

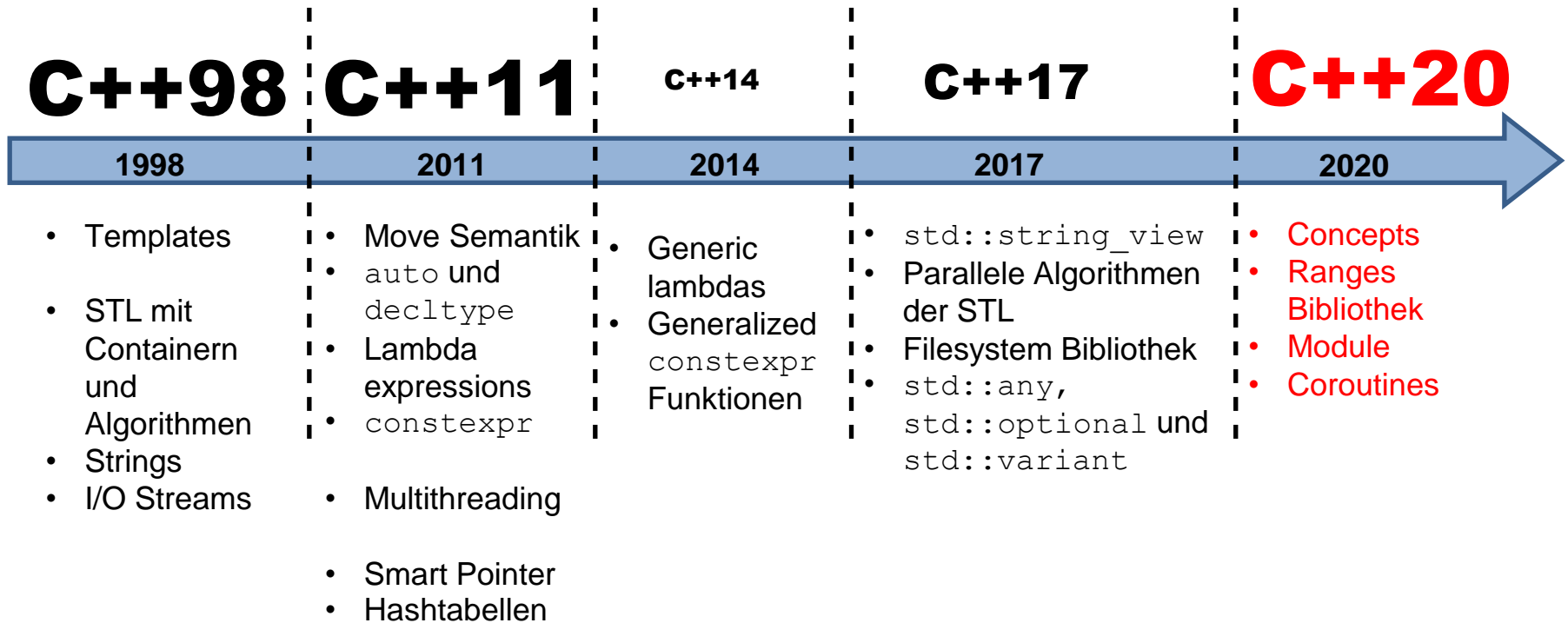


C++20

Die Revolution geht weiter

Rainer Grimm
C++ Trainer

C++-Geschichte



Die großen Vier

Concepts

Ranges Bibliothek

Module

Coroutinen

Vorteile der Concepts

- Drückt Anforderungen an die Template-Parameter durch das Interface aus
- Unterstützt das Überladen von Funktionen und die Spezialisierung von Klassen-Templates
- Erzeugt deutlich verständlichere Fehlermeldungen
- Lassen sich als Platzhalter für generischen Code verwenden
- Lassen sich für Klassen-Templates, Funktions-Template und Methoden von Klassen-Templates anwenden

Funktionen

- **Requires clause**

```
template<typename T>  
requires std::integral<T>  
T gcd(T a, T b);
```

- **Abschließende requires clause**

```
template<typename T>  
T gcd(T a, T b) requires std::integral<T>;
```

- **Eingeschränkte Template-Parameter**

```
template<std::integral T>  
T gcd(T a, T b);
```

- **Abbreviated Function Templates Syntax**

```
auto gcd(std::integral auto a, std::integral auto b);
```

Klassen

```
template<std::regular T>  
class MyVector{};
```


```
MyVector<int> v1; // OK
```

```
MyVector<int&> v2; // ERROR: int& is not regular
```

 Eine Referenz ist nicht regulär.

