Unleashing the Power of Rust: Building Safe, Fast, and Reliable Software for the Future

The programming language that empowers everyone to become a systems programmer.

Overview

Rust is a modern systems programming language that is quickly gaining popularity due to its ability to build safe, fast, and reliable software.

01. Memory Management Mastery: Ownership and Borrowing
02. Performance Amplified: Rust’s Optimization Techniques
03. Conquering Real-World Challenges: Applications of Rust
04. Conclusion: Equipped for the Future with Rust

Memory Management Mastery

Memory management has been a notorious challenge in system programming languages. Rust has revolutionized this landscape with its ownership and borrowing system. We'll take a deep dive into the concept of ownership, understanding how it empowers developers to write code that's both memory-efficient and free from common bugs like null pointer exceptions and data races. Borrowing, an integral part of Rust's memory model, will be demystified as we unravel how it enables multiple components of a program to interact seamlessly while adhering to the principle of data safety.

Memory Management Mastery: Memory leaks

A memory leak in a program occurs when the program unintentionally allocates memory (usually on the heap) during its execution but fails to release or deallocate that memory properly before the program terminates. As a result, the memory that was allocated remains reserved and unavailable for other parts of the program or other processes, leading to a gradual increase in memory usage over time.

Memory Management Mastery: Memory leaks

A memory leak in a program occurs when the program unintentionally allocates memory (usually on the heap) during its execution but fails to release or deallocate that memory properly before the program terminates. As a result, the memory that was allocated remains reserved and unavailable for other parts of the program or other processes, leading to a gradual increase in memory usage over time.

main.cpp > ...

```cpp
#include <iostream>
#include <cstdlib>

int main() {
    int* data = new int[1000]; // Allocating memory

    return 0;
}

```

Memory Management Mastery: Memory leaks

A memory leak in a program occurs when the program unintentionally allocates memory (usually on the heap) during its execution but fails to release or deallocate that memory properly before the program terminates. As a result, the memory that was allocated remains reserved and unavailable for other parts of the program or other processes, leading to a gradual increase in memory usage over time.

main.cpp > main()

```cpp
#include <iostream>
#include <cstdlib>

int main() {
    int* data = new int[1000]; // Allocating memory
    delete[] data;
    return 0;
}

```

Memory Management Mastery: Memory leaks

Consequences of memory leaks

- Reduced available memory
- Slower execution
- Program crashes
- Resource Saturation
- Maintenance Challenge

Memory Management Mastery: Garbage collector

Who came to the rescue?

Memory Management Mastery: Garbage collector

The Garbage Collector!!!

Memory Management Mastery: Garbage collector

Does the dirty work of memory management.

What is a Garbage collector

A garbage collector is a component of many programming languages and runtime environments that automates the process of memory management. Its primary purpose is to automatically reclaim memory that is no longer needed by the program, specifically memory that has been allocated for objects or data structures that are no longer accessible or in use.

