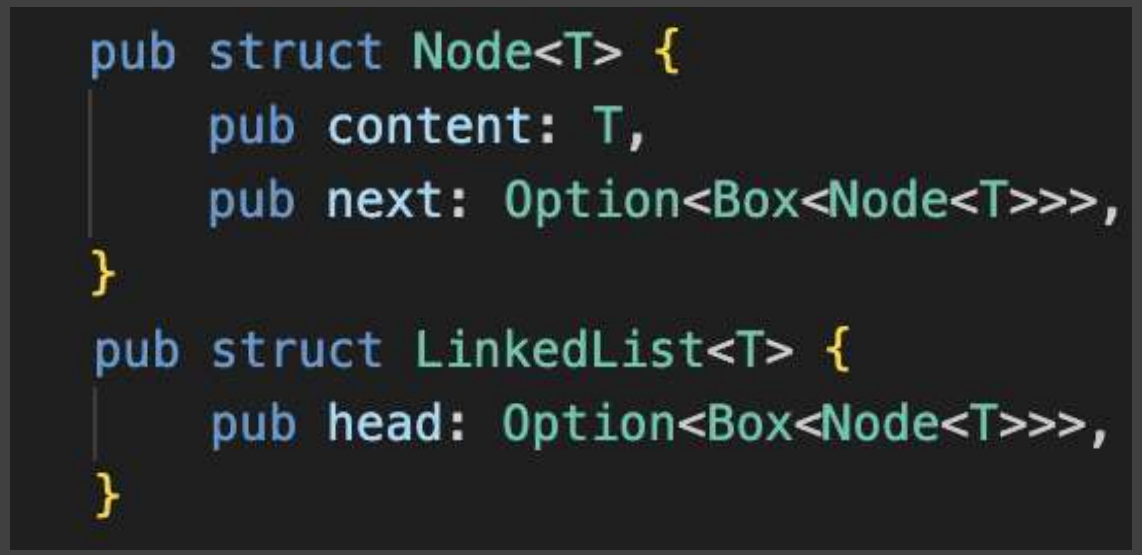
Generische Programmierung & Datenstrukturen

Inhaltsverzeichnis

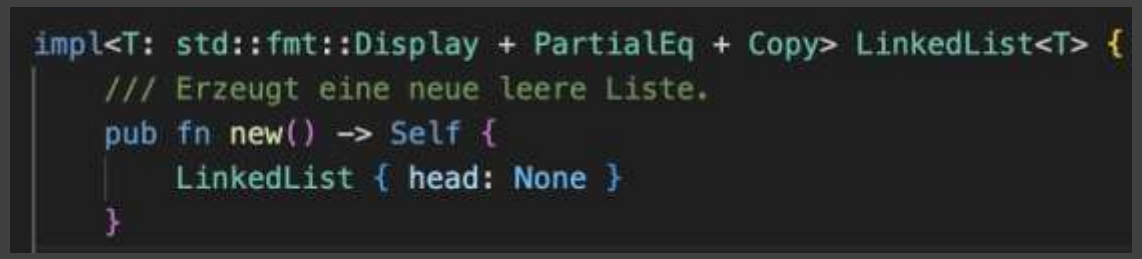
Generische Programmierung

- Programmierparadigma, welches ermöglicht, Code unabhängig von spezifischen Datentypen zu schreiben,
- durch die Verwendung von generischen Typen
- während der Kompilierung durch konkrete Typen ersetzt
- Konzept fördert die Wiederverwendbarkeit, Abstraktion und Flexibilität von Code
- Algorithmen/ Datenstrukturen werden allgemein gehalten, sodass sie mit Vielzahl von Datentypen funktionieren

Structs (Structures)



```rust
pub struct Node<T> {
    pub content: T,
    pub next: Option<Box<Node<T>>>,
}
pub struct LinkedList<T> {
    pub head: Option<Box<Node<T>>>,
}
```



```rust
impl<T: std::fmt::Display + PartialEq + Copy> LinkedList<T> {
    /// Erzeugt eine neue leere Liste.
    pub fn new() -> Self {
        LinkedList { head: None }
    }
```

- ähnlich zu Attributen von Objekten
- ermöglicht Funktionalitäten zusammenzusetzen und Objekte modular und flexibler zu gestalten
- sind stark typisiert, und der Compiler überprüft den Zugriff auf Felder zur Compile-Zeit

Generischer Typ T

```rust
pub struct Node<T> {
    pub data: T,
    pub next: Option<Box<Node<T>>>,
}
pub struct Stack<T> {
    pub head: Option<Box<Node<T>>>,
    pub length: i32,
}
```

```rust
// Demonstration der Stack-Klasse
let mut stack: Stack<i32> = Stack::new();
```

- T als generischer Typ
- Platzhalter werden bei der Kompilierung durch spezifische Typen ersetzt
- T kann primitive & komplexe Datentypen repräsentieren

-> wiederverwendbar & flexibel

Generischer Typ T

- Bsp. für komplexe Datentypen in Stack

```rust
// komplexe Datentypen Array

let a: [i32; 5] = [1, 2, 3, 4, 5];
let b: [i32; 4] = [1,32, 4235, 2324];
let c: [i32; 8] = [1,32, 4235, 2324, 2, 3, 4, 5];

let mut stack_complex: Stack<&[i32]> = Stack::new();
stack_complex.push(data: &a);
stack_complex.push(data: &b);
stack_complex.push(data: &c);

println!("Komplexe Datentype im Stack: {:#?}", stack_complex.peek().unwrap());
stack_complex.pop();
stack_complex.pop();
stack_complex.pop();
```