Methoden

```
template <typename T>
struct MyVector{
    void push_back(const T&) requires std::copyable<T> {}
};
```

 Der Typ-Parameter T muss kopierbar sein.

Überladen

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

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

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

- `std::advance` schiebt den Iterator `n` Positionen weiter
- Abhängig vom Iterator wird eine andere Überladung verwendet

```
std::list<int> lst{1,2,3,4,5,6,7,8,9};
```


```
std::list<int>::iterator i = lst.begin();
```

```
std::advance(i, 2);  std::bidirectional_iterator
```


Spezialisierung

```
template <typename T>  
struct Vector {};
```

```
template <std::regular Reg>  
struct Vector<Reg> {};
```

```
 Vector<int> v1; // std::regular Reg  
Vector<int&> v2; // typename T
```

`Vector<int&>` verwendet den uneingeschränkten Template-Parameter.

`Vector<int>` verwendet den eingeschränkten Template-Parameter.

Mehrere Anforderungen

```
template<typename Iter, typename Val>
    requires std::input_iterator<Iter>    &&
             std::equality_comparable<Value_type<Iter>, Val>
Iter find(Iter b, Iter e, Val v) {
    ...
}
```

- `find` fordert für den Iterator `Iter` und für seinen Vergleich mit `Val`
 - Der Iterator muss ein Input-Iterator sein
 - Der Iterator muss sich mit dem Element `Val` auf Gleichheit vergleichen lassen

Platzhalter: `auto`

C++20

- `auto`: Uneingeschränkte Platzhalter (unconstrained placeholder)
- `Concept`: Eingeschränkte Platzhalter (constrained placeholder)

 Die Verwendung von Platzhaltern erzeugt Templates.

Eingeschränkt und uneingeschränkt

```
#include <concepts>
#include <iostream>
#include <vector>

std::integral auto getInteg(
    int val) {
    return val;
}
```

```
int main(){
    std::vector<int> vec{1, 2, 3};
    for (std::integral auto i: vec) {
        std::cout << i << " ";
    }
    std::integral auto b = true;
    std::cout << b << std::endl;

    std::integral auto i = getInteg(10);
    std::cout << i << std::endl;

    auto i1 = getInteg(10);
    std::cout << i1 << std::endl;
}
```

Syntactic Sugar

Klassisch

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

```
template<std::integral T>
T gcd2(T a, T b){
    if( b == 0 ) return a;
    else return gcd(b, a % b);
}
```

Abbreviated Function Templates

```
std::integral auto gcd3(
    std::integral auto a,
    std::integral auto b) {
    if( b == 0 ) return a;
    else return gcd(b, a % b);
}
```

```
auto gcd4(auto a, auto b){
    if( b == 0 ) return a;
    else return gcd(b, a % b);
}
```

Syntactic Sugar

```
int main(){  
  
    std::cout << std::endl;  
    std::cout << "gcd(100, 10)= " << gcd(100, 10) << std::endl;  
    std::cout << "gcd1(100, 10)= " << gcd1(100, 10) << std::endl;  
    std::cout << "gcd2(100, 10)= " << gcd2(100, 10) << std::endl;  
    std::cout << "gcd3(100, 10)= " << gcd3(100, 10) << std::endl;  
    std::cout << "gcd4(100, 10)= " << gcd4(100, 10) << std::endl;  
    std::cout << std::endl;  
  
}
```



```
gcd(100, 10)= 10  
gcd1(100, 10)= 10  
gcd2(100, 10)= 10  
gcd3(100, 10)= 10  
gcd4(100, 10)= 10
```

Syntactic Sugar: Überladung

```
void overload(auto t){
    std::cout << "auto : " << t << std::endl;
}

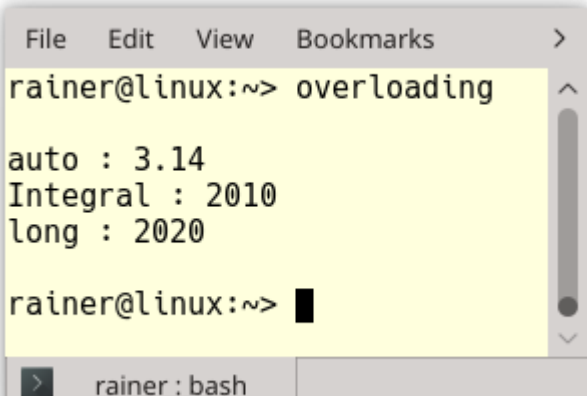
void overload(std::integral auto t){
    std::cout << "Integral : " << t << std::endl;
}

void overload(long t){
    std::cout << "long : " << t << std::endl;
}
```

```
int main(){

    overload(3.14);
    overload(2010);
    overload(20201);

}
```

