Rust语言：功能特性和趋势分析

Demystifying Rust: Features and Trends

Hui Xu

School of Computer Science

Fudan University

10/29/2023

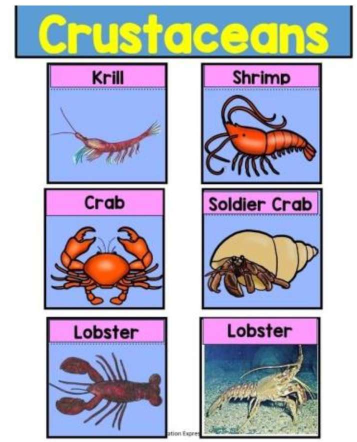
Outline

I. Overview
II. Security
III. Efficiency
IV. Usability
V. Summary

I. Overview of Rust

Origination of Rust

2006	2009	2011	2015	2020	2021
Graydon Hoare (broken elevator)	Supported by Mozilla	Self-hosting (OCaml->Rust)	First stable release	Layoff by Mozilla	Rust Foundation: AWS, Huawei, Google, Microsoft, Mozilla...

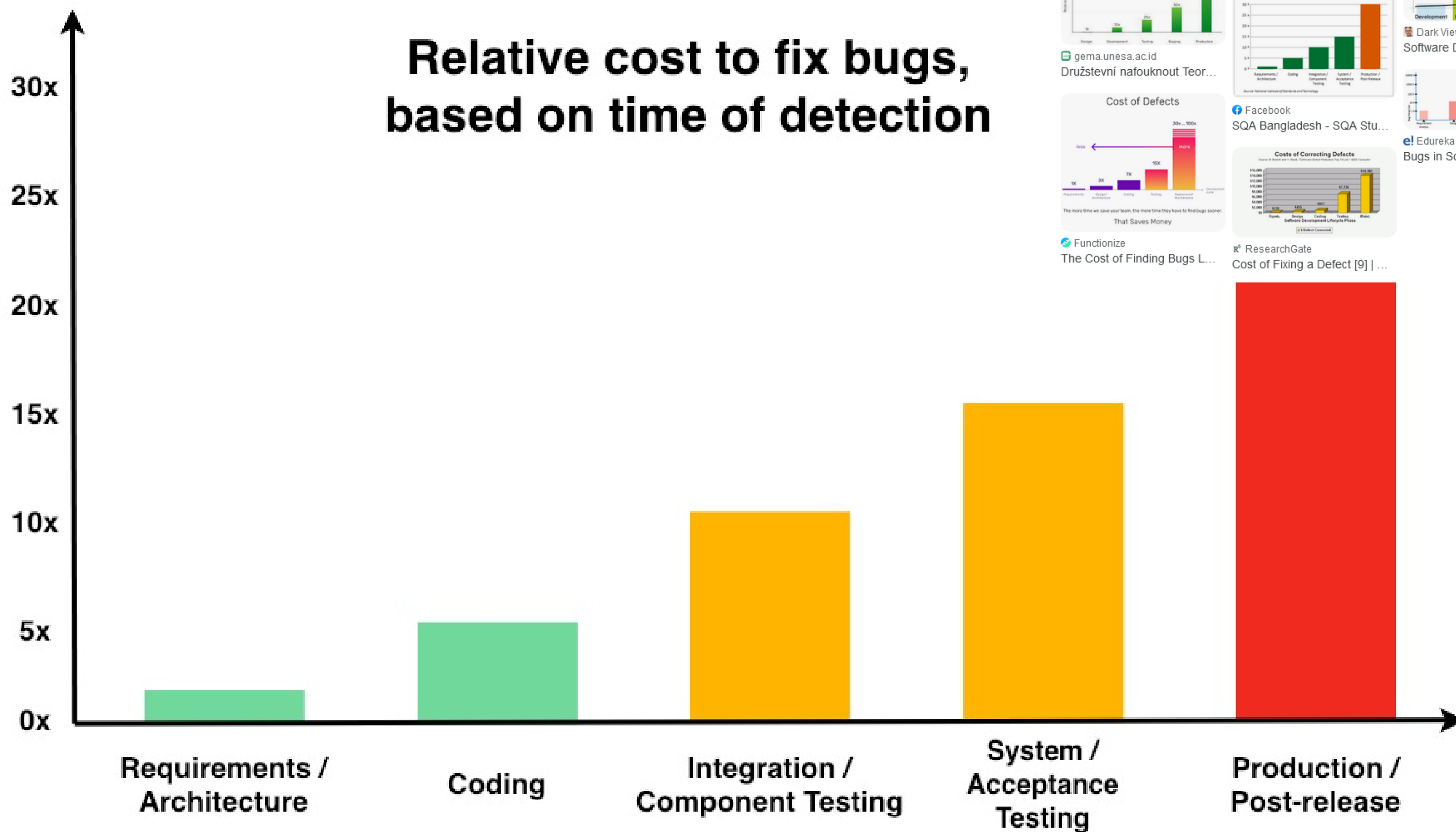


Rust

Oxidize

How Rust Went from a Side Project to the World's Most-Loved Programming Language, *MIT Technology Review*, 2023

A Famous Figure...



https://dzone.com/articles/the-exponential-cost-of-fixing-bugs

Key Design Goals of Rust, But...

