Listing 7: Vergleichsoperatoren in Klassentemplates

```
template<typename T1, typename T2>
class P {
    private:
      T1 x1;
      T2 x2;
    public:
    ...
      friend auto operator<=>(const P&, const P&) = default;
};
```

Listing 8: Jede Sequenz von Integern verdoppeln

```
std::vector<int> v{42, 43, 44, 45};
doubleContent(v);

std::array<int,4> a{-14, 55, 24, 67};
doubleContent(a);

int arr[]{0, 8, 15};
doubleContent(arr);
```

Die Ausgabe der Argumente erfolgt formatiert anhand ihrer Datentypen. Die Reihenfolge der Argumente ist dabei unerheblich:

```
// - Reihenfolge wie spezifiziert
std::format("'{1}' size: {0}\n", str.size(), str);
```

Es sind verschiedene Formatierungen möglich, wie rechtsbündig mit führenden Unterstrichen:

```
std::format("'{:_>20}' has {0} chars\n", str, str.size());
```

Mit `format_to()` kann hierbei auch in existierenden Speicherplatz für Zeichen geschrieben werden. Wie bereits beschrieben, ist der Mechanismus erweiterbar. Die chrono-Bibliothek verwendet diese Erweiterungsmöglichkeit, um Zeitpunkte und Zeitdauern formatiert ausgeben zu können:

```
std::format("Datum: {:%d.%m.%Y %T%z}\n", zonedTimePoint);
```

Der Standard enthält drei weitere Klassen für die Synchronisierung von Threads: Latches, Barriers und Semaphore. Ein Latch ist eine Art Countdown, der initialisiert mit einem Startwert bestimmte Ereignisse herunterzählt und bei 0 „fertig“ meldet. So kann man 10 Threads beauftragen, etwas zu tun, und den Zähler mit 5 initialisieren. Wenn dann jeder Thread sein Ende dem Latch als Ereignis mitteilt, meldet dieser nach 5 fertigen Threads, dass er die Ausführung abgeschlossen hat (Listing 9).

Beliebig viele Threads können einen Latch zum Herunterzählen auf 0 verwenden. Jeder Thread darf beliebig oft und viel herunterzählen. Ist er abgeschlossen und steht somit auf 0, kann er seinen Zustand nicht mehr verändern; er eignet sich also nur für einen einmaligen Countdown.

Die hier verwendete Threadklasse `std::jthread` ist neu und behebt einige Designschwächen von `std::thread`. Es handelt sich um einen echten RAII-Typ, der aufräumt, wenn man das Threadobjekt zerstört (ohne `join()` aufgerufen zu haben). Der Destruktor teilt dem laufenden Thread sogar vorher mit, dass dieser sich beenden soll. Das kann ein kooperativ implementierter Thread nun bequem auswerten, um sich sauber zu beenden.

Barriers sind so organisiert, dass sie mehrfach Ereignisse verschiedener Threads synchronisieren können. Auch hier handelt es sich um einen Zähler, den jeder Thread aber nur einmal herunterzählen kann. Steht der Barrier auf 0, wird eine vordefinierte Aktion aufgerufen und der Barrier lässt sich erneut verwenden. Das ist hilfreich, wenn das Programm nach einem bestimmten Ereignis immer wieder darauf warten muss, dass eine bestimmte Anzahl paralleler Reaktionen erfolgt ist. Ein Semaphor ist ein verallgemeinerter Mutex. Damit kann der Zugriff auf eine Ressource synchronisiert werden. Im Gegensatz zu einem Mutex müssen der Thread, der eine Ressource anfordert, und der Thread, der die

Listing 9: Latch mit zehn Threads, von denen fünf fertig werden müssen

```
#include <latch>

// Menge von Threads:
const int numThreads{10};
std::vector<std::jthread> threads;

// 5 muessen fertig werden:
std::latch ready{5};

// alle Threads starten, die dann fertig melden:
for (int i=0; i<numThreads; ++i) {
    threads.push_back(std::jthread{[&ready]{
      doSomeStuff();
      ready.count_down();
      }});
}
// warten, dass 5 fertig sind:
ready.wait();

...  // Reaktion auf mindestens 5 fertige Threads
```