A terminal window with a menu bar (File, Edit, View, Bookmarks) and a title bar (rainer@linux:~> overloading). The terminal output shows the results of the program execution: 'auto : 3.14', 'Integral : 2010', and 'long : 2020'. The prompt 'rainer@linux:~>' is visible at the bottom.

```
File Edit View Bookmarks >
rainer@linux:~> overloading
auto : 3.14
Integral : 2010
long : 2020
rainer@linux:~> █
rainer : bash
```

Das Concept Ord

Das Concept Equal

```
template<typename T>
concept Equal =
    requires(T a, T b) {
        { a == b } -> std::convertible_to<bool>;
        { a != b } -> std::convertible_to<bool>;
    };
```


Das Concept Ord

Das Concept Ord

```
template <typename T>
concept Ord =
    Equal<T> &&
    requires(T a, T b) {
        { a <= b } -> std::convertible_to<bool>;
        { a < b } -> std::convertible_to<bool>;
        { a > b } -> std::convertible_to<bool>;
        { a >= b } -> std::convertible_to<bool>;
    };
```

Das Concept Ord

```
int main() {  
  
    bool areEqual(Equal auto a,  
                  Equal auto b) {  
        return a == b;  
    }  
  
    Ord getSmaller(Ord auto a,  
                  Ord auto b) {  
        return (a < b) ? a : b;  
    }  
  
    std::cout << areEqual(1, 5);  
  
    std::cout << getSmaller(1, 5);  
  
    std::unordered_set<int> set1{1, 2, 3};  
    std::unordered_set<int> set2{5, 4, 3};  
  
    std::cout << areEqual(set1, set2);  
  
    // auto smallerSet= getSmaller(set1,  
                                   set2);  
  
}
```

Das Concept Ord

```
<source>:19:13: note: the required expression '(a <= b)' is invalid
 19 |         { a <= b } -> std::convertible_to<bool>;
    |         ~~~~~
<source>:20:13: note: the required expression '(a < b)' is invalid
 20 |         { a < b } -> std::convertible_to<bool>;
    |         ~~~~~
<source>:21:13: note: the required expression '(a > b)' is invalid
 21 |         { a > b } -> std::convertible_to<bool>;
    |         ~~~~~
<source>:22:13: note: the required expression '(a >= b)' is invalid
 22 |         { a >= b } -> std::convertible_to<bool>;
    |         ~~~~~
```

Die großen Vier

Concepts

Ranges Bibliothek

Module

Coroutinen

Die Ranges Bibliothek

Die Ranges Bibliothek bietet Algorithmen an,

- die direkt auf dem Container arbeiten.
- die Lazy evaluiert werden.
- die sich komponieren lassen.

 Die Ranges Bibliothek erweitert C++20 um funktionale Pattern.

Direkt auf dem Container

```
std::array<int, 6> arr{3, 1, 4, 1, 5, 9};  
std::ranges::reverse_view revRang{arr};  
for (int i : revRange) std::cout << i << " ";
```

 **9 5 1 4 1 3**

```
std::map<std::string, int> freqWord{ {"witch", 25},  
    {"wizard", 33}, {"tale", 45}, {"dog", 4} };  
auto names = std::views::keys(freqWord);  
for (const auto& n : names){ std::cout << n << " " ; };
```

 **dog tale witch wizard**


Lazy evaluiert

```
#include <ranges>
```

```
#include <iostream>
```


```
int main() {
```

```
    for (int i : std::views::iota{1, 5}) {
```

```
        std::cout << i << ' ';  1 2 3 4 5
```

```
    }
```

```
    for (int i : std::views::iota(1) | std::view::take(5)) {
```

```
        std::cout << i << ' ';  1 2 3 4 5
```


```
    }
```

```
}
```

Funktionskomposition

```
#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
    }
}
```


Die großen Vier

Concepts

Ranges Bibliothek

Module

Coroutinen

Module

Modul

```
// module declaration  
export module math;  
  
// exported function  
export int add(int fir, int sec) {  
    return fir + sec;  
}
```

Client

```
// imported module  
  
import math;  
  
int main(){  
    add(2000, 20);  
  
}
```

Vorteile von Modulen

- Ein Modul wird nur einmal importiert. Dieser Prozess ist buchstäblich umsonst.
- Es stellt keinen Unterschied dar, in welcher Reihenfolge Module importiert werden.
- Identische Namen mit Modulen sind sehr unwahrscheinlich.
- Module erlauben es, die logische Struktur des Codes auszudrücken.
 - Module erlauben das explizite Exportieren von Namen
 - Module lassen sich einfach in neue Module verpacken

Module Interface Unit

```
export module math;
```

```
export int add(int fir, int sec);
```

Die Module Interface Unit

- Enthält die exportierende Moduldeklaration: `export module math`
 - Namen können nur in der Module Interface Unit exportiert werden
 - Namen, die nicht exportiert werden, sind außerhalb des Moduls nicht sichtbar
-
- Ein Modul kann nur eine Module Interface Unit besitzen.

