Mächtige Analysefunktionen
Data Science mit R

iX SPECIAL 2020

Anzeige
B1 SYSTEMS
Sie schreiben den Betrieb von Linux-Umgebungen aus?
Kontaktieren Sie uns – wir übernehmen!
info@b1-systems.de ROCKOLDING · BERLIN · KÖLN · DRESDEN Mehr dazu auf S. 9!

Moderne Programmiersprachen

Per Anhalter durchs Code-Universum

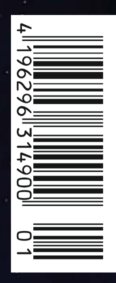
Neue Programmieransätze
Quanten-computing

Haskell, Idris, F#, Elixir, Elm
Funktionale Programmierung

Kotlin und Clojure vs. Java
Alternativen für die JVM

Von TypeScript bis WebAssembly
Moderne Webentwicklung

Go, Rust, D
C weitergedacht



14,90 €
Österreich 16,40 €
Schweiz 27.90 CHF
Luxemburg 17,10 €
www.ix.de

Hochverfügbarer S2D Micro-Cluster

Inklusive Azure Stack HCI Badges & Windows Server Solution Brief

Der extrem leistungsstarke S2D Micro-Cluster bietet Ihnen ein perfektes Rundumpaket. Zusammen mit der Kombination aus neuesten Datacenter Lizenzen von Microsofts Windows Server 2019 sorgt er für eine effiziente Storage-Verwaltung mit größtmöglicher Flexibilität. Zusätzlich haben wir extra für Sie eine spezielle Windows Admin Center Extension entwickelt: Damit haben Sie über eine grafische Weboberfläche das hochverfügbare 2-Node-System jederzeit im Blick . Neben der exzellenten Storage-Performance überzeugt der S2D Micro-Cluster außerdem mit seinem besonders kompakten Format – ideal für den Einsatz in Büroumgebungen.

Preisvorteil:
Basic Version statt
~~17.150,00 EUR~~

12.990,00
EUR

Jetzt erhältlich!

Im Thomas-Krenn-Onlineshop

Kontaktieren Sie uns:
+49 (0) 8551.9150-300

thomas-krenn.com/basic

Probieren Sie doch mal was Neues!

Laut einer aktuellen Studie von HackerRank suchen Unternehmen vor allem nach Fullstack-Entwicklern. An der Frage, was genau hinter dieser Bezeichnung steckt, scheiden sich die Geister. Sicher lässt sich aber sagen, dass Fullstack-Entwickler zumindest ein grundlegendes Verständnis aller Schichten eines Technikstacks benötigen – und damit mit unterschiedlichen Programmiertechniken und -sprachen umgehen können müssen.

In der Praxis trifft man dann freilich häufig auf Berge von Softwarealtlasten. Ein Freund und Ex-Entwickler hat einmal gesagt: „Kaum eine Softwareentwickler*in findet sich in IT-Projekten auf der sprichwörtlichen grünen Wiese wieder. Allzu oft türmen sich Legacy-Berge auf, die verwaltet oder abgetragen werden wollen.“

Doch die technische Erblast ist nur das eine. Das andere ist der Ruf nach Innovation. HackerRank berichtet, dass fast zwei Drittel der Befragten im letzten Jahr ein völlig neues Framework lernen mussten. Entwickler*innen müssen also den Spagat zwischen alt und neu schaffen. Oft kommen sie dabei auch nicht daran vorbei, eine neue Programmiersprache zu lernen. Persönliche Weiterbildung und permanentes Dazulernen ist unumgänglich in einem Arbeitsbereich, der sich so schnell ändert wie die Softwareentwicklung.

In den letzten Jahren ist die Auswahl an Programmiersprachen beachtlich gewachsen. Viele moderne Sprachen bringen neue Möglichkeiten, erleichtern beispielsweise das Programmieren nebenläufiger Anwendungen, bauen dabei jedoch auf altbekannten Konzepten auf. Entsprechend hält sich der Lernaufwand in Grenzen. Es lohnt sich also, gezielt nach Weiterentwicklungen oder Ablegern klassischer Sprachen zu schauen. Besonders nahe liegen dabei Verwandte der Sprachen, die in den eigenen Projekten bereits Verwendung finden. Aber auch völlig neue Welten haben ihren Reiz, wie alle Entwickler*innen bestätigen werden, die den Einstieg in die funktionale Programmierung gewagt haben.

Wer sich auf die Suche nach einer neuen Sprache macht, ist jedoch gut beraten, eine Reihe von Fragen im Gepäck zu haben. Zuerst einmal geht es darum, zu überprüfen, ob sich alt und neu vertragen. An welchen Stellen könnte es haken? Natürlich sind die Features der potenziellen Kandidatin wichtig, aber ebenso sehr stellt sich die Frage nach einer brauchbaren Dokumentation, nach Bibliotheken und Schnittstellen. Nicht zuletzt ist auch ein Blick in die Community aufschluss- und hilfreich. Wer treibt die Sprache voran, wie viele Leute arbeiten mit und wie rege ist die Beteiligung?

Den Blick über den Tellerrand zu wagen ist nicht nur faszinierend, sondern hilft auch, Probleme neu zu denken. Nicht jede neue Sprache muss gleich im nächsten Projekt zum Einsatz kommen. Schon die Beschäftigung mit neuen Ansätzen reicht aus, den eigenen Horizont zu erweitern.

CARINA SCHIPPER

C weitergedacht

C ist die Wurzel zahlreicher moderner Programmiersprachen. Dazu gehören nicht nur die Weiterentwicklungen C++ und C#, sondern auch Newcomer wie Rust, D und Go. Sie orientieren sich an der C-Syntax, bieten aber viele Verbesserungen und zeitgemäße Features.

