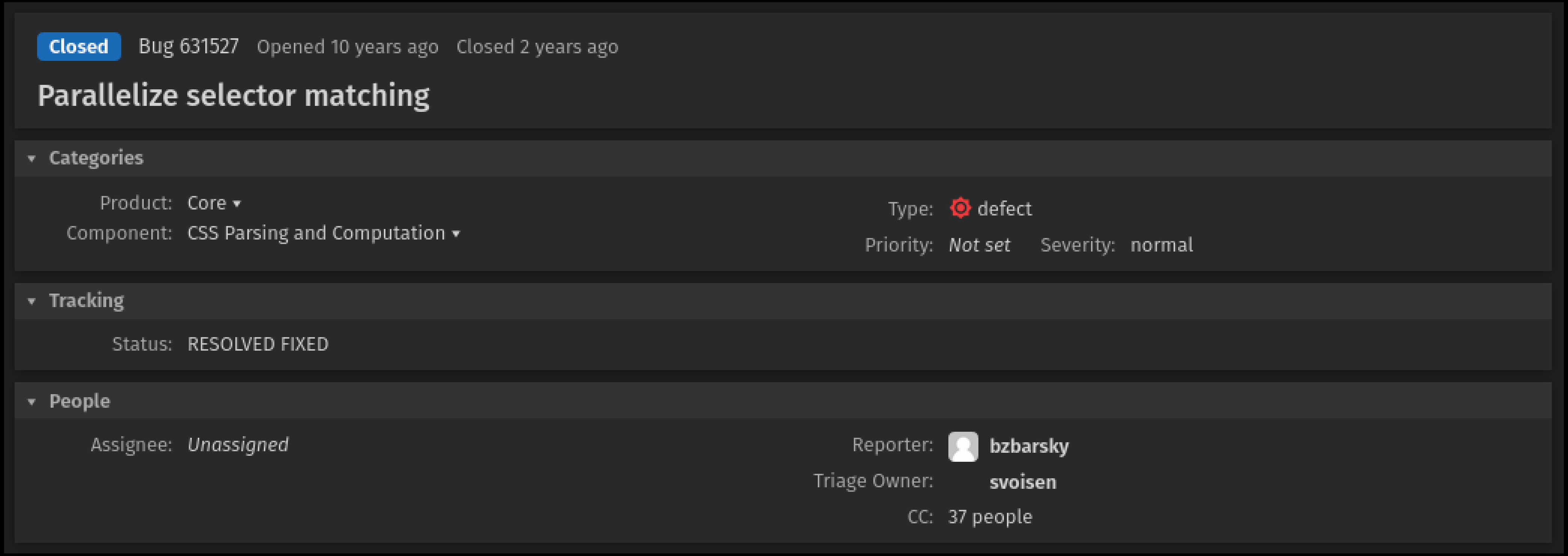
c.R.U.S.T of Rust

Gaurav Saini

Software Engineer, IoT

Github: @metamemelord

Need tools to leverage the multiple cores of the CPU



Closed Bug 631527 Opened 10 years ago Closed 2 years ago

Parallelize selector matching

▾ **Categories**

Product: Core ▾

Component: CSS Parsing and Computation ▾

Type: defect

Priority: *Not set* Severity: normal

▾ **Tracking**

Status: RESOLVED FIXED

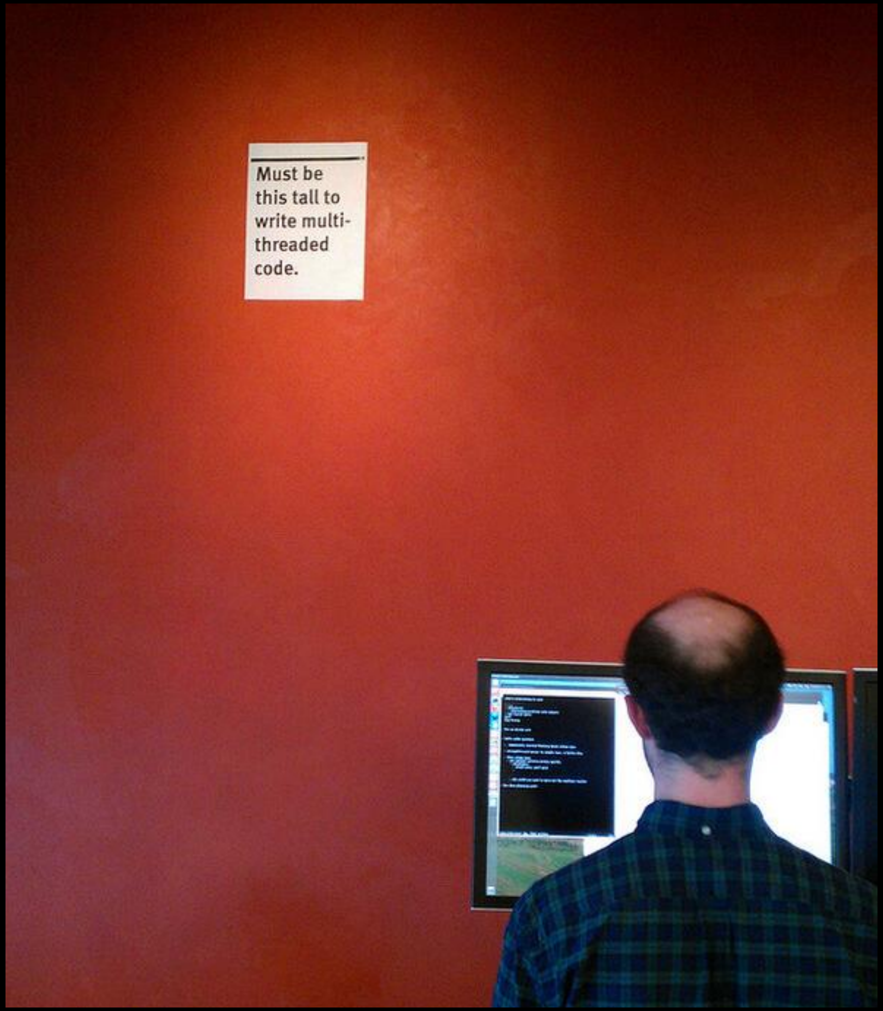
▾ **People**

Assignee: *Unassigned*

Reporter: **bzbarsky**

Triage Owner: **svoisen**

CC: 37 people


Must be this tall to write multi-threaded code.

What is Rust's deal?

From the official website:

A language empowering everyone to build reliable and efficient software.

- Performance
- Reliability
- Productivity

What do these really mean?

- Rust provides top-notch performance with execution speed comparable to C++ and promise of "Fearless concurrency".
- Rust enforces security rules at compile time: "If it compiles, it works".
- Rust provides developers with great documentation, dependency management, and error messages.

All these..

With NO Garbage collector or Runtime.

```rust
fn main() {
    let x = "people!";
    println!("Hello, {}", x);

    for it in 0..10 { // Iteration from 0 to 9
        println!("{} ", it);
    }

    let v: Vec<i32>  = vec![1, 4, 6]; // Syntactical sugar for Vec<i32>
    println!("{:?}", v);
}
```

What are these "memory issues" that Rust is protecting us from?