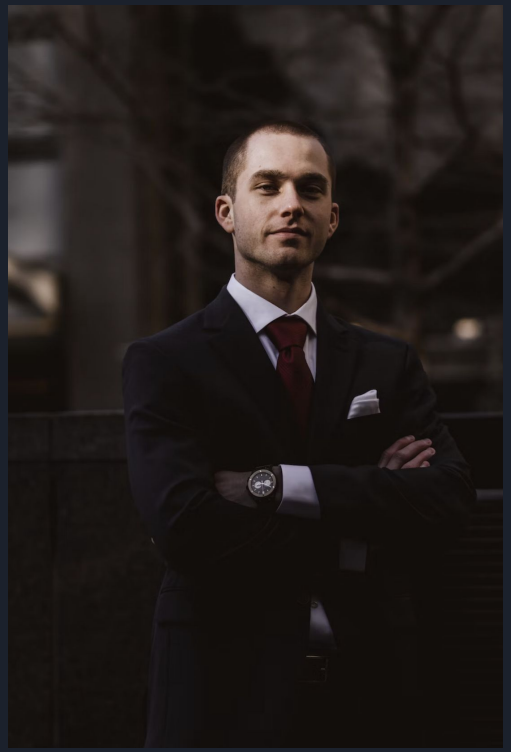
The Garbage Collector: Drawbacks

- Performance Overhead
- Predictability
- Resource usage

Not ideal for low latency use cases

In comes Rust

WITHOUT a garbage collector

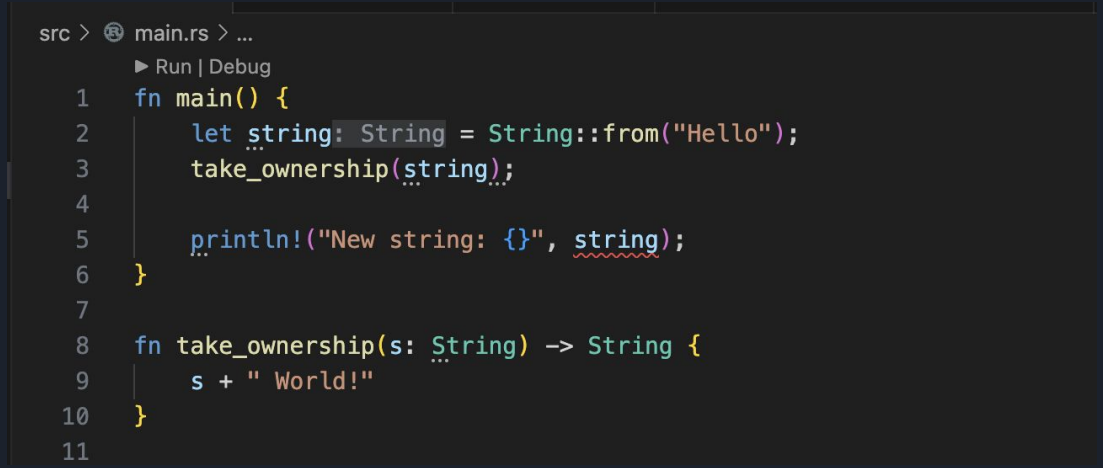


How does Rust handle memory management?

How does Rust handle memory management?

- Ownership and Borrowing
- Lifetimes
- Ownership Transfers and Moves

Ownership and Borrowing



Ownership and Borrowing

src > main.rs > ...

▶ Run | Debug

```rust
fn main() {
    let string: String = String::from("Hello");
    take_ownership(string);

    println!("New string: {}", string);
}

fn take_ownership(s: String) -> String {
    s + " World!"
}

```

```
error[E0382]: borrow of moved value: `string`
 --> src/main.rs:5:32
  |
2 |     let string = String::from("Hello");
  |         ------ move occurs because `string` has type `String`, which does not implement the `Copy` trait
3 |     take_ownership(string);
  |                    ------ value moved here
4 |
5 |     println!("New string: {}", string);
  |                                ^^^^^^ value borrowed here after move
  |
```