- Security: shift the bug detection phase to compile time
  - Memory safety
  - Concurrency safety
  - No undefined behaviors
- Efficiency: zero-cost abstraction, no garbage collection

Efficiency

Security ***VS***

Usability

Rust has built a Prosperous Ecosystem

Apps

Operating System

Tock
Redox
rCORE
Theseus

Database

TiKV
SurrealDB
influxdb
IndraDB

Other Apps

Servo
Leptos
burn
SOLANA
deno
BEVY

3rd-party Lib

Concurrency

tokio
rayon

Web-related

Actix
Rocket
hyper
DIESEL

Other Libs

serde
rkyv
PyO3
thiserror

Official Rust

Compiler

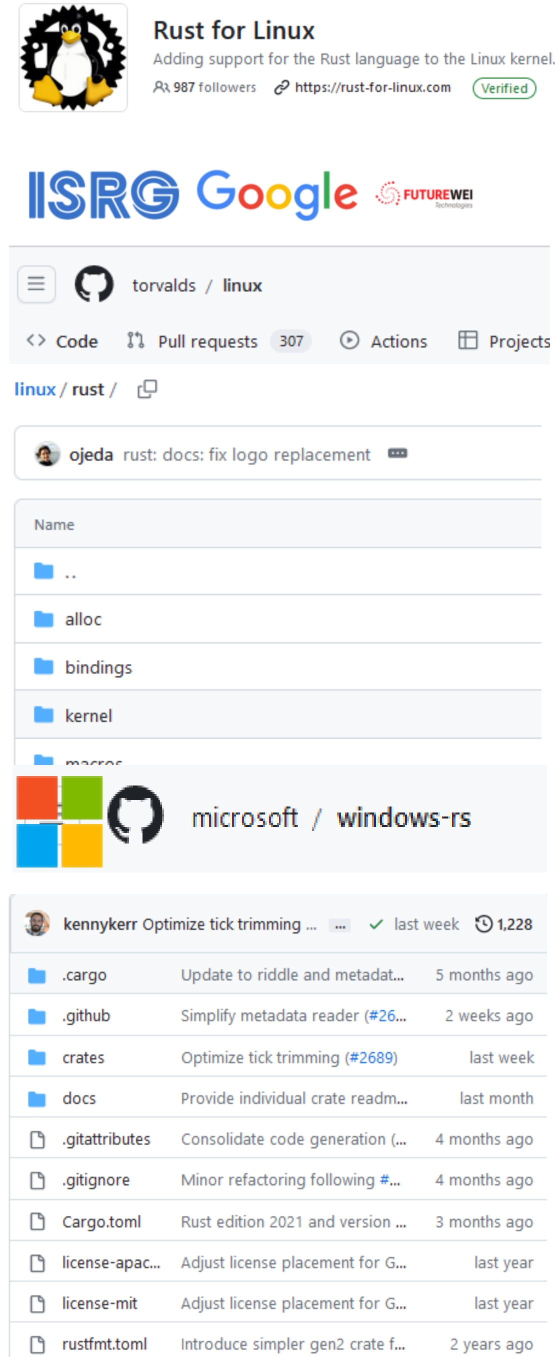
Std-lib

Cargo

IDE

Adopted by Linux and Windows

```
From: ojeda@kernel.org
To: Linus Torvalds <torvalds@linux-foundation.org>,
        Greg Kroah-Hartman <gregkh@linuxfoundation.org>
Cc: rust-for-linux@vger.kernel.org, linux-kbuild@vger.kernel.org,
        linux-doc@vger.kernel.org, linux-kernel@vger.kernel.org,
        Miguel Ojeda <ojeda@kernel.org>
Subject: [PATCH 00/13] [RFC] Rust support
Date: Wed, 14 Apr 2021 20:45:51 +0200   [thread overview]
Message-ID: <20210414184604.23473-1-ojeda@kernel.org> (raw)

From: Miguel Ojeda <ojeda@kernel.org>

Some of you have noticed the past few weeks and months that
a serious attempt to bring a second language to the kernel was
being forged. We are finally here, with an RFC that adds support
for Rust to the Linux kernel.

This cover letter is fairly long, since there are quite a few topics
to describe, but I hope it answers as many questions as possible
before the discussion starts.

If you are interested in following this effort, please join us
in the mailing list at:

    rust-for-linux@vger.kernel.org

and take a look at the project itself at:

    https://github.com/Rust-for-Linux

Cheers,
Miguel
```



https://lore.kernel.org/lkml/20210414184604.23473-1-ojeda@kernel.org/
https://github.com/Rust-for-Linux
https://github.com/microsoft/windows-rs

Why Scientists Are Turning to Rust? By Nature

High Performance

Ergonomic to Use

Easy Debugging

Johannes Köster, *Duisburg-Essen University*
Compare millions of sequence reads against billions of genetic bases to identify genomic variants

Heng Li, *Harvard Medical School*
Tested multiple languages on a biology task that involved parsing 5.7 million records. Rust edged out C to take the top spot.

Luiz Irber, *University of California, Davis*
Genomic searches and taxonomic profiling

Rob Patro, *University of Maryland*
Analyze transcript-level abundance estimates from RNA-seq data

https://www.nature.com/articles/d41586-020-03382-2

Scientific Computing in Rust 2023

Scientific Computing in Rust 2023

13-14 July 2023

Home | Timetable | List of talks | Organisers

Scientific Computing in Rust 2023

The Scientific Computing in Rust 2023 workshop took pla[c]
and 17:00 BST, with over 100 people joining the worksho[p]

This workshop was held virtually and was free to attend.
YouTube.

Timetable

The Scientific Computing in Rust 2023 workshop feature[d]
minute talks by workshop attendees. The full schedule for
talks page.

The talks were be complimented by informal discussion [s]
meet and talk to speakers and other attendees.

The majority of the talks were be recorded and are graduall[y]

Organisers

Timo Betcke

Timo is a professor of computational mathematics at University College London. He is working on developing rlst (Rust linear solver toolbox), a library for dense and sparse matrix routines; and bempp-rs, a Rust-based boundary element method library.

Jed Brown

Jed is a professor of computer science at CU Boulder. Jed maintains Rust crates for MPI, PETSc, and libCEED. He is interested in applying Rust and Enzyme in computational science and engineering, especially computational mechanics.