- Segmentation faults - Are you accessing something that you're not supposed to?
    - Use-after free
    - Double free
- Concurrency issues - Are multiple tasks trying to modify some data concurrently?
    - Deadlock
    - Data Races

And why do these happen?

Lack of a garbage collection leads developers to manage the memory on their own ..and humans do mistakes.

Small mistakes like freeing the memory twice causes memory corruption. Rust is built ground-up to prevent these from happening.

In multithreaded applications, typical tasks around data are:

- Mutation
- Aliasing
- Ordering

They're great, until things start to interleave, what happens when mutation is done via multiple aliases?

```cpp
#include<iostream>

int main() {
    int *x = new int[30];
    std::cout << x[0];
    delete x;
    delete x; // Double free
}
```

```go
package main

import (
  "math/rand"
  "sync"
)

func main() {
  m := make(map[string]int)
  var wg sync.WaitGroup
  for i:=0;i<1000;i++ {
    wg.Add(1)
    go worker(m, &wg);
  }
  wg.Wait()
}
```

```go
func worker(m map[string]int, wg *sync.WaitGroup) {
  key := randString(1)
  if _, ok := m[key]; !ok {   // <------------ Data race
    m[key] = 0
  }
  m[key]++                     // <------------ Data race
  wg.Done()
}

func randString(s int) string {
  lowercase := "abcdefghijklmnopqrstunwxyz"
  result := []byte{}
  for len(result) != s {
    result = append(result, lowercase[rand.Intn(26)])
  }
  return string(result)
}
```

