



Rust

Programowanie współbieżne*

Łukasz Neumann

lukasz.neumann@pw.edu.pl

https://staff.elka.pw.edu.pl/~lneumann/rust_2.pdf



Brendan Dolan-Gavitt

@moyix



My objection to Rust, OCaml, Haskell, etc. is that I am a bad programmer who wants to write bad programs. Python is naturally suited to this task

2:07 AM · Jan 7, 2022 · Twitter Web App

306 Retweets **41** Quote Tweets **3,188** Likes

Enumerations - tagged unions / sum types / algebraic data types (ADT)



```
1 enum WebEvent {
2     PageLoad,
3     PageUnload,
4     KeyPress(char),
5     Paste(String),
6     Click { x: i64, y: i64 },
7 }
8
9 fn inspect(event: WebEvent) {
10     match event {
11         WebEvent::PageLoad => println!("page loaded"),
12         WebEvent::PageUnload => println!("page unloaded"),
13         WebEvent::KeyPress(c) => println!("pressed '{c}'."),
14         WebEvent::Paste(s) => println!("pasted '{s}'."),
15         WebEvent::Click { x, y } => {
16             println!("clicked at x={x}, y={y}.");
17         },
18     }
19 }
```

std::optional jako enumeracja



```
1 enum Option<T> {  
2     Some(T),  
3     None  
4 }  
5  
6 fn maybe_print<T: Display>(value: Option<T>) {  
7     match value {  
8         Option::Some(v) => println!("{v}"),  
9         Option::None => println!("Nothing to print!"),  
10    }  
11 }
```

Failure is not an **Option**, it's a **Result**

```
1 enum Result<T, E> {  
2     Ok(T),  
3     Err(E),  
4 }
```



Obsługa błędów



```
1 fn divide(a: f32, b: f32) → Result<f32, String> {
2     if b ≠ 0. {
3         Ok(a / b)
4     } else {
5         Err("Can't divide by zero".into())
6     }
7 }
8
9 fn mean_std(scores: &[f32]) → Result<(f32, f32), String> {
10     let length = scores.len() as f32;
11     let mean = divide(scores.iter().sum(), length)?;
12     let summed_deviations = scores.iter().map(|x| (x - mean).powf(2.)).sum();
13     let std_dev = divide(summed_deviations, length)?.sqrt();
14     Ok((mean, std_dev))
15 }
16
17 fn main() {
18     let (mean, std) = mean_std(&[1., 2., 3., 4., 5.]).unwrap();
19     println!("{mean} {std}");
20 }
```

Program: *Returns a wrong result*

Me: Ok, I will set up some breakpoints.
Show me how you got there

Program:



Wyzwania programowania współbieżnego

- niedeterministyczna kolejność wykonania - problemy z reprodukcją błędów
- subtelne błędy, których wyłapanie jest ciężkie bądź niemożliwe
- “nielokalność” kodu
- myląca intuicja oraz błędne założenia o działaniu programu

Programowanie wielowątkowe



```
1 use std::thread;
2 use std::time::Duration as D;
3
4
5 fn main() {
6     thread::spawn(|| {
7         for i in 1..10 {
8             println!("{i} Thread");
9             thread::sleep(D::from_millis(1));
10        }
11    });
12
13    for i in 1..5 {
14        println!("{i} Main");
15        thread::sleep(D::from_millis(1));
16    }
17 }
```



```
1 #include <iostream>
2 #include <thread>
3 using namespace std;
4
5 int main() {
6     thread my_thread([] {
7         for (auto i = 1; i < 10; ++i){
8             cout << i << " Thread" << endl;
9             this_thread::sleep_for(1ms);
10        }
11    });
12
13    for (auto i = 1; i < 5; ++i){
14        cout << i << " Main" << endl;
15        this_thread::sleep_for(1ms);
16    }
17    return 0;
18 }
```