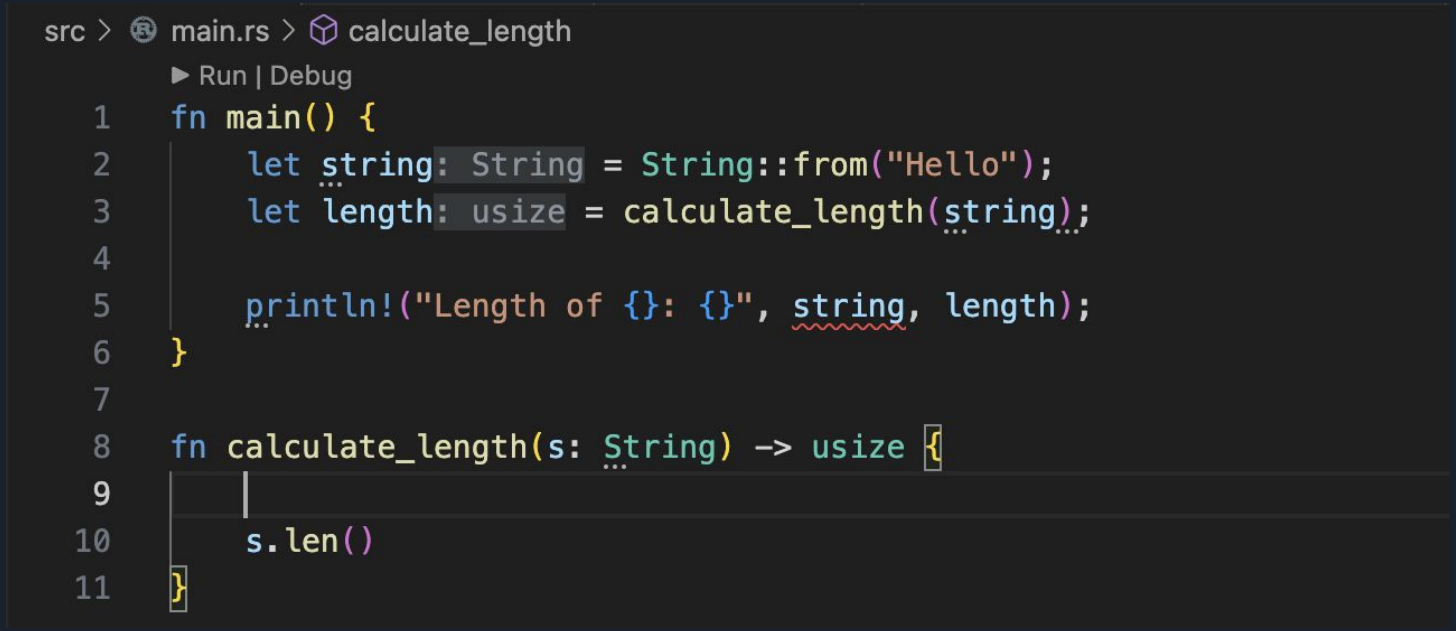
Ownership and Borrowing

src > main.rs > main

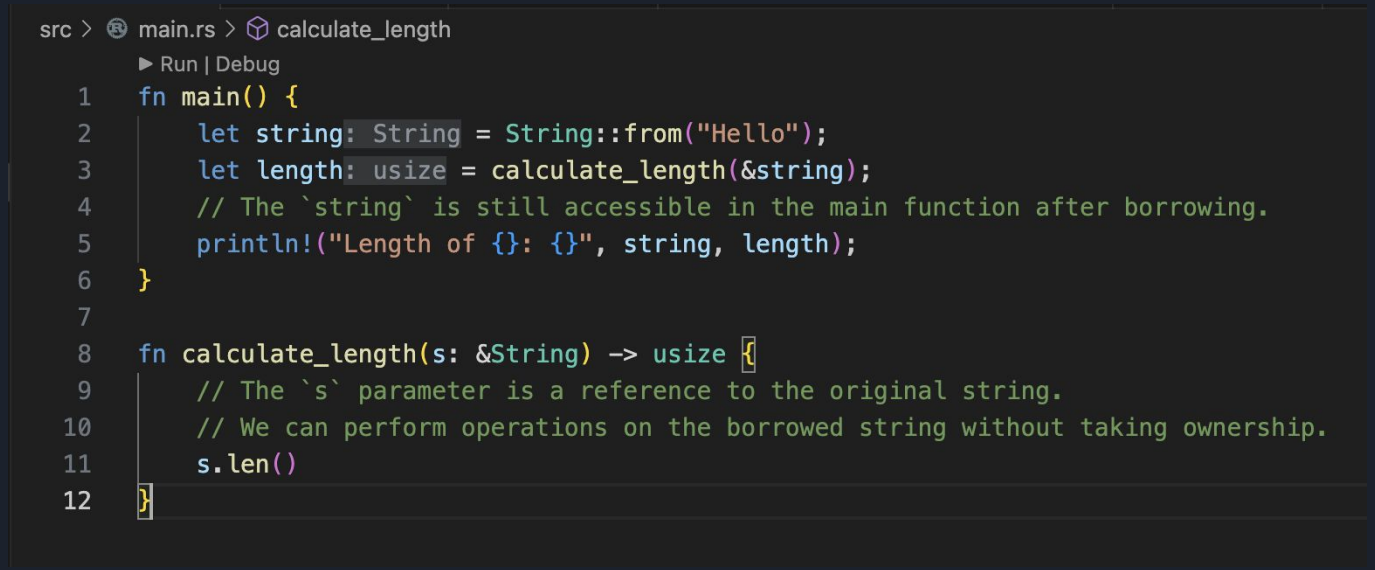
▶ Run | Debug

```rust
fn main() {
    let string: String = String::from("Hello");

    let string2: String = take_ownership(string); // string is moved

    println!("New string: {}", string2);
}

fn take_ownership(s: String) -> String {
    s + " World!"

    // s is dropped
}
```

Ownership and Borrowing



Ownership and Borrowing



TS main.ts > cannotModifyUser

```ts
class User {
    firstName: string
    lastName: string
    age: number

    constructor(firstName: string, lastName: string, age: number) {
        this.firstName = firstName
        this.lastName = lastName
        this.age = age
    }

    setAge(age: number) {
        this.age = age
    }
}

function main() {
    const user = new User("David", "Oyinbo", 12);

    cannotModifyUser(user);

    console.log("User: ", user);
}//end main function

function cannotModifyUser(user: User) {
    // Does some work
}//end method cannotModifyUser

main();
```

```
davidoyinbo@davidoyinbo conf42 % ts-node main.ts
User:  User { firstName: 'David', lastName: 'Oyinbo', age: 12 }
davidoyinbo@davidoyinbo conf42 %
```


main.ts > cannotModifyUser
class User {
firstName: string
lastName: string
age: number
constructor(firstName: string, lastName: string, age: number) {
this.firstName = firstName
this.lastName = lastName
this.age = age
}
setAge(age: number) {
this.age = age
}
}
function main() {
const user = new User("David", "Oyinbo", 12);
cannotModifyUser(user);
console.log("User: ", user);
}//end main function
function cannotMod
setAge(age: number): void
// Does some w
user.setAge(99)
}//end method cannotModifyUser
main();

