Why I am learning a new programming language - and why you should too!

A brief introduction to the Rust programming language

by Dr. Michael O. Distler <distler@uni-mainz.de>

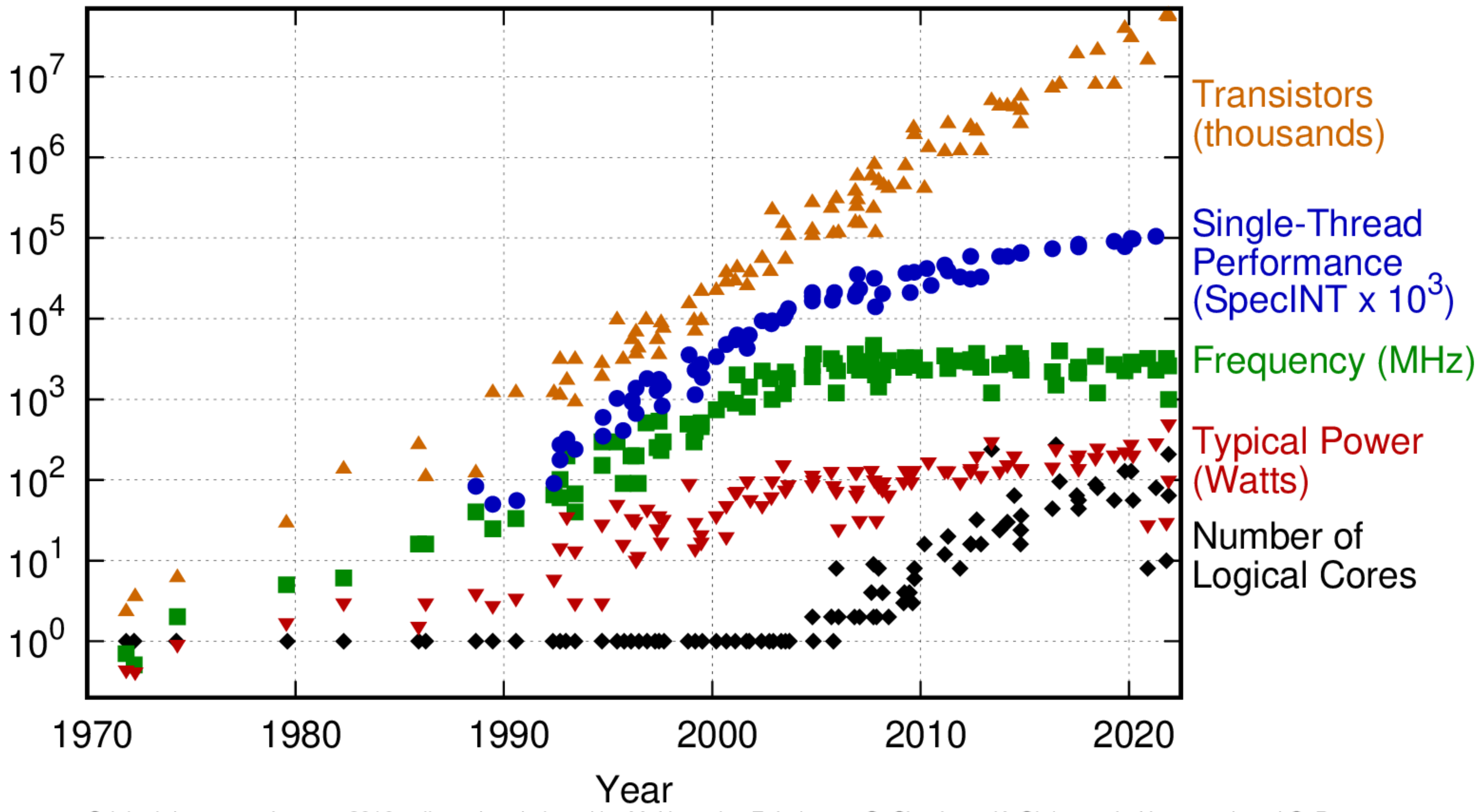
Mainz, 26 April 2023

Workshop “Tools for Physicists” organized by Dr. Bernd-Peter Otte

Content

- Introduction - Moore’s Law
- Functional Programming
- Introduction to Rust
- Concurrency in Rust by example

50 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

https://github.com/karlrupp/microprocessor-trend-data/

What about the future?

- Well, frequency and power will not experience any significant changes.
- Further improvements in instructions per clock may slightly increase single-threaded performance further, but not by a big margin.
- Transistor counts and number of cores are the two interesting quantities:
  How long can we keep Moore's Law going?
- We will (probably) see an increase in the number of cores in proportion to the number of transistors.
- ☞ **massively parallel algorithms are required**

Programming Paradigms

- Structured/Procedural
- Object-Oriented Programming
- Functional Programming
- …

Python Paradigms

- Structured - Functions, loops, conditionals
- OOP - Classes, objects, methods
- FP - ??? functions ???

"Uncle Bob" Martin - "The Future of Programming"

"If we have made any advances in software since 1945 it is almost entirely in what not to do"

YouTube

- Structured Programming:

  Don't use unrestrained GOTOs

- Object Oriented Programming:

  Don't use pointers to functions

- Functional Programming:

  Don't use assignment

What is wrong with assignment?

State

```
Door.open = true
Door.open = false

coding = "awesome"
coding = coding + "!!"
```

What is wrong with assignment?

