B1 SYSTEMS
Support & Managed Service für
Open Source IT-Umgebungen
Linux · OpenStack · Ceph · DevOps · Kubernetes
info@b1-systems.de ROCKOLDING · BERLIN · KÖLN · DRESDEN Mehr dazu auf S. 148

iX DEVELOPER
Herbst 2020
14,90 €
Österreich 16,40 €
Schweiz 27.90 CHF
Luxemburg 17,10 €
www.ix.de

Modernes C++

C++20: Was Entwickler wissen müssen!

C++20

Concepts, Ranges, Coroutinen und Module

Im Interview: C++-Erfinder Bjarne Stroustrup

Concurrency für Multicore-Architekturen

Ausblick auf C++23

Praxis & Methoden

Programmieren mit dem Compiler Explorer

Parallelisierung: Multithreading mit HPX

Weniger Code dank Fold Expressions

Design-Patterns in C++

Tools

Auf dem Prüfstand: Testwerkzeuge zur Codeanalyse

Google Benchmarking Library in der Praxis

Sicher programmieren mit MISRA-C++

Compiler-Überblick

Rheinwerk Konferenz für

KOTLIN

12.–13. Oktober 2020
Live im KOMED / Köln

Zwei Tage Kotlin, zwei Tage volles Programm!

Tauschen Sie sich mit den Kotlin-Profis aus und holen Sie sich wertvolles Wissen für Ihre Projekte. Ganztägige **Intensivworkshops** und **Vorträge** von führenden Experten machen Sie fit in Kotlin – live vor Ort oder bequem zuhause mit dem **Video-Ticket**.

Mit den internationalen Kotlin-Koryphäen **Annyce Davis**, **Amanda Hinchman-Dominguez** sowie **Sebastian Aigner** von JetBrains.

Acht Gründe, dabei zu sein!

- Zwei Tage mit den Kotlin-Profis
- 20 Vorträge zu allen Kotlin-Themen
- Vier Intensivworkshops
- Alle Vorträge auf Video
- Leckeres Essen, tolle Location
- 100 Prozent Rheinwerk
- Viel Platz für alle Teilnehmer
- **Hygienekonzept für Ihre Sicherheit**

Nur noch wenige Tickets!

www.rheinwerk-kkon.de

Unsere Sponsoren und Partner:

MATHEMA · heise Developer · REWE digital · grandcentrix Ein Vodafone Unternehmen · apploft. · JET BRAINS

Die Pflicht vor der Kür

Die Programmiersprache C++ gibt es bereits seit über 40 Jahren. Ihre Community setzt sich aus mehr als 4,5 Millionen aktiven Entwicklern zusammen. Dies allein wären schon gute Gründe für ein Sonderheft zu C++. Aber es kommt noch besser: Trotz der besonderen Situation durch die COVID-19-Pandemie sieht es danach aus, dass der C++20-Standard in diesem Jahr wie geplant veröffentlicht wird.

C++20 steht ganz in der Tradition seiner zwei großen Vorgänger C++98 und C++11. Mit C++20 wird sich die Art und Weise, wie modernes C++ geschrieben wird, ähnlich fundamental ändern wie zuvor mit C++11. Letzteres ist die Wegscheide, an der sich modernes C++ von Legacy-C++ trennt. Große Veränderungen wie bei C++20 müssen sukzessive konsumiert und verdaut werden. Zuerst gilt es, die neuen Features kennenzulernen und zu verstehen, bevor sie ihr volles Potenzial entfalten können. Es wird noch ein paar Jahre dauern, bis sich Best Practices zu C++20-Features im Allgemeinen und zu Concepts, Ranges, Modulen oder Coroutinen im Besonderen herauskristallisiert haben. Vor der Kür steht zuerst einmal die Pflicht.

Anlässlich des neuen C++20-Standards haben die *iX*, *heise Developer* und ich dieses Sonderheft zusammengestellt. Es enthält mehrere Artikel, die die zentralen Features des Standards vorstellen. Neben einem spannenden Einblick in die vier großen Neuerungen gibt das Heft eine Übersicht über unbekanntere Highlights, beispielsweise bei der Kernsprache, der Bibliothek und der Concurrency. Dabei ist zu beachten, dass die Artikel zu C++20 bereits vor der Veröffentlichung des Standards verfasst worden sind. Sie erheben keinen Anspruch auf Vollständigkeit. Wir haben uns zum Ziel gesetzt, Sie beim frühzeitigen Kennenlernen der neuen Features zu unterstützen.

Für einen praktischen Überblick hat das Sonderheft zudem eine Auswahl aus den in den vergangenen zwei Jahren in der *iX* erschienenen Artikeln zu C++ in teils aktualisierter Form im Gepäck. Doch legen wir den Fokus auf eines der Highlights in diesem Sonderheft: das Gespräch mit dem „Vater" der Programmiersprache C++, Bjarne Stroustrup. Die Redaktion hat nachgehorcht, warum es sich noch heute lohnt, C++ zu erlernen. Dazu gibt es auch ein paar Tipps und wichtige Hinweise für Neulinge. Darüber hinaus liefert Stroustrup spannende Informationen zum neuen Sprachstandard, beispielsweise darüber, welche Features es nicht in die neue Version geschafft haben und warum.

Auch wenn viele Features des neuen Standards nur mit einem der hochaktuellen GCC-, Clang- oder Microsoft-Compiler zur Verfügung stehen und einige der C++-Bibliotheken noch von keinem dieser großen Drei unterstützt werden – jetzt ist genau der richtige Zeitpunkt, sich mit dem C++20-Standard auseinanderzusetzen. Denn vor allem der aktuelle GCC-Compiler unterstützt nahezu alle Funktionalitäten der C++20-Kernsprache, und viele der restlichen Features stehen mit Online-Compilern wie der Wandbox oder dem Compiler Explorer zur Verfügung. Damit lassen sich bereits wertvolle Erfahrungen mit den mächtigen C++-20-Neuerungen sammeln. Das gilt insbesondere für Concepts, Ranges, Module und Coroutinen.

Ich freue mich auf jeden Fall darauf, Ihnen den neuen C++20-Standard zusammen mit anderen sachkundigen Autoren genauer vorstellen zu dürfen. Wir wünschen Ihnen viel Spaß beim Lesen!

RAINER GRIMM

Softwarearchitekt, Schulungsleiter, Buchautor und Blogger bei *heise Developer*

C++20-Highlights

Mit dem neuen Sprachstandard halten nützliche Werkzeuge wie Concepts, Ranges, Coroutinen und Module Einzug in C++. Die genannten Features zählen zu den Highlights von C++20.

ab Seite 6

C++20-Highlights

Die großen Vier
Concepts, Ranges, Coroutinen und Module erweitern C++ 6

Concepts
Herkulesaufgabe: Zwei Extreme und die Rettung dank Concepts 12

Concepts als (un-)eingeschränkte Platzhalter und syntaktischer Zucker 18

Concepts definieren und wiederverwenden 24

Evolution oder Revolution? Entwicklung von Concepts 32

Ranges
Funktionskomposition mit der Ranges-Bibliothek 36

Coroutinen
Nebenläufige Systeme organisieren mit anhaltbaren Funktionen 42

Coroutinen implementieren mit cppcoro 48

Module
Unabhängige Ordnung: Die Vorteile von Modulen 56

C++20-Basics

Kernsprachen-Features
Unbekannte Highlights: Drei-Wege-Vergleichsoperator und Designated-Initialisierer 66