Programowanie wielowątkowe - koniec wykonania



```
> rustc naive_threads.rs && ./naive_threads
1 Main
1 Thread
2 Main
2 Thread
3 Main
3 Thread
4 Main
4 Thread
5 Thread
```



```
> g++ -Wall -Wextra -pedantic -Werror \
    -std=c++20 naive_threads.cpp && ./a.out
1 Main
1 Thread
2 Main2
  Thread
3 Main
3 Thread
4 Main
4 Thread
terminate called without an active exception
5 Thread
[1]      39523 IOT instruction (core
dumped)  ./a.out
```

Czekanie na wszystkie wątki



```
1 use std::thread;
2 use std::time::Duration as D;
3
4
5 fn main() {
6     let my_thread = thread::spawn(|| {
7         for i in 1..10 {
8             println!("{i} Thread");
9             thread::sleep(D::from_millis(1));
10        }
11    });
12
13    for i in 1..5 {
14        println!("{i} Main");
15        thread::sleep(D::from_millis(1));
16    }
17    my_thread.join().unwrap();
18 }
```



```
1 #include <iostream>
2 #include <thread>
3 using namespace std;
4
5 int main() {
6     thread my_thread([] {
7         for (auto i = 1; i < 10; ++i){
8             cout << i << " Thread" << endl;
9             this_thread::sleep_for(1ms);
10        }
11    });
12
13    for (auto i = 1; i < 5; ++i){
14        cout << i << " Main" << endl;
15        this_thread::sleep_for(1ms);
16    }
17    my_thread.join();
18    return 0;
19 }
```

Poprawny koniec wykonania



```
> rustc naive_threads.rs && ./naive_threads
```

```
1. Main  
1. Thread  
2. Main  
2. Thread  
3. Thread  
3. Main  
4. Thread  
4. Main  
5. Thread  
6. Thread  
7. Thread  
8. Thread  
9. Thread
```

```
> g++ -Wall -Wextra -pedantic -Werror \  
-std=c++20 naive_threads.cpp && ./a.out
```

```
1. Thread  
1. Main  
2. Thread  
2. Main  
3. Thread  
3. Main  
4. Thread  
4. Main  
5. Thread  
6. Thread  
7. Thread  
8. Thread  
9. Thread
```

Sprytne wskaźniki - `std::unique_ptr`



```
1 {  
2   let val: u8 = 5;  
3   let boxed: Box<u8> = Box::new(val);  
4 }
```

Sprytne wskaźniki - `std::shared_ptr`

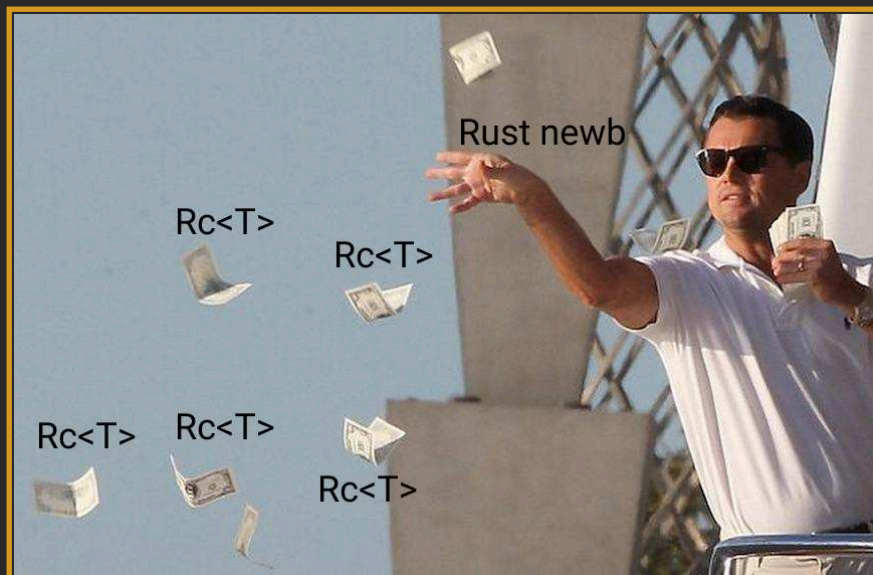


```
1 let rc_value = Rc::new(5_i32);
2 let second = Rc::clone(&rc_value);
3 let weak = Rc::downgrade(&rc_value); // Weak is an equivalent of std::weak_ptr
4 let maybe_strong_again = weak.upgrade(); // Option<Rc<_>>
5 println!("{second} {}", second.pow(3));
```