Side-effects

```
names = ['Jan', 'Kim', 'Sara']

def double_name():
    for (i, name) in enumerate(names):
        names[i] = name + name
    print(names)

# prints out: ['JanJan', 'KimKim', 'SaraSara']
```

Problems with state

- Race conditions
- Complexity
- Unpredictability

Race conditions

```
groceries = ["apple", "banana", "orange",
             "strawberries", "cherries"]
basket = []


def get_groceries():
    for item in groceries:
        if item not in basket:
            basket.append(item)
        print(basket)
```

Unpredictable results

```
x = 1

def times_two():
    x = x*2


print(times_two())
# => 2

print(times_two())
# => 4
```

stateless

```
x = 1

def times_two():
    x = x*2
```

```
def times_two(x):
    return x*2

times_two(1)
```

NO STATE means:

- **Immutability**
- **Predictability: f(x)==f(x)**

HPC2018 lecture02: calculate π

```c
#include <stdio.h>

#define f(A) (4.0/(1.0+A*A))
const int n = 1000000000;

int main(int argc, char* argv[])
{
  int i;
  double w, x, sum, pi;

  w = 1.0/n;
  sum = 0.0;
  for (i=0; i<n; i++) {
    x = w * ((double)i + 0.5);
    sum = sum + f(x);
  }

  printf("pi = %.15f\n", w*sum);

  return 0;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

What is the problem?

except that one should better use the Gauss-Legendre quadrature here

lecture02: calculate π

```c
#include <stdio.h>

#define f(A) (4.0/(1.0+A*A))
const int n = 1000000000;

int main(int argc, char* argv[])
{
  int i;
  double w, x, sum, pi;

  w = 1.0/n;
  sum = 0.0;
  for (i=0; i<n; i++) {
    x = w * ((double)i + 0.5);
    sum = sum + f(x);
  }

  printf("pi = %.15f\n", w*sum);

  return 0;
}
```

```rust
const N: usize = 1_000_000_000;
const W: f64 = 1f64/(N as f64);

fn f(x: f64) -> f64 {
    4.0/(1.0+x*x)
}

fn main() {
    let mut sum = 0.0;

    for i in 0..N {
        let x = W*((i as f64) + 0.5);
        sum = sum + f(x);
    }

    println!("pi = {}", W*sum);
}
```

lecture02: calculate π

stateful ☞ bad

functional ☞ good

```
const N: usize = 1_000_000_000;
const W: f64 = 1f64/(N as f64);

fn f(x: f64) -> f64 {
    4.0/(1.0+x*x)
}

fn main() {

    let mut sum = 0.0;
    for i in 0..N {
        let x = W*((i as f64) + 0.5);
        sum = sum + f(x);
    }

    println!("pi = {}", W*sum);
}
```

```
const N: usize = 1_000_000_000;
const W: f64 = 1f64/(N as f64);

fn f(x: f64) -> f64 {
    4.0/(1.0+x*x)
}

fn main() {

    let sum : f64 = (0..N)
        .into_iter()
        .map(|i| f(W*((i as f64)+0.5)))
        .sum::<f64>();

    println!("pi = {}", W*sum);
}
```

lecture02: calculate π

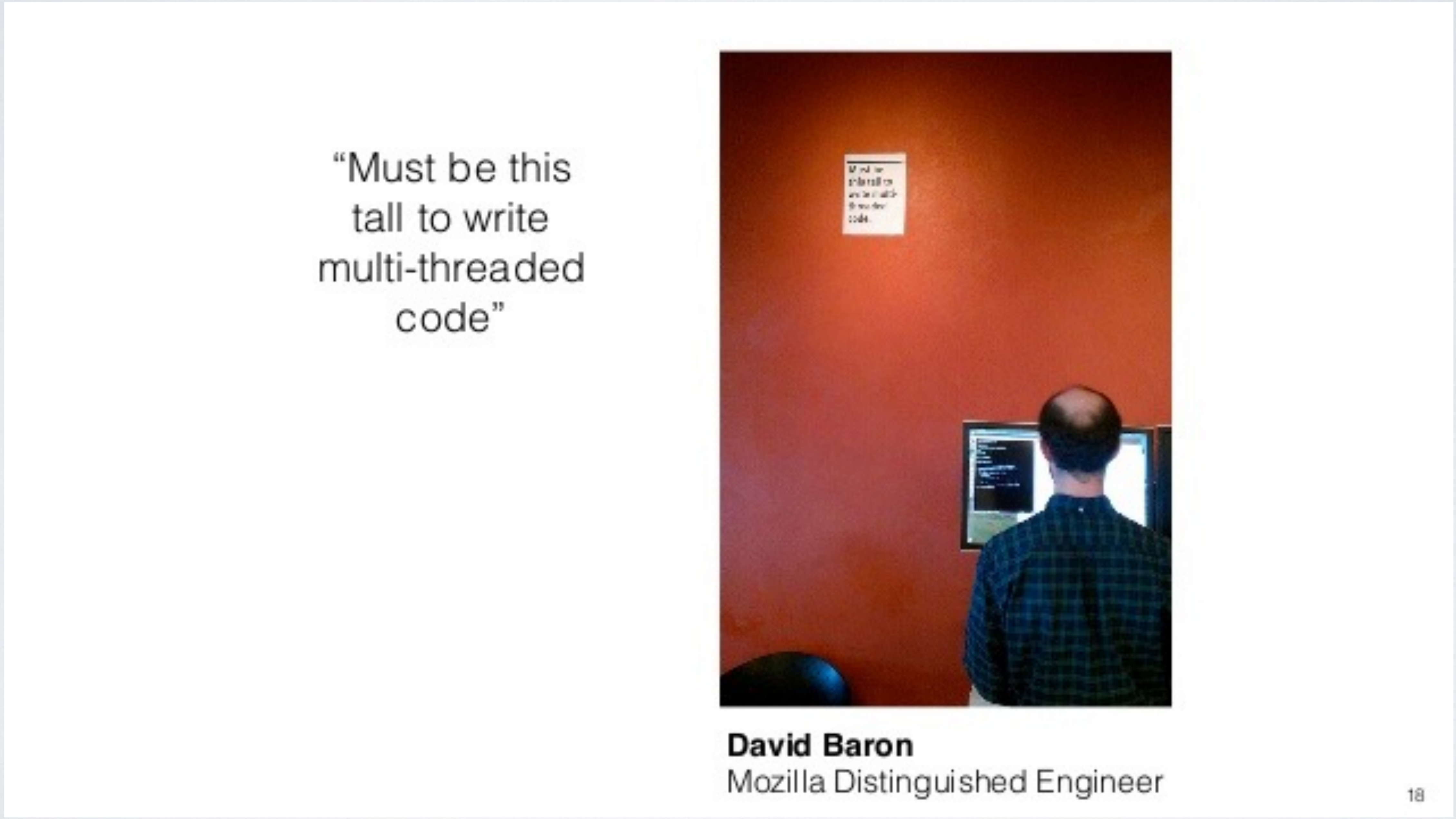
functional program ☞

multithreading is almost trivial