Bibliotheken
Erweiterung der Chrono-Bibliothek und zwei neue Datentypen 70

Nebenläufigkeit
Überblick zur Concurrency 74

Enumerationen
Scoped Enums in C++ 78

ISO-Standardisierung
Wie geht die Standardisierung von C++? 84

Ausblick auf C++23
Ein Blick in die Proposals für künftige Neuerungen 92

Interview
C++-Erfinder Bjarne Stroustrup im Gespräch 100

Praxis & Methoden

Variadische Templates
Weniger Code dank Fold Expressions 106

Memory Leaks
Speicherlecks finden 112

Programmiertechniken
Metaprogrammierung mit dem Compiler Explorer 116

Multithreading-Programmierung mit HPX 120

SFINAE für Member Checker und bedingte Kompilierung anwenden 124

Design-Patterns
Das Builder-Pattern und Alternativen in C++ 130

Tools

C++-Tricks
C++-Compiler im Überblick 136

Micro-Benchmarking mit der Google Benchmark Library 143

Sicheres C++
MISRA-C++: Regelwerke und Konformität 150

Werkzeugkasten
Prüfstand für Testwerkzeuge zur Codeanalyse 157

Sonstiges

Editorial 3

Impressum 133

C++20-Basics

C++20 hat auch unbekanntere Neuerungen zu bieten. Dazu zählen die neuen Features der Kernsprache, die Erweiterung der Chrono-Bibliothek sowie die Verbesserung der Concurrency. Und mit einem Blick in die Proposals lassen sich bereits jetzt Rückschlüsse auf C++23 ziehen.

ab Seite 66

Interview

Warum sollte man heute noch C++ lernen, und wie kann Neulingen der Einstieg gelingen? Welche neuen Features aus C++20 werden langfristig am meisten Bedeutung haben? Diese und weitere Fragen beantwortet C++-Erfinder Bjarne Stroustrup im Gespräch.

ab Seite 100

Praxis & Methoden

Auf das Basiswissen folgt die Umsetzung in den Arbeitsalltag: Mit weniger Code dank Fold Expressions, Methoden zum Aufspüren von Speicherlecks und Metaprogrammierung per Compiler Explorer tauchen Entwickler in die Praxis ein.

ab Seite 106

Tools

Entwickler sind wie Handwerker auf gute Werkzeuge angewiesen. Allerdings sollte man diese Hilfsmittel auch beherrschen. Es gilt, den Überblick zu behalten, ob bei der Vielzahl an Versionen verschiedener C++-Compiler oder beispielsweise den MISRA-C++-Regeln für sicheres Programmieren.

ab Seite 136

Concepts, Ranges, Coroutinen und Module erweitern C++

Die großen Vier

Rainer Grimm

Wie sahen bisherige Standards in C++ aus? Mit den „Big Four“ sind vier wesentliche Elemente von C++20 gemeint, die neu hinzukommen, nämlich Concepts, Ranges, Coroutinen und Module. Dieser Artikel stellt ihre Neuerungen im Überblick vor.

Mit C++20 erscheint der nächste maßgebliche C++-Standard, Entwicklern stehen damit Veränderungen ins Haus. Dieser Artikel soll einen Überblick zu C++20 liefern mit einem Rückblick auf die Versionsgeschichte der C++-Standards und eine Vorschau auf die wesentlichen Neuerungen der kommenden Version. Wie der letzte große Standard (C++11) wird C++20 die Art und Weise ändern, wie Entwickler in modernem C++ programmieren – das gilt vor allem für Ranges, Coroutinen, Concepts und Module. Um den neuen Standard einzuordnen, lohnt sich zunächst ein Blick auf die bisherigen Evolutionsphasen dieser Programmiersprache in Abbildung 1.

The Big Four

Neu sind die „großen Vier“ im Kern von C++20: Concepts, Ranges, Coroutinen und Module. Außerdem bringt der neue Standard Neuerungen bei zahlreichen Features, die die Kernsprache, die Bibliothek und die Concurrency betreffen: den Drei-Wege-Vergleichsoperator, Verbesserungen an den Lambdas und Templates, eine Kalender- und eine Formatierungsbibliothek, Semaphoren und einen verbesserten Thread, der sich unterbrechen lässt.

Concepts

Concepts sind eine Erweiterung der Templates von C++, mit denen sich semantische Kategorien für die Menge der zulässigen Datentypen definieren lassen. Dank Concepts wird das Anwenden und Definieren von Templates deutlich einfacher und ausdrucksreicher.

Die zentrale Idee der generischen Programmierung mit Templates ist es, Funktionen und Klassen zu definieren, die sich mit verschiedenen Datentypen verwenden lassen. Oft passiert es Entwicklern aber, dass sie ein Template mit einem falschen

Die C++-Standards im chronologischen Überblick

C++98 und C++03

Ende der 1980er-Jahre gab es mehrere unabhängige C++-Implementierungen. Daher legten Bjarne Stroustrup und Margaret A. Ellis 1989 in ihrem Buch „The Annotated C++ Reference Manual (ARM C++)“ [1] den Funktionsumfang von C++ fest. Darüber hinaus erfüllte ARM C++ noch eine weitere wichtige Aufgabe: Es bildete die Grundlage für den ersten ISO-C++-Standard ISO/IEC 14882:1998 (C++98). Wichtige Features von C++98 sind Templates, die Standard Template Library (STL) mit ihren Containern und Algorithmen sowie Strings und IO-Streams.

Mit C++03 (ISO/IEC 14882:2003) erfuhr C++98 eine technische Korrektur, die so marginal war, dass sie im hier skizzierten Zeitstrahl keine Berücksichtigung findet. In der Community wird C++03, das C++98 einschließt, schlicht „Legacy C++“ genannt.

C++11

Der Technical Report 1 (TR1) 2007 ist zwar kein offizieller Standard, beschreibt aber viele Komponenten, die in den offiziellen Standard C++11 aufgenommen wurden. Die neuen Bibliotheken zu regulären Ausdrücken, Smart Pointern, Hashtabellen oder Zufallszahlengeneratoren basieren alle auf dem TR1 und somit auf den entsprechenden Boost-Bibliotheken. Das Boost-Projekt, das von Mitgliedern des C++-Standardisierungskomitees gegründet wurde, ist die eigentliche Ideenwerkstatt für die aktuellen Erweiterungen der C++-Bibliothek. Der TR1 enthielt 13 Bibliotheken. Lediglich die speziellen mathematischen Funktionen mussten bis C++17 warten.

C++11 steht für den daran anschließenden C++-Standard. Er wird oft auch schlicht „Modern C++“ genannt. Dieser Name steht auch für C++14 und C++17. C++11 besitzt zahlreiche Features, die die Art und Weise, wie Entwickler C++ programmieren, grundlegend änderten. Diese fundamentalen Änderungen an C++11 betreffen neben den mit dem Technical Report 1 (TR1) eingeführten Komponenten die Kernsprache, die Standardbibliothek und die Concurrency. Für die Kernsprache stehen neue Features wie die Move-Semantik, das Perfect Forwarding, die Variadic Templates, Lambdas und `constexpr` auf der Habenseite. Die Standardbibliothek kann mit neuen Bibliotheken zu Smart Pointern, Hashtabellen und Type Traits punkten. Darüber hinaus besitzt C++11 ein Speichermodell als fundamentale Grundlage zur Concurrency und eine Multithreading-Schnittstelle.

C++14 und C++17

C++14 ist ein kleiner Standard. Mit ihm wurde C++ um Reader-Writer Locks, verallgemeinerte Lambdas und verallgemeinerte `constexpr`-Funktionen erweitert.