Listing 10: Limitierung auf je drei parallel laufende Threads

```
std::counting_semaphore sem{3};
for (int i=0; i<numThreads; ++i) {
    threads.push_back(std::jthread{[&sem]{
                                    sem.acquire();
                                    doSomeStuff();
                                    sem.release();
                                  }});
}
```

Anforderung wieder freigibt, nicht identisch sein. Außerdem kann man dabei zulassen, dass eine bestimmte maximale Anzahl paralleler Zugriffe möglich ist. So lässt der Code in Listing 10 immer nur maximal drei Threads auf einmal `doSomeStuff()` aufrufen.

Weitere neue Merkmale von C++20 sind:

- Range-based for-Schleifen, die jetzt mit einer zusätzlichen Initialisierung beginnen;
- Gleitkommawerte und Objekte von Klassen, die nun Templateparameter sein können;
- Erweiterungen bei atomaren Datentypen;
- bessere Unterstützung zum Debuggen (`std::source_location`).

Fazit

Der C++-Standard ist zwar nur um etwa 15 Prozent gewachsen, aber die Neuerungen werden die Art, in Zukunft C++ zu programmieren, wesentlich beeinflussen. Zu den genannten großen Themen Konzepte, Ranges, Module und Koroutinen wird es Verbesserungen in den nächsten C++-Standards C++23 und C++26 geben. Daher sollten sie mit etwas Vorsicht genutzt werden, auch wenn sie in einigen Compilern schon zur Verfügung stehen.

Bis zur vollständigen Implementierung von C++20 in Compilern werden sicher noch zwei bis drei Jahre vergehen. Alle Beispiele in diesem Artikel sind deshalb ohne Gewähr, weil eine standardkonforme Verwendung bisher nur marginal und experimentell unterstützt wird. Aber kleinere Erweiterungen werden zeitnah zur Verfügung stehen. Wer in C++ programmiert, sollte wie immer wissen, was er tut. Er wird dann aber weiter mit guter Performance bei hoher Flexibilität belohnt. (nb@ix.de)

Quellen

Alle Listings zum Download: ix.de/zwbr

Nicolai Josuttis

ist seit 25 Jahren deutscher Vertreter bei der Standardisierung von C++ und Buchautor etlicher Standardwerke zur Programmierung mit C++.

Rust: nicht nur für den Browser

Lichtblick

Jens Breitbart und Stefan Lankes

Die Sprache Rust eignet sich für den Einsatz im Browser. Ihre Effizienz und Eigenständigkeit macht sie aber auch für Cloud-Services und den Einsatz im Embedded-Bereich interessant.

Rust entsprang 2010 einem privaten Projekt des Mozilla-Mitarbeiters Graydon Hoare. Fünf Jahre später erschien die erste stabile Version der noch jungen Programmiersprache, deren Weiterentwicklung ursprünglich Mozilla Research vorantrieb. Mittlerweile hat sich jedoch eine rege Community herausgebildet, die die Weiterentwicklung übernimmt. Hier gibt es eine Reihe fester Teams und anwendungsgetriebene Working Groups; eine genaue Übersicht findet sich auf der Rust-Homepage.

Die Entwicklung der Sprache orientiert sich an drei Zielen. Zum Ersten ist das die Performance. Rust arbeitet daher ohne Laufzeitumgebung und Garbage Collector, was Anwendungen mit derselben Laufzeitgeschwindigkeit und Speichereffizienz wie vergleichbare in C entwickelte Programme ermöglicht. Das zweite Ziel ist Verlässlichkeit. Bereits zur Kompilierzeit kann der Rust-Compiler verschiedene Klassen von Bugs entdecken, insbesondere im Bereich illegaler Speicherzugriffe und Race Conditions. Wenn Rust Code ohne Fehlermeldung kompiliert, können Entwickler*innen im Allgemeinen davon ausgehen, dass tatsächlich keine entsprechenden Fehler vorhanden sind.