- **Rc** (reference counted) pozwala tylko na **niemutowalny** dostęp do obiektu
- Stworzenie zależności cyklicznej nie jest zabronione i może doprowadzić do wycieku pamięci
- Wartość alokowana na stacku

Interior mutability

- Wzorzec, który pozwala na przeniesienie reguł borrow checker'a z kompilacji na czas działania programu
- Złamanie reguł kończy się paniką programu
- Zaimplementowany za pomocą `unsafe`



Interior mutability - `RefCell` example

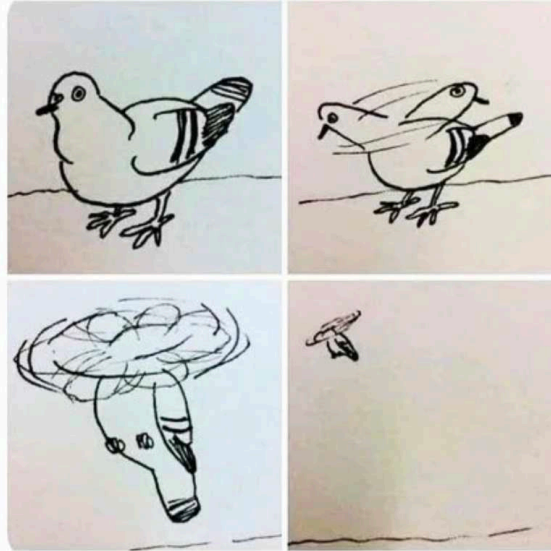


```
1 use std::cell::{RefCell, RefMut};
2 use std::collections::HashMap;
3 use std::rc::Rc;
4
5 fn main() {
6     let inflation: Rc<RefCell<_>> = Rc::new(RefCell::new(HashMap::new()));
7     // Create a new block to limit the scope of the dynamic borrow
8     {
9         let mut map: RefMut<_> = inflation.borrow_mut();
10        map.insert("slovakia", 14.);
11        map.insert("spain", 9.8);
12        map.insert("brazil", 10.06);
13        map.insert("poland", 16.5);
14    }
15
16    // Note that if we had not let the previous borrow of the inflation fall out
17    // of scope then the subsequent borrow would cause a dynamic thread panic.
18    // This is the major hazard of using `RefCell`.
19    let total: f32 = inflation.borrow().values().sum();
20    println!("{}", total);
21 }
```

When you first learn Rc<T>, RefCell<T> in Rust

```
println!("{}", daughter);  
println!("{}", f.  
    .unwrap()  
    .upgrade()  
    .unwrap()  
    .borrow()  
    .name);
```