Der Standard C++17 ist weder groß noch klein, er besitzt zwei herausragende Features: die parallele Standard Template Library (STL) und die Dateisystembibliothek. Über 80 Algorithmen der STL lassen sich mit einer sogenannten Ausführungsstrategie (Execution Policy) ausführen. Das heißt, dass ein Aufruf der Form `std::sort(std::execute::par, vec.begin(), vec.end())` die C++-Implementierung anweist, parallel zu sortieren. Zusätzlich lässt sich spezifizieren, ob die Sortierung sequenziell (`std::execute::seq`) oder vektorisiert (`std::execute::par_unseq`) ausgeführt werden soll. Analog zu C++11 übten die Mitglieder des Boost-Projekts starken Einfluss auf diesen C++-Standard aus. Boost erweiterte C++ um ein Dateisystem und die neuen Datentypen `std::optional`, `std::variant` sowie `std::any`.

C++20

C++20 wird die Art und Weise, mit C++ zu programmieren, ähnlich tief greifend ändern wie C++11 bei seiner Einführung. Das gilt insbesondere für die „großen Vier“: Ranges, Coroutinen, Concepts und Module. Noch 2019 hätte der Verfasser von den „großen Fünf“ geschrieben. Leider wurden Contracts mittlerweile aus dem C++20-Standard entfernt. Ein Contract spezifiziert in einer exakten und überprüfbaren Weise eine Schnittstelle für eine Softwarekomponente. Der Blog-Beitrag des Verfassers „C++ Core Guidelines: Ein kleiner Umweg über Kontrakte“ [a] liefert weitere Details. So weit der Rückblick auf die Geschichte von C++.

C++98	C++11	C++14	C++17	C++20
1998	2011	2014	2017	2020
• Templates • STL mit Containern und Algorithmen • Strings • I/O Streams	• Move-Semantik • vereinheitlichte Initialisierung • `auto` und `decltype` • Lambda-Funktionen • `constexpr` • Multithreading und das Speichermodell • reguläre Ausdrücke • Smart Pointer • Hashtabellen • `std: array`	• Reader-Writer Locks • verallgemeinerte Lambda-Funktionen	• Fold Expressions • constexpr if • Structured Binding • string_view • parallele Algorithmen der STL • Dateisystembibliothek • `std: any, std:optional, und std: variant`	kommen neu hinzu: • Concepts • Ranges Library • Coroutinen • Module entfallen: • Contracts

Seit 2011 verabschiedet das ISO-Standardkomitee im Dreijahresrhythmus neue Versionen von C++ (Abb. 1)

Listing 1: Implementierung und Einsatz des Concepts Integral

```
template<typename T>
concept bool Integral(){
    return std::is_integral<T>::value;
}

Integral auto gcd(Integral auto a,
                  Integral auto b){
    if( b == 0 ) return a;
    else return gcd(b, a % b);

}
```

Listing 2: Einsatz des Concept Integral

```
template<typename T>
requires Integral<T>
T gcd(T a, T b){
    if( b == 0 ) return a;
    else return gcd(b, a % b);
}
```

Listing 3: Funktionskomposition mit dem Pipe-Operator

```
#include <vector>
#include <ranges>
#include <iostream>

int main(){
  std::vector<int> ints{0, 1, 2, 3, 4, 5};
  auto even = [](int i){ return 0 == i % 2; };
  auto square = [](int i) { return i * i; };

  for (int i : ints | std::view::filter(even) |
                      std::view::transform(square)) {
    std::cout << i << ' ';            // 0 4 16
  }
}
```

Datentyp instanziieren. Das typische Ergebnis sind ein paar Seiten voller kryptischer Fehlermeldungen. Mit Concepts endet diese traurige Geschichte. Sie erlauben es, Bedingungen an Templates zu stellen, die der Compiler verifiziert. Concepts revolutionieren die Art und Weise, wie C++-Entwickler über generischen Code denken und diesen implementieren können. Hier sind die Gründe:

- Die Anforderungen an die Templates sind Bestandteil des Interface.
- Das Überladen von Templates oder die Spezialisierung von Templates sind aufgrund von Concepts möglich.
- Entwickler erhalten verbesserte Fehlermeldungen, denn der Compiler kann die Bedingungen an die Templateparameter mit den Templateargumenten vergleichen.

Das ist aber bei Weitem nicht das Ende der Geschichte, Concepts können noch mehr:

- Man kann vordefinierte Concepts einsetzen oder eigene Concepts definieren.
- Die Verwendung von `auto` und Concepts wird vereinheitlicht – daher kann man anstelle von `auto` einfach auf ein Concept zurückgreifen.
- Wenn eine Funktionsdeklaration ein Concept verwendet, wird die Funktion automatisch zum Funktionstemplate. Das Schreiben von Funktionstemplates wird damit so einfach wie das Schreiben von Funktionen.

Listing 1 zeigt die Definition und die Anwendung des Concepts `Integral`: Das ist ein Concept, das von seinem Typparameter fordert, dass `std::is_integral<T>::value` „true“ ergibt. Dies ist eine Funktion aus der Type-Traits-Bibliothek, die ihre Prüfung genau zur Compilezeit ausführt. Wenn `std::is_integral<T>::value true` ergibt, ist alles gut. Falls nicht, erhält man einen Fehler zur Compilezeit. Der `gcd`-Algorithmus bestimmt den größten gemeinen Teiler zweier Zahlen, basierend auf dem euklidischen Algorithmus [c].

Der Verfasser hat die sogenannte Abbreviated Function Template Syntax verwendet, um `gcd` zu definieren. `gcd` fordert gemäß seinen Argumenten und seinem Rückgabewert, dass sie das Concept `Integral` unterstützen. `gcd` ist eine Art Funktionstemplate, das Bedingungen an seine Argumente und an seinen Rückgabewert stellt. Wenn man den „Syntactic Sugar“ entfernt [d], sollte man in Listing 2 den eigentlichen Gehalt der `gcd`-Funktion sehen.

Wer die wahre Natur der `gcd`-Funktion noch nicht erkennt, kann im Artikel zu Concepts ab Seite 12 nachschlagen, dort wird dieses Thema ausführlich behandelt.

Ranges-Bibliothek

Die Ranges-Bibliothek erlaubt es, die Algorithmen der Standard Template Library direkt auf den Containern der STL anzuwenden, diese mit dem aus der Unix-Shell bekannten Pipe-Operator zu verknüpfen und auf unendlichen Datenströmen zu definieren.

Die Ranges-Bibliothek ist der erste „Kunde“ von Concepts. Sie bietet Algorithmen an, die auf den ganzen Container bezogen arbeiten. Entwickler müssen daher keinen Bereich mehr mittels Iteratoren angeben. Sie können Algorithmen komponieren und „lazy“ evaluieren.

Compiler-Unterstützung für C++20

Um sich mit den neuen Features vertraut zu machen, ist es am einfachsten, mit ihnen herumzuspielen. Dabei stellt sich gleich die grundlegende Frage: Welche C++20-Features werden durch welchen Compiler unterstützt? Die Website cppreference.com gibt unter Compiler Support zur Kernsprache und zur Bibliothek eine aussagekräftige Antwort [b]. Um es vorwegzunehmen: Die neuen GCC-, Clang- und EDG-Compiler bieten die breiteste Unterstützung für die Kernsprache an. Zusätzlich unterstützen der MSVC und der Clang-Compiler zahlreiche C++20-Features (s. Abb. 2).

Die Geschichte der Bibliothek ist ähnlich: GCC besitzt die breiteste Unterstützung für die Bibliothek, gefolgt vom Clang-Compiler und dem MSVC (s. Abb. 3).