Als drittes Ziel hat sich die Community das Thema Produktivität vorgenommen. Für sie haben Tools und Dokumentation einen hohen Stellenwert: Der Compiler soll hilfreiche Fehlermeldungen liefern und das eingebaute Build-Tool und die Paketverwaltung Cargo sollen die typischen Probleme von Build-Umgebungen für Rust-Projekte lösen. Auch die meisten bekannten Editoren beherrschen inzwischen den Umgang mit Rust.

iX-TRACT

- Die von Mozilla geförderte Sprache Rust erschien 2015 in einer ersten stabilen Version.
- Die Syntax lehnt sich der von C an. Aufeinanderfolgende Anweisungen lassen sich mit Semikolon abtrennen und Blöcke über geschweifte Klammern markieren.
- Die Ziele der Sprachentwicklung sind Performance, Verlässlichkeit und Produktivität.

Rust ist nicht nur für den Webbrowser von Mozilla

Die genannten Ziele bieten Vorteile für eine Reihe verschiedener Einsatzszenarien. Bekannt ist sicherlich, dass Rust im Firefox-Browser von Mozilla Verwendung findet, insbesondere ist die Browser-Engine Servo in Rust geschrieben. Aber auch System-

komponenten bekannter Anwendungen setzen darauf. So benutzt Dropbox Rust für Teile ihres Storage Backends und auch Amazons leichtgewichtige Firecracker-VM ist komplett in Rust geschrieben. Überhaupt bieten sich Cloud-Services für eine Umsetzung in Rust an, da sich die Effizienz der Sprache, abhängig von der Anzahl der Nutzer, für Betreiber direkt finanziell bezahlt machen kann. Cloudflare nutzt das für ihre HTML Rewriting Engine und auch NPM implementiert verschiedene Services in Rust. Dass die Rust-Standardbibliothek explizit Teile ausweist, die ohne Betriebssystem funktionieren, ist schließlich auch im Kontext von eingebetteten Systemen interessant, zumal die Cargo-Paketverwaltung die Handhabe entsprechender Projekte unterstützt.