Rust's solution

- Mandated initialization: No un-initialized variables
- Restrictive aliasing: Can create aliases, but within the provided rules.
- Ownership

Ownership

Quoting from the Rust book:

- Each value in Rust has a variable that’s called its *owner*.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

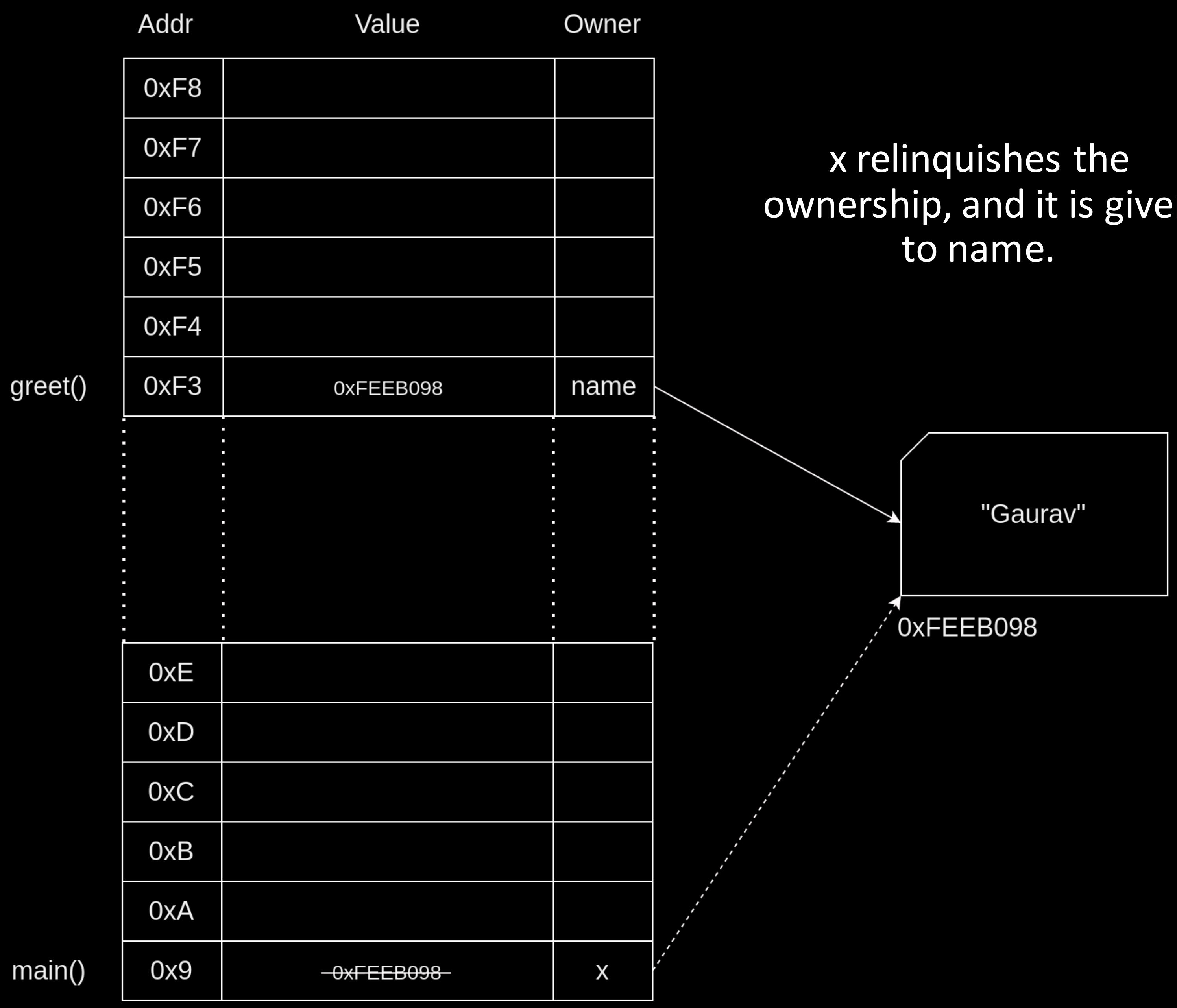
Passing data around

By default, Rust passes data across the functions by transferring the ownership of a variable. This is known as moving the ownership.