Composition over Inheritance

- Komposition bedeutet, dass Klasse andere Klassen als Komponenten verwendet
- Dadurch kann Klasse das Verhalten und Funktionalität anderer Klassen nutzen, ohne von ihnen zu erben
- Komposition kann durch Verwendung von Structs erreicht werden
- Rust unterstützt keine Vererbung von Structs

Composition over Inheritance

- Structs miteinander verknüpfen
- Vorteil: wiederverwenden von Stack-Funktionen

```rust
pub struct Queue<T> {
    stack_in: Stack<T>,  // Stack für das Einfügen von Elementen
    stack_out: Stack<T>, // Stack für das Entfernen von Elementen
}

impl<T: PartialEq + std::fmt::Display> Queue<T> {
    pub fn new() -> Self {
        Queue {
            stack_in: Stack::new(),
            stack_out: Stack::new(),
        }
    }
```

```rust
// Fügt ein Element in die Queue ein (Push auf stack_in)
pub fn enqueue(&mut self, data: T) {
    self.stack_in.push(data);
}
```

-> In manchen Situationen unpassend

Traits

Stack

- equals(other)
- to_string()
- peek()
- size()
- Is_empty
- is_full
- push(elem)
- pushAll(elems)
- pop()

Queue

- equals(other)
- to_string()
- peek()
- size()
- Is_empty
- is_full
- enqueue(elem)
- dequeue()

List

- equals(other)
- to_string()
- peek()
- size()
- Is_empty
- is_full
- get(pos)
- add(elems)
- insert(elem,pos)
- remove()
- replace()

Traits

- ähnlich zu Interfaces
- Verhaltens-Schnittstellen
- wiederverwendbar & kombinierbar
- Implementierung von Traits verursacht keine zusätzliche Laufzeitkosten, da der tatsächliche Code zur Compile-Zeit integriert wird
- Funktionalitäten

```
pub trait Datastructure{
    // isEmpty(): true wenn leer ist sonst fa
    fn is_empty(&self) -> bool ;

    // isFull(): false wenn leer ist sonst tr
    fn is_full(&mut self) -> bool {
        !self.is_empty()
    }

    // Gibt die Anzahl der Elemente in der Li
    fn size(&self) -> i32;

    // Gibt das älteste Element zurück, ohne
    pub fn peek(&mut self) -> Option<&T>;

    // Gibt true zurück, wenn zwei Datenstruk
    // die gleichen Werte (in gleicher Reihen
    fn equals(&self, other: &Self) -> bool {
        // konvertieren wir beide zu Strings
        self.to_string() == other.to_string()
    }

    //eine Repräsentation als String haben (t
    //und die die programmatische Rekonstrukt
    fn to_string(&self) -> String;
} trait Datastructure
```

Zero Cost Abstractions

- Idee, dass Entwickler abstrakten Code schreiben, die Code klarer und besser organisieren
- allerdings keinen Performance Nachteil durch zusätzlichen Laufzeit erzeugen sollten

Into Iterator

- IntoIterator ist Trait
- Typkonvertierung

```rust
// Fügt mehrere Elemente oben auf den Stack
pub fn push_all<I: IntoIterator<Item = T>>(&mut self, data: I) {
    for item: T in data {
        self.push(data: item);
    }
}
```

```rust
// Mehrere Elemente auf Stack pushen
let mut more_stack: Stack<i32> = Stack::new();
more_stack.push_all(data: vec![1, 2, 3, 4, 5]);
```

Zero Cost Abstractions

Mit Generics

- Compiler schreibt für jeden Typ mit dem Generics Funktion aufgerufen wurde spezielle

Mit Traits

- Traits verursachen keine zusätzliche Laufzeitkosten, da der tatsächliche Code zur Compile-Zeit integriert wird

Mit Iterator

- Compiler wandelt in einfache Schleifen um
- Dadurch keine Laufzeitkosten

-> In Rust ist möglich generischen Code für abstrakte Konzepte zu schreiben ohne Laufzeiteffiziens zu verlieren dank Compiler

Error Handling

- Option<T>

```rust
pub enum Option<T> {
    None,
    Some( /* … */ ),
}
```

Tagged Union

```rust
// Pop-Funktion: Entfernt das oberste Element vom Stack und gibt es zurück
pub fn pop(&mut self) -> Option<T> {
    if let Some(node: Box<Node<T>>) = self.head.take() {
        self.head = node.next;
        self.length -= 1;
        Some(node.data)
    } else {
        None // Wenn der Stack leer ist, geben wir None zurück
    }
}
```

Tests

Quellen

- https://medium.com/comsystoreply/28-days-of-rust-part-2-composition-over-inheritance-cab1b106534a
- https://doc.rust-lang.org/beta/embedded-book/static-guarantees/zero-cost-abstractions.html
- https://doc.rust-lang.org/book/ch10-01-syntax.html
- https://doc.rust-lang.org/book/ch10-02-traits.html
- https://stackoverflow.com/questions/69178380/what-does-zero-cost-abstraction-mean
- https://de.wikipedia.org/wiki/Statische_Typisierung
- https://doc.rust-lang.org/book/ch03-02-data-types.html
- https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html
- https://doc.rust-lang.org/book/ch11-01-writing-tests.html
- https://users.rust-lang.org/t/rust-file-organization-imports-etc/103713
- https://doc.rust-lang.org/book/ch03-02-data-types.html
- https://stackoverflow.com/questions/28951503/how-can-i-create-a-function-with-a-variable-number-of-arguments