When your program
is a complete mess,
but it does its job



Synchronizacja i jej brak



```
1 fn main() {
2     let mut x = 0;
3     let pendulum = || {
4         loop {
5             x += 1;
6             x -= 1;
7             assert_eq!(x, 0);
8         }
9     };
10
11     let t1 = thread::spawn(pendulum);
12     let t2 = thread::spawn(pendulum);
13
14     t1.join().unwrap();
15     t2.join().unwrap();
16 }
```



```
1 int main() {
2     auto x = 0;
3     auto pendulum = [&x]{
4         while (true) {
5             x += 1;
6             x -= 1;
7             assert(x == 0);
8         }
9     };
10
11     std::thread t1(pendulum);
12     std::thread t2(pendulum);
13
14     t1.join();
15     t2.join();
16 }
```

```

> rustc pendulum.rs
error[E0373]: closure may outlive the current function, but it borrows `x`, which is owned by the current function
  -> pendulum.rs:6:20
   |
6  |     let pendulum = || {
   |                   ^^ may outlive borrowed value `x`
7  |         loop {
8  |             x += 1;
   |             - `x` is borrowed here
   |
note: function requires argument type to outlive `static`
  -> pendulum.rs:14:14
   |
14 |     let t1 = thread::spawn(pendulum);
   |           ^^^^^^^^^^^^^^^^^^^^^^^^^
help: to force the closure to take ownership of `x` (and any other referenced variables), use the `move` keyword
6  |     let pendulum = move || {
   |                   +++++

error[E0382]: use of moved value: `pendulum`
  -> pendulum.rs:15:28
   |
14 |     let t1 = thread::spawn(pendulum);
   |                       _____ value moved here
15 |     let t2 = thread::spawn(pendulum);
   |                       ^^^^^^^^ value used here after move
   |
note: closure cannot be moved more than once as it is not `Copy` due to moving the variable `x` out of its environment
  -> pendulum.rs:8:13
   |
8  |         x += 1;
   |         ^

```

Bird driven programming



```
1 fn main() {
2     let x = Rc::new(RefCell::new(0));
3     let pendulum = move || {
4         let local_x = x.clone();
5         loop {
6             *local_x.borrow_mut() += 1;
7             *local_x.borrow_mut() -= 1;
8             assert_eq!(*local_x.borrow(), 0);
9         }
10    };
11
12    let t1 = thread::spawn(pendulum.clone());
13    let t2 = thread::spawn(pendulum);
14
15    t1.join().unwrap();
16    t2.join().unwrap();
17 }
```

```

> rustc pendulum_with_rc_refcell.rs
error[E0277]: `Rc<RefCell<i32>>` cannot be sent between threads safely
  → pendulum_with_rc_refcell.rs:17:28
   |
8  |         let pendulum = move || {
   |                        _____ within this `{closure@pendulum_with_rc_refcell.rs:8:20: 8:27}`
...
17 |         let t1 = thread::spawn(pendulum.clone());
   |                        ^^^^^^^^^^^^^^^^^^^^^^^^^ `Rc<RefCell<i32>>` cannot be sent between threads safely
   |
   |         required by a bound introduced by this call
   |
   = help: within `{closure@pendulum_with_rc_refcell.rs:8:20: 8:27}`, the trait `Send` is not implemented
for `Rc<RefCell<i32>>`
note: required because it's used within this closure
  → pendulum_with_rc_refcell.rs:8:20
   |
8  |         let pendulum = move || {
   |                        ^^^^^^^
note: required by a bound in `spawn`
  → /lib/rustlib/src/rust/library/std/src/thread/mod.rs:681:8
   |
678 | pub fn spawn<F, T>(f: F) → JoinHandle<T>
   |         _____ required by a bound in this function
...
681 |     F: Send + 'static,
   |         ^^^^ required by this bound in `spawn`

```

Send trait

- Jedna z dwóch cech bezpośrednio dotycząca współbieżności w języku
- Marker trait
- Oznacza typy, które mogą być przekazywane między wątkami
- Większość typów automatycznie implementuje `Send`:

Almost every Rust type is `Send`, but there are some exceptions, including `Rc<T>`

- Manualna implementacja `Send` wymaga `unsafe`
- Borrow checker wymusza poprawność tej cechy przez regułę unikalnego właściciela
- `Send` pozwala różnym wątkom na korzystanie z tego samego obiektu w **różnym czasie**, po transferze obiektu.