TS main.ts > main

```ts
class User {
    firstName: string
    lastName: string
    age: number

    constructor(firstName: string, lastName: string, age: number) {
        this.firstName = firstName
        this.lastName = lastName
        this.age = age
    }

    setAge(age: number) {
        this.age = age
    }
}

function main() {
    const user = new User("David", "Oyinbo", 12);
    Object.freeze(user)
    cannotModifyUser(user);

    console.log("User: ", user);
}//end main function

function cannotModifyUser(user: User) {
    // Does some work
    user.setAge(99)
}//end method cannotModifyUser

main();
```

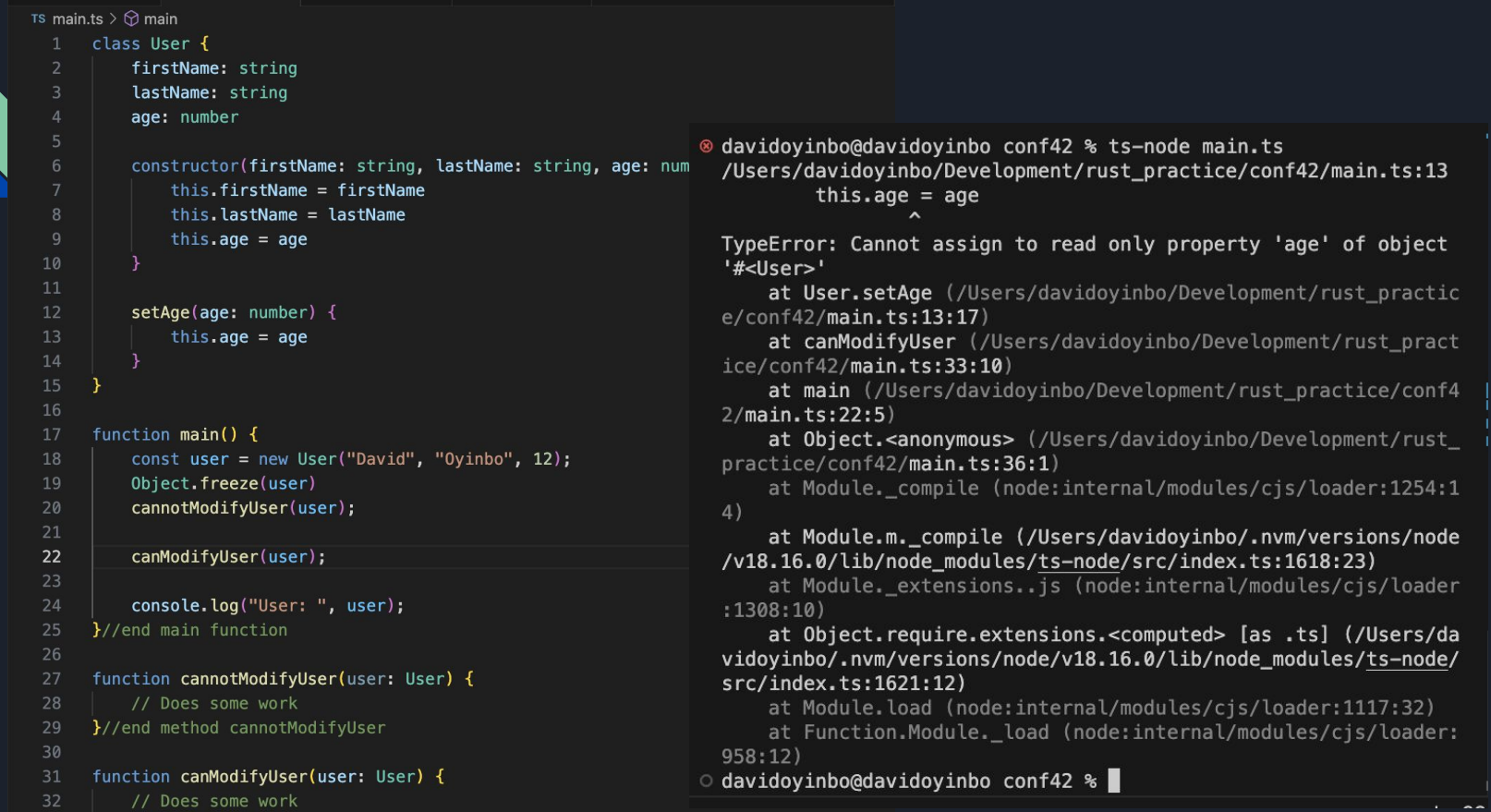
TS main.ts > main

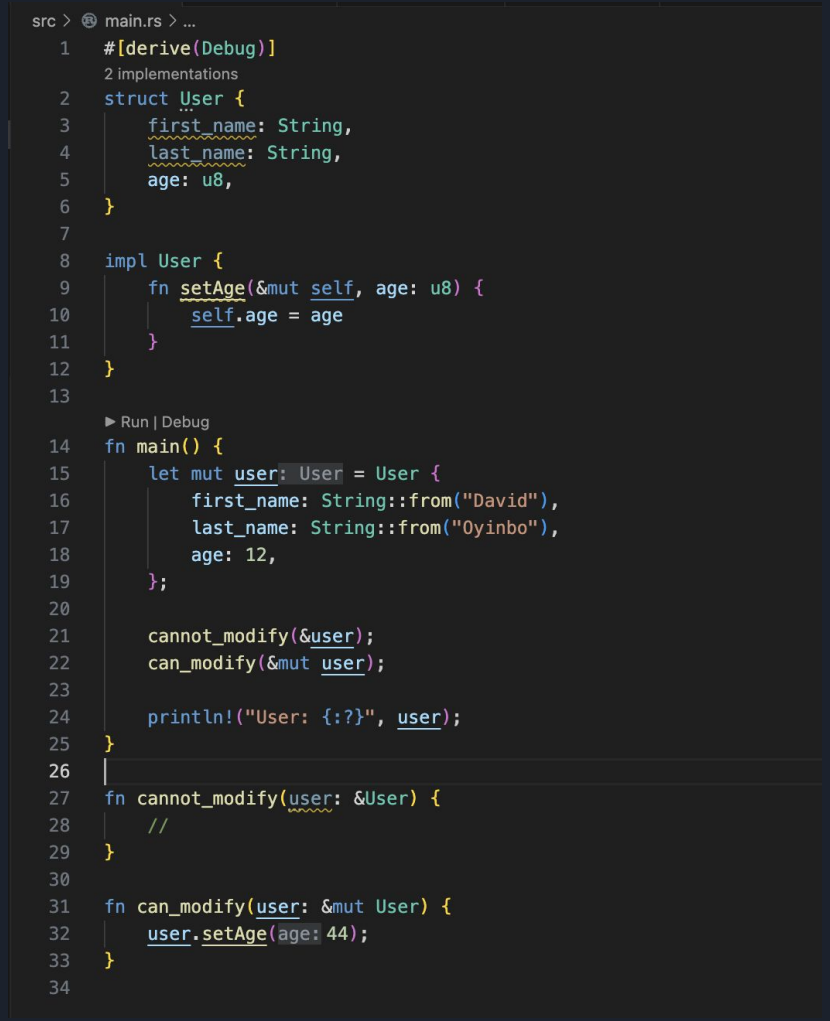
```ts
class User {
    firstName: string
    lastName: string
    age: number

    constructor(firstName: string, lastName: string, age: number) {
        this.firstName = firstName
        this.lastName = lastName
        this.age = age
    }

    setAge(age: number) {
        this.age = age
    }
}

function main() {
    const user = new User("David", "Oyinbo", 12);
    Object.freeze(user)
    cannotModifyUser(user);

    console.log("User: ", user);
}//end main function

function cannotModifyUser(user: User) {