Move

```
fn greet(name: String) {
    println!("Hello, {}!", name);
}

fn main() {
    let x = "Gaurav".to_string();
    show(x);
}
```



	Addr	Value	Owner
	0xF8		
	0xF7		
	0xF6		
	0xF5		
	0xF4		
greet()	0xF3	0xFEEB098	name
	0xE		
	0xD		
	0xC		
	0xB		
	0xA		
main()	0x9	~~0xFEEB098~~	x

x relinquishes the ownership, and it is given to name.

The copy way!

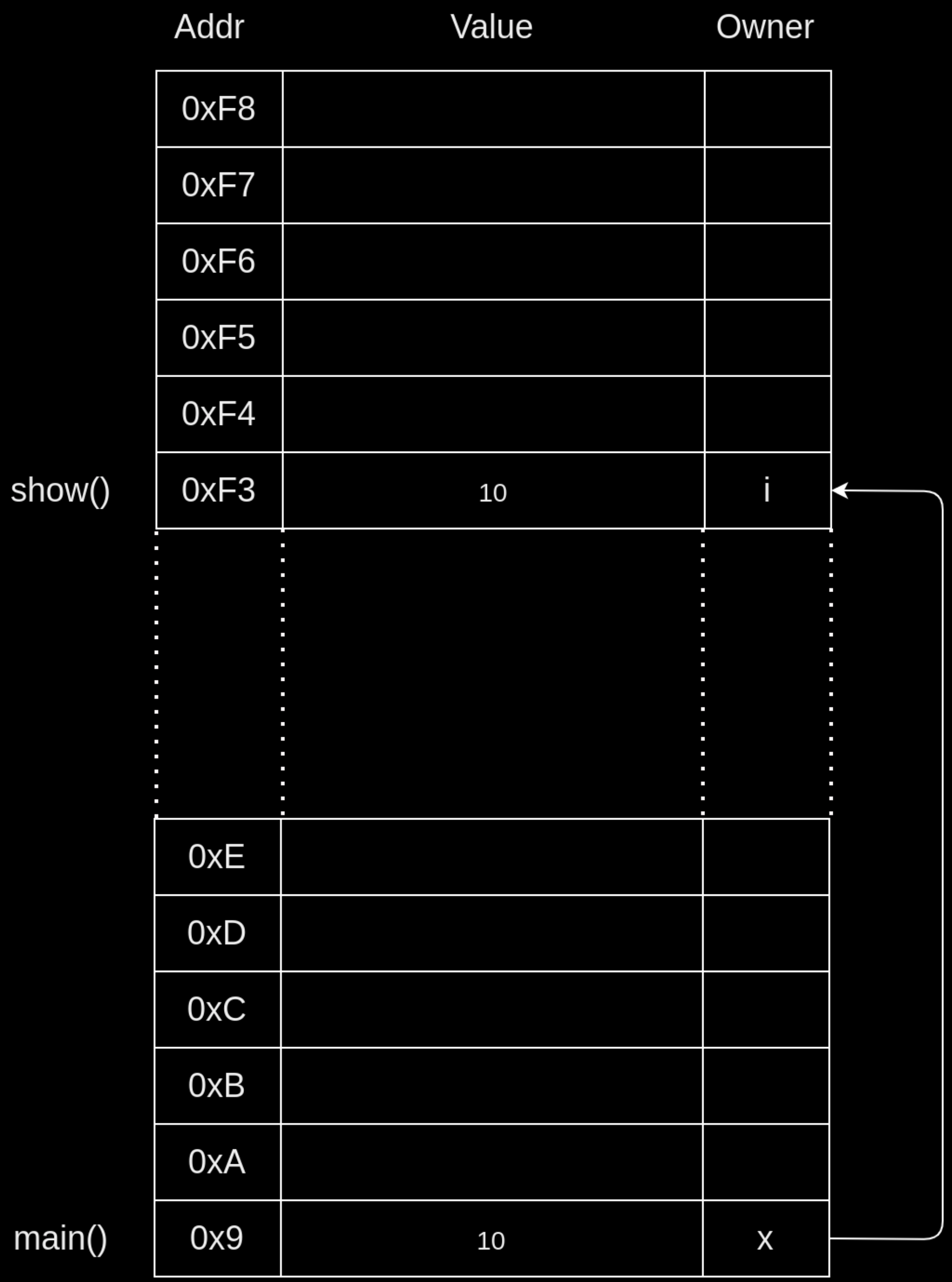
Some data types in Rust have a trait (a trait is a construct similar to an interface), which allows these types to be copied.