Atomic Reference Counted (**Arc**) +



```
1 fn main() {
2     let x = Arc::new(RefCell::new(0));
3     let pendulum = move || {
4         let local_x = x.clone();
5         loop {
6             *local_x.borrow_mut() += 1;
7             *local_x.borrow_mut() -= 1;
8             assert_eq!(*local_x.borrow(), 0);
9         }
10    };
11
12    let t1 = thread::spawn(pendulum.clone());
13    let t2 = thread::spawn(pendulum);
14
15    t1.join().unwrap();
16    t2.join().unwrap();
17 }
```

Atomic Reference Counted (**Arc**) +

```
> rustc pendulum_with_arc.rs
error[E0277]: `RefCell<i32>` cannot be shared between threads safely
  → pendulum_with_arc.rs:15:28
   |
15 |         let t1 = thread::spawn(pendulum.clone());
   |                                ^^^^^^^^^^^^^^^^^^^ `RefCell<i32>` cannot be shared between threads safely
   |                                |
   |                                required by a bound introduced by this call
   |
   = help: the trait `Sync` is not implemented for `RefCell<i32>`
   = note: if you want to do aliasing and mutation between multiple threads, use `std::sync::RwLock` instead
   = note: required for `Arc<RefCell<i32>>` to implement `Send`
note: required because it's used within this closure
  → pendulum_with_arc.rs:6:20
   |
 6 |         let pendulum = move || {
   |                        ^^^^^^^
note: required by a bound in `spawn`
  → /lib/rustlib/src/rust/library/std/src/thread/mod.rs:681:8
   |
678 | pub fn spawn<F, T>(f: F) → JoinHandle<T>
   |         ———— required by a bound in this function
   |
...
681 |     F: Send + 'static,
   |         ^^^^ required by this bound in `spawn`
```

Sync trait

- Druga po `Send` cecha bezpośrednio dotycząca współbieżności w języku
- Marker trait
- Oznacza typy, których referencje mogą być współdzielone między wątkami:

The precise definition is: a type `T` is `Sync` if and only if `&T` is `Send`. In other words, if there is no possibility of undefined behavior (including data races) when passing `&T` references between threads.

- Większość typów jest `Sync`:

Types that are not `Sync` are those that have “interior mutability” in a non-thread-safe form, such as `Cell` and `RefCell`. These types allow for mutation of their contents even through an immutable, shared reference.

`Sync` trait

- `Send` pozwala różnym wątkom na korzystanie z tego samego obiektu w **tym samym czasie**
- Manualna implementacja `Sync` wymaga `unsafe`



Wahadło z synchronizacją



```
1 int main() {
2     auto x = 0;
3     std::mutex my_mutex;
4     auto pendulum = [&x, &my_mutex]{
5         while (true) {
6             my_mutex.lock();
7             x += 1;
8             x -= 1;
9             assert(x == 0);
10            my_mutex.unlock();
11        }
12    };
13    std::thread t1(pendulum);
14    std::thread t2(pendulum);
15
16    t1.join();
17    t2.join();
18 }
```

Problemy z mutexem

Mutexes have a reputation for being difficult to use because you have to remember two rules:

- You must attempt to **acquire the lock** before using the data.
- When you're done with the data that the mutex guards, you must **unlock the data** so other threads can acquire the lock.

Rozwiązanie drugiego problemu - `std::scoped_lock`



```
1 int main() {
2     auto x = 0;
3     std::mutex my_mutex;
4     auto pendulum = [&x, &my_mutex]{
5         while (true) {
6             std::scoped_lock guard(my_mutex);
7             x += 1;
8             x -= 1;
9             assert(x == 0);
10        } // guard dropped here
11    };
12    std::thread t(pendulum);
13    std::thread t2(pendulum);
14
15    t1.join();
16    t2.join();
17 }
```

Mutex w Rustcie



```
1 let my_mutex = Mutex::new(5);  
2 {  
3     let mut number = my_mutex.lock().unwrap();  
4     *number += 1;  
5 } // guard dropped here
```