✉ jed.brown@colorado.edu

https://scientificcomputing.rs

II. Security of Rust

Memory safety

Concurrency safety

No undefined behaviors

Memory Safety

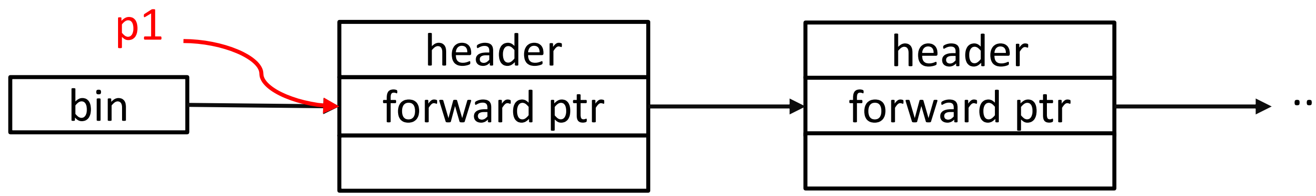
- A security notion stronger than type safety
- Types of memory-safety bugs:
  - Out-of-bound read
  - Out-of-bound write (stack smashing, heap overflow)
  - Dangling pointer (use-after-free, double free)
  - Concurrency issue
- Most dangerous software vulnerabilities (by MITRE, 2023)

Rank	ID	Name
1	CWE-787	Out-of-bounds Write
4	CWE-416	Use After Free
7	CWE-125	Out-of-bounds Read
12	CWE-476	NULL Pointer Dereference
21	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
36	CWE-401	Missing Release of Memory after Effective Lifetime

Source: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

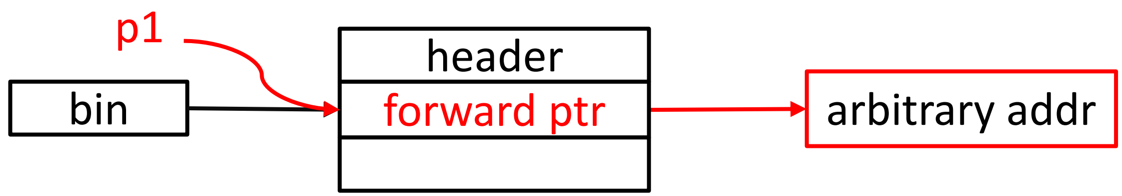
Why Heap Bugs are Dangerous? UAF as an Example

1. free(p1)



Add the block of *p1 to the free list

2. write(p1)

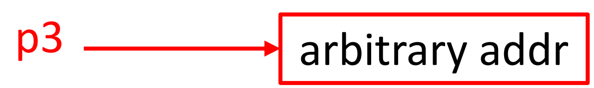


Attackers modify the forward pointer through p1

3. p2 = malloc()

Remove the block from the list; now bin points to the arbitrary address

4. p3 = malloc()

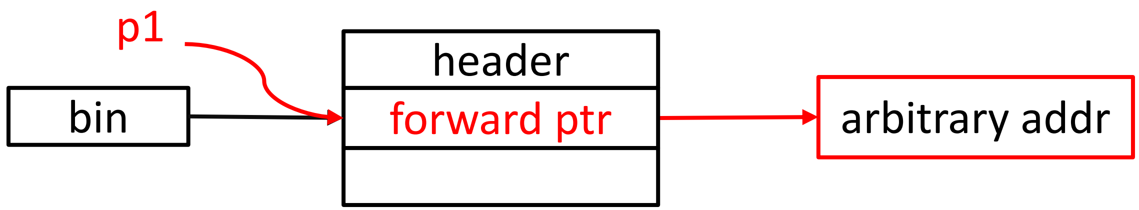


p3 points to the arbitrary address

Detecting UAF is Hard: via Allocator Design

- **Option 1: prevent writing the dangling pointer p1**

2. write(p1)



Problem: incur overhead during each pointer access

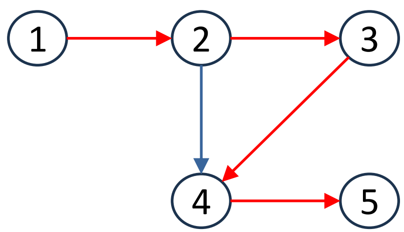
- **Option 2: prevent adding invalid blocks to the free list**

3. p2 = malloc()

How to design an efficient and robust mechanism?

Detecting UAF is Hard: via Static Analysis

- Alias analysis is NP-Hard
  - Hamiltonian path problems => Flow-insensitive may-alias analysis



Hamiltonian Path Problem

```
v4 = &v5
v2 = &v4
v3 = &v4
v2 = &v3
v1 = &v2
```

****v1 = v5 ?

May-Alias Analysis

  - More complicated alias analysis problems：
    - Flow-sensitive, path-sensitive, control-sensitive, context-sensitive…
    - Raw pointer, point-to
    - Concurrent code

[Horwitz'97] Susan Horwitz, Precise Flow-Insensitive May-Alias Analysis is NP-Hard, 1997

Rust Tackles the Problem via Ownership

- Each object is owned by one variable
- Ownership can be moved or borrowed
  - Mode of borrowing: immutable/mutable

```
let mut alice = Box::new(1);
let bob = alice;
println!("alice:{}", alice);
```

alice owns the Box object

move the ownership from alice to bob

```
let mut alice = Box::new(1);
let bob = &mut alice;
**bob = **bob + 1;
println!("alice:{}", alice);
```

bob borrows the ownership

bob is dead; return the ownership

- **Exclusive mutability principle**: an object cannot be both mutable and shared at any program points. How?

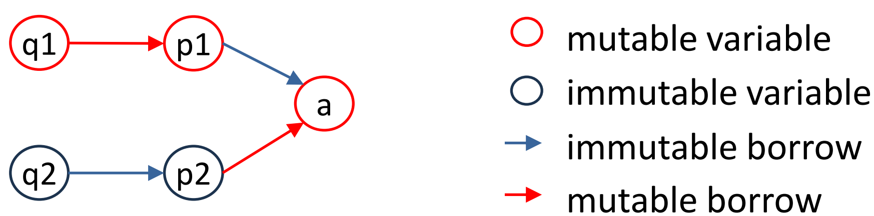
Why the Approach is Efficient?