```rust
const N: usize = 1_000_000_000;
const W: f64 = 1f64/(N as f64);

fn f(x: f64) -> f64 {
  4.0/(1.0+x*x)
}

fn main() {

  let sum : f64 = (0..N)
    .into_iter()
    .map(|i| f(W*((i as f64)+0.5)))
    .sum::<f64>();

  println!("pi = {}", W*sum);
}
```

```rust
! extern crate rayon;

  const N: usize = 1_000_000_000;
  const W: f64 = 1f64/(N as f64);

  fn f(x: f64) -> f64 {
    4.0/(1.0+x*x)
  }

  fn main() {
!   use rayon::prelude::*;
    let sum : f64 = (0..N)
!     .into_par_iter()
      .map(|i| f(W*((i as f64)+0.5)))
      .sum::<f64>();

    println!("pi = {}", W*sum);
  }
```

writing safe concurrent code is, at present, rocket science - or is it ???



"Must be this tall to write multi-threaded code"

David Baron
Mozilla Distinguished Engineer

18

https://bholley.net/blog/

Rust (programming language)

From Wikipedia, the free encyclopedia

Rust is a **systems programming language**

- with a focus on safety, especially safe **concurrency**,
- supporting both **functional** and **imperative** paradigms.

Rust is **syntactically** similar to C++,

- but its designers intend it to provide better **memory safety** while still maintaining **performance**.

Rust is on its seventh year as the "**most loved** programming language" in the **Stack Overflow** Developer Survey in 2016 - 2022.

Rust (programming language)

From Wikipedia, the free encyclopedia

- Rust was originally designed by Graydon Hoare at Mozilla Research (~2010), with contributions from Dave Herman, Brendan Eich, and many others.
- Version 1.0 stable in May 2015
- Its designers have refined the language through the experiences of writing the Servo web browser layout engine and the Rust compiler.
- The compiler is free and open-source software, dual-licensed under the MIT License and Apache License 2.0.

Rust's Buzzwords

Rust can be relevant to nuclear physicists for several reasons:

1. **Performance**: Rust is designed for high performance, which is crucial in nuclear physics simulations and data processing tasks. The language's low-level control, zero-cost abstractions, and efficient memory management allow it to achieve performance comparable to C and C++.

2. **Memory safety**: Nuclear physics often involves complex simulations with numerous calculations and data manipulations. Rust's ownership system and borrowing mechanism ensure memory safety without sacrificing performance, reducing the likelihood of memory-related bugs and crashes.

Rust's Buzzwords

3. **Parallel and concurrent programming**: Nuclear physicists frequently deal with large datasets and computationally intensive tasks, which benefit from parallel and concurrent processing. Rust's safe concurrency model, along with libraries like Rayon, make it easier to write parallel code without introducing data races or other concurrency-related bugs.

4. **Functional programming**: Rust supports functional programming, which can help nuclear physicists write more concise, composable, and easily parallelizable code. The use of higher-order functions, pattern matching, and iterators can lead to more maintainable and efficient code for complex nuclear physics simulations.

Rust's Buzzwords

5. **Interoperability**: Rust can interface with other languages, such as Python or C/C++, allowing nuclear physicists to leverage existing code or libraries. This makes it easier to integrate Rust into existing nuclear physics projects, benefiting from its safety and performance features while retaining compatibility with widely-used scientific tools and libraries.

6. **Growing ecosystem**: Rust has a growing ecosystem of scientific computing libraries, which can be beneficial for nuclear physicists. Additionally, the Rust community is active and supportive, providing valuable resources for learning and troubleshooting.

In summary, Rust's performance, memory safety, support for parallel programming, functional programming features, interoperability, and growing ecosystem make it a relevant and attractive choice for nuclear physicists working on complex simulations and data processing tasks.

Aside: Safety & Garbage Collection

Memory must be reused,
but there are different strategies:

- **C: “Just follow these rules perfectly, you’re smart”**
- **Java, JS, etc: “Wait a minute, I’ll take care of it”**
- **Rust: “I’ll prove correctness at compile time”**

What Rust has to offer

- Strong safety guarantees...
  No seg-faults, no data-races,
  expressive type system.
- ...without compromising on performance.
  No garbage collector, no runtime.
- Goal:
  Confident, productive systems programming

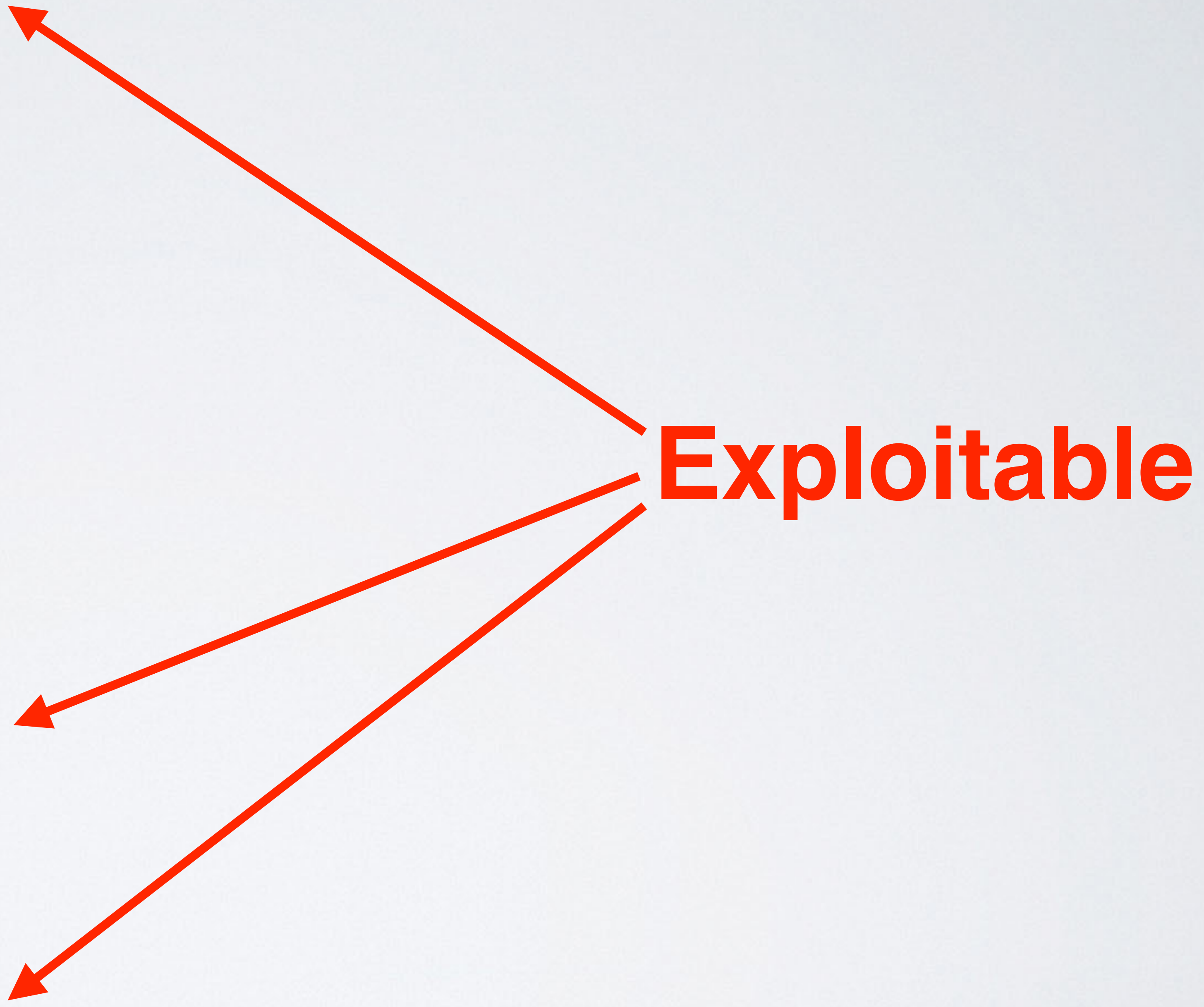
What’s concurrency?

In computer science, concurrency is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other.