Die Screenshots zeigen jeweils nur den Anfang der Tabellen, doch schon dieser Ausschnitt genügt, um zu bemerken, dass der Status quo die meisten Entwickler nicht zufriedenstellen wird. Selbst wenn man alle aktuellen Compiler einsetzt, bleiben viele Features übrig, die bislang noch keine Unterstützung durch einen Compiler besitzen.

C++20 feature	Paper(s)	GCC	Clang	MSVC	Apple Clang	EDG eccp
Allow lambda-capture [=, this]	P0409R2	8	6	19.22*	10.0.0*	5.1
__VA_OPT__	P0306R4 P1042R1	8 (partial)* 10 (partial)*	9	19.25*	11.0.3*	5.1
Designated initializers	P0329R4	4.7 (partial)* 8	3.0 (partial)* 10	19.21*	(partial)*	5.1
template-parameter-list for generic lambdas	P0428R2	8	9	19.22*	11.0.0*	5.1
Default member initializers for bit-fields	P0683R1	8	6	19.25*	10.0.0*	5.1
Initializer list constructors in class template argument deduction	P0702R1	8	6	19.14*	Yes	5.0
const&-qualified pointers to members	P0704R1	8	6	19.0*	10.0.0*	5.1
Concepts	P0734R0	6 (TS only) 10	10	19.23* (partial)*		6.1
Lambdas in unevaluated contexts	P0315R4	9				
Three-way comparison operator	P0515R3	10	8 (partial) 10	19.20*		5.1
DR: Simplifying implicit lambda capture	P0588R1	8		19.24*		5.1
init-statements for range-based for	P0614R1	9	8	19.25*	11.0.0*	6.0
Default constructible and assignable stateless lambdas	P0624R2	9	8	19.22*	10.0.1*	5.1

Quelle: C++-Reference, Stand: 31.07.2020 [b]

Core Language Features und deren Compiler-Unterstützung [b] (Abb. 2)

C++20 feature	Paper(s)	GCC libstdc++	Clang libc++	MSVC Standard Library	Apple Clang
std::endian	P0463R1	8	7	19.22*	10.0.0*
Extending std::make_shared() to support arrays	P0674R1			19.27*	
Floating point atomic	P0020R6	10		19.22*	
Synchronized buffered ostream	P0053R7				
constexpr for <algorithm> and <utility>	P0202R3	10	8 (partial)	19.26*	10.0.1* (partial)
More constexpr for <complex>	P0415R1	9	7 (partial)	19.27*	10.0.0* (partial)
Make std::memory_order a scoped enumeration	P0439R0	9	9	19.25*	11.0.3*
String prefix and suffix checking	P0457R2	9	6	19.21*	10.0.0*
Library support for operator<=> <compare>	P0768R1	10	7 (partial)	19.20* (partial)	10.0.0* (partial)
std::remove_cvref	P0550R2	9	6	19.20*	10.0.0*
[[nodiscard]] in the standard library	P0600R1	9	7 (partial)	19.13* (partial) 19.22*	10.0.0* (partial)
Using std::move in numeric algorithms	P0616R0	9		19.23*	
Utility to convert a pointer to a raw pointer	P0653R2	8	6	19.22*	Yes
Atomic std::shared_ptr and std::weak_ptr	P0718R2			19.27*	
std::span	P0122R7	10	7	19.26*	10.0.0*
Calendar and timezone	P0355R7		7 (partial)		10.0.0* (partial)

Quelle: C++-Reference, Stand: 31.07.2020 [b]

Library Features und deren Compiler-Unterstützung [a] (Abb. 3)

Um es kurz zu machen, die Ranges-Bibliothek bietet funktionale Patterns an. Da Code mehr sagt als Worte, bietet Listing 3 ein Beispiel, das die Funktionskomposition anhand des Pipe-Operators zeigt.

`even` ist eine Lambda-Funktion, die ermittelt, ob `i` gerade ist. Die Lambda-Funktion `square` bildet ihr Argument auf das Quadrat ab. Der Rest des Codes ist Funktionskomposition, die man von links nach rechts lesen muss:

```
for (int i : ints | std::view::filter(even) | 7
                                  std::view::transform(square))
```

Man muss auf jedes Argument von `ints` den Filter `even` anwenden und die verbleibenden Argumente auf ihr Quadrat abbilden. Wenn man mit funktionaler Programmierung vertraut ist, liest sich die Verknüpfung der Filterschritte durch Pipes wie Prosa. Diese Art von Workflow wird auch Pipes und Filter genannt [e].

Coroutinen

Mit Couroutinen unterstützt C++20 die asynchrone Programmierung. Damit sollen sich in C++20 kooperatives Multitasking, unendliche Datenströme, Eventschleifen oder auch Pipelines elegant umsetzen lassen. Sie sind verallgemeinerte Funktionen, die ihre Ausführung anhalten und wieder aufnehmen können. Dabei behalten sie ihren Zustand. Coroutinen stellen die typische Art dar, eventgetriebene Applikationen zu schreiben. Diese können Simulationen, Spiele, Server, Benutzerinterface und selbst Algorithmen sein. Oft kommen sie auch für kooperatives Multitasking zum Einsatz [f].

Mit C++20 erhalten Entwickler keine konkreten Coroutinen, sondern ein Framework für das Schreiben von Coroutinen. Dieses Framework besteht aus mehr als zwanzig Funktionen, die teilweise implementiert werden müssen oder können. Daher lässt sich das Verhalten von Coroutinen auf die eigenen Anforderungen genau zuschneiden.

Der Verständlichkeit halber stellt Listing 4 eine einfache Coroutine vor. Das Programm verwendet einen Erzeuger für einen unendlichen Datenstrom – das heißt, die Funktion `getNext` gibt einen Generator zurück, der unendlich viele Werte erzeugen kann: Dazu sind ein paar Erläuterungen nötig, denn dies ist nur ein Codeschnipsel. Die Funktion `getNext` ist eine Coroutine, da sie das Schlüsselwort `co_yield` verwendet. `getNext` besitzt eine Endlosschleife, die den Wert nach `co_yield` zurückgibt. Ein Aufruf `next()` in den Zeilen 2 und 3 weckt die Coroutine auf, und der anschließende `getValue`-Aufruf erlaubt es, den Wert abzuholen. Nach dem `getNext`-Aufruf pausiert die Coroutine bis zum nächsten `next()`-Aufruf. Es gibt noch eine große Unbekannte in diesem Beispiel, und zwar den Rückgabewert `Generator<int>` der `getNext`-Funktion. Hier starten die komplizierten Aspekte von Coroutinen, die der Artikel zu Coroutinen ab Seite 42 ausführlicher erläutert.

Listing 4: Ein Generator für unendlich viele Werte

```
Generator<int> getNext(int start = 0, int step = 1){
    auto value = start;
    for (int i = 0;; ++i){
        co_yield value;            // 1
        value += step;
    }
}

int main() {

    std::cout << std::endl;

    std::cout << "getNext():";
    auto gen = getNext();
    for (int i = 0; i <= 10; ++i) {
        gen.next();                // 2
        std::cout << " " << gen.getValue();
    }

    std::cout << "\n\n";

    std::cout << "getNext(100, -10):";
    auto gen2 = getNext(100, -10);
    for (int i = 0; i <= 20; ++i) {
        gen2.next();               // 3
        std::cout << " " << gen2.getValue();
    }

    std::cout << std::endl;

}
```

Dank des Online-Compilers Wandbox lässt sich die Ausgabe des Programms darstellen [g].