ab Seite 12

Programmiersprachen heute

Am Anfang war … die Maschine **6**

Sprachwelten **10**

C weitergedacht

Modernes Programmieren mit C++20 **12**

Rust: nicht nur für den Browser **16**

Go für skalierbare und verteilte Systeme **20**

D – die C-Alternative **26**

Typsicherheit mit Swift **30**

C# 8.0: Änderungen bei Schnittstellen und Compiler **34**

Webanwendungen mit Blazor und C# **38**

Alternativen für die JVM

Java – die jüngsten Entwicklungen **46**

Einstieg in Kotlin **50**

Clojure: funktional programmieren auf der JVM **56**

Moderne Webentwicklung

Eine kurze Geschichte von ECMAScript **60**

TypeScript: JavaScript mit Typsystem **66**

ClojureScript: funktional im Browser **72**

WebAssembly für mehr Performance **78**

PHP: Programmiersprache fürs Web **84**

Das JavaScript-Framework Svelte **88**

Funktionale Programmierung

Elm-Apps als Web Components **98**

Elixir: pragmatisch und leicht zu erlernen **104**

Funktionale Programmierung mit Haskell und Idris **110**

Python: erste Wahl für Data Science und Machine Learning **118**

Perl-Nachfolger: Raku verstehen und anwenden **126**

F# in der Enterprise-Entwicklung **132**

Wissenschaftliches Rechnen

R: Statistikumgebung für Datenanalyst*innen **137**

Sentimentanalyse mit R **140**

Quantencomputer programmieren – ein Einstieg **146**

Julia für die Datenanalyse **152**

Rubriken

Editorial: Probieren Sie doch mal was Neues! **3**

Impressum, Inserentenverzeichnis **121**

Alternativen für die JVM

Seit über 20 Jahren belegt Java durchgängig Platz eins oder zwei im TIOBE-Ranking der populärsten Programmiersprachen. Allerdings lässt sich das Java-Ökosystem inzwischen auch mit anderen Sprachen nutzen: Kotlin glänzt mit kompakterer Syntax, Clojure bringt funktionale Programmierung auf die JVM.

ab Seite 46

Moderne Webentwicklung

Der Browser wird zunehmend zur Plattform der Wahl für die Anwendungsentwicklung. Neben JavaScript-Alternativen wie TypeScript und ClojureScript macht derzeit WebAssembly als Format zum Ausführen von Binärcode im Browser von sich reden.

ab Seite 60

Funktionale Programmierung

Funktionale Sprachen galten lange als theoretisch faszinierend, aber in der Praxis sperrig und umständlich. Doch in der neuen Welt der skalierbaren Anwendungen für verteilte Systeme liefert das funktionale Programmierparadigma einfache Lösungen für komplexe Probleme.

ab Seite 98

Wissenschaftliches Rechnen

Im Bereich des wissenschaftlichen Rechnens versuchen spezialisierte Sprachen wie R und Julia der Datenflut Herr zu werden. Quantencomputer sind mittlerweile mehr als technische Spielereien und verschieben die Grenzen des Möglichen.

ab Seite 137

Welche Programmiersprache ist die beste im Land?

Am Anfang war … die Maschine

Michael Stal

Sie sind nur ein Werkzeug für die Softwareentwicklung, wenn auch das wichtigste. Um sie entstehen Glaubenskriege und unterschiedliche Weltanschauungen: Gehen wir auf eine Reise durchs fabelhafte Universum der Programmiersprachen.

Ein historischer Blick auf die Entwicklung von Programmiersprachen verhilft zu einer besseren Perspektive auf die heutige Sprachlandschaft. Die ersten Programmiersprachen waren gar keine, sondern zwangen Programmierer*innen dazu, nah an der Maschine zu codieren. Stichwort Maschinensprache. Das erwies sich als fehlerträchtig und mühsam, erforderte viel Spezialisierung und machte eine Portierung auf andere Hardware so gut wie unmöglich. Das Lambdakalkül (Church, Kleene) sowie Zuses Plankalkül markieren die ersten Versuche, höhere Abstraktionsebenen einzuführen.

Die weitere Evolution von Programmiersprachen war letztendlich von dem Wunsch getrieben, immer höhere Abstraktionsebenen einzuführen, um Entwickler*innen näher an die Problemdomäne und weiter weg von den Niederungen der Maschine zu bringen. Seit den 1960er-Jahren kamen zunehmend Konzepte hinzu, um die Entwicklung großer Softwaresysteme zu fördern. Durch immer leistungsfähigere Computersysteme ließen sich auch stärkere Programmierumgebungen realisieren, um das Programmieren von einer Kunst weniger zu einem Handwerk für viele zu transformieren. Zudem hat die Durchdringung aller Lebensbereiche und Domänen mit Software zu einer Inflation allgemeiner und spezialisierter Programmiersprachen geführt. Es fällt zunehmend schwerer, einen Überblick zu behalten und für eine Problemstellung die geeignete Sprache auszuwählen.

Die Qual der Wahl

Mittlerweile lesen sich Aufreihungen von Programmiersprachen schon fast wie ein Adressbuch. Beispielsweise gibt es in Wikipedia eine große Vergleichsliste bekannter Sprachen, deren Unterscheidung anhand der unterstützten Paradigmen erfolgt. Solche Verzeichnisse erlauben einen tiefergehenden Einblick in die Vielzahl existierender Programmiersprachen, ermöglichen aber keine gezielte Auswahl. Ihre Beliebtheit ist die naheliegendste Art, verschiedene Programmiersprachen zu vergleichen und einzuordnen. Vergleichsindizes wie TIOBE oder RedMonk stellen eine Reihenfolge der beliebtesten Programmiersprachen dar. Als Basis dient dabei die Zahl der Erwähnungen auf Webseiten wie Stack Overflow. Im Mai 2020 lautete zum Beispiel die Reihenfolge im TIOBE-Index wie in der Abbildung gezeigt.

Die üblichen Verdächtigen wie Java, C, Python rangieren dort auf den Spitzenplätzen. Ganz abgesehen davon, dass die diversen Indizes verschiedene Reihenfolgen produzieren, sind sie zwar

interessant, aber wenig hilfreich bei der Entscheidung, welche Programmiersprache sich für ein Problem am besten eignet. Für manche Aufgabenstellung kann eine Nischensprache die richtige Wahl sein, obwohl sie auf dem TIOBE-Index erst in den unteren Regionen auftaucht.

Sprachen klassifizieren

Zum Vergleich und zur Betrachtung verschiedener Programmiersprachen existiert eine ganze Reihe von Kriterien. Eine rein technische Liste von systemnahen Aspekten wie Sprachmerkmalen oder Syntax genügt allerdings nicht. Stattdessen ist essenziell, welches Ziel ein Projekt verfolgt, um die dafür geeigneten Sprachen zu eruieren. Folgender Katalog versucht, ein paar Bezugspunkte für die Auswahl zu setzen, ohne Anspruch auf Vollständigkeit zu erheben.