- Compute the 'minimum' liveness of each variable
- Avoid hard alias-analysis problems
  - No need to track multi-level aliases
  - The mutability does not propagate

```
let mut a = 1;
let mut p1 = &a;
let p2 = &mut a;
let mut q1 = &mut p1;
let q2 = &p2;
```

live variable analysis (backward)	**live variables** with borrow constraint
{a}	{a}
{a, p1}	{a, p1}
{p1, p2}	{a, p1, p2}
{p2}	{a, p2}

borrow constraint: 'a > 'p2



Limitations of Ownership

- We may need both shared & mutable, *e.g.,* double-linked list


```
struct Node {
    val: u64,
    prev: Option<Weak<RefCell<Node>>>,
    next: Option<Weak<RefCell<Node>>>,
}
```

Option 1: Shared Pointer
(with runtime cost)

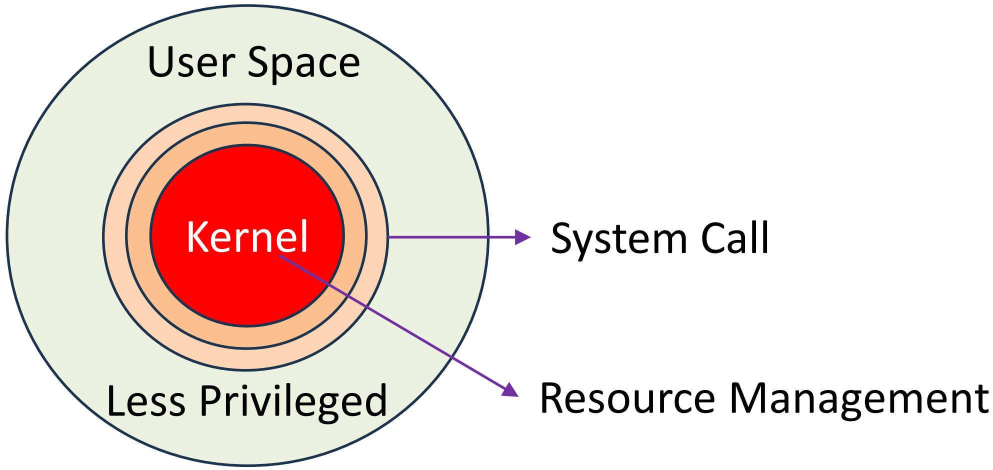
```
struct Node {
    val: u64,
    next: *mut Node,
    prev: *mut Node,
}
```

Option 2: Raw Pointer
(unsafe code, bypass borrow check)

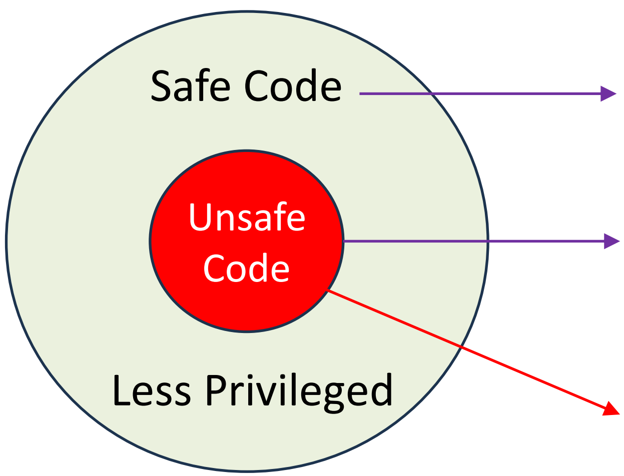
- Ownership also requires RAII because dropping unit data is bad
  - Use unsafe code to create uninitialized object

Code Privilege Model

Operating System:
Ring Model



Rust:
Code Privilege Model



Security objective of Rust:

- Memory safety
- Concurrency safety
- No other undefined behaviors

Interior unsafe: encapsulate privileged code within safe APIs

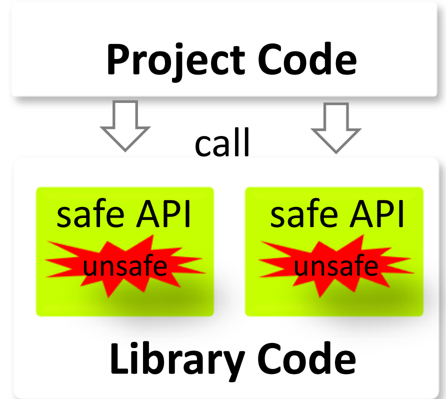
May break the security protocol

Interior Unsafe: Encapsulate Unsafe Code within Safe APIs

- Encourage developers not to use unsafe code directly
- However, API soundness with unsafe code cannot be verified by compiler

