Fakultät Informatik

Sicherheitsaspekte Rust

Bericht im Studiengang Informatik

vorgelegt von
Robin Rosner

Matrikelnummer 362 5303

Abstract

Dieser Bericht behandelt die zentralen Sicherheitsaspekte der Programmiersprache Rust, die entwickelt wurde, um sichere und effiziente Software zu schreiben. Rust kombiniert leistungsstarke Kontrolle über Systemressourcen mit modernen Spracheigenschaften, die viele häufige Sicherheitsprobleme in der Softwareentwicklung verhindern [1].

Zunächst wird die statische Typisierung von Rust erläutert, die durch strikte Typenkontrolle zur Vermeidung typischer Fehler beiträgt. Anschließend wird gezeigt, welche Möglichkeiten Rust bietet, mit Integer-Überläufen umzugehen. Ein weiterer Schwerpunkt liegt auf den Zero-Cost-Abstraktionen, die es Entwicklern ermöglichen, effizienten und dennoch sicheren Code zu schreiben.

Der Bericht beleuchtet auch Rusts Ansätze zur Fehlerbehandlung, die auf das Konzept der Ausnahmen verzichten und stattdessen explizite Fehlerwerte nutzen.

Besonders hervorzuheben ist Rusts Besitz- und Ausleihmodell, das durch strikte Regeln für Speicherzugriffe und Lebensdauern von Objekten die Sicherheit und Stabilität des Codes verbessert.

Ergänzend dazu wird das Lebensdauer-Tracking vorgestellt, das durch präzise Kontrolle über die Lebensdauer von Referenzen die Gefahr von hängenden Referenzen minimiert.

Schließlich wird Rusts Ansatz zur sicheren Nebenläufigkeit untersucht, der Datenrennen (data races) und damit verbundene Sicherheitsprobleme verhindert.

Dieser Bericht gibt einen Überblick über die wichtigsten Spracheigenschaften von Rust und zeigt anhand von Beispielen, wie diese in der Praxis angewendet werden können.

Inhaltsverzeichnis

1 Statische Typen

Rust ist eine statisch typisierte Programmiersprache, was bedeutet, dass die Typen aller Variablen zur Kompilierzeit bekannt sein müssen. Dies ermöglicht dem Compiler, viele Fehler frühzeitig zu erkennen und zu verhindern. Unsichere Operationen wie Dereferenzierungen von Nullzeigern, ungesicherte Typumwandlungen, Typfehler und Typinkompatibilitäten werden so vermieden.

Listing 1.1: i32 in i64

```
pub fn i32_in_i64() {
    let smol: i32 = 128;

    // panic: i32 will not get converted to i64
    let big: i64 = smol;

    // explicit conversion from i32 to i64
    let big: i64 = smol.into();

    println!("i32 in i64: {}", big);
}
```

Beispiel 1.1 zeigt den Versuch, einen `i32`-Wert direkt in einen `i64`-Wert zu konvertieren, was nicht erlaubt ist. Stattdessen kann die Methode `into()` verwendet werden, um explizit eine Typenkonvertierung vorzunehmen. Rust führt keine impliziten Konvertierungen durch; der Programmierer muss an dieser Stelle explizit sein [2, Kapitel 1, Types].

Listing 1.2: Signiertheit

```
pub fn signdness() {
    let unsigned_val: u32 = 150;
    let signed_val: i32 = -100;

    // will panic
    let signed: i32 = unsigned_val;
    let unsigned: u32 = signed_val;

    // explicit conversion from unsigned to signed
    let signed_from_unsigned: i32 = unsigned_val as i32;
    println!("Signed from Unsigned: {}", signed_from_unsigned);

    // explicit conversion from signed to unsigned
    // has a logial error in this context and overflows
    let unsigned_from_signed: u32 = signed_val as u32;
    println!("Unsigned from Signed: {}", unsigned_from_signed);
    }
```

Das Beispiel 1.2 zeigt die Funktion `signdness()`, die mögliche Probleme bei der Konvertierung zwischen signierten und unsignierten Typen darstellt. Direkte Konvertierungen können zu logischen Fehlern oder Überläufen führen. Daher ist es wichtig, explizite und sichere Methoden zu verwenden. er Programmierer muss hierbei explizit eine Konvertierung veranlassen.