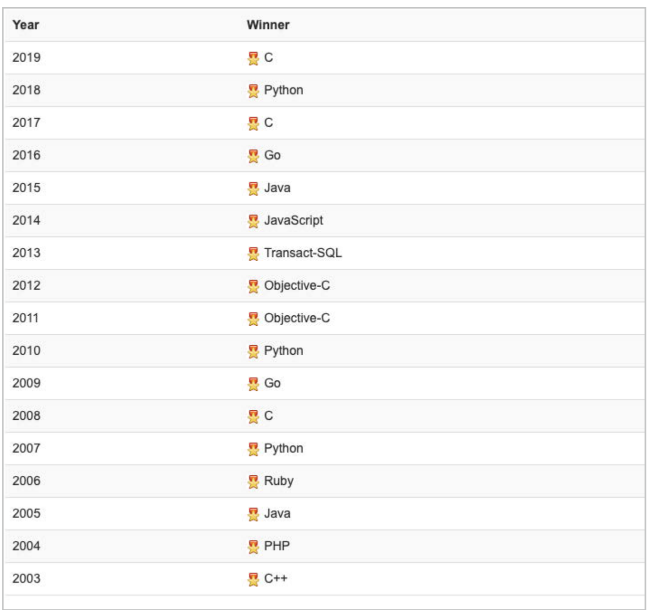
Enthalten sind sowohl technische als auch nicht technische Aspekte. Welche Art von System möchten Entwickler*innen erstellen? Zum Beispiel eine reine Geschäftslösung, ein eingebettetes System, einen Echtzeitprozess zur Automatisierung, eine browserbasierte Anwendung, eine Edge-Anwendung, eine Software-as-a-Service-Anwendung?

Für jede Problemdomäne existieren entsprechende Programmiersprachen. Im Falle einer Echtzeitanwendung erweist sich zum Beispiel JavaScript als völlig ungeeignet, ganz im Gegensatz zu webbasierten Frontends oder Backends. Traditionelle Sprachen wie C oder C++ glänzen hingegen für systemnahe Domänen à la Embedded und Echtzeit. Go gilt als die Sprache der Wahl für verteilte Netzwerkanwendungen, während Elixir als Sprache mit Ruby-Wurzeln hervorragend für IoT-Edge-Knoten oder verteilte eingebettete Systeme prädestiniert ist.

Moderne Programmiersprachen wie Swift oder Rust implementieren in der Regel mehrere Paradigmen, wobei objektorientiertes Programmieren mittlerweile ebenso zu den Standards gehört wie funktionale Programmierkonzepte. Generisches oder generatives Programmieren taucht häufig auf, während Ansätze für sicheres Programmieren wie Ownership in Rust selten anzutreffen sind.

Ein weiteres Gebiet ist die Unterstützung ereignisorientierten Programmierens mit Sprachen wie F#, Groovy oder Kotlin. Konzepte zur Modularisierung wiederum spielen in vielen Sprachen eine wichtige Rolle. Meta Programming, also die Fähigkeit einer Software, dynamisch Näheres über sich selbst in Erfahrung zu bringen oder sogar eigene Eigenschaften zu verändern, findet sich heute in Sprachen wie zum Beispiel Clojure oder Prolog.

Prolog und Erlang gehören zur Familie der Programmiersprachen, die deklarativ statt imperativ arbeiten. Die meisten Sprachen verfolgen also einen Multiparadigmenansatz. Typisches Beispiel ist Scala, das beispielsweise Generizität, funktionale Programmierung und objektorientierte Programmierung in sich vereint und sogar interne DSLs ermöglicht.



Year	Winner
2019	C
2018	Python
2017	C
2016	Go
2015	Java
2014	JavaScript
2013	Transact-SQL
2012	Objective-C
2011	Objective-C
2010	Python
2009	Go
2008	C
2007	Python
2006	Ruby
2005	Java
2004	PHP
2003	C++

TIOBE-Index Mai 2020

Unterstützung durch Dritte

Das Laufzeitsystem einer Sprachimplementierung ist die eine Seite der Medaille. Die andere ist die Verfügbarkeit von Zusatzbibliotheken. Moderne Sprachen wie Rust versuchen die Sprache möglichst klein zu halten und wichtige Funktionen stattdessen in Standardbibliotheken vorzuhalten. Da eine Programmiersprache und ihre Bibliotheken in der Praxis nur die Spitze des Eisbergs darstellen, gehört auch eine Betrachtung der Unterstützung durch Dritte zu den wichtigen Aspekten. Beispielsweise die Verfügbarkeit weiterer Werkzeuge wie IDEs, Bibliotheken, Frameworks, Datenbanken, Cloud-Integration und Tools. Oder auch die Verbreitung entsprechender Communitys und die Verfügbarkeit von Fachliteratur oder anderen Lernquellen.

Um moderne Multicoreprozessoren optimal auszureizen, sollte die Laufzeitumgebung einer Programmiersprache entsprechende Concurrency-Konzepte bereitstellen. Dazu gehören Parallelisierung durch Multithreading und Synchronisationsmechanismen für den Zugriff auf gleiche Ressourcen. Im Idealfall verlässt sich eine Sprache auf ein eigenes einheitliches konzeptionelles Modell, statt direkt auf Betriebssystemebene durchzugreifen.

Beispielsweise glänzt Rust durch ein integriertes Modell für Concurrency. Sein Ownership-Modell bei Daten vermeidet mögliche Race Conditions schon zur Übersetzungszeit gezielt. Hier zeigt sich übrigens, dass eine geschickte Wahl von Sprachmerkmalen positive Konsequenzen auf Parallelität besitzt. In Sprachen auf Basis der BEAM VM wie Erlang und Elixir laufen leichtgewichtige Threads geschützt voneinander auf separaten CPU-Kernen. In diesen Fällen bietet das Laufzeitsystem Unterstützung für Concurrency.