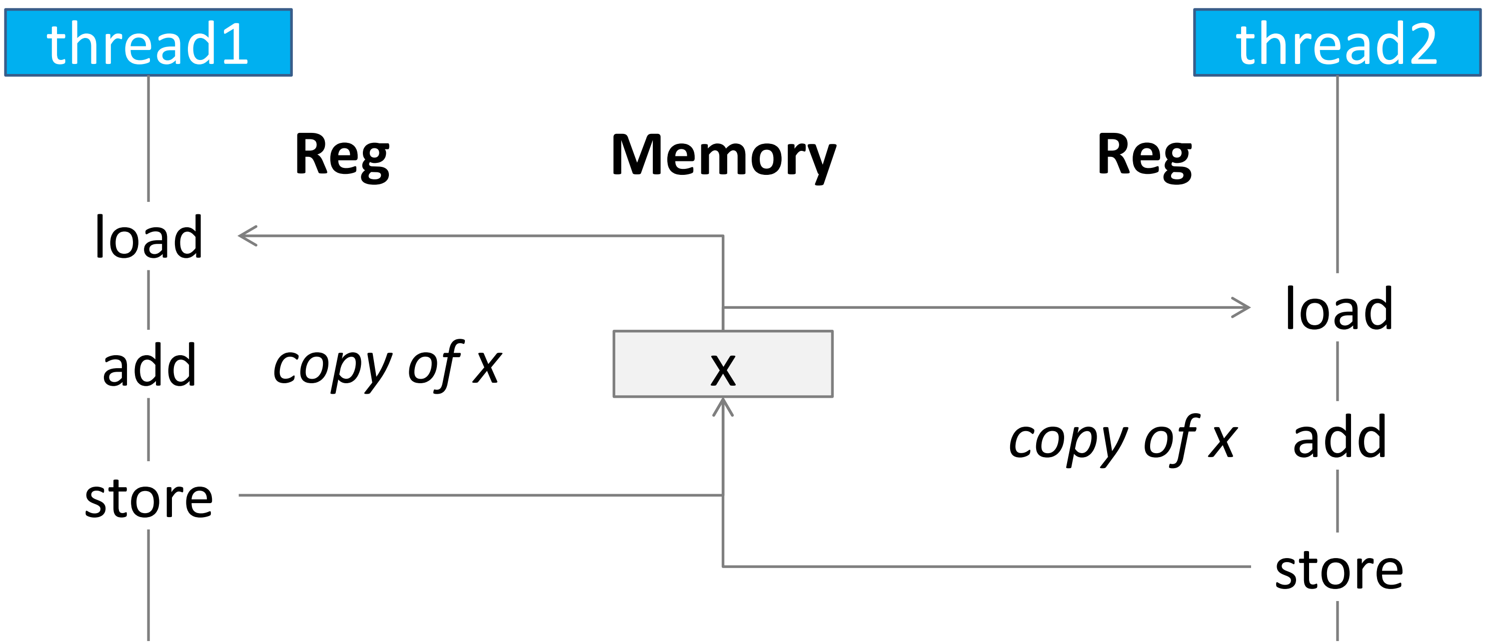
```
impl<T> Vec<T> {
    //safe API encapsulation
    pub fn push(&mut self, value: T) {
        if self.len == self.buf.capacity() {
            self.buf.reserve_for_push(self.len);
        }
        unsafe {
            let end = self.as_mut_ptr().add(self.len);
            ptr::write(end, value);
            self.len += 1;
        }
    }
}
```

Code of Vec from Rust std-lib



Concurrency Safety

- Non-atomic code is vulnerable to race condition




```
int global_cnt = 0;
void *mythread(void *in) { global_cnt++;  }
assert(pthread_create(&tid[0], NULL, mythread, NULL)==0);
assert(pthread_create(&tid[1], NULL, mythread, NULL)==0);
pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);
assert(global_cnt==2);
```

Data Sharing Among Threads in Rust

```
let mut x = 1;
let tid = thread::spawn(move|| {
    x = 10;
    println!("x = {}", x); //x = 10
});
tid.join().unwrap();
println!("x = {}", x); // x= 1
```

move the ownership or copy the value

copied x

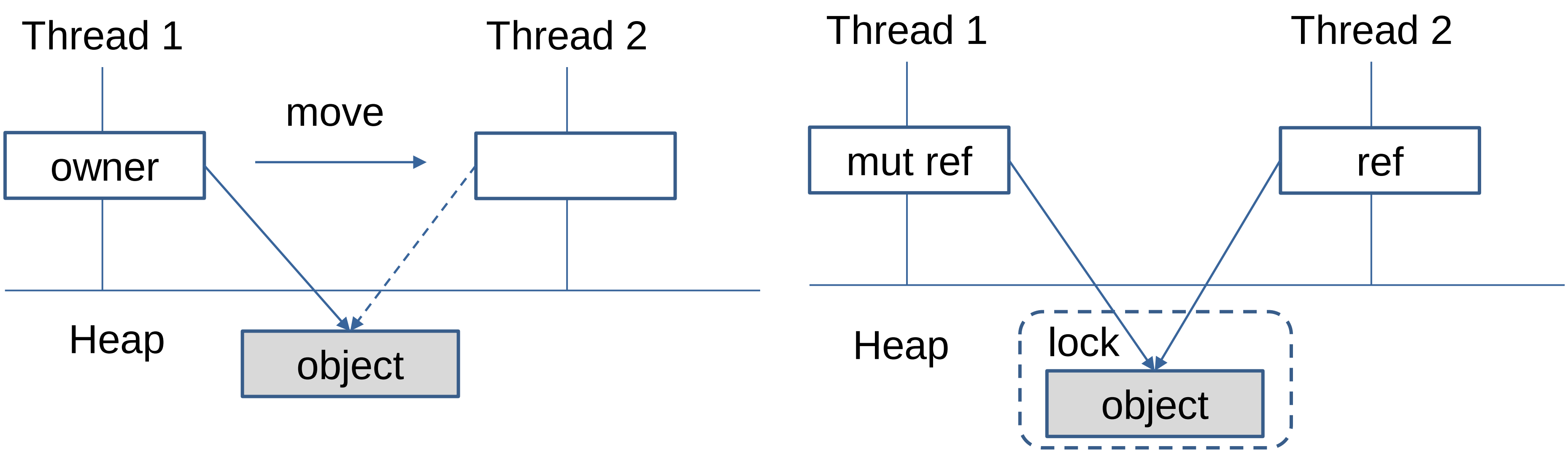
```
let mut x = Box::new(1);
//let mut y = x.clone();
let tid = thread::spawn(move|| {
    *x = 10;
    println!("spawn: x = {}", x);
});
tid.join().unwrap();
println!("main: x= {}", x);
```

move the ownership of x to the thread

Illegal to access x

Declare Types with Send/Sync Trait (unsafe)

- Send Trait: The type can be transferred (moved) between threads
  - For types of Copy trait, make a copy of the object
  - For non-copy, transfer the ownership
- Sync Trait: The type is safe to be referenced from multiple threads
  - Any type T is Sync if &T is Send
  - Sync is usually more rigid than Send
- Raw pointers are neither Send or Sync by default



Other Reliability Features

- Prevent dangling pointer via lifetime specification
- Perform boundary check for slice/vector
- Prevent false monomorphism via trait bound
- Enforce error handling via Monad
- Check integer overflow in debug mode
- ...