Die Rust-Syntax: fast wie in C/C++

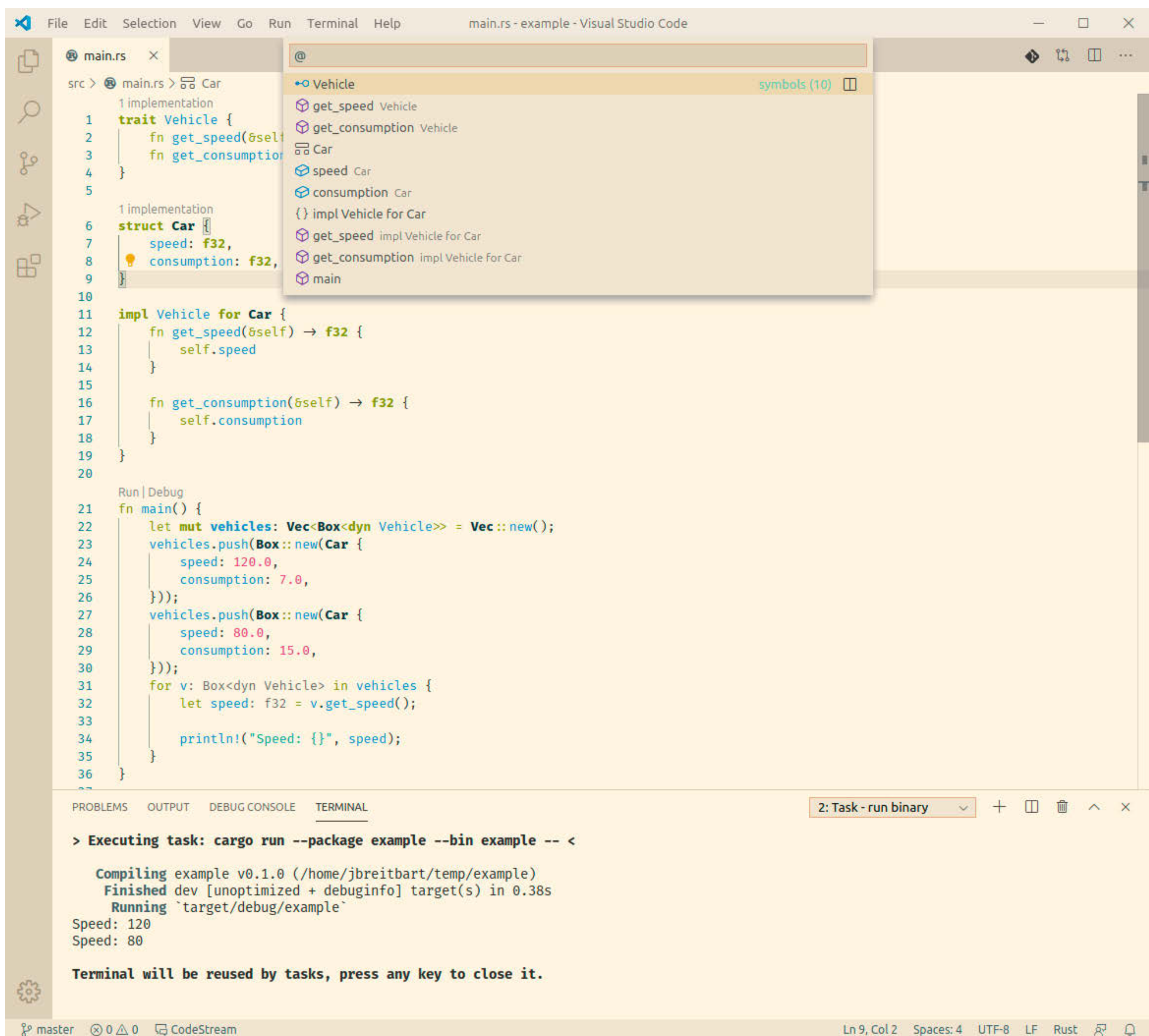
Die Syntax von Rust lehnt sich an C/C++ an. Entwickler*innen trennen aufeinanderfolgende Anweisungen mit Semikolon ab und markieren Blöcke über geschweifte Klammern. Ein einfaches Hello-World-Programm sieht entsprechend wie folgt aus:

```
fn main() {
    println!("Hello, world!");
}
```

Im Detail gibt es dabei durchaus Abweichungen von C, so sind Variablen wie folgt zu definieren:

```
let x: Typ = Wert;
let mut v: Typ = Initialwert;
v = NeuerWert;
x = v;
```

Das Snippet zeigt nicht nur, dass die Syntax für die Variablendefinition deutlich von der von C abweicht, sondern stellt auch einen entscheidenden semantischen Unterschied heraus: Variablen sind im Normalfall unveränderlich, außer Entwickler*innen markieren sie über das Schlüsselwort `mut` als `mutable`. Im gezeigten



Mit Visual Studio Code und dem Plug-in rust-analyzer ist es möglich, die Symbole, also Funktionen, Strukturen und Traits des Projektes, anzeigen zu lassen und an die entsprechende Stelle im Code zu springen. Weiterhin lässt sich die Anwendung auch direkt mit oder ohne Debugger ausführen.