Brückenbau

Viele Anwendungen sind über mehrere Systeme verteilt, zum Beispiel Geschäftsanwendungen oder IoT-Systeme. Daher erweisen sich Mechanismen für die prozessübergreifende Kommunikation als wichtig. Die Art der Kommunikation, etwa synchron, asynchron oder nachrichtenbasiert, spielt dabei eine ebenso große Rolle wie die darunterliegenden Kommunikationsprotokolle und Datenübertragungsformate, etwa RESTful, JSON oder

iX-TRACT

- Die ersten Programmiersprachen waren Maschinensprachen, die eine hohe Spezialisierung der Programmierer*innen erforderten. Erst mit der Zeit entwickelten sich höhere Abstraktionsebenen, die den Fokus weg von der Maschine hin zu den Problemdomänen lenkten.
- Da sich jede Sprache aufgrund ihres Charakters nur für bestimmte Einsatzzwecke eignet, ist es ratsam, nicht an einer festzuhalten, sondern mehrere zu erlernen.
- Bei der Wahl der Sprache gilt es, das Projektziel im Blick zu haben. Es ist zu fragen, welche Art von System zu entwerfen ist: ein eingebettetes, eine Edge- oder SaaS-Anwendung, ein Echtzeitprozess zur Automatisierung oder eine reine Geschäftslösung.

TCP/IP. Die Implementierung von Verteilungs- und Kommunikationsaspekten erfolgt meistens in Bibliotheken.

Da nicht alle Projekte auf der grünen Wiese beginnen, sondern sich in eine bereits bestehende Umgebung einfügen müssen, ist entscheidend, wie weit die gewünschte Programmiersprache dies überhaupt zulässt. Gibt es zum Beispiel Anbindungen an C- oder C++-Code, ist der Aufruf existierender C- oder C++-APIs möglich. Zum Zugriff auf Microservices bedarf es REST-Unterstützung. Was auf den ersten Blick oft einfach erscheint, offenbart auf den zweiten Blick manchmal eine große Komplexität. Wer schon einmal versucht hat, die C-Anbindung von Python zu durchdringen, kann ein Lied davon singen.

Für die einen gelten statische Typsysteme als erste Wahl, weil sie schon zur Übersetzungszeit Typfehler erkennen und als Komfort Typinferenz mitbringen, während andere auf dynamische Typsysteme schwören, frei nach dem Erlang-Mantra: „Let it crash." Statische Sprachen sind meist übersetzte Sprachen, für die ein Compiler Maschinencode erzeugt, dessen Ausführung auf einer realen oder virtuellen Maschine erfolgt. Die meisten Skriptsprachen sind hingegen dynamisch, das heißt, ihre Ausführung erfolgt in einem Interpreter – Stichwort: REPL (Run-Eval-Print Loop).

Lernkurve

Wer bereits einschlägige Erfahrungen mit Java oder C# besitzt, dürfte mit dem Erlernen anderer Sprachen aus der C-Familie wie Swift nur wenig Mühe haben. Erfahrenen Entwickler*innen mit JDK- oder .NET-Vorkenntnissen fällt es leichter, sich mit anderen Sprachen aus dem JVM- oder CLR-Universum anzufreunden. Die Lernkurve ist somit ein wichtiges Kriterium bei der Sprachenwahl. Das gilt speziell in solchen Fällen, in denen eine komplette Organisation oder ein Projekt sich für eine „neue" Programmiersprache entscheiden muss.

Zur Komplexität einer Programmiersprache trägt auch fehlende Transparenz bei. In C++ gibt es eine ganze Menge von Seiteneffekten, die Entwickler*innen beachten müssen, um in keine Falle zu laufen, etwa in Hinblick auf das Erzeugen und Löschen temporärer Variablen durch das Laufzeitsystem. Sprachen wie C übertragen die Bürde der Speicherverwaltung weitgehend auf Programmierer*innen. Dieses Vorgehen sorgt für die notwendige Freiheit bei systemnaher oder effizienzorientierter Software, verpflichtet Entwickler*innen aber andererseits zu größerer Disziplin und Vorsicht. Garbage Collection bei Java und C# reduziert die Freiheiten zugunsten der Programmiersicherheit, macht aber Speicherfreigaben intransparent.

In der Compilersprache Go findet sich zwar ein Garbage Collector, aber im Gegensatz zu Java und C# organisiert Go den Heap nicht in Generationen und führt des Weiteren eine Analyse durch, ob Objekte auf dem Heap oder auf dem Stack landen sollen. Sprachen mit Automatic Reference Counting wie Swift verfolgen einen Mittelweg, bei dem der Compiler Retain- und Release-Instruktionen in den Code einfügt. Zyklische Abhängigkeiten lassen sich allerdings nicht automatisieren, sondern verlangen die explizite Mithilfe der Entwickler*innen.

In RAII-Ansätzen (Resource Acquisition is Initialization) wie bei C++ oder Rust nutzt das Laufzeitsystem ein Lebenszyklusmodell für Objekte. Sie beginnen ihr Leben im Konstruktor und beenden es im Destruktor. Das Ownership-Modell in Rust setzt auf RAII auf. Es erfordert, dass jeder Datenwert eine Variable als Besitzer hat. Die Ownership kann zum Beispiel bei Funktionsaufrufen auf den Empfänger übergehen. Gerät ein Objekt außerhalb des Bereiches seines Besitzers, gibt es das Laufzeitsystem mithilfe eines Referenzzählermechanismus frei.

Erfolgsgeschichten

Sobald eine bestimmte Programmiersprache in den Fokus und damit in die engere Auswahl gerät, lohnt sich der Blick darauf, welche Erfahrungen andere Projekte mit der Sprache der Wahl gemacht haben. In der Ära von Open-Source-Software und dem Internet finden sich zum Glück recht schnell entsprechende Informationsquellen und Codebasen. Im Optimalfall lassen sich dank Google, Bing oder DuckDuckGo sogar vergleichbare Projekte aufspüren, die zum eigenen Profil passen.

Für einzelne Programmierer*innen wahrscheinlich wenig relevant, aber für kommerzielle Projekte von maßgeblicher Bedeutung dürfte die Frage sein, wer eine Programmiersprache kontrolliert. Einige Programmiersprachen unterliegen der Kontrolle von Forschungseinrichtungen, Open-Source-Konsortien oder kommerziellen Unternehmen, etwa Rust, Scala, Swift, Java oder Kotlin. Andere haben einen Standardisierungsprozess durchlaufen wie C++ oder Ruby.

Beide Ansätze bringen Vor- und Nachteile mit sich. So hat Andrew Tanenbaum bereits vor Jahren treffend zum Thema Standardisierung formuliert: „Das Schöne an Standards ist, dass es so viele davon gibt, aus denen sich auswählen lässt." Standardisierungsprozesse erlauben Nutzer*innen bessere Einflussnahme, können aber auch sehr zäh verlaufen.

