C++20
Rainer Grimm
Training, Coaching und Technologieberatung
www.ModernesCpp.de

Geschichte von C++

C++98 — 1998

- Templates
- STL mit Containern und Algorithmen
- Strings
- I/O Streams

C++11 — 2011

- Move Semantik
- Vereinheitlichte Initialisierung
- `auto` und `decltype`
- Lambda-Funktionen
- `constexpr`

- Multithreading und das Speichermodell

- Reguläre Ausdrücke
- Smart Pointer
- Hashtabellen
- `std::array`

C++14 — 2014

- Reader-Writer Locks
- Verallgemeinerte Lambda-Funktionen

C++17 — 2017

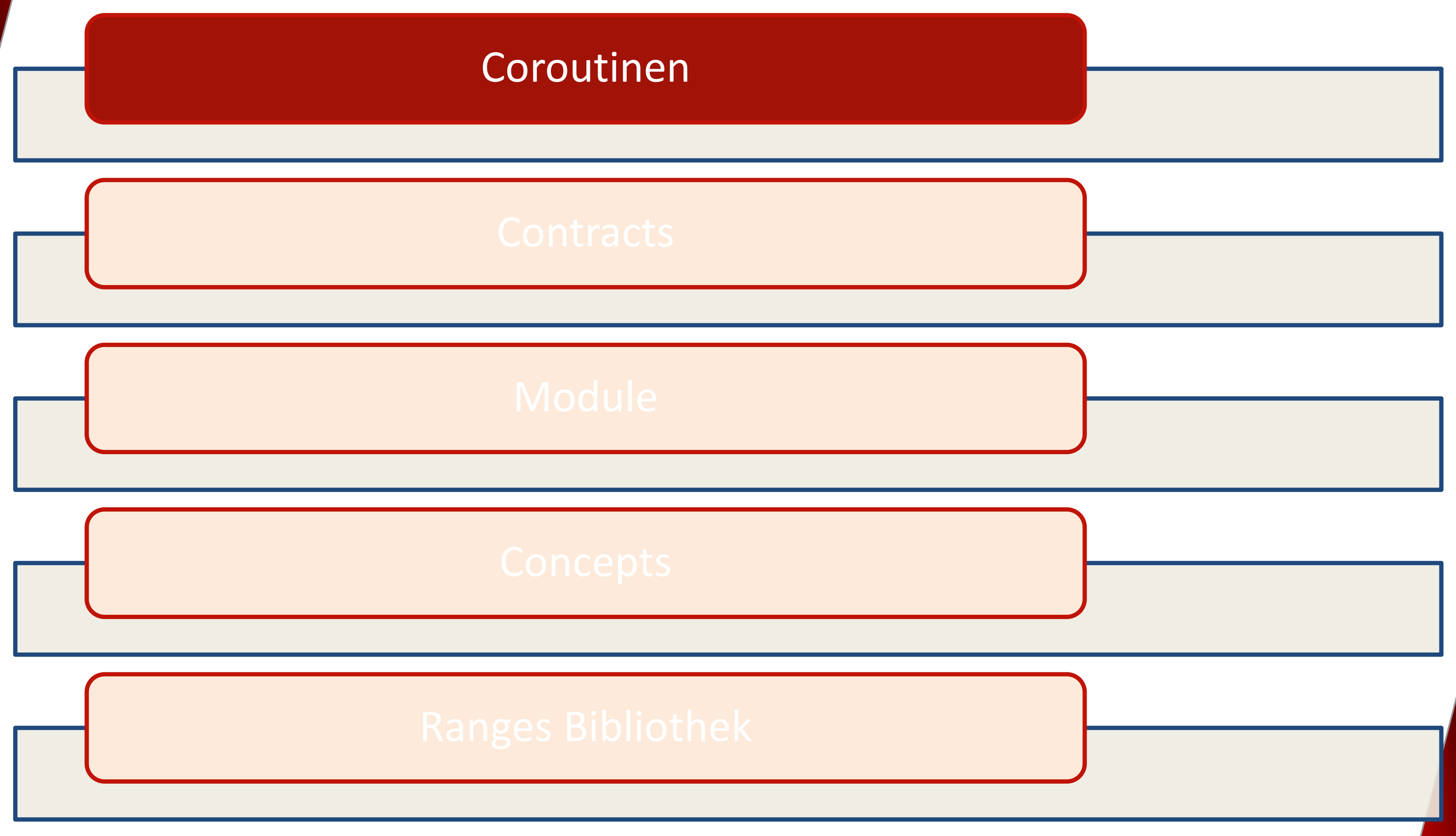
- Fold expressions
- constexpr if
- Structured binding

- `std::string_view`
- Parallele Algorithmen der STL
- Dateisystem Bibliothek
- `std::any`, `std::optional` und `std::variant`

C++20 — 2020

- Contracts
- Concepts
- Concepts
- Range Bibliothek
- Couroutinen

Die großen Fünf



Coroutinen

Coroutinen sind verallgemeinerte Funktionen, die ihre Ausführung unterbrechen und wieder aufnehmen können und dabei ihren Zustand speichern.

