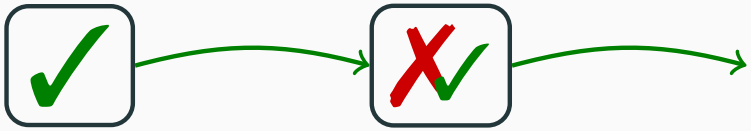
To prove:



Problem: unsafe blocks are **not** well-typed...

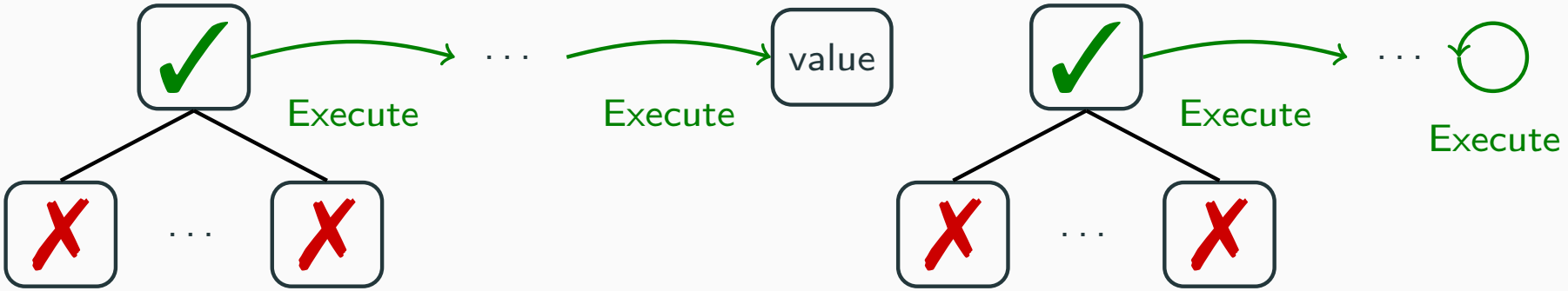
Solution: prove that unsafe libraries **behave like** safe code at interfaces

$\Rightarrow$ **semantic** type safety



Unsafe: modular type safety

To prove:



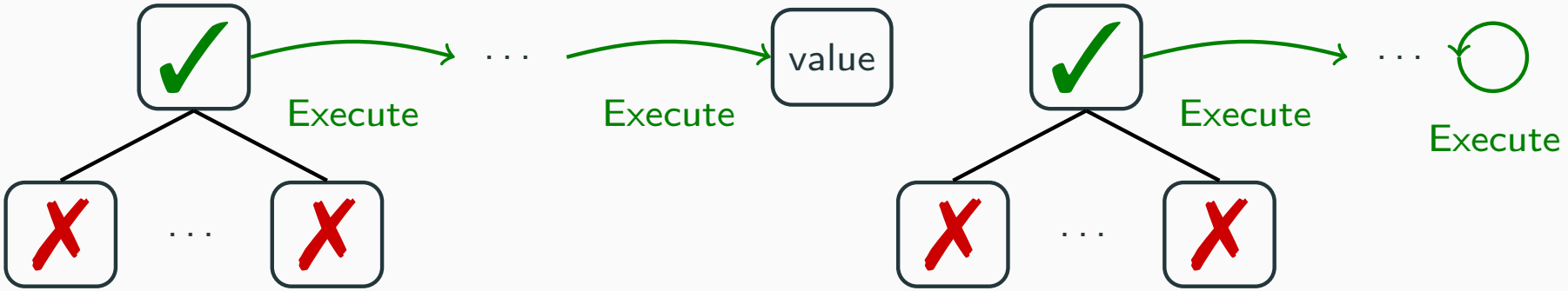
Problem: unsafe blocks are **not** well-typed...

Solution: prove that unsafe libraries **behave like** safe code at interfaces

$\Rightarrow$ **semantic** type safety

Unsafe: modular type safety

To prove:



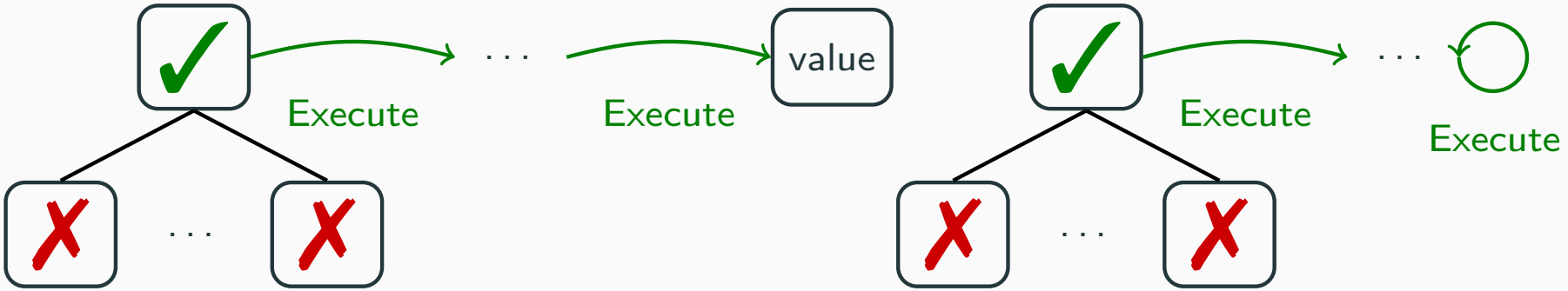
Problem: unsafe blocks are **not** well-typed...