In vielen dynamisch typisierten Sprachen wie Python oder JavaScript werden Typfehler erst zur Laufzeit entdeckt, was zu unvorhersehbarem Verhalten und potenziellen Sicherheitslücken führen kann. Zudem werden ebenfalls in diesen Sprachen implizite Typen-Konvertierungen, vorgenommen. [2, Kapitel 1: Types]. Auch dies unterbindet rust wie in Beispiel 1.1 und 1.2 gezeigt.

2 Integer-Überlauf

Der Integer-Überlauf ist ein häufiges Problem in vielen Programmiersprachen, das zu schwerwiegenden Fehlern und Sicherheitslücken führen kann. Ein Integer-Überlauf tritt auf, wenn eine arithmetische Operation das Maximum (oder Minimum) des für den Datentyp zulässigen Wertebereichs überschreitet. In vielen Programmiersprachen führt dies zu undefiniertem Verhalten oder unvorhersehbaren Ergebnissen, was potenziell ausnutzbare Schwachstellen zur Folge haben kann [3].

Rust hingegen bietet verschiedene Mechanismen, um Integer-Überläufe sicher zu handhaben und das Verhalten explizit zu definieren [4, Kapitel 3.2].

Listing 2.1: Integer-Überläufe

```rust
pub fn overflow_how_not(){
    let a: i32 = i32::MAX;
    // panic in release mode due to runtime checks
    // wil wrap in release mode -> defined behaviour
    let b = a + 1;

    println!("{}", b);
}
pub fn overflow(){
    let a: i32 = i32::MAX;

    // checked addition
    let checked_sum = a.checked_add(1);
    println!("Checked sum: {:?}", checked_sum);

    // wrapping addition
    let wrapping_sum = a.wrapping_add(1);
    println!("Wrapping sum: {}", wrapping_sum);

    // saturating addition
    let saturating_sum = a.saturating_add(1);
    println!("Saturating sum: {}", saturating_sum);

    // overflowing addition
    let (overflowing_sum, overflowed) = a.overflowing_add(1);
    println!("Overflowing sum: {}, overflowed: {}",
        overflowing_sum, overflowed);
}
```

Wie in Beispiel 2.1 zu sehen, bietet Rust mehrere Möglichkeiten, Integer-Überläufe sicher zu handhaben:

- **Überprüfte Addition (checked addition)**: Gibt `None` zurück, wenn ein Überlauf auftritt.
- **Wrappende Addition (wrapping addition)**: Wickelt den Wert bei Überlauf um, entsprechend dem Wertebereich des Datentyps.
- **Sättigende Addition (saturating addition)**: Begrenzt den Wert auf den maximalen oder minimalen Wert des Datentyps bei Überlauf.
- **Überlaufende Addition (overflowing addition)**: Gibt ein *tupel* zurück, bestehend aus dem Ergebnis der Addition und einem booleschen Wert, der anzeigt, ob ein Überlauf aufgetreten ist.

3 Zero-Cost-Abstraktionen

Zero-Cost-Abstraktionen sind ein zentrales Konzept in Rust, das es ermöglicht, hohe Abstraktionen zu verwenden, ohne die Laufzeit des Programms zu beeinträchtigen. Dies bedeutet, dass der Overhead, der durch die Abstraktionen entsteht, zur Kompilierzeit eliminiert wird, sodass der generierte Maschinencode ähnlich effizient ist wie handgeschriebener Low-Level-Code [5]. In anderen Sprachen kann dies zu einem Laufzeit-Overhead führen [6].