An Example

"Compare the performance of matrix multiplication with different languages"

Python: done!

R: done!

Java: done!

C++: done!

Go: done!

Rust: panic...

```python
#矩阵乘法（每次生成一个新的随机矩阵）
def matrix_multiply(matrix):
    # 获取矩阵的维度
    n = len(matrix)
    # 创建结果矩阵，初始化为全零
    result = [[0 for _ in range(n)] for _ in range(n)]
    # 随机生成本轮所乘矩阵
    matrix0 = generate_random_matrix(n)
    # 进行矩阵乘法运算
    for i in range(n):
        for j in range(n):
            for k in range(n):
                result[i][j] += matrix[i][k] * matrix0[k][j]

    return result
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/matmul 10 100`
thread 'main' panicked at 'attempt to add with overflow', src/main.rs:91:17
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
ran@gaishuijiaoladeMacBook-Pro matmul %
```

Trust Rust with Reservations

Rust for Linux
Organization for adding support for the Rust language to the l
456 followers rust-for-linux@vger.kernel.org

Wedson Almeida Filho
@Rust for Linux

```
Date     Mon, 19 Sep 2022 19:05:23 +0100
From     Wedson Almeida Filho <>
Subject  Re: [PATCH v9 12/27] rust: add `kernel` crate
```

```
We generally have two routes to avoid undefined behaviour: detect at
compile time (and fail compilation) or at runtime (and stop things
before they go too far). The former, while feasible, would require some
static analysi or passing tokens as arguments to guarantee that we're in
sleepable context when sleeping (all ellided at compile time, so
zero-cost in terms of run-time performance), but likely painful to
program use.
```

Linus Torvalds

```
 You need to realize that

  (a) reality trumps fantasy

  (b) kernel needs trump any Rust needs
```

```
Or, you know, if you can't deal with the rules that the kernel
requires, then just don't do kernel programming.

Because in the end it really is that simple.  I really need you to
understand that Rust in the kernel is dependent on *kernel* rules. Not
some other random rules that exist elsewhere.

                Linus
```

Rust for Linux
Organization for adding support for the Rust language to t
456 followers rust-for-linux@vger.kernel.org

Wedson Almeida Filho

```
While I disagree with some of what you write, the point is taken.

But I won't give up on Rust guarantees just yet, I'll try to find
ergonomic ways to enforce them at compile time.

Thanks,
-Wedson
```

Linus Torvalds

```
If you cannot get over the fact that the kernel may have other
requirements that trump any language standards, we really can't work
together.
```

https://lkml.org/lkml/2022/9/19/1105#1105.php

III. Efficiency of Rust

Zero Cost Abstraction

What you don't use, you don't pay for.

What you do use, you couldn't hand-code any better.

-- Bjarne Stroustrup

There are no Zero Cost Abstractions

-- Chandler Carruth @ CppCon 2019

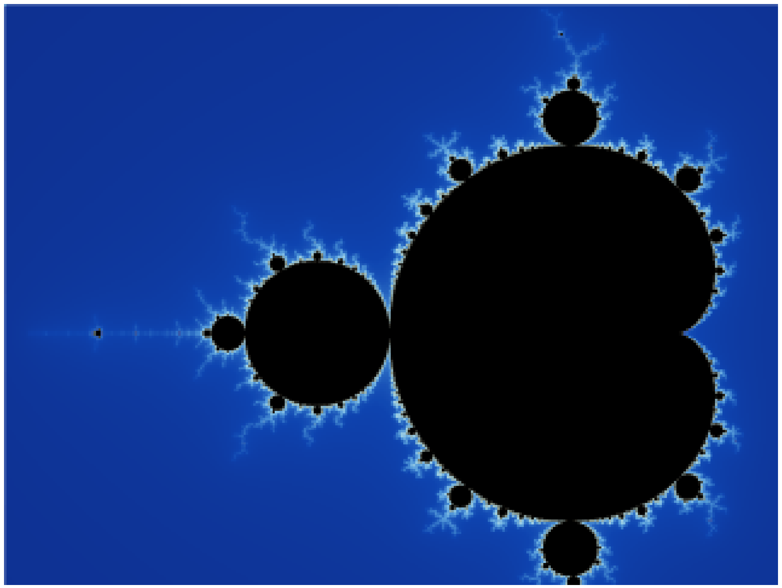
Bjarne Stroustrup, "The Design and Evolution of C++." 1994.

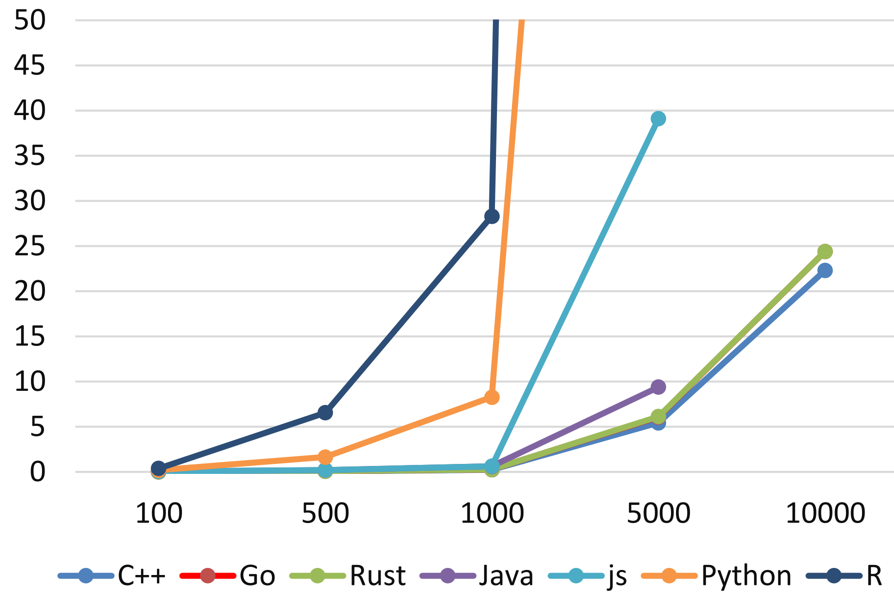
Example of Abstractions

- Dynamic memory management: garbage collection or manual?
- Polymorphism: compile-time binding or dynamic dispatch?
- Functions: inline or not?
- ...