Module

Module stellen gegenüber Headerdateien eine deutliche Verbesserung dar: Sie garantieren schnellere Kompilierungszeiten, da sie ein wiederholtes Ersetzen (wie es bei den Headerdateien notwendig ist) überflüssig machen. Mit Modulen können Entwickler Präprozessor-Makros isolieren. Damit ist die Reihenfolge, in der man Headerdateien inkludiert, nicht mehr entscheidend. Module drücken die logische Struktur des Codes aus und erlauben es, neue Modulpakete aus bestehenden Modulen zusammenzustellen. Die Notwendigkeit, auf fehleranfällige Makro-Tricks zurückzugreifen, entfällt dabei, und Verletzungen der One Definition Rule (ODR) gehören der Vergangenheit an. Zur Erinnerung: Im Falle von Funktionen besagt die One Definition Rule, dass diese nicht mehr als eine Definition pro Übersetzungseinheit besitzen dürfen.

Weitere Höhepunkte

Doch ist das bei Weitem nicht der vollständige Inhalt von C++20, es gibt noch weitere Höhepunkte, die dieses Sonderheft genauer beleuchtet:

- den Drei-Wege-Vergleichsoperator: `<=>`,
- String-Literale als Templateparameter,
- Kalender- und Zeitzonenerweiterungen für die Chrono-Bibliothek,
- `std::span` als einen View auf ein zusammenhängendes Array,
- `constexpr` für virtuelle Funktionen,
- die Neudefinition von `volatile`,
- Erweiterungen zu atomaren Variablen: `std::atomic_ref<T>` und `std::atomic<std::shared_ptr<T>>>`,
- Warten mit atomaren Variablen,
- neue Synchronisationsmechanismen wie Semaphoren, Latches und Barriers,
- `constexpr`-Container wie `std::string` und `std::vector`,
- einen verbesserten Thread mit `std::jthread`, der automatisch den `join`-Aufruf anstößt und gestoppt werden kann.

Nach dem ersten groben Überblick zur Geschichte von C++ beschäftigen sich die weiteren Artikel im Detail mit den Neuerungen in C++20. Diese betreffen einerseits die herausragenden Features Concepts, Ranges, Coroutinen und Module – andererseits betreffen sie aber auch alle weiteren Features rund um die Kernsprache, die Standardbibliothek und Concurrency in C++20. (sih@ix.de)

Onlinequellen

[a] Rainer Grimm; C++ Core Guidelines über Kontrakte: heise.de/-4117736.html
[b] Compiler Support auf cppreference.com: n.cppreference.com/w/cpp/compiler_support
[c] Euklidischer Algorithmus: de.wikipedia.org/wiki/Euklidischer_Algorithmus
[d] Syntaktischer Zucker: de.wikipedia.org/wiki/Syntaktischer_Zucker
[e] Pipes und Filter: de.wikipedia.org/wiki/Pipes_und_Filter
[f] Cooperative Multitasking: en.wikipedia.org/wiki/Cooperative_multitasking
[g] Wandbox Compiler: wandbox.org

Namensgebung für die kommende C++-Generation

Die Frage, wie das jüngste Mitglied der C++-Familie nun heißen soll, ist offen. Die bestehenden C++-Standards tragen bereits informelle Namen: C++98 und C++03 werden mittlerweile als „Legacy C++“ bezeichnet, C++11, C++14 und C++17 als „Modern C++“. Welche Bezeichnung kommt für C++20 und die folgenden Standards in Betracht? Hierzu gab es aus der Community heraus viele kreative Vorschläge. Heute gilt als sicher, dass der Standard einfach C++20 heißen wird – ein passender Spitzname wird sich erst nach dessen Einführung herauskristallisieren. Als nächster großer Standard ist in etwa drei Jahren C++23 geplant.

Literatur

[1] Margaret A. Ellis, Bjarne Stroustrup; The Annotated C++ Reference Manual (ARM C++); Addison-Wesley 1990

Rainer Grimm

ist Trainer und Coach für C++ und Python. Seine Bücher „C++11 für Programmierer“, „C++“, „C++ Standardbibliothek“, „The C++ Standard Library“ und „Concurrency with Modern C++“ sind bei dpunkt und Leanpub erschienen.

B1 Consulting 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 130 festangestellten Linux- und Open-Source-Experten.

Unsere Kernthemen:

Linux Server & Desktop · Private Cloud (OpenStack & Ceph) · Public Cloud (AWS, Azure, OTC & GCP) · Container (Docker, Kubernetes, Red Hat OpenShift, Rancher & SUSE CaaSP) · Monitoring (Icinga, Nagios & ELK) · Patch Management · Automatisierung (Ansible, Salt, Puppet & Chef) · Videokonferenzen

Unser in Deutschland ansässiges Support- und Betriebsteam ist immer für Sie da – mit qualifizierten Reaktionszeiten ab 10 Minuten und Supportzeiten von 8x5 bis 24x7!

Zwei Tage Linux-/Open-Source-Consulting zum Preis von einem!
Mail an **info@b1-systems.de** und Aktionscode **IX2020** angeben*!

*Aktionscode einmal pro Unternehmen einlösbar

B1 SYSTEMS

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

Zwei Extreme und die Rettung dank Concepts

Herkulesaufgabe

Rainer Grimm

Bis C++20 gab es in C++ zwei diametrale Optionen, um mithilfe von Funktionen und Klassen Abstraktionen zu schaffen. Sie ließen sich für konkrete Datentypen oder generische Datentypen definieren. Im zweiten Fall nennt man sie Funktions- oder Klassentemplates. Warum sind beide Wege falsch?

Es ist nahezu ein Mammutprojekt, für jeden konkreten Datentyp eine Funktion oder Klasse zu definieren. Um sich die daraus resultierende Last von den Schultern zu nehmen, greifen Entwickler beispielsweise auf Typkonvertierungen zurück. Doch was wie eine Rettung scheint, entpuppt sich oft als Fluch (s. Listing 1). Der erste Fall (1) startet mit einem `double`- und schließt mit einem `int`-Wert (2). Fall zwei (3) beginnt mit einem `bool`- und endet wieder mit einem `int`-Wert (4). Abbildung 1 zeigt die Ausgabe in der Konsole.

Der Aufruf von `getInt(int a)` mit einem `double`-Wert verursacht eine einschränkende Konvertierung (Narrowing Conversion). Dieses ist eine Konvertierung mit Verlust der Datengenauigkeit. Das war sicherlich nicht die Absicht des Autors. Andersherum ist es aber auch nicht besser. Ruft man `getInt(int a)` mit einem `bool`-Wert auf, wird dieser auf einen `int`-Wert aufgeblasen. Das kommt vielleicht überraschend, denn viele C++-Entwickler wissen nicht, was passiert, wenn man zwei Wahrheitswerte addiert:

Listing 1: Narrowing Conversion und Integral Promotion mit der Funktion needInt

```
// tooSpecific.cpp

#include <iostream>

void needInt(int i){
    std::cout << "int: " << i << std::endl;
}

int main(){

    std::cout << std::boolalpha << std::endl;

    double d{1.234};                          // (1)
    std::cout << "double: " << d << std::endl;
    needInt(d);                               // (2)

    std::cout << std::endl;

    bool b{true};                             // (3)
    std::cout << "bool: " << b << std::endl;
    needInt(b);                               // (4)

    std::cout << std::endl;

}
```

Listing 2: std::sort auf std::list angewandt

```
// sortList.cpp

#include <algorithm>
#include <list>

int main(){

    std::list<int> myList{1, 10, 3, 2, 5};

    std::sort(myList.begin(), myList.end());

}
```

```
template <typename T>
auto add(T first, T second){
    return first + second;
}

int main(){
    add(true, false);

}
```

C++ Insights [a] zeigt die ganze Wahrheit (s. Abb. 2). Die Template-Instanziierung des Funktionstemplates `add` erzeugt eine vollständige Spezialisierung (Zeile 6 bis 12), die den Rückgabetyp `int` besitzt.