Listing 1: Beispiel für das Verwenden von Traits

```
trait Vehicle {
    fn get_speed(&self) -> f32;
    fn get_consumption(&self) -> f32;
}

struct Car {
    speed: f32,
    consumption: f32,
}

impl Vehicle for Car {
    fn get_speed(&self) -> f32 {
      self.speed
    }

    fn get_consumption(&self) -> f32 {
      self.consumption
    }
}
```

Listing 2: Einen Vektor mit Trait Objects erzeugen

```
let mut vehicles: Vec<Box<dyn Vehicle>> = Vec::new();

vehicles.push(Box::new(Car::new()));
vehicles.push(Box::new(Truck::new()));

for v in vehicles {
    let speed = v.get_speed()

    // do something with speed
}
```

Beispiel erkennt der Compiler den Versuch, der Variablen x einen neuen Wert zuzuweisen, und wirft eine entsprechende Fehlermeldung.

Rust bietet an C angelehnte Strukturen an, allerdings mächtiger und in vielen Bereichen den Strukturen und Klassen aus C++ ähnlich. Konkret können Structs in Rust Member-Funktionen enthalten und auch sogenannte Traits implementieren. Letztere beschreiben eine Sammlung von Methoden, im Unterschied zu einer abstrakten Oberklasse erlauben sie jedoch eine horizontale Wiederverwendung von Methodensammlungen und vermeiden so Mehrfachvererbungen. Anders als C++ und Java stellt Rust also keine Vererbung zur Verfügung.

Listing 1 zeigt die Definition des Traits `Vehicle`, das die maximale Geschwindigkeit und den Treibstoffverbrauch eines Fahrzeugs beschreibt (alle Listings siehe ix.de/z1rn). Die Struktur `Car` implementiert die Eigenschaften von `Vehicle`, das heißt, sie implementiert die entsprechenden Methoden. Analog können die Fahrzeuge `Truck` und `Caravan` angelegt werden, die jeweils unterschiedliche Implementierungen besitzen. Im Unterschied zu C übergeben Entwickler*innen den Methoden die Selbstreferenz `self`, worüber diese auf die Elemente zugreifen können. Das `&` beschreibt wie in C eine Übergabe per Referenz. Auffällig ist, dass der Rückgabewert nicht unbedingt mit dem Schlüsselwort `return` zurückgeliefert wird, sondern Entwickler*innen im Code dem Compiler durch Weglassen des Semikolons signalisieren, dass hier das Ergebnis der Zeile zurückzugeben ist.

Es ist naheliegend, alle Fahrzeuge in einer gemeinsamen Datenstruktur zu verwalten. Dafür kommt ein Vektor (`Vec`) infrage, der ein dynamisches Feld darstellt und mit der Anzahl der Elemente wächst. Um dies zu implementieren, benötigt der Compiler zur Kompilierzeit die Größe der einzelnen Elemente. Für ein Trait wie `Vehicle` kann der Compiler das jedoch nicht bestimmen. Entwickler*innen haben aber die Möglichkeit, Referenzen zu einem `Vehicle` zu verwalten, denn deren Größe ist bekannt.

Das Beispiel in Listing 2 verpackt das Fahrzeug in eine Box. Der Datentyp `Box` stellt einen sogenannten Owning Smart Pointer dar, der das Objekt auf den Heap legt und automatisch löscht, sobald es nicht mehr in Verwendung ist.

Entsorgungssystem

Wie bereits erwähnt, bietet Rust ein sicheres Speichermodell an und verzichtet hierbei auf Garbage Collection, die zur Laufzeit versucht, nicht länger benötigte Speicherbereiche zu identifizieren und freizugeben. In Rust erfolgt dies bereits zur Kompilierzeit. Das Konzept der Ownership, das einen der größten Unterschiede zu C/C++ ausmacht, ermöglicht das.

Das Beispiel in Listing 3 zeigt die Möglichkeit, gültigen C++-Code zu schreiben, der zwar erfolgreich kompiliert wird, aber ungültige Speicherzugriffe enthält. Der Vektor z wird nach dem Verlassen des Codeblocks zwischen den geschweiften Klammern gelöscht, obwohl noch die Referenz x darauf existiert.