3.1 Iteratoren

Iteratoren helfen dabei, über eine Menge von Elementen zu iterieren und ersetzen eine explizite Schleife. Sie bieten eine abstrahierte Möglichkeit zur Verarbeitung von Sequenzen, wodurch die Programmlogik klarer und fehlerresistenter wird.

Listing 3.1: Iterator

```
use std::sync::mpsc;
use std::thread;

pub fn iterator(){
    let numbers = vec![1, 2, 3, 4, 5];

    // Using an iterator to iterate through the vector safely
    for num in numbers.iter() {
        println!("{}", num); // Safely prints each number
    }
}
```

Ein solcher Iterator löst die Programmlogik von der Schleifenlogik, wodurch typische Fehler wie *off-by-one* und daraus resultierende, z.B. falsche Array-Indexierung (Buffer-Overflow), vermieden werden. Darüber hinaus bieten Iteratoren eine einheitliche und oft effizientere Möglichkeit, mit Datenstrukturen zu arbeiten.

3.2 Filter-Methode

Die `filter`-Methode ermöglicht es, Elemente eines Iterators basierend auf einer Bedingung zu filtern. Dies ergänzt die Funktionalität von Iteratoren und hilft, spezifische Elemente aus einer Sequenz herauszufiltern.

Listing 3.2: Filter-Funktion

```
pub fn filter(){
    let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

    // Use filter to find even numbers
    let even_numbers: Vec<_> = numbers.iter()
        .filter(|&&x| x % 2 == 0)
        .collect();

    println!("Even numbers: {:?}", even_numbers);
}
```

Ein Filter auf einem Iterator separiert zusätzlich die Filterlogik. An dieser Stelle könnte auch eine Methode verwendet werden, was für besser lesbaren Code sorgt. Die Kombination von `filter` mit anderen Iterator-Methoden erlaubt eine deklarative Beschreibung von Datenverarbeitungsabläufen, was die Verständlichkeit und Wartbarkeit des Codes weiter erhöht.

3.3 Message Passing

Message Passing ist eine sichere und effiziente Methode zur Kommunikation zwischen Threads.

Listing 3.3: Message Passing

```
pub fn message_passing(){
    let (tx, rx) = mpsc::channel();

    let sender_thread = thread::spawn(move || {
        let msg = "Hello␣from␣the␣sender␣thread";
        // Sends a message safely to the receiver
        tx.send(msg).unwrap();
        println!("Sent␣message:␣'{}'", msg);
    });

    let receiver_thread = thread::spawn(move || {
        // Receives message safely
        let received = rx.recv().unwrap();
        println!("Received␣message:␣'{}'", received);
    });

    sender_thread.join().unwrap();
    receiver_thread.join().unwrap();
}
```

Das Beispiel 3.4 zeigt, wie mit `mpsc::channel()` ein Sender- und Empfängerkanal erstellt wird, über den Nachrichten sicher zwischen Threads, mittels `.send()` und `.recv()`, ausgetauscht werden können. Durch eine solche einfache Abstraktion können Datenrennen zwischen Threads vermieden werden. Trotzdem können Threads effektiv kommunizieren und der Code bleibt lesbar.

3.4 Option Typ

Der `Option`-Typ wird verwendet, um die mögliche Abwesenheit eines Wertes sicher zu handhaben. Hier ist ein Beispiel, wie man eine Division sicher durchführen kann:

Listing 3.4: Option Typ

```
pub fn option_type(){
    let dividend = 10;
    let divisor = 0;

    match find_divisor(dividend, divisor) {
        Some(result) => println!("Result of division: {}", result),
        None => println!("Cannot divide by zero!"),
    }
}

fn find_divisor(dividend: i32, divisor: i32) -> Option<i32> {
    if divisor == 0 {
        None // Safely handle division by zero
    } else {
        Some(dividend / divisor) // Safely return the result
    }
}
```

Eine `Option` gibt entweder einen Wert `Some` zurück oder einen `None`-Wert. Mittels `match` kann geprüft werden, ob ein Wert vorliegt und dementsprechend verarbeitet werden.