Solution: prove that unsafe libraries **behave like** safe code at interfaces

$\Rightarrow$ **semantic** type safety

Unsafe: modular type safety

To prove:



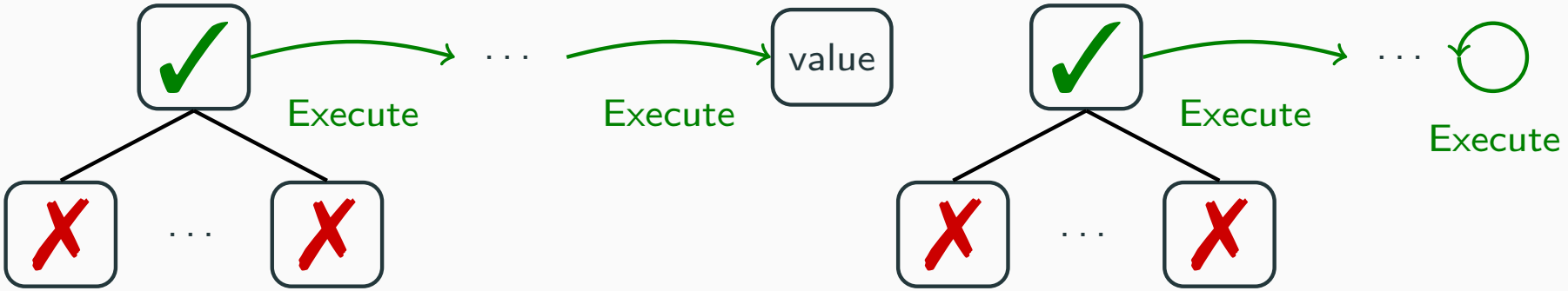
Problem: unsafe blocks are **not** well-typed...

Solution: prove that unsafe libraries **behave like** safe code at interfaces

$\Rightarrow$ **semantic** type safety

Unsafe: modular type safety

To prove:



Problem: unsafe blocks are **not** well-typed...

Solution: prove that unsafe libraries **behave like** safe code at interfaces
$\Rightarrow$ **semantic** type safety

RustBelt: several important Rust libraries proven semantically type-safe

Key takeaway

Rust is a systems programming language that runs blazingly fast, yet still guarantees memory and thread safety.

Exercise sessions (optional)

When?		Where?
Tuesday 15/11	13:30 – 16:00	Q&A on Blackboard Collaborate
Wednesday 16/11	10:30 – 13:00	Q&A on Blackboard Collaborate
Friday 18/11	10:30 – 13:00	Currently Unallocated

As before: enroll on Tolinto!

Friday session will be allocated in case of sufficient demand;

$\Rightarrow$ you can still change your sessions if this happens

Exercise sessions (optional)

When?		Where?
Tuesday 15/11	13:30 – 16:00	Q&A on Blackboard Collaborate
Wednesday 16/11	10:30 – 13:00	Q&A on Blackboard Collaborate
Friday 18/11	10:30 – 13:00	Currently Unallocated

As before: enroll on Tolinto!

Friday session will be allocated in case of sufficient demand;

$\Rightarrow$ you can still change your sessions if this happens

Online editor: `https://play.rust-lang.org/`

Or install Rust on your computer: `https://www.rustup.rs/`

General References

- "The Rust Programming Language", Alex Crichton, Google Tech Talk, 2015, `https://www.youtube.com/watch?v=d1uraoHM8Gg`
- `http://manishearth.github.io/Presentations/Rust/`
- `https://doc.rust-lang.org/stable/rust-by-example/`
- `http://doc.rust-lang.org/book/`
- Rust courses by Jake Goulding at SecAppDev 2018, `https://www.secappdev.org/handouts-2018.html` (focus on software security)

Type Safety References

- Pierce, Benjamin C., and C. Benjamin. Types and programming languages. MIT press, 2002.
- Robin Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, 17(3):348–375, 1978.
- Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. **RustBelt**: Securing the Foundations of the Rust Programming Language. Proc. ACM Program. Lang. 2, POPL