Mutex w Rustcie



```
1 use std::{sync::Mutex, thread};
2
3 fn main() {
4     let my_mutex = Arc::new(Mutex::new(0));
5     let pendulum = move || {
6         loop {
7             let mut x = my_mutex.lock().unwrap();
8             *x += 1;
9             *x -= 1;
10            assert_eq!(*x, 0);
11        }
12    };
13
14    let t1 = thread::spawn(pendulum.clone());
15    let t2 = thread::spawn(pendulum);
16
17    t1.join().unwrap();
18    t2.join().unwrap();
19 }
```

Z czego wynika podział na **Mutex** i **Arc**?



```
1 type KeyBindings = HashMap<String, String>;
2 type Items = Vec<String>;
3
4 #[derive(Default)]
5 struct Player {
6     id: String,
7     score: AtomicUsize,
8     items: Mutex<Items>,
9     keybindings: Mutex<KeyBindings>,
10 }
11
12 fn main() {
13     let player = Arc::new(Player::default());
14     let modify_player = move || {
15         println!("{}", player.id);
16         player.score.fetch_add(1, Ordering::SeqCst);
17         let mut items = player.items.lock().unwrap();
18         items.push("Sword".into());
19     };
20     let my_thread = thread::spawn(modify_player);
21     my_thread.join().unwrap();
22 }
```


Scoped thread - “lokalne” wątki



```
1 use std::sync::Mutex;
2 use std::thread;
3
4 fn main() {
5     let my_mutex = Mutex::new(0);
6     thread::scope(|s| {
7         let pendulum = || {
8             loop {
9                 let mut x = my_mutex.lock().unwrap();
10                *x += 1;
11                *x -= 1;
12                assert_eq!(*x, 0);
13            }
14        };
15
16        let t1 = s.spawn(pendulum);
17        let t2 = s.spawn(pendulum);
18
19        t1.join().unwrap();
20        t2.join().unwrap();
21    });
22 }
```

Data race vs race condition



```
1 use std::thread;
2 use std::sync::atomic::{AtomicUsize, Ordering};
3 use std::sync::Arc;
4
5 let data = vec![1, 2, 3, 4];
6 let idx = Arc::new(AtomicUsize::new(0));
7 let other_idx = idx.clone();
8
9 thread::spawn(move || {
10     other_idx.fetch_add(10, Ordering::SeqCst);
11 });
12
13 println!("{}", data[idx.load(Ordering::SeqCst)]);
```



```
1 let behind1 = Arc::new(Mutex::new(String::new()));
2 let ticks1 = Arc::new(RwLock::new(0));
3 let behind2 = behind1.clone();
4 let ticks2 = ticks1.clone();
5
6 let thread1 = thread::spawn(move || {
7     let mut behind = behind1.lock().unwrap();
8     *behind += "thread_1";
9     thread::sleep(Duration::from_millis(200));
10    let mut ticks = ticks1.write().unwrap();
11    *ticks += 1;
12 });
13 let thread2 = thread::spawn(move || {
14     let mut ticks = ticks2.write().unwrap();
15     *ticks += 1;
16     thread::sleep(Duration::from_millis(200));
17     let mut behind = behind2.lock().unwrap();
18     *behind += "thread_2";
19 });
20
21 thread1.join().unwrap();
22 thread2.join().unwrap();
```



Do not communicate by sharing memory; instead, share memory by communicating



```
1 use std::sync::mpsc;
2 use std::thread;
3
4 fn main() {
5     let (tx, rx) = mpsc::channel();
6
7     thread::spawn(move || {
8         let message = String::from("hi");
9         tx.send(message).unwrap();
10    });
11
12    let received = rx.recv().unwrap();
13    println!("Got: {received}");
14 }
```

Kanał - wiele wiadomości



```
1 use std::{sync::mpsc, thread, time::Duration};
2
3 fn main() {
4     let (tx, rx) = mpsc::channel();
5
6     thread::spawn(move || {
7         let messages = vec![
8             String::from("hi"),
9             String::from("from"),
10            String::from("the"),
11            String::from("thread"),
12        ];
13
14        for message in messages {
15            tx.send(message).unwrap();
16            thread::sleep(Duration::from_secs(1));
17        }
18    });
19
20    for received in rx {
21        println!("Got: {received}");
22    }
23 }
```