Dadurch wird vermieden, wie es oft in C++ der Fall ist, dass spezielle Rückgabewerte wie `-1` oder andere verwendet werden müssen, um besondere Zustände zu signalisieren. Ein bekanntes Beispiel dafür ist die Methode `std::string::find` aus der C++ Standardbibliothek, die `std::string::npos` (typischerweise `-1`) zurückgibt, wenn ein Wert nicht gefunden wird. Solche Werte können unter Umständen zu Konflikten führen [7] [8].

4 Fehlerbehandlung

Rust bietet einen robusten Ansatz zur Fehlerbehandlung, der ohne Ausnahmen (exceptions) auskommt und eine sichere und vorhersehbare Fehlerbehandlung ermöglicht. Im Gegensatz zu vielen anderen Programmiersprachen verwendet Rust das `Result` und `Option`-Typsystem, um Fehler und fehlende Werte explizit zu behandeln.

Fehler werden durch den `Result<T, E>`-Typ dargestellt. `Result<T, E>` ist ein Typ, der entweder einen Wert vom Typ `T` oder einen Fehler vom Typ `E` enthält.

Listing 4.1: Sichere Fehlerbehandlung

```
1  use std::fs::File;
2  use std::io::Read;
3
4  pub fn get_file() {
5      match read_file_content() {
6          Ok(contents) => println!("File␣contents:␣{}", contents),
7          Err(e) => println!("Error␣reading␣file:␣{}", e),
8      }
9  }
10
11 fn read_file_content() -> Result<String, std::io::Error> {
12     let mut file = File::open("example.txt")?;
13     let mut contents = String::new();
14     file.read_to_string(&mut contents)?;
15     Ok(contents)
16 }
```

Die Funktion `read_file_content()` gibt einen `Result`-Typ zurück, welcher einen Fehler beinhaltet, sollte die Dateioperation fehlschlagen. Mittels `match` kann überprüft werden, ob das Result `OK` ist und ein Wert vorliegt oder ob das `Result` ein Fehler ist.

Generell müssen in Rust alle möglichen Fehler behandelt werden. Dies kann auch durch `unwrap()` geschehen, welches den `OK`-Wert zurückgibt oder im Falle eines Fehlers eine *panic* wirft. In beiden Fällen muss der Programmierer diese jedoch explizit behandeln.

5 Besitz und Ausleihe

In Rust sind Besitz und Ausleihe grundlegende Konzepte, die zur Gewährleistung der Speicher- und Datensicherheit beitragen. Das Besitzmodell sorgt dafür, dass es stets einen eindeutigen Besitzer eines Datenobjekts gibt, während das Ausleihmodell ermöglicht, dass andere Teile des Codes temporär auf diese Daten zugreifen können, ohne deren Besitzer zu ändern. Dies verhindert Datenrennen (*data races*) und Speicherfehler.

5.1 Copy-Typen

Copy-Typen sind einfache Datentypen, die eine bitweise Kopie beim Zuweisen erstellen. Primitive Typen wie `i32` sind Beispiele dafür.

Listing 5.1: Copy-Typen

```
pub fn copy_type()
{
    let i1 = 32;
    let i2 = i1;

    println!("{}, {}", i1, i2);
}
```

Bei der Besitzübertragung wird der Besitz eines Datenobjekts von einer Variablen auf eine andere übertragen, was bedeutet, dass die ursprüngliche Variable nicht mehr auf das Objekt zugreifen kann. Dies ist der Fall bei komplexen Typen wie `String`.