Der Autor ist fest davon überzeugt, dass man nur aus praktischen Gründen die ganze Magie der Typkonvertierungen in C/C++ besitzt, um mit der Unzulänglichkeit umgehen zu können, dass Funktionen nur spezifische Datentypen annehmen können.

Der folgende Abschnitt wendet sich der zweiten oben genannten Option zu und nutzt keine spezifischen, sondern generische Datentypen. Vielleicht bieten gerade Templates eine Möglichkeit der Rettung.

Zu generisch

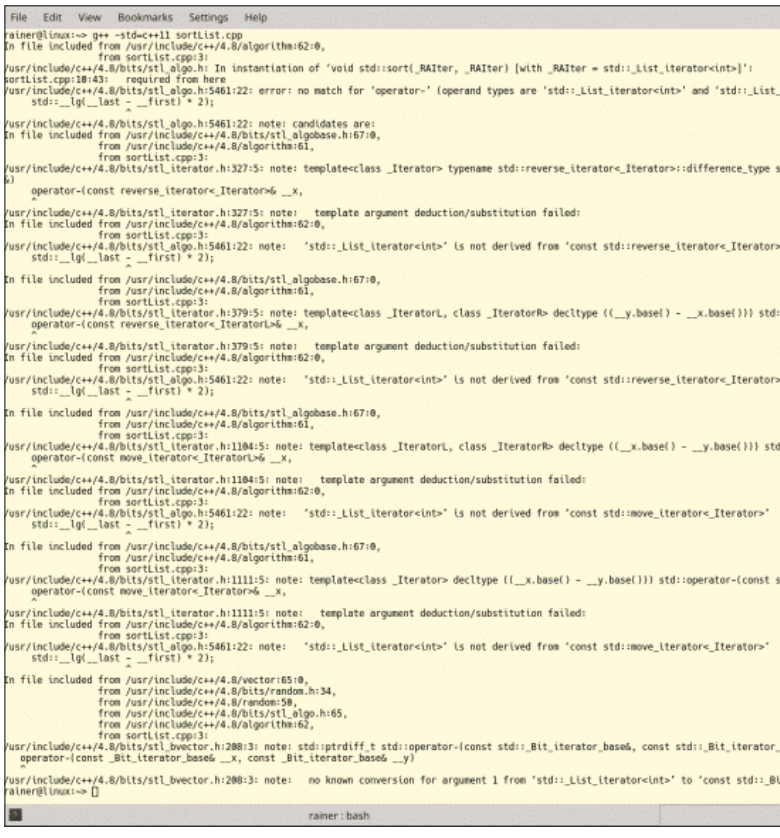
Der erste Versuch: Sortieren ist eine ganz allgemeine Idee. Daher sollte sie auf jeden Container anwendbar sein, falls sich die Elemente des Containers sortieren lassen. Listing 2 zeigt die Anwendungen von `std::sort` auf eine `std::list`. Abbildung 3 zeigt, was passiert, wenn man versucht, das kleine Programm zu übersetzen. Diese Fehlermeldung wird nicht dechiffriert. Was läuft

```
C:\Users\rainer>tooSpecific.exe

double: 1.234
int: 1

bool: true
int: 1


C:\Users\rainer>
```

Narrowing Conversion und Integral Promotion in Aktion (Abb. 1)

```
1  template <typename T>
2  auto add(T first, T second){
3      return first + second;
4  }
5
6  #ifdef INSIGHTS_USE_TEMPLATE
7  template<>
8  int add<bool>(bool first, bool second)
9  {
10   return static_cast<int>(first) + static_cast<int>(second);
11 }
12 #endif
13
14
15 int main()
16 {
17   add(true, false);
18 }
```

C++ Insights generiert den Code (Abb. 2).



Der Compiler quittiert das Sortieren einer std::list mit einer eindeutigen Fehlermeldung (Abb. 3).

Ein std::list ist eine doppelt verkettete Liste, die ein Bidirectional Iterator anbietet (Abb. 4).

Listing 3: std::sort fordert einen LegacyRandomAccessIterator

```
template<typename It>
concept LegacyRandomAccessIterator =
  LegacyBidirectionalIterator<It> &&        // (1)
  std::totally_ordered<It> &&
  requires(It i, typename std::incrementable_traits<It>::difference_type n) {
    { i += n } -> std::same_as<It&>;        // (2)
    { i -= n } -> std::same_as<It&>;
    { i +  n } -> std::same_as<It>;
    { n +  i } -> std::same_as<It>;
    { i -  n } -> std::same_as<It>;
    { i -  i } -> std::same_as<decltype(n)>;
    {  i[n]  } -> std::convertible_to<std::iter_reference_t<It>>;
  };
```

Listing 4: Anwendung eines Concepts bei einem Klassentemplate

```
template<Object T>
class MyVector{};

MyVector<int> v1;   // OK
MyVector<int&> v2;  // ERROR: int& does not satisfy
                    //        the constraint Object
```

Listing 5: Definition des Type-Traits is_object, auf dem das Concept Object basiert

```
template< class T>
struct is_object : std::integral_constant<bool,
                     std::is_scalar<T>::value ||
                     std::is_array<T>::value  ||
                     std::is_union<T>::value  ||
                     std::is_class<T>::value> {};
```

Listing 6: Die Funktionstemplates all, any und none

```
 // allAnyNone.cpp

#include <iostream>
#include <type_traits>

template<typename T>
concept Arithmetic = std::is_arithmetic<T>::value;

template<Arithmetic... Args>
bool all(Args... args) { return (... && args); }

template<Arithmetic... Args>
bool any(Args... args) { return (... || args); }

template<Arithmetic... Args>
bool none(Args... args) { return !(... || args); }

int main(){

    std::cout << std::boolalpha << std::endl;

    std::cout << "all(5, true, 5.5, false): " << all(5, true, 5.5, false) << std::endl;

    std::cout << "any(5, true, 5.5, false): " << any(5, true, 5.5, false) << std::endl;

    std::cout << "none(5, true, 5.5, false): " << none(5, true, 5.5, false) << std::endl;

}
```

hier falsch? Vielleicht hilft ein genauerer Blick auf die verwendete Überladung von `std::sort`.

```
template< class RandomIt >
void sort( RandomIt first, RandomIt last );
```

`std::sort` greift auf den seltsam klingenden Namen `RandomIt` zurück. Er steht für einen Random Access Iterator. Das ist der Grund für die Fehlermeldung, für die Templates berühmtberüchtigt sind. Eine `std::list` bietet nur einen Bidirectional Iterator an, aber `std::sort` benötigt einen Random Access Iterator. Die Struktur einer `std::list` (s. Abb. 4) offenbart diese Einschränkungen. Ein Blick in die Dokumentation zu `std::sort` [b] zeigt etwas sehr Interessantes, nämlich Typanforderungen an `std::sort`.

Concepts als Rettung

Concepts sind die Rettung, denn sie definieren semantische Einschränkungen auf Templateparametern. Hier sind die bereits erwähnten Typanforderungen zu `std::sort`:

- `RandomIt` muss die Anforderungen von `ValueSwappable` [c] und `LegacyRandomAccessIterator` [d] erfüllen.
- Der Typ des dereferenzierten `RandomIt` muss die Anforderungen von `MoveAssignable` [e] und `MoveConstructible` [f] erfüllen.
- `Compare` [g] muss ein binäres Prädikat sein, mit dem sich zwei Elemente vergleichen lassen.