Das Projektziel sich immer vor Augen halten

Das Erlernen einer neuen Programmiersprache erfordert eine große Investition von Organisationen und Entwickler*innen. Daher fällt es oft schwer, ausgetretene Pfade zu verlassen, um neues Terrain zu erforschen. Heutige Entwicklungsprojekte adressieren zumeist verschiedene Lösungsdomänen, weshalb das starre Festhalten an einer einzigen Sprache schon längst nicht mehr funktioniert.

Hemmungsloses Anwenden einer „Jeder Topf bekommt seinen eigenen Deckel"-Strategie erweist sich aber ebenfalls als kontraproduktiv. Wer möchte schon eine polyglotte Codebasis verwalten, die unübersichtlich viele Sprachen umfasst. Wenn aber schon eine neue Programmiersprache her muss, dann sollte sich die Auswahl nicht nur am „Fun"-Faktor, sondern hauptsächlich an den eigenen Bedürfnissen orientieren. Immerhin gibt es mittlerweile eine Vielzahl von Programmiersprachen mit unterschiedlichsten Konzepten und Neuerungen. Da fällt die Auswahl zunehmend schwer.

Auf der positiven Seite bleibt festzuhalten, dass man heute aus einer Vielzahl von Sprachen auswählen kann und dass andererseits alle Sprachen auf Konzepte von Vorgängersprachen aufbauen, weshalb sich die Lernkurve für professionelle Entwickler*innen mit jeder neuen Sprache abflacht. Nicht zuletzt macht es Spaß, den eigenen Horizont durch eine neue Programmiersprache zu erweitern. Jedes neue Werkzeug erhöht die eigenen Möglichkeiten. Bekanntlich ist Spaß die beste Motivation. (nb@ix.de)

Prof. Dr. Michael Stal

beschäftigt sich bei der Corporate Technology der Siemens AG mit Software- und Systemarchitekturen, Digitalisierung und KI. An der University of Groningen in den Niederlanden hält er Vorlesungen und betreut Doktoranden.

B1 Managed Service & Support

individuell – umfassend – kundenorientiert

Neue oder bestehende Systemlandschaften stellen hohe Anforderungen an Ihr IT-Personal. Mit einem individuellen Support- und Betriebsvertrag von B1 Systems ergänzen Sie Ihr Team um die Erfahrung und das Wissen unserer über 120 festangestellten Linux- und Open-Source-Experten.

Unsere Kernthemen:

Linux Server & Desktop · Private Cloud (OpenStack & Ceph) · Containerization (Docker) · Orchestration (Kubernetes) · Monitoring (Icinga, Nagios & ELK) · Patch Management · Automatisierung (Ansible, Salt, Puppet & Chef)

Wir sind für Sie da – mit qualifizierten Reaktionszeiten ab 10 Minuten und Supportzeiten von 8x5 bis 24x7!

B1 Systems GmbH - Ihr Linux-Partner

Linux/Open Source Consulting, Training, Managed Service & Support

ROCKOLDING · KÖLN · BERLIN · DRESDEN

www.b1-systems.de · info@b1-systems.de

Sprachwelten

Jenseits der eigenen Galaxie

Carina Schipper

Das Universum der Softwareentwicklung erstreckt sich über schier unendliche Weiten. Wer sich darin bewegt, braucht vor allem eins: Entdecker*innengeist.

Angesichts des Sprachspektrums, um das sich vielfältige Ökosysteme entwickelt haben, fällt es oft schwer, vor lauter Bäumen den sprichwörtlichen Wald zu sehen. Gefühlt jagt eine Innovation die nächste. Projekte verlangen von Softwareentwickler*innen ein hohes Maß an Spezialisierung und viel Detailwissen. Zeitgleich zwingt der stetige Wandel innerhalb der IT sie, sich mit neuen Herausforderungen auseinanderzusetzen und das gesamte Umfeld um das eigene Projekt im Blick zu behalten. Als eines von zahlreichen Beispielen für den Wandel und dessen weitreichende Konsequenzen lässt sich die Cloud mit Entwicklungen wie Platform as a Service und Serverless Computing anführen.

Von I-Shape zu T-Shape

Wissen in die Tiefe in einem oder nur wenigen Gebieten reicht nicht mehr aus, um im Arbeitsalltag als Programmierer*in zu bestehen. Immer neue Anforderungen beispielsweise durch den Einsatz von KI, die Verarbeitung immer größerer Datenmengen oder neue Programmierparadigmen wie Serverless führen das Bild der hoch spezialisierten I-Shape-Entwickler*in ad absurdum. Breitenwissen ist gefragt, auf drei Ebenen, der technischen, fachlichen und sozialen. Daraus ergibt sich vereinfacht das Bild eines T-Modells (siehe Abbildung). Bekannt ist dieses Konzept vor allem aus dem Bereich der agilen Softwareentwicklung – Stichwort crossfunktionale Teams.

Übertragen auf den Aspekt der Programmiersprachen bedeutet das, die Sphäre der eigenen Sprache zu verlassen und in neue Gefilde einzutauchen. Dabei geht es nicht unbedingt darum, sich in eine polyglotte Entwickler*in zu verwandeln, die alle möglichen Programmiersprachen fließend spricht. Schon sich mit den grundlegenden Merkmalen anderer Sprachen oder beispielsweise der Frage, wie sie mit Problem der Nebenläufigkeit umgehen, zu beschäftigen, kann hilfreich für das eigne Projekt sein. Daneben lohnt sich auch ein Blick auf Themen wie Programmierparadigmen und Softwarearchitekturen.

Spitzenreiter, Dauerbrenner und Newcomer

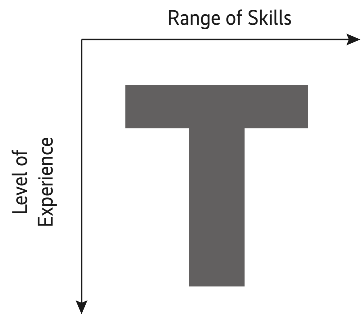
Ein Blick auf den TIOBE-Index liefert eine monatliche Momentaufnahme der Beliebtheit verschiedener Programmiersprachen. Die oberen fünf Plätze der Hitliste sind seit Jahren mit den üblichen Verdächtigen besetzt. Hier rangiert die alteingesessene Prominenz der Szene. Beispielsweise gewinnt im Mai 2020 C gegen Java den scheinbar ewig währenden Kampf um

die absolute Spitzenposition. Vergleicht man die Liste der Dauerbrenner über die letzten 30 Jahre hinweg, tauchen die beiden dort ebenfalls auf dem Siegertreppchen auf. Aber auch Python, C++, C# oder JavaScript feiern schon lange immer wieder Erfolge. Zu den Neueinsteigern im TIOBE-Index zählen Sprachen wie Go, Kotlin oder TypeScript, wobei Go der einzige relativ junge Kandidat ist, der es in die Top 20 schafft. Die anderen Newcomer versammeln sich eher im Mittelfeld des Rankings.