Listing 5.2: Unveränderliche Ausleihe

```
pub fn transfer_ownership() {
    let s1 = String::from("Hello");
    // Ownership of the string is transferred from s1 to s2
    let s2 = s1;

    // This line causes a compile-time error because
    // s1 no longer owns the string.
    println!("{}", s1);
    // This works perfectly, s2 now owns the data.
    println!("{}", s2);
}
```

5.2 Ausleihe

Ausleihe ermöglicht es, dass eine Variable auf die Daten einer anderen Variable zugreift, ohne deren Besitzer zu ändern. Dies kann entweder als unveränderliche oder veränderliche Referenz erfolgen.

Listing 5.3: Unveränderliche Ausleihe

```
pub fn borrowing() {
    let s1 = String::from("Hello");
    // s2 is a reference to s1, s1 is borrowed
    let s2 = &s1;

    // s1 is still valid and hasn't been moved.
    println!("{}", s1);
    // s2 is a valid reference to s1.
    println!("{}", s2);
}
```

Veränderliche Referenzen ermöglichen das Ändern der Daten, auf die sie zeigen. Allerdings kann zu einem Zeitpunkt nur eine veränderliche Referenz existieren.

Listing 5.4: Veränderliche Referenz

```
pub fn mut_reference() {
    let mut s1 = String::from("Hello");
    // s2 is a mutable reference to s1
    let mut s2 = &mut s1;

    // only 1 mutable reference allowed
    // will panic
    let mut s3 = &mut s1;

    // Modifying s1 through its mutable reference s2
    s2.push_str(",␣world!");
    // This works and prints "Hello, world!"
    println!("{}", s2);
}
```

Im Gegensatz zu Rust erlauben viele andere Programmiersprachen wie C und C++ eine flexiblere, aber auch unsicherere Speicherverwaltung. In diesen Sprachen kann leicht unbeabsichtigter Code entstehen, der zu Speicherlecks oder Datenrennen führt, da hier Speicherplatz manuell allokiert und dessen Freigabe manuell verwaltet werden muss. Rusts strikte Regeln für Besitz und Ausleihe verhindern solche Probleme, indem sie sicherstellen, dass immer geregelt ist, wer für den Speicher eines Datenobjekts verantwortlich ist und wie darauf zugegriffen werden kann.

6 Lebensdauer-Tracking

Das Lebensdauer-Tracking ist ein zentrales Konzept in Rust, das dazu beiträgt, Speicherfehler zu verhindern. Lebensdauern (lifetimes) sind eine Art von generischen Parametern, die sicherstellen, dass Referenzen nur so lange gültig sind, wie es der Kontext erfordert. Dies verhindert das Auftreten von hängenden Referenzen und ermöglicht eine präzise Kontrolle über die Lebensdauer von Daten im Speicher.

Rust verwendet Lebensdauern, um sicherzustellen, dass Referenzen nicht auf ungültige Daten zeigen. Eine Lebensdauer beschreibt den Gültigkeitsbereich einer Referenz. Rust prüft zur Kompilierzeit, dass alle Referenzen gültig sind. In einfachen Beispielen kann Rust die Lebensdauer selbst ableiten. Bei komplizierteren Strukturen mit Referenzen muss der Programmierer die Lebensdauer angeben.

Listing 6.1: Expliziete lifetime

```
1 pub fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
2     if x.len() > y.len() {
3         x
4     } else {
5         y
6     }
7 }
```

Beide Eingabereferenzen und die Rückgabereferenz im Beispiel 6.1 haben dieselbe Lebensdauer `'a`. Das bedeutet, dass die Rückgabereferenz nur so lange gültig ist, wie beide Eingabereferenzen gültig sind.

Die Angabe der Lebensdauer ist notwendig, da der Compiler ohne diese Annotation nicht in der Lage ist, die Beziehung zwischen den Lebensdauern der Eingabereferenzen und der Rückgabereferenz zu erkennen. Rusts *borrow checker* kann ohne die explizite Angabe der Lebensdauer `'a` nicht sicherstellen, dass die Rückgabe einer der Eingabereferenzen keine ungültigen Referenzen erzeugt. Die Lebensdauer `'a` stellt sicher, dass die Rückgabereferenz nur so lange gültig ist, wie beide Eingabereferenzen gültig sind.