- Typische Einsatzgebiete
  - Kooperative Tasks
  - Eventschleifen
  - Unendliche Datenströme
  - Pipelines

Coroutinen

Design Principles (James McNellis)

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

Coroutinen

	Function	Coroutine
`invoke`	`func(args)`	`func(args)`
`return`	`return statement`	`co_return statement`
`suspend`		`co_await expression` `co_yield expression`
`resume`		`coroutine_handle<>::resume()`

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: Generatoren

```
generator<int> genForNumbers(int begin, int inc = 1){
    for (int i = begin;; i += inc){
        co_yield i;
    }
}

int main(){
    auto numbers = genForNumbers(-10);
    for (int i= 1; i <= 20; ++i) std::cout << numbers << " ";
    for (auto n: genForNumbers(0, 5)) std::cout << n << " ";
}
```

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

0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 ...

Coroutinen: Warten statt blockieren

Blockieren

```
Acceptor accept{443};

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

Warten

```
Acceptor accept{443};

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

Die großen Fünf

- Coroutinen
- Contracts
- Module
- Concepts
- Ranges Bibliothek

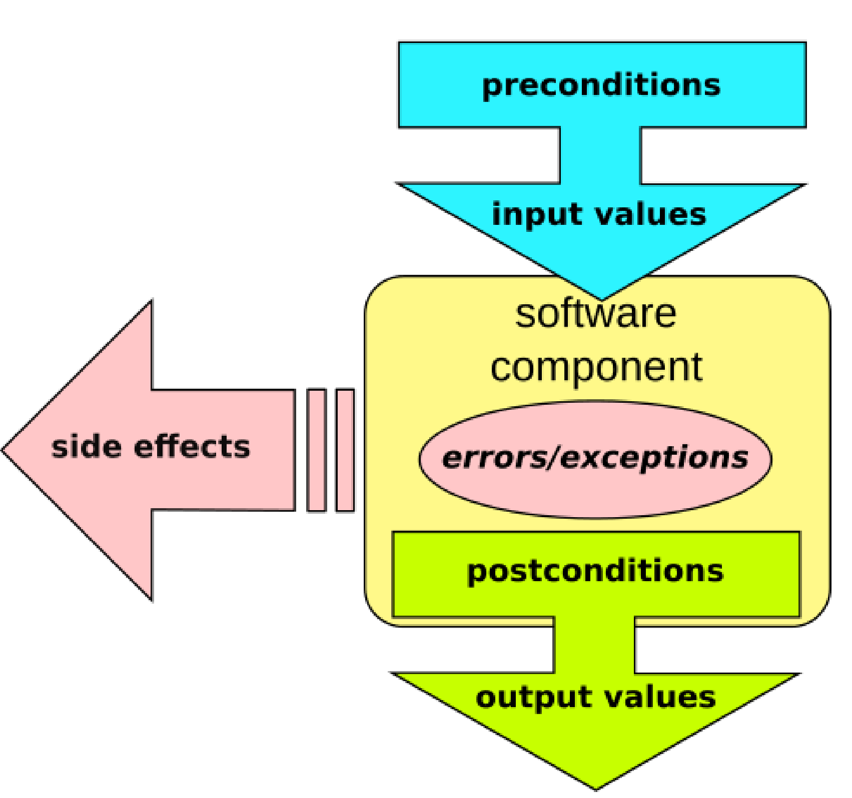
Contracts

Ein Contract stellt in einer präzisen und prüfbaren Art ein Interface für eine Softwarekomponente dar.

⇒ Design By Contract

- Softwarekomponenten sind typischerweise Funktionen oder Methoden.
- Design By Contract geht auf Bertrand Meyer (Eiffel) zurück.
- Das reibungslose Zusammenspiel der Komponenten wird durch Vorbedingungen, Nachbedingungen und Invarianten gewährleistet.