Embarrassingly parallel problems



```
1 fn sum_input(input: &[&str]) → usize {  
2     input.iter()  
3         .map(|s| s.parse::<usize>())  
4         .filter(|r| r.is_ok())  
5         .map(|r| r.unwrap())  
6         .sum()  
7 }  
8  
9 fn main() {  
10     let input = ["1", "two", "NaN", "4"];  
11     let sum = sum_input(&input);  
12     assert_eq!(sum, 5)  
13 }
```

Rayon parallel iterators



```
1 fn sum_input(input: &[&str]) → usize {
2     input.iter()
3         .map(|s| s.parse::<usize>())
4         .filter(Result::is_ok)
5         .map(Result::unwrap)
6         .sum()
7 }
8
9 fn main() {
10     let input = ["1", "two", "NaN", "4"];
11     let sum = sum_input(&input);
12     assert_eq!(sum, 5)
13 }
```

```
1 use rayon::prelude::*;
2
3 fn sum_input(input: &[&str]) → usize {
4     input.par_iter()
5         .map(|s| s.parse::<usize>())
6         .filter(Result::is_ok)
7         .map(Result::unwrap)
8         .sum()
9 }
10
11 fn main() {
12     let input = ["1", "two", "NaN", "4"];
13     let sum = sum_input(&input);
14     assert_eq!(sum, 5)
15 }
```


Czy w C++ można napisać ekwiwalent Rust'owego mutex'a?

Jeśli tak napisz w jaki sposób (może być kod/pseudokod/opis słowny),
jeśli nie napisz dlaczego i podaj przykład niepoprawnego działania.



```
1 #include <boost/thread/synchronized_value.hpp>
2 #include <string>
3 #include <iostream>
4
5 typedef boost::synchronized_value<std::string> Mutex;
6 typedef boost::strict_lock_ptr<std::string> Guard;
7
8 auto& init_path(Mutex& path) {
9     Guard guard = path.synchronize();
10
11     if (guard->empty() || (*guard->rbegin() != '/')) {
12         *guard = "/my_path";
13     }
14     std::cout << *guard << std::endl;
15     return *guard;
16 }
17
18 int main () {
19     Mutex my_path;
20     auto& path_ref = init_path(my_path);
21     path_ref += ":0";
22     for (auto& letter : path_ref) {
23         std::cout << letter << std::endl;
24     }
25     Guard guard = my_path.synchronize();
26     std::cout << *guard << std::endl;
27     return 0;
28 }
```



```
> g++ -Wall -Wextra -pedantic -Werror -fsanitize=undefined -std=c++20 boost_mutex.cpp && ./a.out  
/my_path  
/  
m  
y  
-  
p  
a  
t  
h  
:  
0  
/my_path:0
```



```
1 use std::{sync::Mutex, ops::DerefMut};
2
3 fn init_path(path: &Mutex<String>) → &mut String {
4     let mut guard = path.lock().unwrap();
5     if guard.is_empty() || !guard.starts_with("/") {
6         *guard = "/my_path".to_string();
7     }
8     guard.deref_mut()
9 }
10
11 fn main() {
12     let my_path = Mutex::new(String::from("hia!"));
13     let path_ref = init_path(&my_path);
14     path_ref.push_str(":0");
15     for letter in path_ref.chars() {
16         println!("{letter}");
17     }
18     let mut guard = my_path.lock().unwrap();
19     println!("{guard}");
20 }
```



```
> rustc path_mutex.rs
error[E0515]: cannot return value referencing local variable `guard`
  → path_mutex.rs:8:5
   |
8  |     guard.deref_mut()
   |     ^^^^^^^^^^^^^^^
   |
   | returns a value referencing data owned by the current function
   | `guard` is borrowed here
```

C++



Rust