Mit diesem System kann Rust häufige Speicherfehler wie in C++ vermeiden, wo der Programierrer für die Freigabe selbst verantwortlich ist. All dies ohne die zuhilfenahme eines garbage-collectors, wie in viele moderne Sprachen einsetzen.

7 Sichere Nebenläufigkeit

Nebenläufigkeit ist ein wesentlicher Aspekt moderner Softwareentwicklung, insbesondere bei der Entwicklung von Systemen, die eine hohe Leistung und Reaktionsfähigkeit erfordern. Jedoch birgt die Nebenläufigkeit zahlreiche Herausforderungen, insbesondere im Hinblick auf die Sicherheit und Korrektheit des Codes. Datenrennen sind ein häufiges Problem, das auftritt, wenn mehrere Threads gleichzeitig auf dieselben Daten zugreifen und diese ändern, ohne dass die Zugriffe korrekt synchronisiert sind.

Rust bietet einen Ansatz zur sicheren Nebenläufigkeit, indem es die Konzepte der Besitzrechte (Ownership) und des Ausleihens (Borrowing) auf Threads anwendet. Dies wird durch die Einbindung von Mechanismen wie `Arc` (Atomic Reference Counting) und `Mutex` (Mutual Exclusion) erreicht, die sicherstellen, dass Datenrennen verhindert werden.

`Arc` (Atomic Reference Counting) ist ein Thread-sicherer Referenzzähler, der es ermöglicht, dass mehrere Threads gleichzeitig Besitz an den gleichen Daten haben, indem er die Daten im Heap speichert und den Zähler atomar verwaltet.

`Mutex` (Mutual Exclusion) ist ein Synchronisationsprimitiv, das sicherstellt, dass nur ein Thread zu einem bestimmten Zeitpunkt auf die geschützten Daten zugreifen kann

`Arc` übernimmt das Zählen der Referenzen, während `Mutex` den exklusiven Zugriff auf die Daten garantiert. Dies funktioniert reibungslos mit Rusts Ownership- und Borrowing-System, da `Arc` die Besitzrechte an den Daten teilt und `Mutex` den Zugriff kontrolliert, um Datenrennen zu vermeiden.

Listing 7.1: Sicheres Erstellen von Threads

```rust
use std::sync::{Arc, Mutex};
use std::thread;
use std::time::Duration;
use rand::{thread_rng, Rng};

pub fn create_increment_threads(){
    let counter = Arc::new(Mutex::new(0));
    let mut children = vec![];

    for _ in 0..5 {
        let counter_clone = Arc::clone(&counter);
        let child = thread::spawn(move || {
            let mut rng = thread_rng();
            // Loop to add to the counter 10 times
            for _ in 0..10 {
                {
                    let mut num = match counter_clone.lock() {
                        Ok(x) => x,
                        Err(_) => todo!(),
                    };
                    *num += 1;
                } // MutexGuard goes out of scope here,
                  //releasing the lock

                let sleep_time = rng.gen_range(1..=3);
                // Sleep for random duration
                thread::sleep(Duration::from_millis(sleep_time));
            }
        });
        children.push(child);
    }

    // Wait for all threads to complete
    for child in children {
        child.join().unwrap();
    }

    // Print the result
    println!("Final counter value: {}", *counter.lock().unwrap());
}
```

In diesem Beispiel wird ein gemeinsamer Zähler von mehreren Threads erhöht. Die Verwendung von `Arc<Mutex<i32>>`, stellt sicher, dass nur ein Thread gleichzeitig auf den Zähler zugreifen kann, wodurch Datenrennen verhindert werden.