Contracts



- Eine **Vorbedingung** (Precondition) ist ein Prädikat, das gelten muss, bevor die Komponente aufgerufen wird.
- Eine **Nachbedingung** (Postcondition) ist ein Prädikat, das gelten muss, nachdem die Komponente aufgerufen wurde.
- Eine **Zusicherung** (Invariante) ist ein Prädikat, das an der Stelle im Code gelten muss, an der es platziert ist.

Contracts

```
int push(queue& q, int val)
  [[ expects: !q .full() ]]
  [[ ensures !q.empty() ]]{
   ...
  [[assert: q.is_ok() ]]
}


class X {
public:
  void f(int n)
    [[ expects: n < m ]] // error
  {
    [[ assert: n < m ]]; // OK
    ...
  }
private:
  int m;
};
```

- `expects:` Vorbedingung
- `ensures:` Nachbedingung
- `assert:` Zusicherung

- Vor- und Nachbedingungen:
  - Teil des Interfaces
  - werden außerhalb der Funktionsdefinition verwendet

- Zusicherungen:
  - Teil der Implementierung
  - werden innerhalb der Funktionsdefinition verwendet

Contracts

```
int mul(int x, int y)

  [[expects: x > 0]] // => default
  [[expects default: y > 0]]
  [[ensures audit res: res > 0]]{

  return x * y;
}
```

- `default`: die Kosten den Contract zur Laufzeit zu prüfen sind gering (Standard)
- `audit`: die Kosten den Contract zur Laufzeit zu prüfen sind hoch
- `axiom`: das Prädikat wird nicht zur Laufzeit geprüft

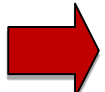
- `res`: der Identifier erlaubt es, den Rückgabewert anzusprechen

Verletzungen des Kontrakts

Flags zur Übersetzungszeit:

- **off**: keine Contracts werden geprüft
- **default**:
  - `default`: Contracts werden geprüft (Standard)
- **audit**:
  - `default` und `audit`: Contracts werden geprüft

Bei Verletzung eines Kontrakts wird der Violation Handler aufgerufen.

 `std::terminate`

```
class contract_violation{
public:
  uint_least32_t line_number() const noexcept;
  string_view file_name() const noexcept;
  string_view function_name() const noexcept;
  string_view comment() const noexcept;
  string_view assertion_level() const
noexcept;
};
```

Famous Last Words zu Contractsn

Sutter's Mill:

"contracts is the most impactful feature of C++20 so far, and arguably the most impactful feature we have added to C++ since C++11."

Die großen Fünf

- Coroutinen
- Contracts
- **Module**
- Concepts
- Ranges Bibliothek

Module

```
// math.cppm

// module declaration
export module math;


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

```
// main.cpp

// imported module
import math;

int main(){

   add(2000, 20);

}
```

Vorteile von Modulen

- Schnellere Übersetzungszeiten
  - ein Modul wird nur einmal buchstäblich umsonst importiert.
- Isolation von Makros
  - Namenskollisionen und Abhängigkeiten mit Makros vermeiden
- Ausdruck der logischen Struktur des Codes
  - explizites Exportieren von Namen steuern
  - Module können neu verpackt werden
- Keine Headerdateien mehr notwendig
  - Keine Trennung Interface- und Implementierungseinheiten
  - Anzahl der Quelldateien wird halbiert
- Hässliche Workaround loswerden
  - keine Inklude Guards oder Makros mit LONG_UPPERCASE_NAMES sind mehr notwendig

Modul-Interface-Einheit

```
// math1.cppm

export module math1;

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

Die Modul-Interface-Einheit

- enthält die exportierende Moduldeklaration: `export module math1`.
- Namen können nur in der Modul-Interface-Einheit exportiert werden.
- Namen, die nicht exportiert werden, sind außerhalb des Moduls nicht sichtbar.

- Ein Modul kann nur eine Modul-Interface-Einheit besitzen.