Der TIOBE-Index und andere Quellen zur Verbreitung von Programmiersprachen liefern zwar eine Rangliste, sagen aber nichts darüber aus, warum sich einzelne Vertreter mehr oder weniger großer Beliebtheit erfreuen. Wer wissen will, was es damit auf sich hat, braucht mehr als ein Kratzen an der Oberfläche. Die Zusammenstellung der Artikel in diesem Sonderheft erhebt sicherlich keinen Anspruch auf Vollständigkeit. Trotzdem können die Texte als erster Einstieg dabei helfen, den Sprung in neue (Sprach-)Welten zu wagen.

Und wie geht es dann weiter?

Zweifellos, das Berufsbild der Software*entwicklerin geht Hand in Hand mit der Evolution der Technik. Vor allem in der jüngsten Vergangenheit scheint die Digitalisierung vehement um sich zu greifen und in nahezu alle Bereiche des Lebens vorzustoßen. Programmieren ist schon lange nicht mehr nur die Sache dezidierter Expert*innen. Da gibt es zum Beispiel die Schar der Blogger*innen, die sich mit HTML, CSS und JavaScript eigene



Softwareentwickler*innen brauchen neben Detail- auch Breitenwissen.

Webseiten bauen, oder Fachkräfte in Unternehmen, die eigene Datenauswertungen programmieren.

Auch in Jobs, die vermeintlich fernab der IT liegen, sind daher IT-Skills immer wichtiger. Karrierebibel etwa bezeichnet Fähigkeiten wie einen Onlineshop zu modifizieren oder ein Add-on für eine Anwendung zu schreiben als unschlagbare Zusatzqualifikationen (siehe ix.de/z8ey).

Zusätzlich werfen neue Techniken wie Low- und No-Code-Plattformen eine Reihe von Fragen auf. Wo geht die Reise in der IT hin und welche Rolle spielen Softwareentwickler*innen darin? Zwei Dinge sind sicher: Die Zeit steht nicht still und es lohnt sich, mit ihr zu gehen und den eigenen Horizont zu erweitern.

(nb@ix.de)

Quellen

Alle Links unter ix.de/z8ey

Online-Konferenz

Neue Cyberangriffe –

Wie können Unternehmen sich schützen – Kritische Infrastrukturen & Industrie 4.0 im Fokus

27. August 2020, 9 – 16 Uhr

Neben der Notwendigkeit von Cyberabwehrmaßnahmen werden auf dieser Online-Konferenz auch verschiedene Konzepte und Vorgehensweisen zur Sicherung der unternehmensinternen IT- und Prozessnetze dargelegt.

Dabei sollen folgende Fragen behandelt werden: Welche Tools benötigt ein Unternehmen, um sich gegen aktuelle Cyberangriffe abzusichern? Wie werden potentielle Angriffe möglichst schnell erkannt? Und welche Vorgehensweisen sind am besten zur Bedrohungsanalyse geeignet?

In Kooperation mit

Preis: 159,00 Euro

©LuckyStep

www.heise-events.de/konferenzen/it-sicherheitstag-august

Modernes Programmieren mit C++20

Paradigmenwechsel

Nicolai Josuttis

Mit C++20 steht eine große Release vor der Tür. Die neuen Features Modules, Coroutines, Concepts und Ranges ändern die Art, mit C++ zu programmieren.

C++20 ist seit Februar 2020 inhaltlich finalisiert und wartet auf die formale Verabschiedung als Standard. Nach zwei kleineren Releases (derzeit erscheint alle drei Jahre eine neue C++-Version) handelt es sich diesmal wieder um eine große, die ähnlich wie C++11 das Programmieren mit C++ nachhaltig verändern wird. Die großen Neuerungen sind Concepts, Ranges, Modules und Coroutines.

Die erste Neuerung betrifft generische Programmierung. Man kann nun auch außerhalb von Lambdas `auto` als Parameter verwenden:

```
void printColl(const auto& coll) {
    for (const auto& elem : coll) {
      std::cout << elem << '\n';
    }
}
```

Diese Funktion ist im Prinzip ein Funktionstemplate für einen Parameter `coll` von beliebigem Typ. Konzepte erlauben es nun, Anforderungen und Einschränkungen an generische Datentypen zu formulieren. Dabei handelt es sich um eine mit einem Namen versehene Spezifikation der von einem Datentyp unterstützten Eigenschaften und Operationen. Damit kann man dokumentieren und sicherstellen, dass ein Template wie oben nur bestimmte Typen unterstützt (Listing 1; alle Listings siehe ix.de/zwbr).

Die in Listing 1 definierte generische Funktion `print()` lässt sich nur für Datentypen einsetzen, für deren Objekte man `begin()` und `end()` aufrufen kann. Das schützt nicht nur vor Fehlverhalten, sondern verbessert auch die Fehlermeldungen beim Verwenden von generischem Code. Die C++20-Standardbibliothek definiert hierzu über fünfzig Standardkonzepte wie `predicate` oder `movable`.

Der zweite große Meilenstein sind Ranges. Mit der Aufnahme der Standard Template Library (STL) in den ersten C++-Standard etablierte sich das grundsätzliche Prinzip, dass man zum Bearbeiten aller Elemente einer Menge einen halboffenen Bereich in Form von zwei Parametern, Anfang und Ende, übergibt:

```
std::sort(coll.begin(), coll.end());
```

Die neue Ranges-Bibliothek ersetzt dieses Prinzip nun durch die Option, ganze Mengen als einen Parameter zu übergeben:

iX-TRACT

- Die inhaltlichen Arbeiten von C++20 sind abgeschlossen, der Standard liegt zur endgültigen Genehmigung und Veröffentlichung dem ISO-Komitee vor (DIS, Stand: April 2020).
- Das Update der Sprache enthält zahlreiche Neuerungen. Wesentliche sind Ranges, Module, Konzepte (Concepts) und Koroutinen (Coroutines).
- Weitere Features sind der Drei-Wege-Vergleichsoperator <=>, Synchronisierungsmechanismen wie Semaphore, Latches und Barriers sowie Kalender- und Zeitzonenerweiterungen für die chrono-Bibliothek.