Ein entsprechender Fehler lässt sich auch in Rust programmieren (Listing 4). Im Gegensatz zu einem C/C++-Compiler übersetzt ein Rust-Compiler diesen Code nicht, obwohl er syntaktisch korrekt ist. Er meldet es als Fehler, dass z nicht lange

Listing 3: Beispiel einer ungültigen Referenz in C++

```
std::vector<std::string>* x = nullptr;

{
    std::vector<std::string> z;

    z.push_back("Hello World!");
    x = &z;
}

std::cout << (*x)[0] << std::endl;
```

Listing 4: Versuch zum Erzeugen einer ungültigen Referenz in Rust

```
let x;

{
    let z = vec!("Hello World!");

    x = &z;
}

println!("{}", x[0]);
```

Listing 5: Objekte ausleihen

```
let mut x = vec!("Hello World!");

{
    let z = &mut x;
    // Do something with z...
}

println!("{}", x[0]);
```

Listing 6: Generische Typen verwenden

```
struct Vector<T> {
    x: T,
    y: T,
    z: T,
}

fn do_something<T>(v: Vector<T>) {
    // do something with the vector v
}
```

Listing 7: Fehlerbehandlung in Rust

```
enum MathError {
    DivisionByZero,
    NonPositiveLogarithm,
    NegativeSquareRoot,
}

fn div(x: f64, y: f64) -> Result<f64, MathError> {
    if y == 0.0 {
      Err(MathError::DivisionByZero)
    } else {
      Ok(x / y)
    }
}
```

genug lebt. Um Speichersicherheit zu gewährleisten, führte Rust die Konzepte Ownership (Besitz) und Borrowing (Ausleihen) ein. Dabei gibt es immer nur einen Besitzer. Er hat das Recht, Objekte freizugeben und alle Ressourcen, etwa durch das Objekt benutzten Speicher, zurückzugeben. Durch das Erstellen von Referenzen lässt sich das Objekt verleihen (Listing 5).

Das Erstellen der Referenz auf x verleiht das Objekt. Die geschweiften Klammern sind in diesem Beispiel essenziell. Ohne sie verweigert der Compiler in der letzten Zeile das Ausleihen des Vektors an `println`, da nicht `x`, sondern `z` der Besitzer des Vektors ist. Es gilt die Grundregel, dass nur der aktuelle Besitzer etwas verleihen darf.

Das Ownership-Borrowing-Konzept ist gewöhnungsbedürftig, hat aber deutliche Vorteile: Die Speicherverwaltung ist sicher und nebenläufiger Code bei nur einem Besitzer einfach zu realisieren.

Rust ermöglicht die Verwendung generischer Typen. Datenstrukturen und Funktionen lassen sich so formulieren, dass beim Schreiben der genaue Typ nicht bekannt ist. Erst zur Kompilierzeit kann man ihn bestimmen und einsetzen.

Das Beispiel in Listing 6 definiert einen Vektor, bei dem noch nicht bekannt ist, ob eine einfach genaue (`f32`) oder eine doppelt genaue Fließkommazahl (`f64`) zur Beschreibung der Koordinaten verwendet wird. Daher kommt der generische Typ `T` zum Einsatz.

Um sinnvoll eine Berechnung in `do_something` zu realisieren, müssen Entwickler*innen bekannt geben, welche Methoden auf `T` anwendbar sind. Dazu dienen Traits, die die auf `T` ausgeführten Operationen beschreiben. `T` lässt sich nur für Typen substituieren, die die entsprechenden Traits implementieren. Der Trait `std::ops::Add` beschreibt, dass sich die Werte des Typs, der den Trait implementiert, addieren lassen.