Die Typanforderungen an `std::sort` sind Concepts. Insbesondere fordert `std::sort` einen `LegacyRandomAccessIterator` [d]. Darauf geht der folgende Abschnitt genauer ein, das Beispiel [d] aus Listing 3 wurde etwas aufpoliert.

Das ist die entscheidende Beobachtung. Ein Datentyp `It` unterstützt das Concept `LegacyRandomAccessIterator` genau dann, wenn er das Concept `LegacyBidirectionalIterator` (1) und die weiteren Anforderungen unterstützt. Zum Beispiel sagt die Anforderung in (2) für einen Wert des Datentyps `It` aus: `{ i += n }` muss gültig sein und eine Referenz auf `It` zurückgeben. Um mit dieser Story zum Abschluss zu kommen: `std::list` unterstützt nur einen `LegacyBidirectionalIterator`.

Nach einem technisch versierten Abschnitt folgt nun die Praxis. Mit Concepts bekommt man eine einfache Fehlermeldung wie die in Abbildung 5, wenn man `std::sort` mit einer `std::list` erhält.

Die Fehlermeldung war eine „Ente“, denn kein Compiler setzt zum jetzigen Zeitpunkt die C++20-Syntax für Concepts vollständig um. MSVC 19.23 unterstützt Concepts teilweise, GCC eine vorherige Version von Concepts. Ein Beitrag auf cppreference.com [h] liefert mehr Details zur aktuellen Compilerunterstützung von Concepts.

Mit Concepts ist es möglich, semantische Einschränkungen auf Templateparameter auszudrücken. Bevor dieser Artikel auf die vielen Anwendungsfälle im Detail eingeht, gilt es aber erst einmal, einen Schritt zurückzutreten und die Vorteile von Concepts kompakt zu nennen:

```
cannot call std::sort with std::_List_iterator<int>
concept RandomAccessIterator<std::_List_iterator<int>> was
not satisfied
```

Die kompakte Fehlermeldung beim Versuch, std::sort auf eine std::list anzuwenden (Abb. 5)

- Die Anforderungen an die Templates sind Bestandteil des Interface.
- Das Überladen von Templates oder die Spezialisierung von Templates ist aufgrund von Concepts möglich.
- Man erhält verbesserte Fehlermeldungen, denn der Compiler kann die Bedingungen an die Templateparameter mit den Template-Argumenten vergleichen.
- Die Verwendung vordefinierter Concepts ist möglich, es lassen sich auch eigene Concepts definieren.
- Die Verwendung von `auto` und Concepts wird vereinheitlicht. Daher lässt sich anstelle von `auto` einfach ein Concept verwenden.
- Wenn eine Funktionsdeklaration ein Concept nutzt, wird die Funktion automatisch zum Funktionstemplate. Das Schreiben von Funktionstemplates wird damit so einfach wie das Schreiben von Funktionen.

Die ersten drei Punkte der obigen Zusammenstellung im Blick, geht es mit den verschiedenen Anwendungen von Concepts weiter. Das Concept `Sortable` lässt sich auf drei Arten einsetzen. Der folgende Abschnitt stellt lediglich die Deklaration eines Funktionstemplates vor:

- Requires Clause

```
template<typename Cont>
    requires Sortable<Cont>
void sort(Cont& container);
```

- Trailing Requires Clause

```
template<typename Cont>
void sort(Cont& container) requires Sortable<Cont>;
```

- Constrained Template Parameter

```
template<Sortable Cont>
void sort(Cont& container)
```

Der Algorithmus `sort` fordert, dass der Container sortierbar ist. `Sortable` muss demnach ein konstanter Ausdruck und ein Prädikat sein.

Weitere Details zu Concepts liefern die kommenden Abschnitte im Stakkatotakt.

Klassen

Es ist möglich, ein Klassentemplate zu definieren, das nur Objekte annimmt (s. Listing 4). Der Compiler beschwert sich in diesem Fall, dass eine Referenz kein Objekt ist. Nun kann man fragen, was ein Objekt ist. Ein mögliche Implementierung der Type-Traits-Funktion `std::is_object` in Listing 4 gibt die Antwort. Ein Objekt ist entweder ein Skalar, ein Array, eine Union oder eine Klasse.

Member-Funktionen

```
template<Object T>
class MyVector{
    ...
    void push_back(const T& e) requires Copyable<T>{}
    ...
};
```

```
C:\Users\rainer>allAnyNone.exe

all(5, true, 5.5, false): false
any(5, true, 5.5, false): true
none(5, true, 5.5, false): false

C:\Users\rainer>
```

Die Funktionstemplates all, any und none in Aktion (Abb. 6)

In diesem Fall fordert die Member-Funktion `push_back`, dass der Templateparameter `T` kopierbar sein muss.

Variadic Templates

Concepts lassen sich auch in Variadic Templates anwenden (s. Listing 6). Die Definitionen der Funktionstemplates verwenden Fold Expressions [j]. `all`, `any` und `none` verlangen von ihrem Typparameter, dass er das Concept Arithmetic unterstützt. `Arithmetic` ist `T` dann, wenn es eine Ganzzahl oder eine Fließkommazahl ist. Der aktuelle Microsoft-Compiler 19.23 unterstützt als einziger teilweise die neue Syntax für Concepts (s. Abb. 6).

Anforderungen und Spezialisierungen

Natürlich lässt sich an einen Templateparameter mehr als eine Anforderung stellen:

```
template <SequenceContainer S,
          EqualityComparable<value_type<S>> T>
Iterator_type<S> find(S&& seq, const T& val){
    ...
}
```

Das Funktionstemplate `find` fordert, sein Container `S` soll ein `SequenceContainer` sein und die Elemente `T` des Containers `S` sollen `EqualityComparable` sein.

`std::advance(iter, n)` setzt seinen Iterator `iter` `n` Positionen weiter. Abhängig vom Iterator wendet die Implementierung Zeigerarithmetik an oder geht lediglich `n` Schritte weiter. Im ersten Fall ist die Ausführungszeit konstant; im zweiten Fall hängt die Ausführungszeit von der Schrittweite `n` ab. Dank Concepts kann `std::advance` aufgrund der Iterator-Kategorie überladen werden (s. Listing 7).

Basierend auf der Iterator-Kategorie, die die Container `std::vector`, `std::list` und `std::forward_list` anbieten, kommt die am besten passende Implementierung von `std::advance` zum Einsatz.

Concepts unterstützen natürlich auch die Templatespezialisierung:

```
template<typename T>
class MyVector{};

template<Object T>
class MyVector{};

MyVector<int> v1;     // Object T
MyVector<int&> v2;    // typename T
```

Listing 7: Überladen mithilfe von Concepts

```
template<InputIterator I>
void advance(I& iter, int n){...}

template<BidirectionalIterator I>
void advance(I& iter, int n){...}

template<RandomAccessIterator I>
void advance(I& iter, int n){...}

// usage

std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto vecIt = vec.begin();
std::advance(vecIt, 5);       //  RandomAccessIterator

std::list<int> lst{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto lstIt = lst.begin();
std::advance(lstIt, 5);       //  BidirectionalIterator

std::forward_list<int> forw{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto forwIt = forw.begin();
std::advance(forwIt, 5);      //  InputIterator
```

`MyVector<int&>` führt zum Aufruf des Klassentemplates mit dem uneingeschränkten Templateparameter, `MyVector<int>` zum Aufruf des Klassentemplates mit dem eingeschränkten Templateparameter.