Listing 1: Verwenden von Concepts und auto in Funktionen

```
template <typename T>
concept IsContainer = requires(const T& t) {
    { t.begin() };
    { t.end() };
};

void print(const IsContainer auto& coll) {
    for (const auto& elem : coll) {
      std::cout << elem << ' ';
    }
}
```

Listing 2: Iteration mit einer Ranges-View

```
// Spezifikation einer View:
rng::filter_view over36View{coll | rng::views::filter(isOver36)};

// Verwendung der View (hier wird iteriert):
for (int i : over36View) {
  std::cout << "over Limits: " << i << '\n';
}
```

```
std::ranges::sort(coll);
```

Eine Unix-Pipe-ähnliche Syntax erlaubt es dabei, Mengen in Filterketten zu bearbeiten:

```
auto isOver36 = [] (int val) { return val > 36; };

for (int i : coll | rng::views::filter(isOver36)) {
    std::cout << "over 36: " << i << '\n';
}
```

Dazu lassen sich auch Views und Adapter definieren (Listing 2). Zu beachten ist, dass erst die `for`-Schleife tatsächlich über die Elemente iteriert. Das geschieht allerdings nicht nebenläufig, etwa durch mehrere Threads oder Prozesse.

Mit Modulen arbeiten

Als dritter großer Meilenstein gilt die Aufnahme von Modulen. Diese gehen das Problem der physischen Strukturierung von umfangreichem Quellcode mit Millionen oder Milliarden von Codezeilen an. Die große Verbreitung von Templates führt dazu, dass immer wieder der gleiche Code kompiliert. Auch ist es schwierig, Komponenten zu definieren, die aus mehreren Dateien bestehen und nach außen eine klare Schnittstelle haben. Module bieten nur die Möglichkeit, Komponenten zu definieren. Neben einer sauberen Schnittstellendefinition lässt sich ein Modul, auch wenn es in mehreren Dateien implementiert wird, eigenständig kompilieren.

Hier ein einfaches Beispiel: Das Modul `My.Account` definiert seine Schnittstelle in genau einer Datei (Listing 3). Darin exportiert die Schnittstelle die Klasse `Customer` und die Funktion `print()`. Aber auch private Aspekte sind definierbar. Denn bei der Verwendung des Moduls müssen die Objektgrößen und manche Definitionen bekannt sein, um das Kompilieren von Code und Optimierungen zu ermöglichen.

Um Code zu kompilieren, der das Modul nutzt, ist nur ein Import dieser Schnittstelle erforderlich:

```
import My.Account;

Customer c{"test"};
print(c);
```

Listing 4 zeigt, wie das Modul dann in mehreren Dateien implementiert wird.

Module lösen damit das Konzept von Headerdateien ab, die vor allem den kompletten Code von Templates enthalten. Es entstehen aber auch neue Dateiformate für vorkompilierte Module und Repositories, um den Code später zu Programmen zusammenzubinden. Eine sanfte Migration von Headerdateien als Schnittstellenbeschreibung wird unterstützt.

Listing 3: Definition einer Modulschnittstelle

```
export module My.Account;

export class Customer {
      std::string name;
    public:
      Customer(std::string name);
      std::string getName() const;
};

export void print(const Customer&);

void helper(); // private in module
```

Listing 4: Modul in mehreren Dateien implementieren

```
// 1. Datei mit Modulimplementierungen
module My.Account;

#include <iostream>

void print(const Customer& c) {
    std::cout << c << '\n';
}

// 2. Datei mit Modulimplementierungen
module My.Account;

void helper() {
    ...
}
```

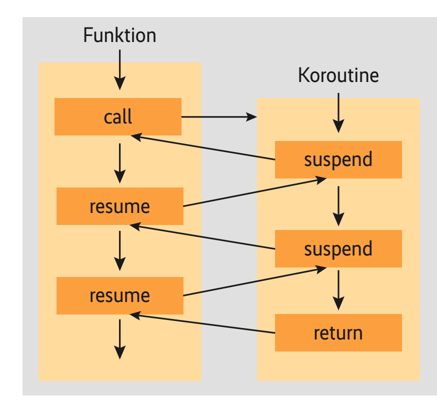
Die praktische Verwendung von Modulen bedarf noch der Klärung technischer Details und Konventionen. Offene Punkte sind deren Dateiendungen, wo und in welchem Format sich vorkompilierte Module speichern lassen, ob und wie diese zwischen verschiedenen Compilern austauschbar sind und wie die C++-Standardbibliothek von Headerdateien auf Module umzustellen ist. Eine Arbeitsgruppe befasst sich im Rahmen der weiteren Standardisierung von C++ damit.

Auch bei einigen anderen C++20-Features wurden erste Teile von neuen Sprachmitteln eingeführt, mit der Idee, diese dann in nachfolgenden C++-Versionen noch zu ergänzen und abzurunden. C++20 stellt in dieser Hinsicht nur den Beginn eines Zyklus von zwei bis drei Releases dar.

Ein Beispiel ist das Verlagern von Berechnungen von der Laufzeit auf die Kompilierzeit. Im Kontext von `constexpr` können jetzt zwar dynamischer Speicher, `try/catch` (nicht `throw`), virtuelle Funktionen, zwei neue Schlüsselwörter (`consteval` und `constinit`) und damit auch Strings und Vektoren zur Kompilierzeit verwendet werden, allerdings lassen sich noch keine Mengen zur Kompilierzeit aufbauen und dann zur Laufzeit verwenden. Alle Features sind erforderlich, um vermutlich mit C++23 Reflection einzuführen.

Asynchrone Programmierung

C++20 ermöglicht auch ein erstes Arbeiten mit Koroutinen. Das sind Funktionen, die (nach einem Zwischenergebnis) unterbrochen und später wieder aufgenommen werden können (siehe Abbildung). Die Aufgabe einer Koroutine könnte zum Beispiel sein, nach und nach das nächste Element einer Menge zu liefern (Listing 5).



Koroutinen können unterbrochen und fortgesetzt werden.