Fehlerbehandlung

Vielen Programmiersprachen ist gemein, dass sie das Ergebnis nicht auf mögliche Fehler überprüfen. Typischerweise werden in Rust Ergebnisse in ein Result-Objekt (`Result<T, E>`) gepackt. Hier kann der Typ `T` ein beliebiger Datentyp für das Ergebnis sein, während `E` den Fehler repräsentiert.

Das Beispiel aus Listing 7 liefert durch die mathematische Division das Ergebnis als doppelt genaue Fließkommazahl zurück, während im Fehlerfall die Fehlerart durch `MathError` beschrieben ist. Die Rückgabe eines validen Ergebnisses erfolgt über `Ok`, ein Fehler wird in `Err` verpackt.

Der Aufruf der Funktion `div` erzwingt eine Auswertung des Rückgabewertes, um zu überprüfen, ob das Ergebnis korrekt war oder ein Fehler vorliegt. Es gibt sonst keine Möglichkeit, an das benötigte Ergebnis zu kommen. Im Aufruf lässt sich dies mit dem Schlüsselwort `match` realisieren, das den Rückgabewert

Listing 8: Auswerten des Rückgabewertes

```
match div(42.0, 2.0) {
    Err(why) => panic!("math error"),
    Ok(result) => println!("result {}", result);
}

let result = div(24.0, 0.0).unwrap();
println!("result {}", result);
```

Listing 9: Optionale Rückgabewerte verwenden

```
fn find(s1: String, s2: String) -> Option<usize> {
    ...
}

match find("Heise", "eise") {
    Some(i) => println!("Found substring at position {}", i),
    None => println!("Unable to found substring"),
}
```

auspackt und in den entsprechenden Zweig springt (Listing 8). Den Vorgang kann man mit `unwrap` verkürzen, das das Programm abbricht, wenn der Rückgabewert ein Fehler ist, und es andernfalls auspackt.

Es gibt aber auch Situationen, in denen Funktionen abhängig von ihren Eingaben kein Ergebnis zurückliefern können. Bei einer Suche innerhalb eines Strings stellt das Nichtauffinden eines Teilstrings nicht zwingend ein Fehler dar, sondern kann ein gewolltes Verhalten sein. In einem solchen Fall ist die Verwendung einer `Option<T>` zu bevorzugen, der Index der Fundstelle zurückzuliefern und diese in `Some` zu packen. Ist der Substring nicht auffindbar, gibt die Funktion `None` zurück. Das Auswerten des Ergebnisses lässt sich äquivalent mit der `match`-Anweisung realisieren (Listing 9).

Beliebte Sprache

Die immer noch relativ neue Programmiersprache Rust erfreut sich großer Beliebtheit und hat zuletzt viermal in Folge die Umfrage nach der beliebtesten Sprache auf Stack Overflow gewonnen. Den Einstieg erleichtert der Onlineeditor Rust Playground, für die ersten Schritte reicht der integrierte Compiler aus und selbst die lokale Installation ist dank rustup einfach gehalten. Wer sich ein Bild verschaffen will, welcher Editor welche Features unterstützt oder welche Plug-ins zu empfehlen sind, geht auf die Webseite „Are we (I)DE yet?“ (ix.de/z1rn). (nb@ix.de)

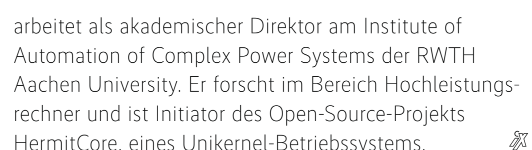
Quellen

Listings zum Download, zu Editoren und Plug-ins: ix.de/z1rn

Dr. Jens Breitbart

arbeitet als Softwarearchitekt bei Bosch Automated Driving. Er arbeitet zudem am Softwareprojekt HermitCore mit.

Dr. Stefan Lankes



arbeitet als akademischer Direktor am Institute of Automation of Complex Power Systems der RWTH Aachen University. Er forscht im Bereich Hochleistungsrechner und ist Initiator des Open-Source-Projekts HermitCore, eines Unikernel-Betriebssystems.