```
// What does this print?
int main() {
  int pid = fork();
  printf("%d\n", pid);
}
```

Concurrency is hard!

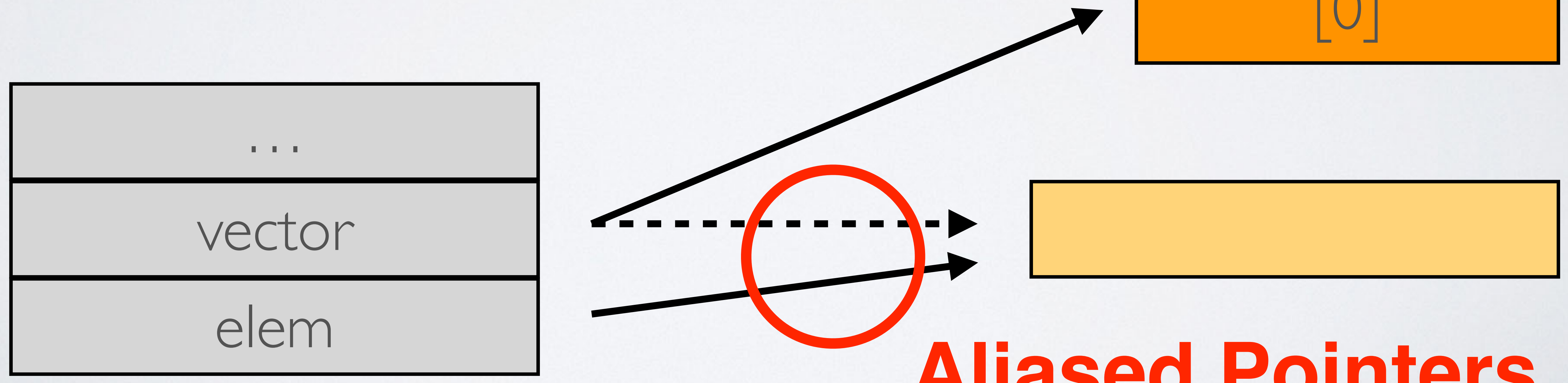
- Data Races
- Race Conditions
- Deadlocks
- Use after free
- Double free



What's safety?