8 Fazit

Rust hat sich als eine Programmiersprache erwiesen, die Sicherheit und Leistung gleichermaßen priorisiert. Durch ihre einzigartigen Sprachmerkmale bietet Rust Lösungen für viele der häufigsten Sicherheitsprobleme, die in der Softwareentwicklung auftreten. Die wichtigsten Sicherheitsaspekte, die wir in diesem Bericht untersucht haben, sind:

- **Statische Typen:** Rusts strenge statische Typisierung stellt sicher, dass Typfehler frühzeitig im Entwicklungsprozess erkannt werden. Dies reduziert die Wahrscheinlichkeit von Laufzeitfehlern und erhöht die Zuverlässigkeit des Codes.
- **Integer-Überlauf:** Rust verhindert Integer-Überläufe durch explizite Überlaufprüfungen und bietet Entwicklern Werkzeuge, um sichere numerische Berechnungen durchzuführen.
- **Zero-Cost-Abstraktionen:** Diese ermöglichen es, hochabstrakten und dennoch performanten Code zu schreiben. Rusts Abstraktionen verursachen keine Laufzeitkosten, was zu effizientem und sicherem Code führt. Gleichzeitig können z.B. *off-by-one*-Fehler vermieden werden.
- **Fehlerbehandlung:** Rusts Ansatz zur Fehlerbehandlung ohne Ausnahmen, durch die Verwendung von `Result` und `Option`, fördert robustes und vorhersehbares Fehlermanagement.
- **Besitz und Ausleihe:** Das Besitz- und Ausleihmodell von Rust garantiert Speicher- und Datensicherheit ohne die Notwendigkeit eines Garbage Collectors. Dies verhindert Speicherlecks und Datenrennen.
- **Lebensdauer-Tracking:** Durch die explizite Angabe von Lebensdauern verhindert Rust Dangling References und gewährleistet sichere Speicherzugriffe.
- **Sichere Nebenläufigkeit:** Rust ermöglicht es, nebenläufigen Code sicher und effizient zu schreiben. Durch die Vermeidung von Datenrennen wird die Zuverlässigkeit nebenläufiger Programme erhöht.

Zusammenfassend lässt sich sagen, dass Rust durch diese Sprachmerkmale eine sichere und zuverlässige Entwicklung von Software ermöglicht. Die Kombination aus statischer Typisierung, sicherer Speicherverwaltung und robustem Fehlerhandling macht Rust zu einer idealen Wahl für sicherheitskritische Anwendungen.

Mit Blick auf die Zukunft verspricht Rust, durch kontinuierliche Weiterentwicklung und Verbesserung, seine Position als eine der sicheren und leistungsfähigen Programmiersprachen weiter zu festigen.

Beispielverzeichnis

Literaturverzeichnis

[1] "Rust by example," accessed: 2024-06-06. [Online]. Available: https://doc.rust-lang.org/rust-by-example/

[2] D. Drysdale, *Effective Rust*. Sebastopol, CA: O'Reilly Media, Inc., 2024, copyright © 2024 Galloglass Consulting Limited. All rights reserved. Printed in the United States of America.

[3] MITRE, "CWE-190: Integer Overflow or Wraparound," https://cwe.mitre.org/data/definitions/190.html, n.d., accessed: 2024-06-05.

[4] Rust Foundation and the Rust Project Developers, *The Rust Programming Language*, 2nd ed. Rust Foundation, 2023, copyright © 2023 by the Rust Foundation and the Rust Project Developers.

[5] R. Sequeira, "Learning rust: Understanding zero-cost abstraction with filter and map," https://ranveersequeira.medium.com/learning-rust-understanding-zero-cost-abstraction-with-filter-and-map-e967d09fff79, 2023, accessed: 2024-06-06.