Fazit

Concepts stellen die Rettung aus den zwei Extremen dar: Funktionen oder Klassen, die für einen konkreten Datentyp definiert sind, und Funktionstemplates oder Klassentemplates, die für alle Datentypen definiert sind. Concepts führen semantische Kategorien auf Templateparameter, die der Compiler prüft. Der Artikel hat gezeigt, dass die Vorteile von Concepts vielfältig sind. So lassen sich Anforderungen an Templates als Teil des Interface ausdrücken und Concepts für das Überladen von Funktionstemplates oder die Spezialisierungen von Templates verwenden. Darüber hinaus quittiert der Compiler den Bruch eines Concepts mit einer sehr aussagekräftigen Fehlermeldung.

Dies ist natürlich nicht die ganze Geschichte zu Concepts. Die Syntax von C++ wird durch Concepts deutlich konsistenter.

(mdo@ix.de)

Onlinequellen

[a] C++ Inisghts: cppinsights.io/s/22249b6a

[b] std::sort: en.cppreference.com/w/cpp/algorithm/sort

[c] ValueSwappable: en.cppreference.com/w/cpp/named_req/ValueSwappable

[d] LegacyRandomAccessIterator: en.cppreference.com/w/cpp/named_req/RandomAccessIterator

[e] MoveAssignable: en.cppreference.com/w/cpp/named_req/MoveAssignable

[f] MoveConstructible: en.cppreference.com/w/cpp/named_req/MoveConstructible

[g] Compare: en.cppreference.com/w/cpp/named_req/Compare

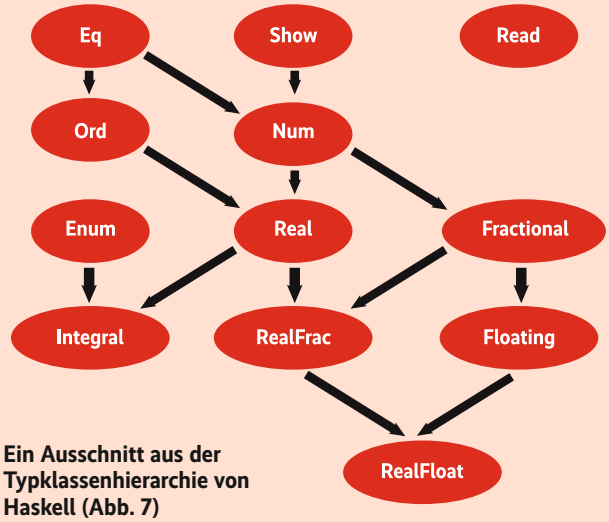
[h] C++ Compiler Support: en.cppreference.com/w/cpp/compiler_support

[i] Concepts Lite: Constraining Templates with Predicates – Andrew Sutton, Bjarne Stroustrup: isocpp.org/blog/2013/02/concepts-lite-constraining-templates-with-predicates-andrew-sutton-bjarne-s

[j] Fold Expressions: grimm-jaud.de/index.php/44-blog/funktional/190-fold-expressions

Die lange, lange Geschichte

Den Autor dieser Zeilen erinnerten Concepts an Haskells Typklassen. Typklassen in Haskell sind Interfaces für ähnliche Datentypen. Abbildung 7 zeigt einen Ausschnitt aus der Typklassenhierarchie von Haskell.



Ein Ausschnitt aus der Typklassenhierarchie von Haskell (Abb. 7)

Doch die Concepts in C++ unterscheiden sich von Haskells Typklassen. Hier ein paar Beobachtungen:

- In Haskell muss ein Datentyp eine Instanz einer Typklasse sein. In C++20 muss ein Datentyp die Anforderungen eines Concepts erfüllen.
- Concepts lassen sich auch auf „Non-Type-Parameter" anwenden. Zum Beispiel kann die Zahl 5 als Non-Type-Parameter verwendet werden. Wenn man ein `std::array` von `int`s mit fünf Elementen haben möchte, wendet man diesen Non-Type-Parameter an: `std::array<int, 5> myVec`.
- Concepts besitzen keine Laufzeitkosten.

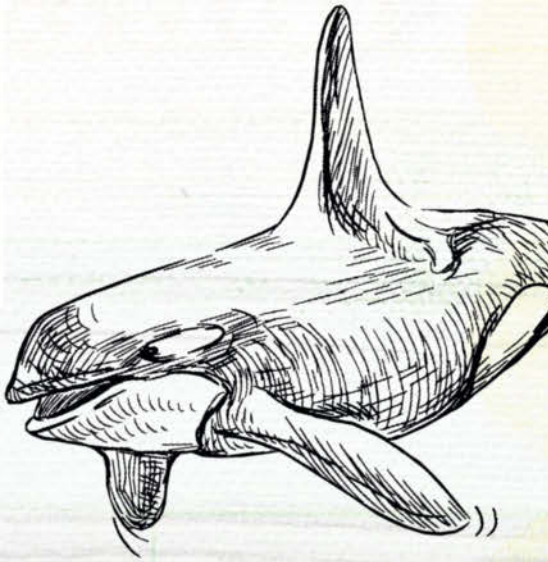
Ursprünglich sollten Concepts bereits das große Feature von C++11 sein. Im Standardisierungsmeeting in Frankfurt am Main wurden sie aber im Juli 2009 entfernt. Das Zitat von Bjarne Stroustrup spricht für sich: „Das C++0x-Concept-Design entwickelte sich zu einem Monster an Komplexität" [i]. Ein paar Jahre später war der nächste Versuch, Concepts zu standardisieren, wieder erfolglos. Concept Lite wurden aus C++17 entfernt. Mit C++20 werden sie nun aber eingeführt.

Rainer Grimm

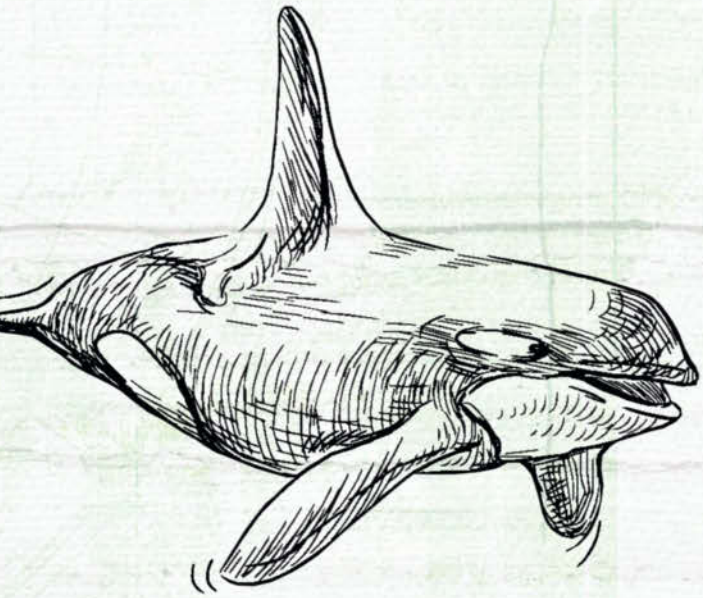
ist Trainer und Coach für C++ und Python. Seine Bücher „C++11 für Programmierer", „C++", „C++ Standardbibliothek", „The C++ Standard Library" und „Concurrency with Modern C++" sind bei dpunkt und Leanpub erschienen.

enterJS

28.9. – 1.10.2020
Online



Die Online-Konferenz für Enterprise JavaScript



- Moderne Softwarearchitektur
- Frameworks & Tools
- Testing & Security
- Frontend & Backend
- JavaScript Deep Dives

Jetzt Ticket sichern!

Silbersponsor
ACCSO ACCELERATED SOLUTIONS

Bronzesponsor
Maiborn Wolff Mensch IT

Veranstalter
iX heise Developer dpunkt.verlag