    // Does some work
    user.setAge(99)
}//end method cannotModifyUser

main();
```

```
davidoyinbo@davidoyinbo conf42 % ts-node main.ts
/Users/davidoyinbo/Development/rust_practice/conf42/main.ts:13
        this.age = age
                 ^
TypeError: Cannot assign to read only property 'age' of object
'#<User>'
    at User.setAge (/Users/davidoyinbo/Development/rust_practic
e/conf42/main.ts:13:17)
    at cannotModifyUser (/Users/davidoyinbo/Development/rust_pr
actice/conf42/main.ts:27:10)
    at main (/Users/davidoyinbo/Development/rust_practice/conf4
2/main.ts:20:5)
    at Object.<anonymous> (/Users/davidoyinbo/Development/rust_
practice/conf42/main.ts:30:1)
    at Module._compile (node:internal/modules/cjs/loader:1254:1
4)
    at Module.m._compile (/Users/davidoyinbo/.nvm/versions/node
/v18.16.0/lib/node_modules/ts-node/src/index.ts:1618:23)
    at Module._extensions..js (node:internal/modules/cjs/loader
:1308:10)
    at Object.require.extensions.<computed> [as .ts] (/Users/da
vidoyinbo/.nvm/versions/node/v18.16.0/lib/node_modules/ts-node/
src/index.ts:1621:12)
    at Module.load (node:internal/modules/cjs/loader:1117:32)
    at Function.Module._load (node:internal/modules/cjs/loader:
958:12)
```