Module Implementation Unit

```
module math;  
  
int add(int fir, int sec) {  
    return fir + sec;  
}
```

Die Module Implementation Unit

- enthält die nicht-exportierende Moduldeklaration: `module math;`
- Ein Modul kann mehr als eine Module Implementation Unit besitzen.

Struktur eines Moduls

```
module;                                // global module fragment

#include <headers for libraries not modularized so far>

export module math;                    // module declaration

import <importing of other modules>

<non-exported declarations>          // names with only visibility
                                     // inside the module

export namespace math {

    <exported declarations>           // exported names

}
```

Die großen Vier

Concepts

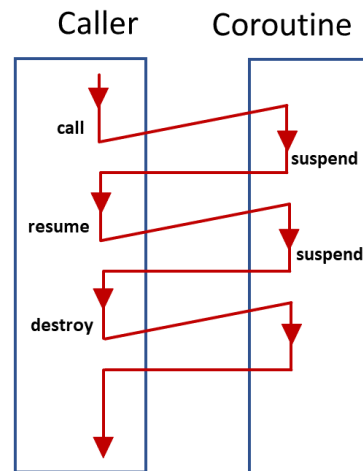
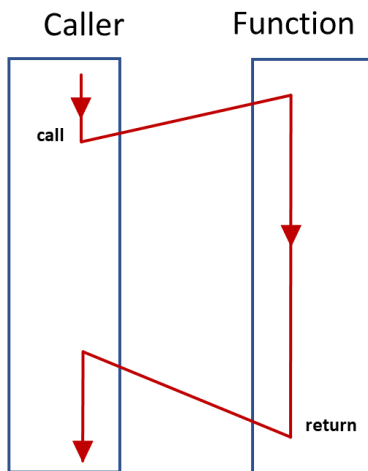
Ranges Bibliothek

Module

Coroutinen

Coroutinen

Coroutinen sind verallgemeinerte Funktionen, die ihre Ausführung unterbrechen und wieder aufnehmen können.



Typische Einsatzgebiete

- Kooperative Task
- Eventschleifen
- Unendliche Datenströme
- Pipelines

Coroutinen

Design Principles (James McNellis)

- **Scalable**, to billions of concurrent coroutines
- **Efficient**: Suspend/resume operations comparable in cost to function call overhead
- **Open-Ended**: Library designers can develop coroutine libraries
- **Seamless Interaction** with existing facilities with no overhead.
- **Usable** in environments where exceptions are forbidden or not available.

Coroutinen

	Funktion	Coroutine
invoke	<code>func (args)</code>	<code>func (args)</code>
return	<code>return statement</code>	<code>co_return statement</code>
suspend		<code>co_await expression</code> <code>co_yield expression</code>
resume		<code>coroutine_handle<>::resume ()</code>

Eine Funktion ist eine Coroutine, falls sie einen Aufruf `co_return`, `co_await`, `co_yield` oder eine Range-basierte for-Schleife `co_await` enthält.

Coroutinen: Warten statt blockieren

Blockieren

```
Acceptor ac{443};

while (true) {
    Socket so= ac.accept(); // block
    auto req= so.read();    // block
    auto resp= handleRequest(req);
    so.write(resp);        // block
}
```

Warten


```
Acceptor ac{443};

while (true) {
    Socket so= co_await ac.accept();
    auto req= co_await so.read();
    auto resp= handleRequest(req);
    co_await so.write(resp);
}
```

Coroutinen: Generatoren

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

```
auto gen = getNext(-10);  
for (int i= 1; i <= 20; ++i) {  
    std::cout << gen.getNextValue() << " ";  
}
```

 **-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10**



Live Demo

[infiniteDataStream.cpp](#)

Die großen Vier

Concepts

Semantische
Kategorien

Module

Software-
komponenten

Ranges
Bibliothek

Bedarfsaus-
wertung und
Funktions-
komposition

Coroutinen

Asynchrone
Programmierung

Fragen?

[Blogposts auf Heise Developer](#)

[C++20](#): LeanPub (digital)

[C++20](#): Amazon (gedruckt)