[6] J. Haberman, "The overhead of abstraction in c/c++ vs. python/ruby," https://blog.reverberate.org/2014/10/the-overhead-of-abstraction-in-cc-vs.html, 2014, accessed: 2024-06-06.

[7] cplusplus.com, "std::string::find," https://cplusplus.com/reference/string/string/find/, accessed: 2024-06-06.

[8] ——, "std::string::npos," https://cplusplus.com/reference/string/string/npos/, accessed: 2024-06-06.

Glossar

borrow checker Ein System in der Programmiersprache Rust, das zur Kompilierzeit überprüft, ob alle Speicherzugriffe sicher sind. Es stellt sicher, dass Daten nie gleichzeitig sowohl mutabel als auch immutabel geborgt werden, und dass keine doppelten mutablen Referenzen existieren, um Race Conditions und andere Speicherfehler zu verhindern. i, 11

data race Ein Datenrennen tritt auf, wenn zwei oder mehr Threads in einem Computerprogramm gleichzeitig auf dieselbe Speicherstelle zugreifen, und mindestens einer der Zugriffe schreibend erfolgt, ohne dass die Zugriffe synchronisiert werden. Dies kann zu unerwarteten und fehlerhaften Verhaltensweisen führen, da die Reihenfolge der Zugriffe nicht garantiert ist und das Ergebnis von der Timing-Synchronisation der Threads abhängt.. i, ii, 9

exception Eine *exception* (dt. Ausnahme) ist ein Mechanismus in vielen Programmiersprachen, der verwendet wird, um auf ungewöhnliche oder fehlerhafte Zustände während der Programmausführung zu reagieren. Ausnahmen treten auf, wenn ein Programm auf ein Problem stößt, das es nicht mit normalem Kontrollfluss handhaben kann, wie z.B. Division durch Null oder der Versuch, auf eine Datei zuzugreifen, die nicht existiert. Beim Auftreten einer Ausnahme wird der normale Ablauf des Programms unterbrochen und es wird ein spezieller Fehlerbehandlungsroutine aufgerufen. . i, 8

off-by-one Ein Fehler in der Programmierung, bei dem eine Schleife oder ein Array um eine Einheit zu viel oder zu wenig iteriert oder indiziert wird, was oft zu unerwartetem Verhalten führt. Dies passiert häufig bei der Implementierung von Schleifen oder bei der Arbeit mit Indizes, wo die korrekte Anzahl der Iterationen oder die korrekten Grenzen leicht übersehen werden können. i, 5, 14

panic In der Programmiersprache Rust und anderen Programmiersprachen bezeichnet *panic* eine Laufzeitausnahme, die auftritt, wenn das Programm auf einen schwerwiegenden Fehler stößt, von dem es sich nicht erholen kann. Ein panic führt in der Regel dazu, dass das Programm sofort abgebrochen wird und die Kontrolle an das Betriebssystem zurückgegeben wird. Dies unterscheidet sich von regulären Fehlern, die durch Rückgabewerte oder Ausnahmen behandelt werden können. . i, 8

tupel Ein Tupel ist eine geordnete Liste von Elementen, die in der Informatik häufig verwendet wird, um eine feste Anzahl von Werten zu speichern, die zu einer Einheit zusammengefasst sind. Die Elemente eines Tupels können unterschiedliche Datentypen haben. Tupel sind unveränderlich, was bedeutet, dass ihre Inhalte nach der Erstellung nicht mehr geändert werden können.. i, 4

wrap-around Der Begriff "Wrap-around" beschreibt ein Phänomen, bei dem ein numerischer Wert, der seinen maximal oder minimal zulässigen Wert erreicht, auf den gegenüberliegenden Extremwert zurückgesetzt wird. Dies tritt häufig in der Computerprogrammierung auf, insbesondere bei der Arbeit mit beschränkten Datentypen wie Ganzzahlen. Ein Wrap-around kann zu Fehlern führen, wenn es nicht ordnungsgemäß gehandhabt wird.. i