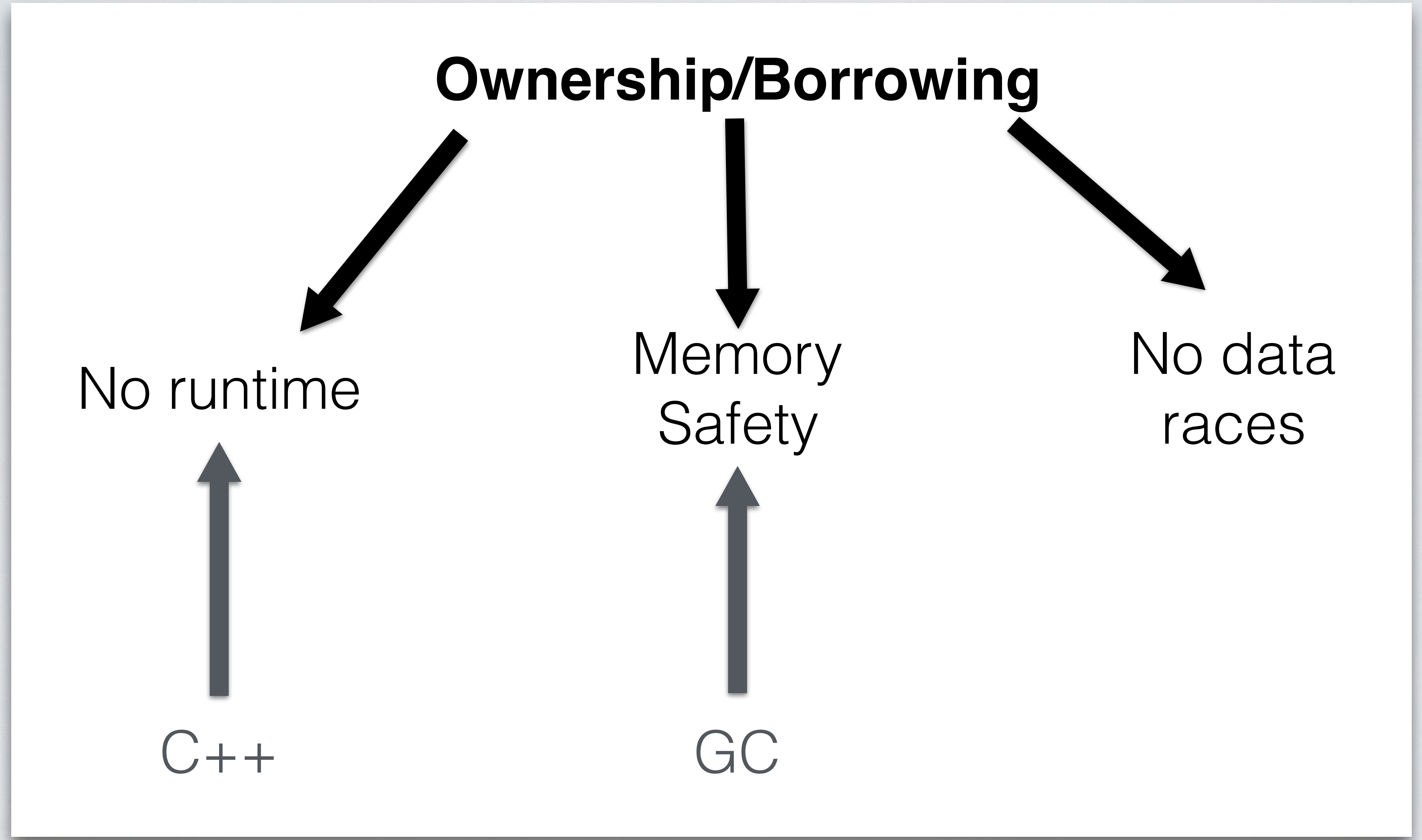
```
void example() {
   vector<string> vector;
   // ...
   auto& elem = vector[0];
   vector.push_back(some_string);
   cout << elem;
}
```

Mutation



Aliased Pointers

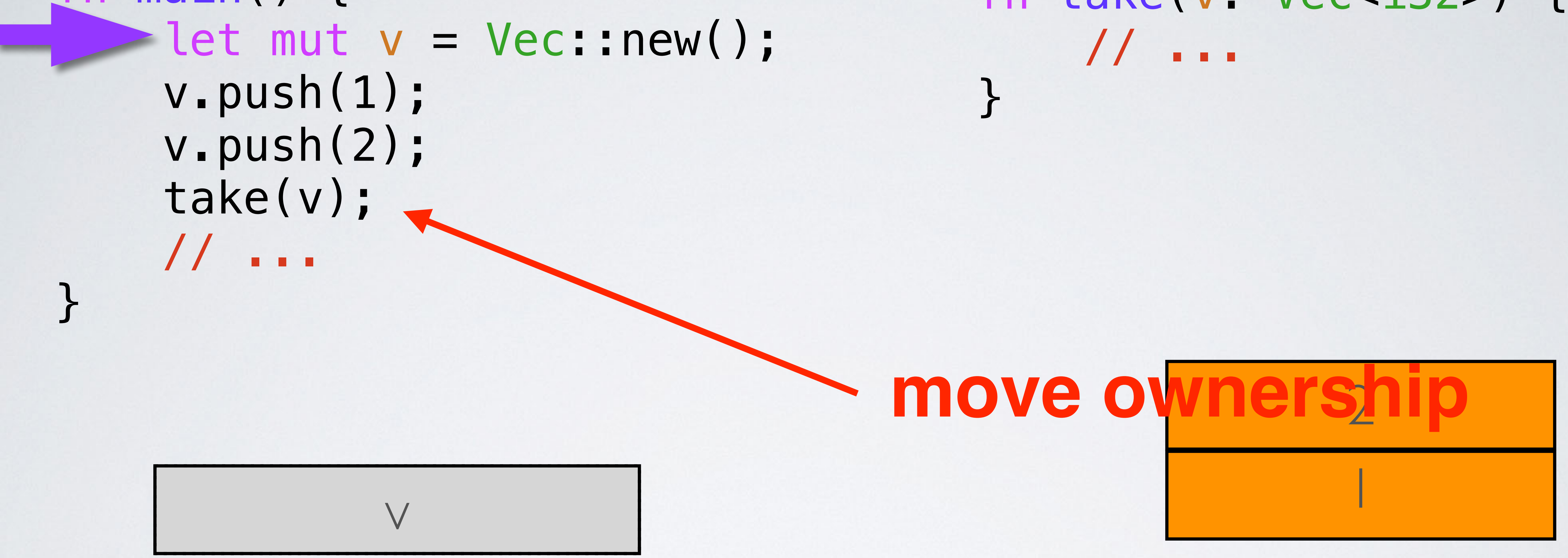
Rust's Solution



Ownership