Modul-Implementierungs-Einheit

```
// math1.cpp

module math1;

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

Die Modul-Implementierungs-Einheit

- enthält die nicht-exportierende Moduldeklaration: `module math1;`

- Ein Modul kann mehr als eine Modul-Implementierungs-Einheit besitzen.

Vordefinierte Module

- **std.regex:** `<regex>`
- **std.filesystem:** `<experimental/filesystem>`
- **std.memory:** `<memory>`
- **std.threading:** `<atomic>, <condition_variable>, <future>, <mutex>, <shared_mutex>, <thread>`
- **std.core:** Rest der Standard Template Library

Die großen Fünf

- Coroutinen
- Contracts
- Module
- Concepts
- Ranges Bibliothek

Die Vorteile von Concepts

- Drücken die Anforderungen der Template-Parameter im Interface aus
- Unterstützen das Überladen von Funktionen und die Spezialisierung von Templates
- Erzeugen deutlich lesbarere Fehlermeldungen, in dem die Anforderungen an die Template-Parameter mit den Template-Argumenten verglichen werden
- Können als Platzhalter für die generische Programmierung verwendet werde
- Können für alle Templates angewandt werden

Funktionen

Verwendung des Concepts `Sortable`.

implizit

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

explizit

```
template<typename Cont>
    requires Sortable<Cont>()
void sort(Cont& container){
    ...
}
```

- Verwendung:

```
std::list<int> lst = {1998, 2014, 2003, 2011};
sort(lst); // ERROR: lst is no random-access container with <
```

- `Sortable`
  - muss ein konstanter Ausdruck, der ein Prädikat darstellt

Klassen

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

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

➡ Eine Referenz ist kein Objekt.

Methoden

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

- Der Type-Parameter `T` muss sich kopieren lassen
- Das Concept muss vor der Methodendeklaration angegeben werden.

Mehrere Anforderungen

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

- `find` verlangt, dass die Elemente des Containers
  - eine Sequenz bilden
  - auf Gleichheit vergleichbar sind

Überladung

```
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){...}
```

- `std::advance` setzt seinen Iterator `n` Positionen weiter.
- abhängig vom Iterator werden verschiedene Funktions-Templates 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);   //  BidirectionalIterator
```

Spezialisierung

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

template<Object T>
class MyVector{};

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

`MyVector<int&>` ruft den unconstrained Template-Parameter auf.

`MyVector<int>` ruft den constrained Template-Parameter auf

Placeholder Syntax: `auto`

Ausflug: Asymmetrie in C++14

```
auto genLambdaFunction= [](auto a, auto b) {
    return a < b;
};

template <typename T, typename T2>
auto genFunction(T a, T2 b){
    return a < b;
}
```

➡ Generische Lambdas stellten eine neue Art dar, Templates zu definieren.

Placeholder Syntax: `auto`

C++20 hebt die Asymmetry auf

- `auto`: Unconstrained placeholder
- Concepts: Constrained placeholder

➡ Die Verwendung von Platzhaltern erzeugt Templates.

Constrained und Unconstrained

Constrained Concepts können anstelle von Unconstrained Concepts `auto` verwenden werden.

```
#include <iostream>
#include <type_traits>
#include <vector>

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

Integral auto getIntegral(int val){
    return val;
}
```

```
int main(){

    std::vector<int> vec{1, 2, 3, 4, 5};
    for (Integral auto i: vec)
        std::cout << i << " ";

    Integral auto b = true;
    std::cout << b << std::endl;

    Integral auto integ = getIntegral(10);
    std::cout << integ << std::endl;

    auto integ1 = getIntegral(10);
    std::cout << integ1 << std::endl;

}
```

Syntaktischer Zucker

Klassisch

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

template<Integral T>
T gcd1(T a, T b){
    if( b == 0 ) return a;
    else return gcd(b, a % b);
}
```

Abbreviated Function Templates

```
Integral auto gcd2(Integral auto a,
                   Integral auto b){
    if( b == 0 ) return a;
    else return gcd(b, a % b);
}

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

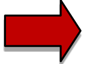
Syntaktischer Zucker

```
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 << std::endl;

}
```

Übersetzt mit GCC 6.3 und dem Flag `-fconcepts`



Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe

```
rainer@suse:~> conceptsIntegralVariations

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

rainer@suse:~>
```

rainer : bash

Placeholder Syntax: Kleine Unterschiede

```
Integral gcd2(Integral a, Integral b){
  if( b == 0 ) return a;
  else return gcd(b, a % b);
}
```

`gcd2's` Typ-Parameter

- muss `Integral` sein
- muss derselbe Typ sein

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

`gcd3's` Typ-Parameter

- kann verschieden Typen annehmen

Syntaktischer Zucker: Überladung

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

void overload(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(2020l);