…but VERY EXPENSIVE!

Copy

```
fn show(i: i32) {
    println!("{}", i);
}

fn main() {
    let x = 10;
    show(x);
    println!("{}", x)
}
```



	Addr	Value	Owner
	0xF8		
	0xF7		
	0xF6		
	0xF5		
	0xF4		
show()	0xF3	10	i
	0xE		
	0xD		
	0xC		
	0xB		
	0xA		
main()	0x9	10	x

Value in x is copied to i

Sharing is ~~caring~~ faster!

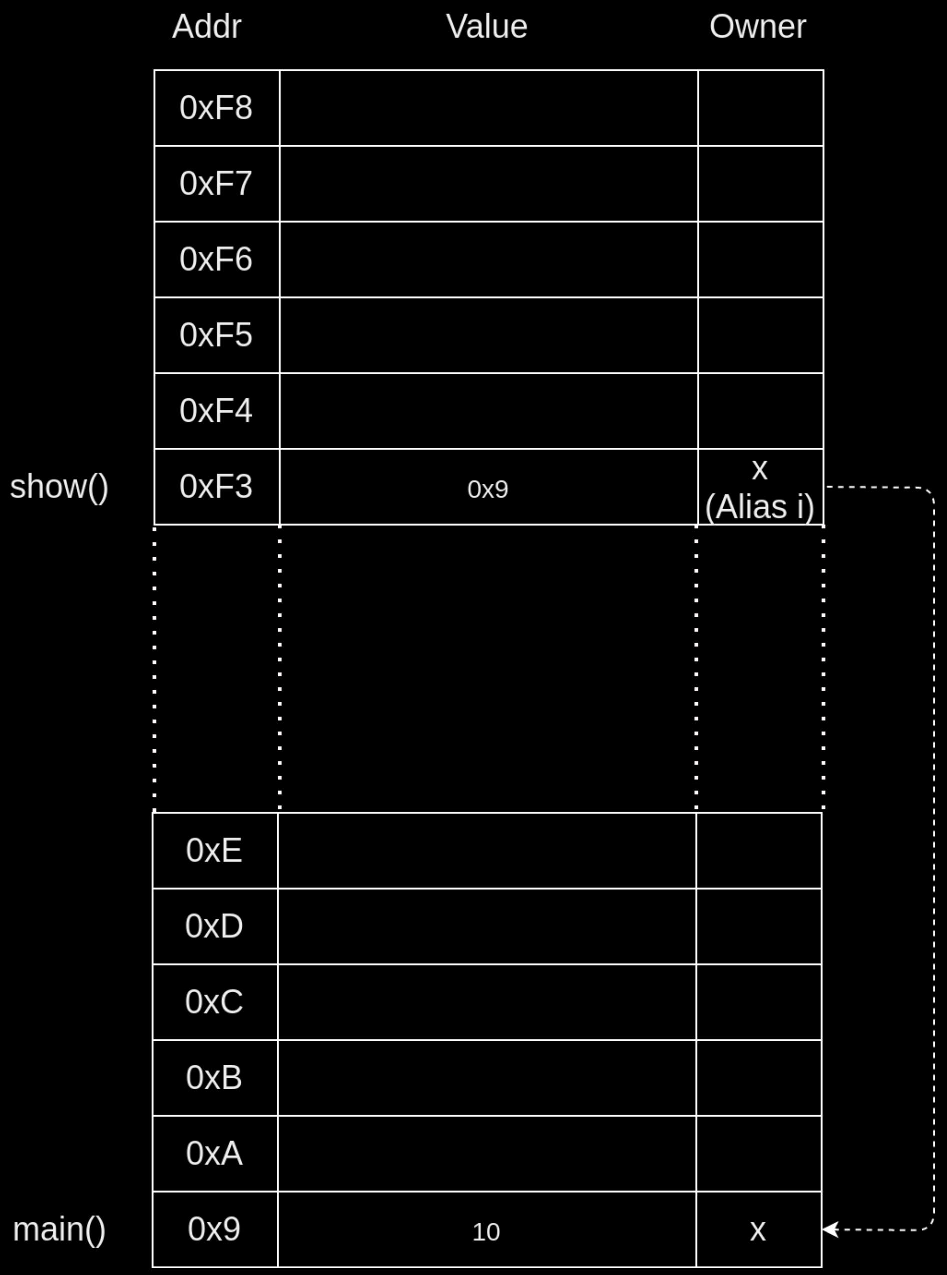
Rust provides 2 types of borrowing:

- (Multiple) Read only references (&T)
- (Single active*) Mutable reference (&mut T)

Borrow

```
fn show(i: &i32) {
    println!("{}", i);
}

fn main() {
    let x = 10;
    show(&x);
    println!("{}", x)
}
```



	Addr	Value	Owner
	0xF8		
	0xF7		
	0xF6		
	0xF5		
	0xF4		
show()	0xF3	0x9	x (Alias i)
	0xE		
	0xD		
	0xC		
	0xB		
	0xA		
main()	0x9	10	x

i is a reference to the value owned by x

Concurrent workloads?

Code with potential race conditions will not compile in Rust; Rust has a famous proverb, "If it compiles, it works".

```go
// Code with data race (Compiles and fails
at runtime)
package main

import (
    "math/rand"
    "sync"
)

func worker(m map[string]int, wg
        *sync.WaitGroup) {
    key := randString(1)
    if _, ok := m[key]; !ok {
        m[key] = 0
    }
    m[key]++
    wg.Done()
}
```

```go
func randString(s int) string {
    lowercase := "abcdefghijklmnopqrstunwxyz"
    result := []byte{}
    for len(result) != s {
        result = append(result, lowercase[rand.Intn(26)])
    }
    return string(result)
}

func main() {
    m := make(map[string]int)
    var wg sync.WaitGroup
    for i:=0;i<1000;i++ {
        wg.Add(1)
        go worker(m, &wg);
    }
    wg.Wait()
}
```

```go
package main

import (
    "fmt"
    "math/rand"
    "sync"
)

func worker(m map[string]int, wg
       *sync.WaitGroup, mx *sync.Mutex) {
    mx.Lock()
    defer mx.Unlock()
    key := randString(1)
    if _, ok := m[key]; !ok {
        m[key] = 0
    }
    m[key]++
    wg.Done()
}
```

```go
func randString(s int) string {
     lowercase := "abcdefghijklmnopqrstunwxyz"
    result := []byte{}
    for len(result) != s {
        result = append(result, lowercase[rand.Intn(26)])
    }
    return string(result)
}

func main() {
    m := make(map[string]int)
    var mx sync.Mutex
    var wg sync.WaitGroup
    for i:=0;i<1000;i++ {
        wg.Add(1)
        go worker(m, &wg, &mx);
    }
    wg.Wait()
    fmt.Println(m)
}
```

```
use rand::Rng;
use std::collections::HashMap;
use std::thread;

fn rand_string(s: i32) -> String {
    let lowercase = "abcdefghijklmnopqrstunwxyz";
    let mut result = String::new();
    for _i in 0..s {
        let num: usize = rand::thread_rng().gen_range(0, 26);
        result.push(lowercase.as_bytes()[num] as char);
    }
    result
}
```

```
fn main() {
    let mut m: HashMap<String, i32> =
        HashMap::new();
    let mut handlers = vec![];
    for _i in 0..1000 {
        handlers.push(thread::spawn(|| {
            let key = rand_string(1);
            *m.entry(key).or_insert(0) += 1;
        }));
    }

    for handler in handlers {
        handler.join();
    }
}
```

```
use rand::Rng;
use std::collections::HashMap;
use std::thread;
use std::sync::{Arc, Mutex};

fn rand_string(s: i32) -> String {
    let lowercase = "abcdefghijklmnopqrstunwxyz";
    let mut result = String::new();
    for _i in 0..s {
        let num: usize = rand::thread_rng()
            .gen_range(0, 26);
        result.push(lowercase.as_bytes()[num] as char);
    }
    result
}
```

```
fn main() {
    let m = Arc::new(Mutex::new(HashMap::new()));
    let mut handlers = vec![];
    for _i in 0..1000 {
        let mp = m.clone();
        handlers.push(thread::spawn(move || {
        let key = rand_string(1);
        *mp.lock()
            .unwrap()
            .entry(key)
            .or_insert(0) += 1;
        }));
    }

    for handler in handlers {
        handler.join().unwrap();
    }
    println!("{:?}", m.lock().unwrap());
}
```