```
fn main() {
    let mut v = Vec::new();
    v.push(1);
    v.push(2);
    take(v);
    // ...
}
```

```
fn take(v: Vec<i32>) {
    // ...
}
```



Ownership

```
fn main() {
    let mut v = Vec::new();
    v.push(1);
    v.push(2);
    take(v);
    ~~v.push(3);~~
}
```

```
fn take(v: Vec<i32>) {
    // ...
}
```

error: use of moved variable v

Borrowing

```
fn main() {
    let mut v = Vec::new();
    push(&mut v);
    read(&v);
    // ...
}
```

```
fn push(v: &mut Vec<i32>)
{
    v.push(1);
}

fn read(v: Vec<i32>) {
    // …
}
```

Safety in Rust

- Rust statically prevents aliasing + mutation
- Ownership prevents double-free
- Borrowing prevents use-after-free
- Overall, no segfaults!

Rust's core concepts: Lifetimes

- In Rust, lifetimes are a way to express the scope in which references to data are valid. They help the Rust compiler ensure that references do not outlive the data they point to, which prevents dangling references and ensures memory safety. Lifetimes are denoted using a single quote and a descriptive name, such as **'a**.

- It's important to note that lifetimes are a compile-time concept, and they don't have any runtime overhead. The Rust compiler uses lifetimes to perform static analysis and verify the safety of the code.

Rust's core concepts: Lifetimes

```
fn longest<'a>(str1: &'a str, str2: &'a str) -> &'a str {
    if str1.len() > str2.len() {
        str1
    } else {
        str2
    }
}
```

- In this example, the **longest** function takes two string references as arguments and returns a reference to the longest string. The lifetime annotation **'a** is used to express that the input references (**str1** and **str2**) and the output reference all share the same lifetime.

- This means that the returned reference will be valid as long as the shortest of the input references is valid. The Rust compiler uses this information to ensure that the returned reference does not outlive the data it points to, preventing memory safety issues.

Rust's core concepts: enums and pattern matching

- In Rust, **enums** (short for enumerations) are a way to define a custom data type that can have one of several possible variants. Each variant can have associated data, making enums a versatile and expressive tool for modeling complex data structures. Enums in Rust are similar to algebraic data types in functional programming languages.

- **Pattern matching** is a powerful language construct in Rust that allows you to destructure and inspect data structures, like enums, in a concise and expressive way. It is typically used with the **match** expression or **if let** statement to handle different cases of enums or other complex data structures.

Rust's core concepts: enums and pattern matching

```rust
enum Color {
    Red,
    Green,
    Blue,
    Custom { r: u8, g: u8, b: u8 },
}

fn describe_color(color: Color) {
    match color {
        Color::Red => println!("This is red"),
        Color::Green => println!("This is green"),
        Color::Blue => println!("This is blue"),
        Color::Custom { r, g, b } => println!("Custom color: R={}, G={}, B={}", r, g, b),
    }
}

fn main() {
    let red = Color::Red;
    let custom_color = Color::Custom { r: 128, g: 64, b: 255 };

    describe_color(red);
    describe_color(custom_color);
}
```

Enums and pattern matching in Rust allow you to express complex data types and handle different cases in a type-safe, clear, and concise manner. They are particularly useful when dealing with multiple possible states or error conditions.

Rust's core concepts: `Result` and `Option` types

The `Result` and `Option` types are part of the Rust core language, and they are defined in the standard library (`std`). They are essential building blocks for handling error cases and optional values in a type-safe and idiomatic way.

- The `Option` type is an enumeration that represents an optional value. It has two variants: `Some(T)` and `None`. The `Some(T)` variant represents the presence of a value of type `T`, while the `None` variant represents the absence of a value. The `Option` type is commonly used in Rust to express the possibility of a value being missing or uninitialized, and it helps prevent null pointer-related bugs.

```
fn find_even_number(numbers: &[i32]) -> Option<i32> {
    for &number in numbers {
        if number % 2 == 0 {
            return Some(number);
        }
    }
    None
}
```

Rust's core concepts: `Result` and `Option` types

- The **Result** type is an enumeration used to represent the outcome of a computation that might fail. It has two variants: **Ok(T)** and **Err(E)**. The **Ok(T)** variant represents a successful computation with a resulting value of type **T**, while the **Err(E)** variant represents a computation that failed with an error of type E. The Result type encourages Rust developers to handle errors explicitly, leading to more robust and maintainable code.