Comparison Study: Mandelbrot Set

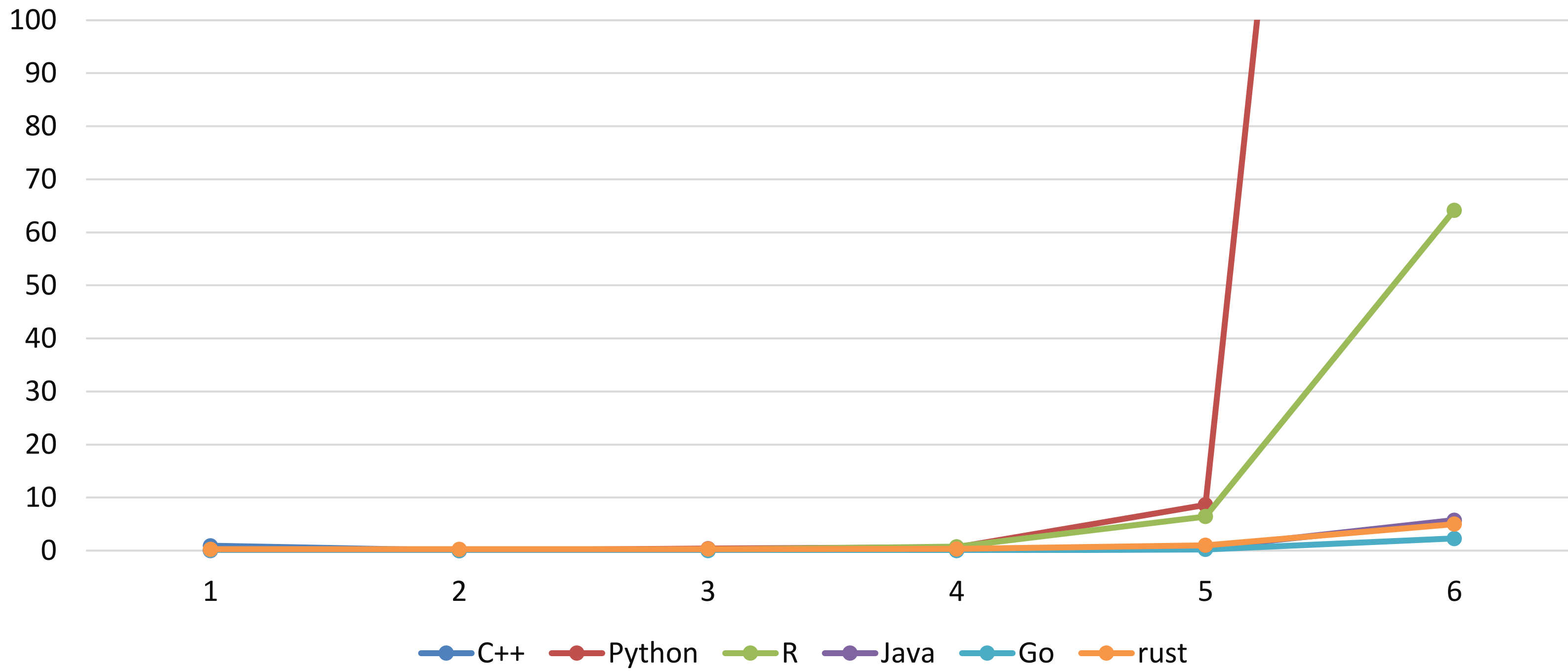
$$f_c(z) = z^2 + c$$





Task (resolution)	C++	Rust	Go	Java	js	Python (numpy)	R (matrix)
100	0.00352	0.00354	0.00404	0.0542	0.0631	0.183	0.374
500	0.0581	0.0663	0.07	0.197	0.204	1.63	6.55
1000	0.23	0.248	0.258	0.613	0.616	8.26	28.3
5000	5.43	6.11	6.06	9.38	39.1	240	889
10000	22.3	24.4	24.4	-	-	-	-

Comparison Study: Matrix Multiplication



Task (dim*round)	C++	Python	R	Java	Go	Rust
10*10	0.924	0.043	0.109	0.004	0.045	0.262
10*100	0.016	0.042	0.116	0.007	0.004	0.266
10*1000	0.061	0.3667	0.191	0.0397	0.006	0.280
100*10	0.074	0.644	0.745	0.061	0.028	0.319
100*100	0.566	8.639	6.427	0.565	0.236	1.005
100*1000	5.590	448.915	64.129	5.734	2.294	4.993

Optimization Study by Crichton: k-CorrSet problem

"given a size k, which set of k questions has the highest correlation with overall performance?"

```
{
  "user": "5ea2c2e3-4dc8-4a5a-93ec-18d3d9197374",
  "question": "7d42b17d-77ff-4e0a-9a4d-354ddd7bbc57",
  "score": 1
},
/* ... more data ... */
```

1) Python => Rust: speedup 8 times
2) Change HashMap to Vec to avoid hash: speedup 4*6 times
3) Disable boundary checks with get_unchecked(): speedup 1.16 times
4) Use bit-set for sparse data handling: speedup 3.4 times
5) Use ass SIMD (std::simd): speedup 34 times
6) Allocation at the beginning: speedup 1.24 times

https://willcrichton.net/notes/k-corrset/

Parallel

- SIMD

```
#![feature(portable_simd)]
use std::simd::f32x4;
fn main() {
    let a = f32x4::splat(10.0);
    let b = f32x4::from_array([1.0, 2.0, 3.0, 4.0]);
    println!("{:?}", a + b);
}
```

- Multi-threading with Rayon: no data races

```
use rayon::prelude::*;
fn sum_of_squares(input: &[i32]) -> i32 {
    //input.iter().map(|&i| i * i) .sum()
    input.par_iter().map(|&i| i * i) .sum()
}
```


https://doc.rust-lang.org/stable/std/simd/struct.Simd.html

IV. Usability

What Makes Rust Difficult?