TS main.ts > main

```ts
class User {
    firstName: string
    lastName: string
    age: number

    constructor(firstName: string, lastName: string, age: number) {
        this.firstName = firstName
        this.lastName = lastName
        this.age = age
    }

    setAge(age: number) {
        this.age = age
    }
}

function main() {
    const user = new User("David", "Oyinbo", 12);
    Object.freeze(user)
    cannotModifyUser(user);

    canModifyUser(user);

    console.log("User: ", user);
}//end main function

function cannotModifyUser(user: User) {
    // Does some work
}//end method cannotModifyUser

function canModifyUser(user: User) {
    // Does some work
    user.setAge(44)
}//end method cannotModifyUser

main();
```


main.ts > main
class User {
firstName: string
lastName: string
age: number
constructor(firstName: string, lastName: string, age: num
this.firstName = firstName
this.lastName = lastName
this.age = age
}
setAge(age: number) {
this.age = age
}
}
function main() {
const user = new User("David", "Oyinbo", 12);
Object.freeze(user)
cannotModifyUser(user);
canModifyUser(user);
console.log("User: ", user);
}//end main function
function cannotModifyUser(user: User) {
// Does some work
}//end method cannotModifyUser
function canModifyUser(user: User) {
// Does some work
user.setAge(44)
}//end method cannotModifyUser
main();
davidoyinbo@davidoyinbo conf42 % ts-node main.ts
/Users/davidoyinbo/Development/rust_practice/conf42/main.ts:13
this.age = age
^
TypeError: Cannot assign to read only property 'age' of object
'#<User>'
at User.setAge (/Users/davidoyinbo/Development/rust_practic
e/conf42/main.ts:13:17)
at canModifyUser (/Users/davidoyinbo/Development/rust_pract
ice/conf42/main.ts:33:10)
at main (/Users/davidoyinbo/Development/rust_practice/conf4
2/main.ts:22:5)
at Object.<anonymous> (/Users/davidoyinbo/Development/rust_
practice/conf42/main.ts:36:1)
at Module._compile (node:internal/modules/cjs/loader:1254:1
4)
at Module.m._compile (/Users/davidoyinbo/.nvm/versions/node
/v18.16.0/lib/node_modules/ts-node/src/index.ts:1618:23)
at Module._extensions..js (node:internal/modules/cjs/loader
:1308:10)
at Object.require.extensions.<computed> [as .ts] (/Users/da
vidoyinbo/.nvm/versions/node/v18.16.0/lib/node_modules/ts-node/
src/index.ts:1621:12)
at Module.load (node:internal/modules/cjs/loader:1117:32)
at Function.Module._load (node:internal/modules/cjs/loader:
958:12)
davidoyinbo@davidoyinbo conf42 %


src > main.rs > ...
#[derive(Debug)]
2 implementations
struct User {
first_name: String,
last_name: String,
age: u8,
}
impl User {
fn setAge(&mut self, age: u8) {
self.age = age
}
}
▶ Run | Debug
fn main() {
let mut user: User = User {
first_name: String::from("David"),
last_name: String::from("Oyinbo"),
age: 12,
};
cannot_modify(&user);
can_modify(&mut user);
println!("User: {:?}", user);
}
fn cannot_modify(user: &User) {
//
}
fn can_modify(user: &mut User) {
user.setAge(age: 44);
}