Listing 5: Eine Koroutine hält die Ausgabe an und setzt sie später wieder fort

```
IntGen nextElemGen(const auto& coll)
{
    for (int elem : coll) {
      co_yield elem;  // Schleife und Funktion mit elem als ↵
                                     Zwischenergebnis anhalten
    }
}
```

Mit `co_yield` wird die Funktion zunächst angehalten und der aktuelle Wert von `elem` zurückgeliefert. Eine weitere Funktion kann diese nun nach und nach nutzen, um zusätzliche Werte abzurufen, was die Koroutine dann fortsetzt:

```
IntGen gen = nextElemGen(coll);
for (int val : gen) {  // iteriert über die jeweils fortgesetzte
                       //Coroutine
    doSomeStuff(val);
}
```

Das Bindeglied zwischen der Koroutine und ihrem Aufruf ist eine Generator-Klasse, die den Zustand von Koroutinen hält. Sie zu implementieren, ist nicht trivial. Man muss dies noch selbst erledigen (Listing 6). Insofern benötigen auch Koroutinen noch Ergänzungen, die die nächsten Releases sicher enthalten.

Ein weiteres Beispiel einer schrittweisen Verbesserung ist der neue Datentyp für UTF-8-Zeichenliterale `char8_t`. Das Typsystem stellt damit endlich die semantische Bedeutung eines UTF-8-Zeichens sicher. Darauf aufbauend wird es in den nächsten C++-Versionen signifikante Verbesserungen im Umgang mit UTF-8 geben.

Raumschiffoperator

Neue Regeln und ein neuer Operator machen die Definition von Vergleichsoperatoren einfacher. Zunächst ist in Klassen mit `operator==` nun auch automatisch `operator!=` definiert. Für andere Vergleiche liegt der neue Drei-Wege-Vergleichsoperator <=> vor, auch Raumschiffoperator genannt: `x <=> y` liefert einen Wert kleiner 0, wenn x kleiner ist, 0, wenn x und y gleich sind, und größer 0, wenn y größer ist. Seine Definition in Klassen definiert implizit die anderen Vergleichsoperatoren (wenn keine Probleme vorliegen). So kann man in Klassentemplates grundsätzlich alle Vergleichsoperatoren wie in Listing 7 definieren. Als Default erfolgt der Vergleich über die ganze Hierarchie Member-weise.

Während in C++17 String-Views Einzug hielten (Read-only-Sichten mit einem String-Interface auf existierende externe Zeichenfolgen), enthält C++20 nun Sichten auf existierende Sequenzen beliebiger Objekte mit Schreibzugriff. Damit können Entwickler*innen Code implementieren, der sequenzielle Mengen oder Teilmengen bearbeitet, ohne sie zu besitzen. Die folgende Funktion verdoppelt alle Elemente einer Sequenz von Integerwerten (siehe auch Listing 8):

```
void doubleContent(std::span<int> coll) {
    for(auto& val : coll) {
      val *= 2;
    }
}
```

Spans sind keine Templates, sondern eine Art Wrapper, mit dem sich auf existierenden Mengen operieren lässt. Das ist so, als hätte man Zeiger oder Iteratoren auf die Elemente übergeben, um die Elemente zu bearbeiten; nur dass weiter eine containerartige Schnittstelle verwendet wird. Deshalb müssen Programmierer*innen auch unbedingt darauf achten, dass die von Span bearbeiteten Mengen lange genug existieren.

Spans können by value übergeben werden, und seit C++17 sind dank Class Template Argument Deduction (CTAD) bei der Initialisierung keine Elementtypen mehr anzugeben. Auch Teilsequenzen sind bequem erzeug- und verwendbar:

```
// verdoppele die ersten beiden Werte in v:
doubleContent(std::span{v}.first(2));
```

Die chrono-Bibliothek wurde so erweitert, dass sie nun mühelos für alle möglichen Kalenderfunktionen (inklusive Behandlung von Zeitzonen) verwendet werden kann. Dabei galt wie immer das Designprinzip, möglichst viele Fehler zur Kompilierzeit zu finden. Ein Tag lässt sich sowohl mit 2019y/11/14 als auch mit 14/11/2019y definieren, das y ist aber wichtig, um zu wissen und zu prüfen, ob Tag, Monat und Jahr korrekt benutzt werden. Auch so etwas wie Thursday[last]/November/2019 für den letzten Donnerstag im November ist möglich. Zeitzonenbehandlung ist einfach (Namespaces sind weggelassen):

```
zoned_time tp1{current_zone(), now()};  // aktuelle lokale Zeit
zoned_time tp2 {"Europe/Paris", tp};    // aktuelle Zeit in Paris
```

Zusätzlich gibt es neue Clock-Datentypen, um Zeitpunkte mit Zeitangaben aus anderen Kontexten zu vergleichen oder diese umzurechnen: `file_clock` (für Filesystem-Einträge), `utc_clock` (zur Behandlung von Schaltsekunden) sowie `gps_clock` und `tai_clock` für Zeitpunkte aus den Standards GPS (Navigation) und TAI (internationale Atomzeit).

Formatierte Ein- und Ausgabe

C++ führte von Anfang an ein neues Konzept für Ein- und Ausgaben ein: I/O Streams. Auch nach 20 Jahren verwenden Entwickler*innen für formatierte Ausgaben noch Funktionen wie `printf()`, weil sie leichter zu benutzen sind und mitunter auch weniger Ressourcen brauchen. C++20 enthält deshalb eine neue, standardisierte Bibliothek zur einfachen formatierten Ausgabe. Sie ist schnell, typsicher und erweiterbar.

`std::format()` erzeugt einen formatierten String, in dem sich mit geschweiften Klammern die nachfolgenden Argumente ansprechen lassen:

```
std::string str{"hello"};
// - String und int nach Reihenfolge der Argumente
std::format("'{}' size: {}\n", str, str.size());
```

Listing 6: Generator-Beispiel für eine Koroutine

```
class IntGen {
    public:
      struct promise_type {
        int current_value;
        ...
        auto yield_value(int value) {
          current_value = value;
          return std::suspend_always{};
        }
      };
    private:
      using handle = std::coroutine_handle<promise_type>;
      handle coro;
      IntGen(handle h) : coro{h} {
      }
    public:
      bool nextValue() {
        return  coro ? (coro.resume(), !coro.done()) : false;
      }
      int getValue() {
        return coro.promise().current_value;
      }
      ...
      using iterator = ...   // ++ setzt die Coroutine fort
      iterator begin();
      iterator end();
};
```

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-