- Confusing rules for memory safety
  - borrow check
  - lifetime inference
- Unfamiliar with the paradigm or design patterns
  - trait (duck typing)
  - closure (functional)
  - macro
  - ...

Confusing Rules: Borrow Check

```
#![allow(unused_variables)]

struct Inner { inner: u8 }
struct Outer1 { a: [Inner; 2] }
struct Outer2 { a: (Inner, Inner) }

fn test(in1: &mut Inner, in2: &Inner){}

fn main() {
    let mut out1 = Outer1 { a:
      [Inner {inner: 1}, Inner {inner: 3}]};
    let mut out2 = Outer2 { a:
      (Inner {inner: 1}, Inner {inner: 3})};
-   test(&mut out1.a[0], &out1.a[1]);
+   let (first, rest) = out1.a.split_first_mut().unwrap();
+   test(first, &rest[0]);
    test(&mut out2.a.0, &out2.a.1);
}
```

```
let r1 = &mut out1.a[0];
let r3 = &mut out2.a.0;
let r2 = &out1.a[1];
let r4 = &out2.a.1;
*r1 += 1;
*r3 += 1;

println!("{:?}", r2);
println!("{:?}", r4);
```

PC-1 changes to PC-3

Shuofei Zhu, *et al*. "Learning and programming challenges of rust: A mixed-methods study." *ICSE* 2022.

Confusing Rules: Lifetime

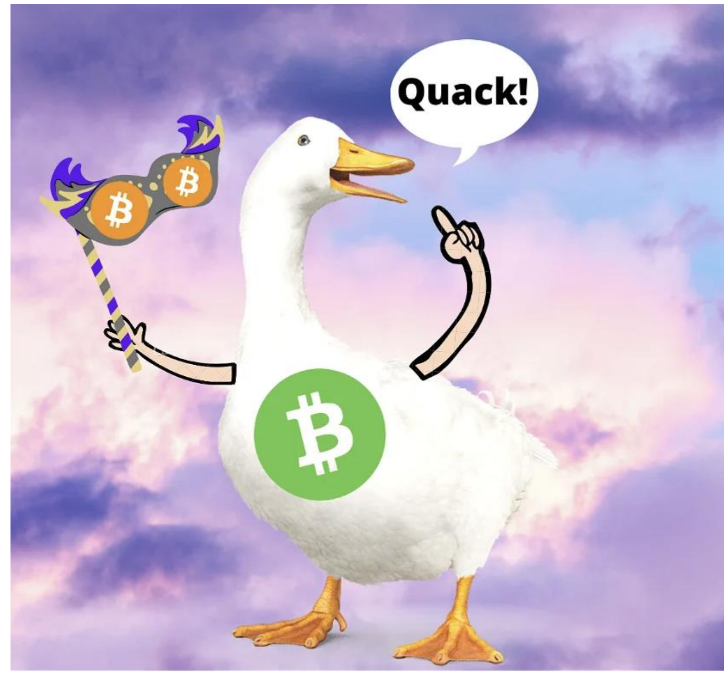
```
struct Foo {}
struct Bar2<'b> { x: &'b Foo,}

impl<'b> Bar2<'b> {
-   fn f(&'b mut self)-> &'b Foo {
+   fn f(&mut self)-> &'b Foo {
        self.x
    }
}
```

```
fn f4() {
    let foo = Foo {};
    let mut bar2 = Bar2 {
        x: &foo };
    bar2.f();
    let z = bar2.f();
}
```


Shuofei Zhu, *et al*. "Learning and programming challenges of rust: A mixed-methods study." *ICSE* 2022.

Design Pattern: Trait

```
struct Sheep { name: &'static str }
trait Animal {
    fn new(name: &'static str) -> Self;
    fn name(&self) -> &'static str;
    fn talk(&self) -> &'static str;
}

impl Sheep {
    fn shear(&mut self) {
        ...
    }
}

impl Animal for Sheep {
    ...
}
```



"If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck"

Design Pattern: Functional Programming

```
fn is_odd(n: u32) -> bool {
    n % 2 == 1
}
fn main() {
    println!("Find the sum of all the squared odd numbers under 1000");
    let upper = 1000;
    let sum_of_squared_odd_numbers: u32 =
        (0..).map(|n| n * n) // All natural numbers squared
             .take_while(|&n_squared| n_squared < upper)
             .filter(|&n_squared| is_odd(n_squared))
             .sum();
    println!("{}", sum_of_squared_odd_numbers);
}
```


https://doc.rust-lang.org/rust-by-example/fn/hof.html

Design Pattern: Macros

```
// `find_min!` will calculate the minimum of any number of arguments.
macro_rules! find_min {
    // Base case:
    ($x:expr) => ($x);
    // `$x` followed by at least one `$y,`
    ($x:expr, $($y:expr),+) => (
        // Call `find_min!` on the tail `$y`
        std::cmp::min($x, find_min!($($y),+))
    )
}

fn main() {
    println!("{}", find_min!(1));
    println!("{}", find_min!(1 + 2, 2));
    println!("{}", find_min!(5, 2 * 3, 4));
}
```


https://doc.rust-lang.org/rust-by-example/macros/repeat.html

V. Summary

Summary

- Attractive features of Rust:
  - Security: memory safety, concurrency safety
  - Reliability: checked add, boundary check, monad,...
  - Efficiency: zero cost abstraction
- Problems of Rust:
  - Usability
  - Verifiability of security
- The community/ecosystem of Rust grows at an incredible pace

Thanks for Watching

Q & A

xuh@fudan.edu.cn