```rust
use std::fs::File;
use std::io::Read;

fn read_file_contents(file_path: &str) -> Result<String, std::io::Error> {
    let mut file = File::open(file_path)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents)
}
```

Rust’s core concepts: immutability

Immutability is a crucial aspect of Rust's programming philosophy and plays a vital role in ensuring memory safety, data consistency, and promoting more predictable code.

- **Memory safety**: By default, Rust enforces immutability, which means that once a variable is initialized, its value cannot be changed. This helps prevent data races and other concurrency-related bugs, especially when working with shared or borrowed data.
- **Easier reasoning about code**: Immutability simplifies reasoning about code as it eliminates side effects caused by shared mutable state. When a variable is immutable, you can be sure that its value will not change unexpectedly, making it easier to understand the flow of data and logic in your program.

Rust’s core concepts: immutability

- **Functional programming**: Immutability is a core concept of functional programming, which Rust supports through its type system and syntax. By embracing immutability, Rust encourages writing code that is more composable, reusable, and easier to test.

- **Optimizations**: Immutability can lead to compiler optimizations, such as better utilization of CPU caches or dead code elimination. The Rust compiler can make certain assumptions about the behavior of the code when variables are immutable, leading to improved performance.

- **Encouraging good programming practices**: Immutability promotes the use of more explicit patterns for managing state and updating data. For example, when using immutable data structures, you need to create a new instance with the updated data rather than modifying the existing instance in place. This approach can prevent bugs caused by unintentional side effects or shared mutable state.

In summary, **immutability** is important in Rust because it helps ensure memory safety, simplifies code reasoning, supports functional programming principles, enables compiler optimizations, and encourages good programming practices. By embracing **immutability**, Rust developers can create more robust, maintainable, and efficient programs.

Smart pointer

… are data structures that not only act like a pointer but also have additional metadata and capabilities.
Examples:

- Vec<T>
- Box<T> for allocating values on the heap
- Rc<T>, a reference counting type that enables multiple ownership

Iterators and Closures

Functional Language Features:

- **Closures, a function-like construct you can store in a variable**
- **Iterators, a way of processing a series of elements**

I did not talk about …

- Macros
- Testing
- Generic Types, Traits
- Cargo and crates.io
- Futures
- Unsafe or advanced Rust
- …

Further reading and viewing

- **The Rust Programming Language** https://doc.rust-lang.org/stable/book/
- Vorlesung „Programmieren in Rust“, Universität Osnabrück, Wintersemester 2016/17. https://github.com/LukasKalbertodt/programmieren-in-rust
- https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/
- https://youtu.be/eclWPzGEbFc
- https://youtu.be/6f5dt923FmQ

Installing Rust

- rustup: the Rust toolchain installer
  https://www.rust-lang.org/tools/install

```
# curl https://sh.rustup.rs \
    --silent --output rustup-init.sh
# sh rustup-init.sh
```

Why I am learning a new programming language - and why you should too!
- part 2 -

A brief introduction to the Rust programming language

by Dr. Michael O. Distler <distler@uni-mainz.de>

Mainz, 26 April 2023

Workshop “Tools for Physicists” organized by Dr. Bernd-Peter Otte

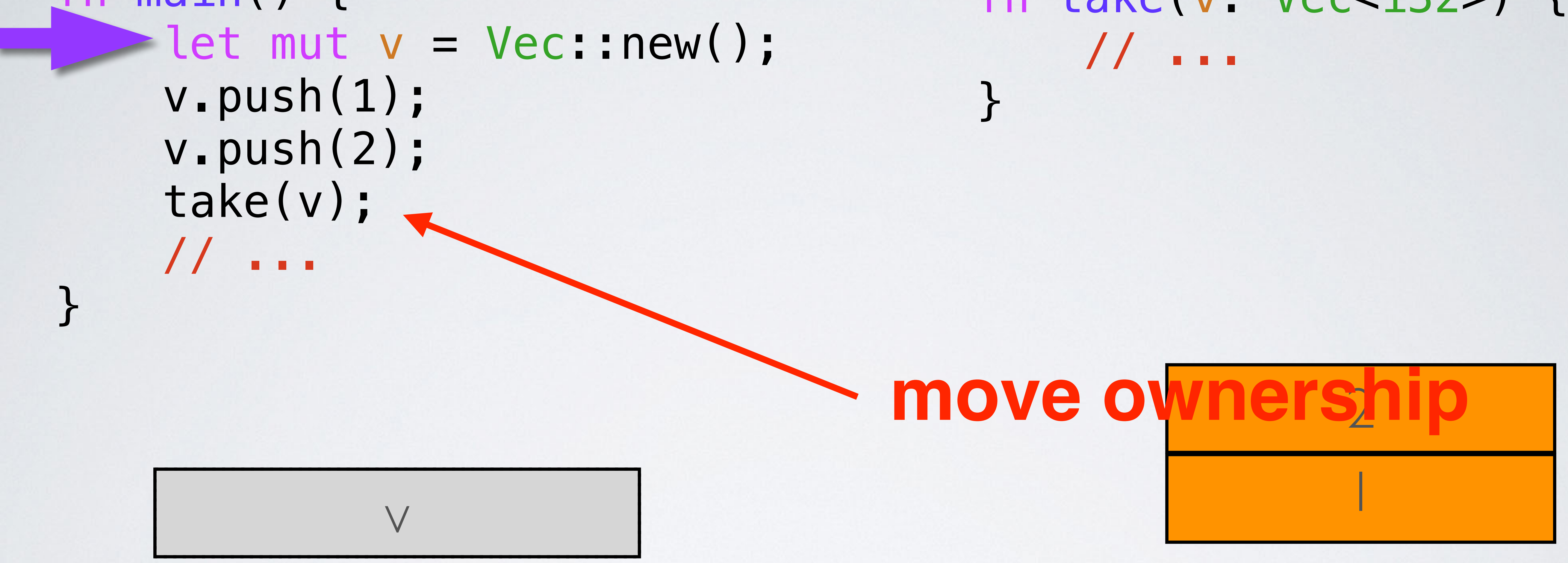
Content

- Ownership and borrowing.
- Traits: Send and Sync.
- Smart pointers: Arc<T> and Mutex<T>.
- Asynchronous communication between threads: mpsc::channel.
- Examples: ping, ring, (dining philosophers problem)

Ownership

