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

An introduction to
“Concurrent programming in Rust”

by Dr. Michael O. Distler

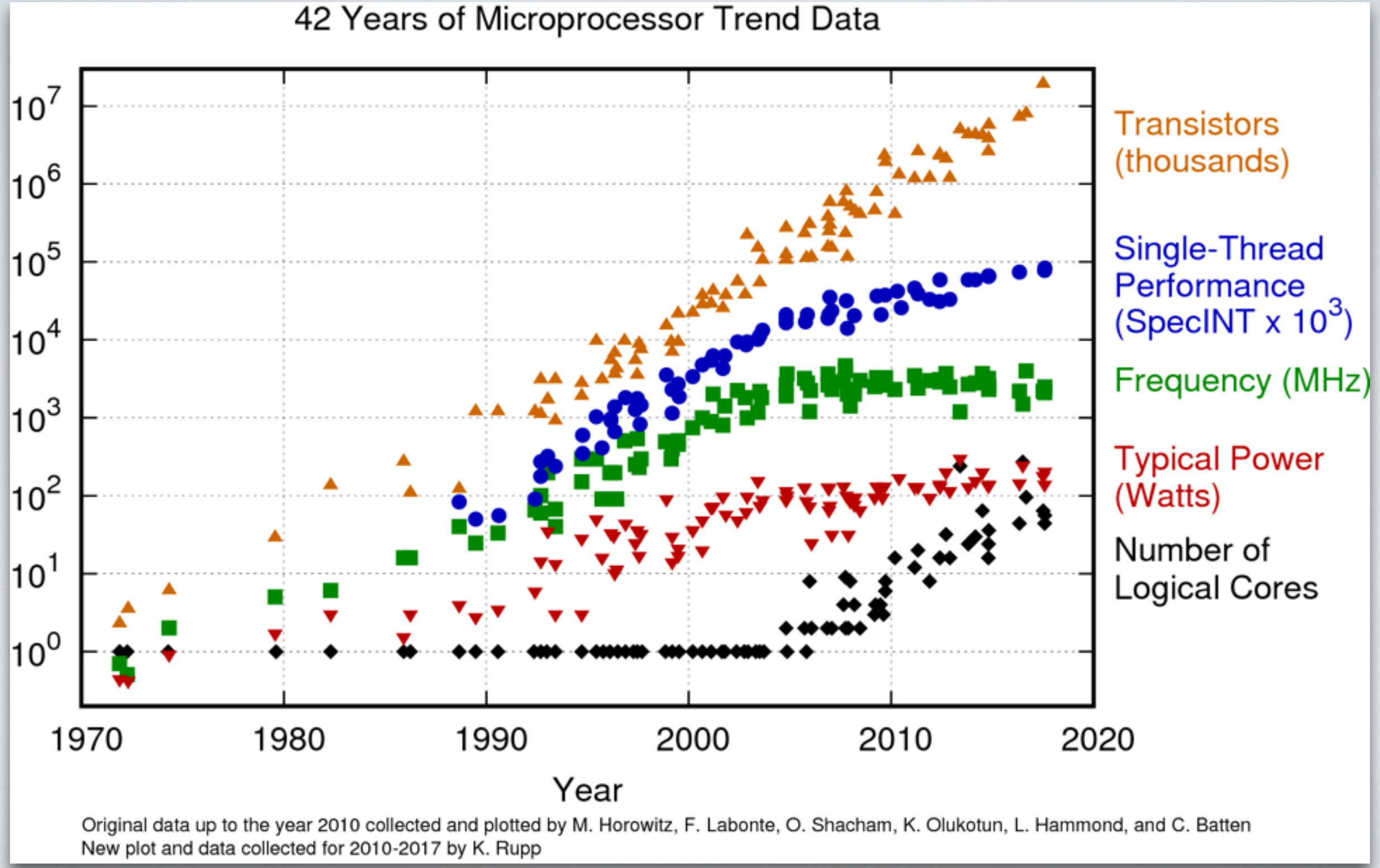
Mainz, 8 January 2019

special lecture as part of “Introduction to HPC Programming”
(Dr. Bernd-Peter Otte)

Content

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

42 Years of Microprocessor Trend Data



https://www.karlrupp.net/blog/

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

- Funtional 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)**

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?