Rust being used at multiple organizations. Some of the use-cases for Rust are:

- Web Assembly

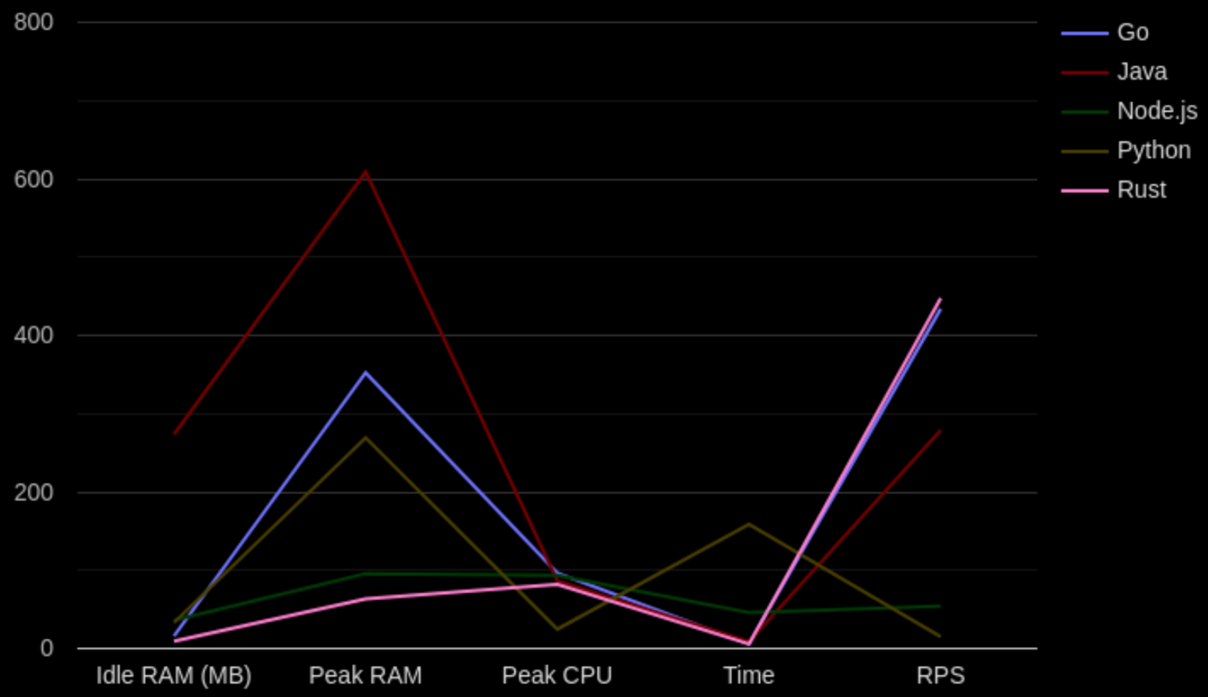
- Command-line Tools

- Networking – Rust is great for high-performance webservers.

Web benchmarks

Total requests: 250,000
Concurrency: 250
Request:
GET / HTTP/1.1
Response:
HTTP/1.1 200 OK
Content-type: application/json

"OK"



*	Idle RAM (MB)	Peak RAM (MB)	Peak CPU (%)	Time (s)	Requests per second / 100
Go (Gin)	15.8	352.6	96.4	5.761	433.94
Java (Spring boot)	273.6	608.9	86.4	8.962	278.94
Node.js (Express.js)	35.9	95.7	93.4	46.061	54.27
Python (Flask) (Failed numerous times)	33.1	269.3	24.6	158.545	15.01
Rust (Actix)	9.2	63.6	82.1	5.664	447.68

Code available on Github

Thank you!

Interested? Find some simple projects here