```rust
fn main() {
    let mut v = Vec::new();
    v.push(1);
    v.push(2);
    take(v);
    // ...
}
```

```rust
fn take(v: Vec<i32>) {
    // ...
}
```



Ownership

```
fn main() {
    let mut v = Vec::new();
    v.push(1);
    v.push(2);
    take(v);
    ~~v.push(3);~~
}
```

```
fn take(v: Vec<i32>) {
    // ...
}
```

error: use of moved variable v

Borrowing

```
fn main() {
    let mut v = Vec::new();
    push(&mut v);
    read(&v);
    // ...
}
```

```
fn push(v: &mut Vec<i32>)
{
    v.push(1);
}

fn read(v: Vec<i32>) {
    // …
}
```

Traits: Copy and Clone

Traits abstract over behavior that types can have in common.

Examples: Copy and Clone

- Copies happen implicitly, for example as part of an assignment y = x. The behavior of Copy is not overloadable; it is always a simple bit-wise copy.

Traits: Copy and Clone

Traits abstract over behavior that types can have in common.

Examples: Copy and Clone

- Cloning is an explicit action, x.clone(). The implementation of Clone can provide any type-specific behavior necessary to duplicate values safely. For example, the implementation of Clone for String needs to copy the pointed-to string buffer in the heap. A simple bitwise copy of String values would merely copy the pointer, leading to a double free down the line. For this reason, String is Clone but not Copy.

Traits: Send and Sync

Send and Sync are fundamental to Rust's concurrency story.

- A type is Send if it is safe to send it to another thread.
- A type is Sync if it is safe to share between threads (&T is Send).

Smart pointer

… are data structures that not only act like a pointer but also have additional metadata and capabilities.
Examples:

- Vec<T>
- Box<T> for allocating values on the heap
- Rc<T>, a reference counting type that enables multiple ownership

Smart pointer: Arc<T>

- Arc<T>: A thread-safe reference-counting pointer. 'Arc' stands for 'Atomically Reference Counted'.
- The type Arc<T> provides shared ownership of a value of type T, allocated in the heap. Invoking clone on Arc produces a new Arc instance, which points to the same value on the heap as the source Arc, while increasing a reference count. When the last Arc pointer to a given value is destroyed, the pointed-to value is also destroyed.
- Shared references in Rust disallow mutation by default, and Arc is no exception: you cannot generally obtain a mutable reference to something inside an Arc.

Smart pointer: Arc<T>

```
// lecture13/src/bin/arc.rs
// cd lecture13; cargo run --bin arc
use std::sync::Arc;
use std::thread;
fn main() {
    let five = Arc::new(5);
    for _ in 0..10 {
        let five = Arc::clone(&five);
        thread::spawn(move || {
            println!("{:?}", five);
        });
    }
}
```

Change the code, so each thread prints it's ID.

Smart pointer: Mutex<T>

- **Mutex<T>: A mutual exclusion primitive useful for protecting shared data**

- **This mutex will block threads waiting for the lock to become available. The mutex can also be statically initialized or created via a new constructor. Each mutex has a type parameter which represents the data that it is protecting. The data can only be accessed through the RAII guards returned from lock and try_lock, which guarantees that the data is only ever accessed when the mutex is locked.**

RAII: Resource acquisition is initialization

Communication between threads

```
pub fn channel<T>() -> (Sender<T>,
Receiver<T>)
```

- Creates a new asynchronous channel, returning the sender/receiver halves. All data sent on the Sender will become available on the Receiver in the same order as it was sent, and no send will block the calling thread (this channel has an "infinite buffer", unlike sync_channel, which will block after its buffer limit is reached). recv will block until a message is available.

Communication between threads

```
pub fn channel<T>() -> (Sender<T>,
Receiver<T>)
```

- The Sender can be cloned to send to the same channel multiple times, but only one Receiver is supported.
- If the Receiver is disconnected while trying to send with the Sender, the send method will return a SendError. Similarly, if the Sender is disconnected while trying to recv, the recv method will return a RecvError.

Communication between threads

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

let (sender, receiver) = channel();

// Spawn off an expensive computation
thread::spawn(move|| {
    sender.send(expensive_computation()).unwrap();
});

// Do some useful work for awhile

// Let's see what that answer was
println!("{:?}", receiver.recv().unwrap());
```

Smart pointer: Mutex<T>

```rust
use std::sync::{Arc, Mutex};
use std::thread;
use std::sync::mpsc::channel;

const N: usize = 10;
let data = Arc::new(Mutex::new(0));
let (tx, rx) = channel();
for _ in 0..N {
    let (data, tx) = (Arc::clone(&data), tx.clone());
    thread::spawn(move || {
        let mut data = data.lock().unwrap();
        *data += 1;
        if *data == N {
            tx.send(()).unwrap();
        }
    });
}

rx.recv().unwrap();
```

Change the code, so the value of data is printed

Exercise 3

1. Try to understand ‘ping.rs’
2. Run the program:

   ```
   cargo run --bin ping -- --cycles 100000
   ```
3. Change the transmitted data size.
   Does the transmit time change?
   Why (not)?

Exercise 4

1. Try to understand ‘ring.rs’
2. Run the program:

   ```
   time target/debug/ring --threads 16
   ```
3. Try to understand ‘MPIring.c’
4. Run the program:

   ```
   time mpirun target/debug/MPIring
   ```
5. Do you notice a difference?
6. Double the number of threads in both cases