src > main.rs > cannot_modify

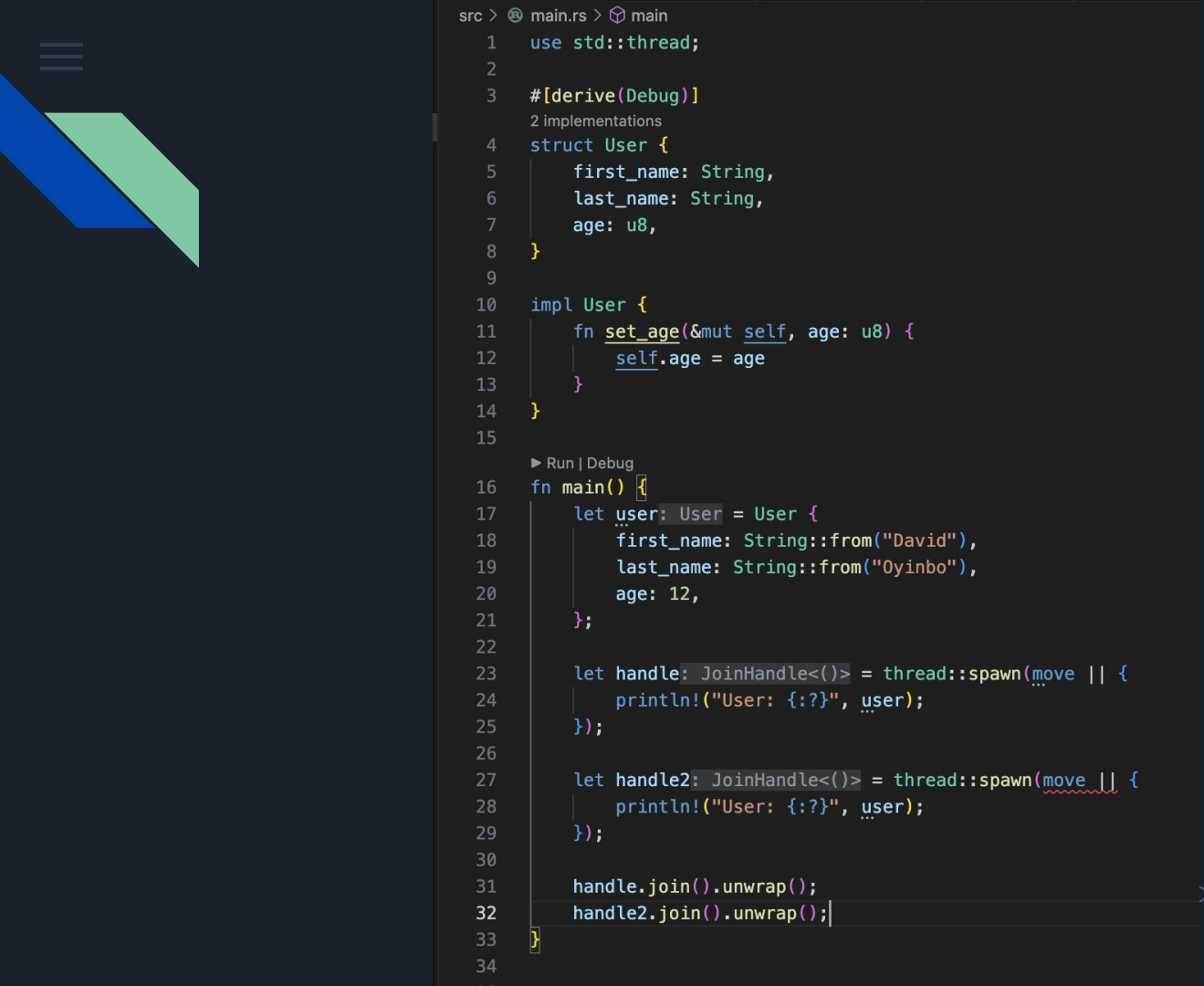
```rust
#[derive(Debug)]
2 implementations
struct User {
    first_name: String,
    last_name: String,
    age: u8,
}

impl User {
    fn setAge(&mut self, age: u8) {
        self.age = age
    }
}

▶ Run | Debug
fn main() {
    let mut user: User = User {
        first_name: String::from("David"),
        last_name: String::from("Oyinbo"),
        age: 12,
    };

    cannot_modify(&user);
    can_modify(&mut user);

    println!("User: {:?}", user);
}

fn cannot_modify(user: &User) {
    user.setAge(age: 49);
}

fn can_modify(user: &mut User) {
    user.setAge(age: 44);
}
```

```
davidoyinbo@davidoyinbo conf42 % cargo run
   Compiling conf42 v0.1.0 (/Users/davidoyinbo/Development/rust_practice/conf42)
error[E0596]: cannot borrow `*user` as mutable, as it is behind a `&` reference
  --> src/main.rs:28:5
   |
28 |     user.setAge(49);
   |     ^^^^^^^^^^^^^^^ `user` is a `&` reference, so the data it refers to cannot be borrowed as mutable
   |
help: consider changing this to be a mutable reference
   |
27 | fn cannot_modify(user: &mut User) {
   |                        ~~~~~~~~~

For more information about this error, try `rustc --explain E0596`.
error: could not compile `conf42` (bin "conf42") due to previous error
```

Ownership and Borrowing: (Multi-Threading)

src > main.rs > ...

```rust
use std::thread;

#[derive(Debug)]
2 implementations
struct User {
    first_name: String,
    last_name: String,
    age: u8,
}

impl User {
    fn set_age(&mut self, age: u8) {
        self.age = age
    }
}

▶ Run | Debug
fn main() {
    let user: User = User {
        first_name: String::from("David"),
        last_name: String::from("Oyinbo"),
        age: 12,
    };

    let handle: JoinHandle<()> = thread::spawn(move || {
        println!("User: {:?}", user);
    });

    handle.join().unwrap();
}
```