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

```
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);
}
```

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 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

```
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);
}
```

!

```
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 won first place for "most loved programming language" in the Stack Overflow Developer Survey in 2016, 2017, and 2018.

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

- Safety, Speed, Concurrency
- Memory safety without garbage collection
- Zero-cost abstractions
- Hack Without Fear

Aside: Safety & GC

- Memory must be reused
- 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

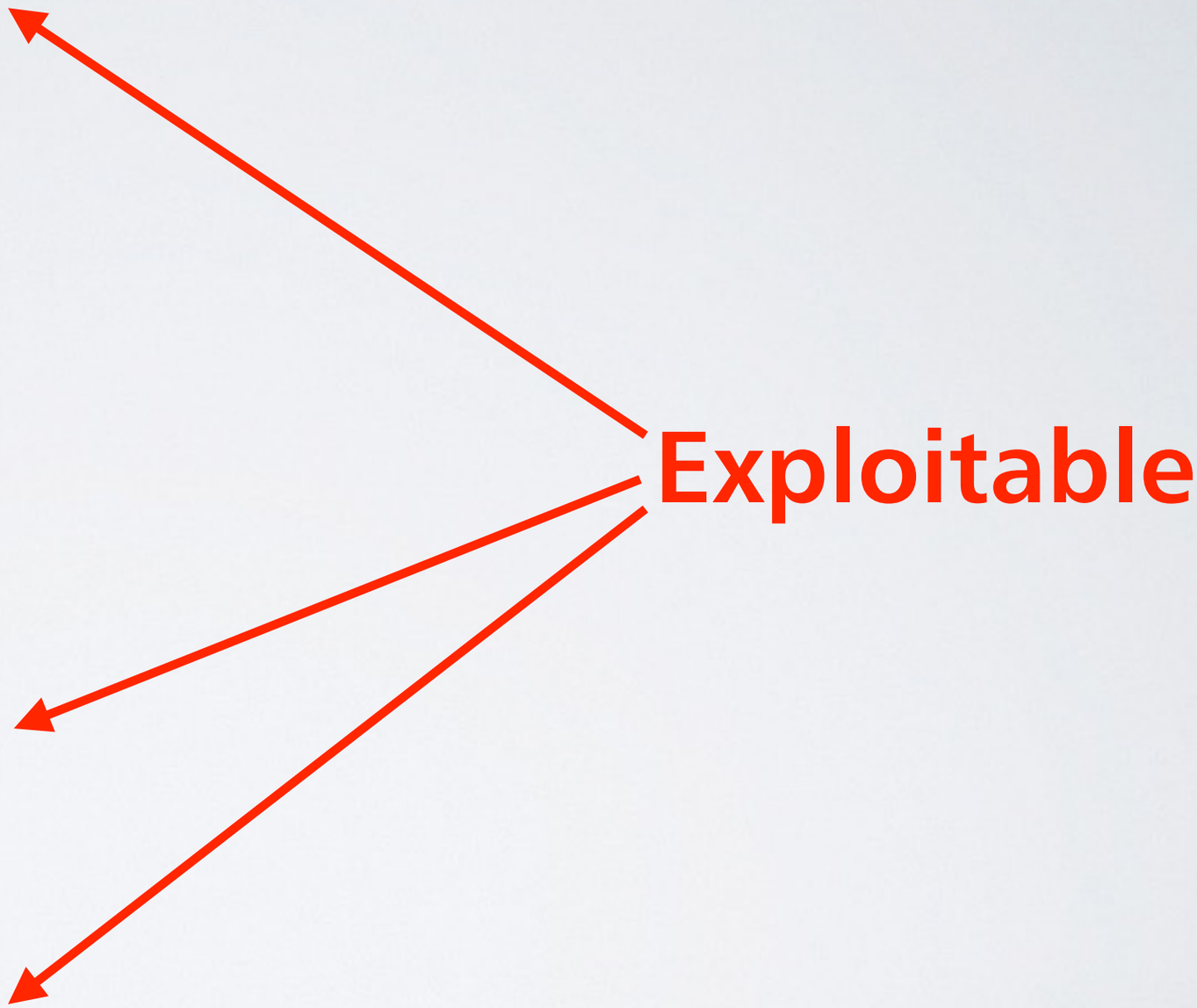
What's concurrency?

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

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

Concurrency is hard!

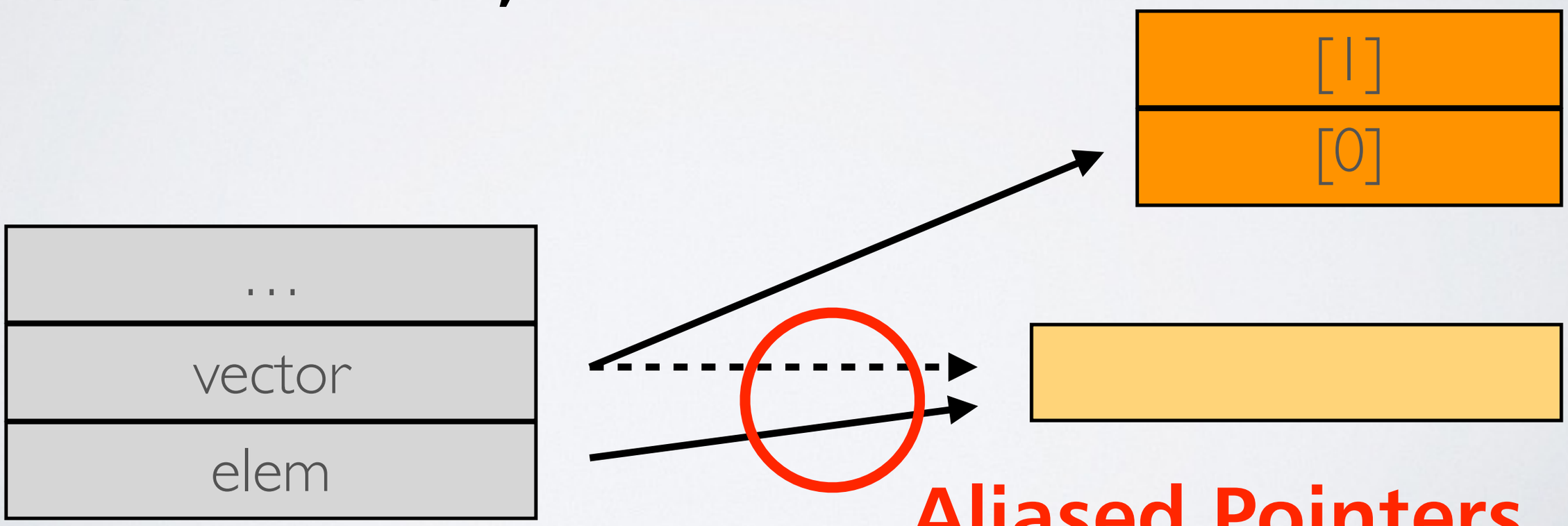
- Data Races
- Race Conditions
- Deadlocks
- Use after free
- Double free



What's safety?

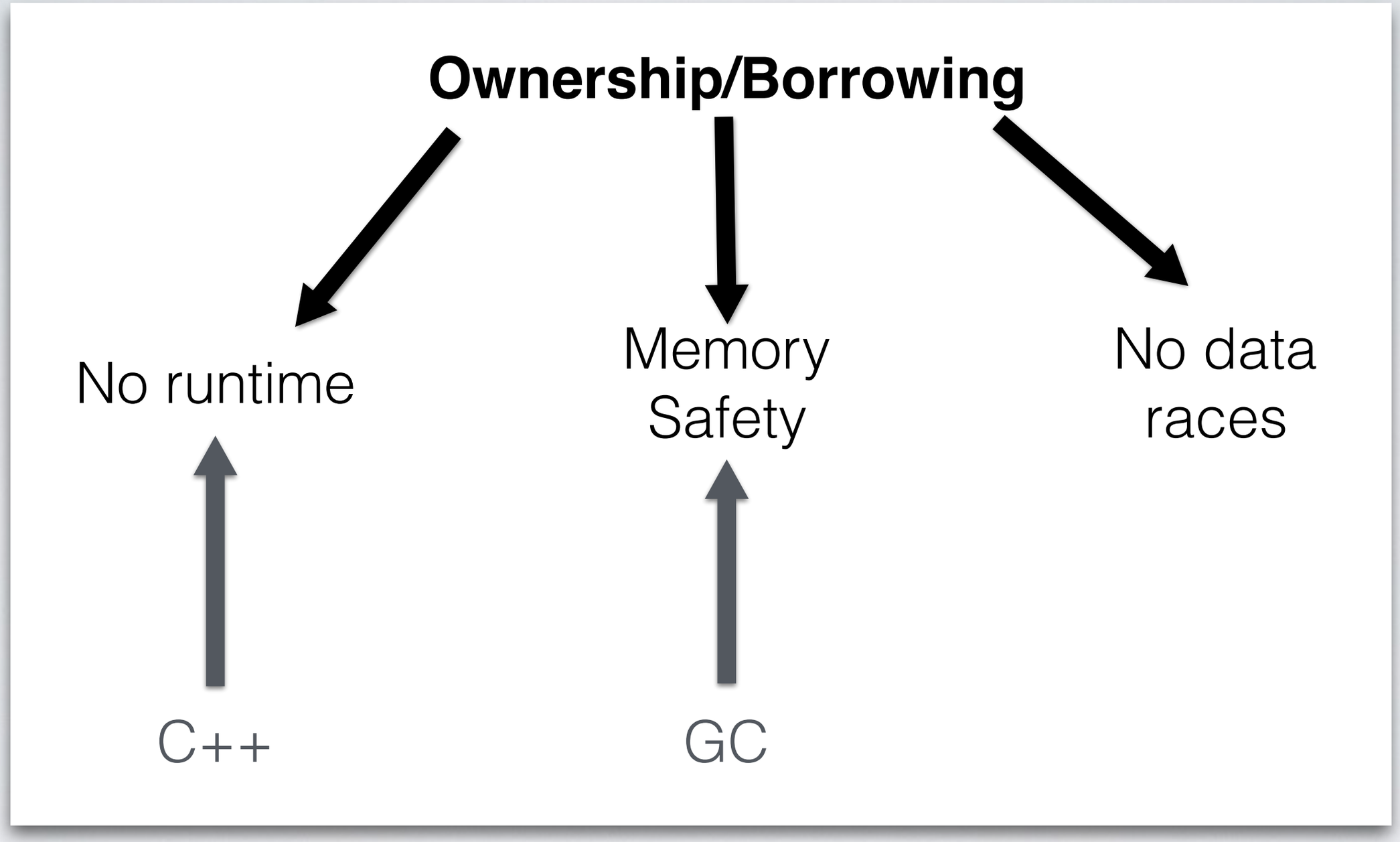
```
void example() {
  vector<string> vector;
  // ...
  auto& elem = vector[0];
  vector.push_back(some_string);
  cout << elem;
}
```

Mutation



Aliased Pointers

Rust's Solution



Ownership

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

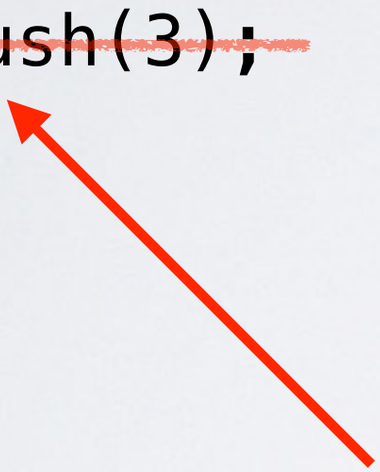
```
fn take(v: Vec<i32>) {
    // ...
}
```

move ownership

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!

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 …

- Testing
- Error Handling
- Generic Types, Traits, Lifetimes
- 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/2015/06/40-years-of-microprocessor-trend-data/
- https://youtu.be/ecIWPzGEbFc
- https://youtu.be/6f5dt923FmQ

Installing Rust

- rustup: the Rust toolchain installer
  https://github.com/rust-lang-nursery/
  rustup.rs

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