Ownership and Borrowing: (Multi-Threading)

src > main.rs > main

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

#[derive(Debug)]
2 implementations
struct User {
    first_name: String,
    last_name: String,
    age: u8,
}

impl User {
    fn set_age(&mut self, age: u8) {
        self.age = age
    }
}

▶ Run | Debug
fn main() {
    let user: User = User {
        first_name: String::from("David"),
        last_name: String::from("Oyinbo"),
        age: 12,
    };

    let user: Arc<User> = Arc::new(data: user);

    let user1: Arc<User> = user.clone();
    let handle: JoinHandle<()> = thread::spawn(move || {
        println!("User: {:?}", user1);
    });

    let user2: Arc<User> = user.clone();
    let handle2: JoinHandle<()> = thread::spawn(move || {
        println!("User: {:?}", user2.clone());
    });

    handle.join().unwrap();
    handle2.join().unwrap();
}
```

Performance Amplified: Rust's Optimization Techniques

Rust is a systems programming language that emphasizes safety, performance, and concurrency. It provides a variety of optimization techniques to help developers write efficient code without sacrificing safety.

Rust's Optimization Techniques: Zero Cost Abstraction

Rust provides high-level abstractions without incurring any runtime overhead. This is achieved through a combination of compile-time checks and optimizations. For example, Rust's ownership and borrowing system allows the compiler to ensure memory safety without introducing runtime garbage collection or reference counting overhead.

Rust's Optimization Techniques: Inline Functions

The #[inline] attribute in Rust is a compiler directive that tells the compiler to inline a function at the call site. This means that the compiler will copy the body of the function into the caller's code, instead of calling the function as a separate entity. This can improve performance by **eliminating the overhead of the function call, such as the stack frame setup and tear down.**

The compiler will not always inline a function that is marked with the #[inline] attribute. The compiler will make a decision based on a number of factors, such as the size of the function, the optimization level, and the target architecture.

Rust's Optimization Techniques: Inline Functions

src > main.rs > factorial

```rust
#[inline]
fn factorial(n: u32) -> u32 {
    if n == 0 {
        return 1;
    }
    return n * factorial(n - 1);
}

▶ Run | Debug
fn main() {
    println!("{}", factorial(5));
}
```

Rust's Optimization Techniques: Constant Propagation

Constant propagation is an optimization technique that the Rust compiler uses to replace expressions that evaluate to constants with their actual values. This can improve performance by eliminating the need to evaluate the expressions at runtime.

The Rust compiler can propagate constants through expressions in a number of ways. For example, if an expression contains a variable that has been assigned a constant value, the compiler can replace the variable with its value. The compiler can also propagate constants through arithmetic operations, such as addition and multiplication.

Rust's Optimization Techniques: const & static

Constants:

In Rust, constants are defined using the const keyword and must have a fixed, compile-time evaluable value. They are usually used for values that are known at compile-time and won't change during program execution. Since constants are evaluated at compile-time, the compiler can substitute their values directly into the expressions where they are used, reducing the need for runtime calculations.

src > main.rs > ...

▶ Run | Debug

```rust
fn main() {
    const PI: f64 = 3.14159;
    let radius: f64 = 5.0;
    let circumference: f64 = 2.0 * PI * radius; // Constant propagation here
}
```

Rust's Optimization Techniques: const & static

Static Variables:

Static variables are values that are allocated once and persist throughout the entire program's execution. They are defined using the static keyword and can also contribute to constant propagation optimizations.

src > main.rs > main

```rust
static CONFIG: u32 = 42;
▶ Run | Debug
fn main() {
    let value: u32 = CONFIG * 2; // Constant propagation here
    println!("Value: {}", value)
}

```

Rust's Optimization Techniques: const & static

By utilizing constants and static variables, Rust's compiler can perform various optimizations, including constant propagation, which can lead to more efficient generated code. These optimizations can eliminate unnecessary runtime calculations and improve the overall performance of Rust programs.

Conquering Real-World Challenges: Applications of Rust

Operating Systems:

Rust's emphasis on memory safety and absence of null pointers makes it an attractive choice for building operating systems. The ability to write low-level systems code without sacrificing safety is a game-changer. Notable projects include Redox OS and Tock OS. Microsoft is also rewriting core window libraries in Rust

Conquering Real-World Challenges: Applications of Rust

Web Server Development:

Rust's performance and safety features make it suitable for building high-performance and secure web servers. Projects like Actix and Rocket provide frameworks that leverage Rust's concurrency model and memory safety to develop robust web applications.

Conquering Real-World Challenges: Applications of Rust

Other Applications of rust

- Databases
- Game Development
- Embedded Systems
- Blockchain and Cryptocurrency
- Networking

Thank you!

Remember, with great power comes great responsibility