}
```

```
File   Edit   View   Bookmarks
rainer@linux:~> overloading

auto : 3.14
Integral : 2010
long : 2020

rainer@linux:~>
rainer : bash
```

Vordefinierte Concepts (Ranges)

- Core language concepts
  - Same
  - DerivedFrom
  - ConvertibleTo
  - Common
  - Integral
  - Signed Integral
  - Unsigned Integral
  - Assignable
  - Swappable
- Compare concepts
  - Boolean
  - EqualityComparable
  - StrictTotallyOrdered
- Object concepts
  - Destructible
  - Constructible
  - DefaultConstructible
  - MoveConstructible
  - Copy Constructible
  - Movable
  - Copyable
  - Semiregular
  - Regular
- Callable concepts
  - Callable
  - RegularCallable
  - Predicate
  - Relation
  - StrictWeakOrder

Concept Definitition:Variable-Concept

```
template<typename T>
concept bool Integral =
    std::is_integral<T>::value;
}
```

- T erfüllt das Variable-Concept `Integral`, falls `std::integral<T>::value true` ergibt

Concept Definition: Funktion-Concept

Concepts TS

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

Entwurf C++20

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

- `T` erfüllt das Funktion-Concept falls `==` und `!=` überladen sind und die Operationen einen Wahrheitswert zurückgeben

Das Concept Equal

```
bool areEqual(Equal auto a, Equal auto b) return a == b;

/*
struct WithoutEqual{
  bool operator == (const WithoutEqual& other) = delete;
};
struct WithoutUnequal{
  bool operator != (const WithoutUnequal& other) = delete;
};
*/
. . .
std::cout << "areEqual(1, 5): " << areEqual(1, 5) << std::endl;

/*
bool res = areEqual(WithoutEqual(),  WithoutEqual());
bool res2 = areEqual(WithoutUnequal(),  WithoutUnequal());
*/
```

The Concept Equal

```
File   Edit   View   Bookmarks   Settings   Help
rainer@suse:~> conceptsDefinitionEqual

areEqual(1, 5): false

rainer@suse:~>
rainer : bash
```

```
File   Edit   View   Bookmarks   Settings   Help
rainer@suse:~> g++ -fconcepts conceptsDefinitionEqual.cpp -o conceptsDefinitionEqual
conceptsDefinitionEqual.cpp: In function 'int main()':
conceptsDefinitionEqual.cpp:37:54: error: cannot call function 'bool areEqual(auto:1, auto:1) [with auto:1 = WithoutEqual]'
   bool res = areEqual(WithoutEqual(),  WithoutEqual());
                                                      ^
conceptsDefinitionEqual.cpp:13:6: note:   constraints not satisfied
 bool areEqual(Equal a, Equal b){
      ^~~~~~~~
conceptsDefinitionEqual.cpp:6:14: note: within 'template<class T> concept bool Equal() [with T = WithoutEqual]'
 concept bool Equal(){
              ^~~~~
conceptsDefinitionEqual.cpp:6:14: note:     with 'WithoutEqual a'
conceptsDefinitionEqual.cpp:6:14: note:     with 'WithoutEqual b'
conceptsDefinitionEqual.cpp:6:14: note: the required expression '(a == b)' would be ill-formed
conceptsDefinitionEqual.cpp:6:14: note: 'b->a.WithoutEqual::operator==()' is not implicitly convertible to 'bool'
conceptsDefinitionEqual.cpp:6:14: note: the required expression '(a != b)' would be ill-formed
conceptsDefinitionEqual.cpp:39:59: error: cannot call function 'bool areEqual(auto:1, auto:1) [with auto:1 = WithoutUnequal]'
   bool res2 = areEqual(WithoutUnequal(),  WithoutUnequal());
                                                           ^
conceptsDefinitionEqual.cpp:13:6: note:   constraints not satisfied
 bool areEqual(Equal a, Equal b){
      ^~~~~~~~
conceptsDefinitionEqual.cpp:6:14: note: within 'template<class T> concept bool Equal() [with T = WithoutUnequal]'
 concept bool Equal(){
              ^~~~~
conceptsDefinitionEqual.cpp:6:14: note:     with 'WithoutUnequal a'
conceptsDefinitionEqual.cpp:6:14: note:     with 'WithoutUnequal b'
conceptsDefinitionEqual.cpp:6:14: note: the required expression '(a == b)' would be ill-formed
conceptsDefinitionEqual.cpp:6:14: note: the required expression '(a != b)' would be ill-formed
conceptsDefinitionEqual.cpp:6:14: note: 'b->a.WithoutUnequal::operator!=()' is not implicitly convertible to 'bool'
rainer@suse:~>
rainer : bash
```

`Eq` versus `Equal`

Die Typklasse Eq

```
class Eq a where
   (==) :: a -> a -> Bool
   (/=) :: a -> a -> Bool
```

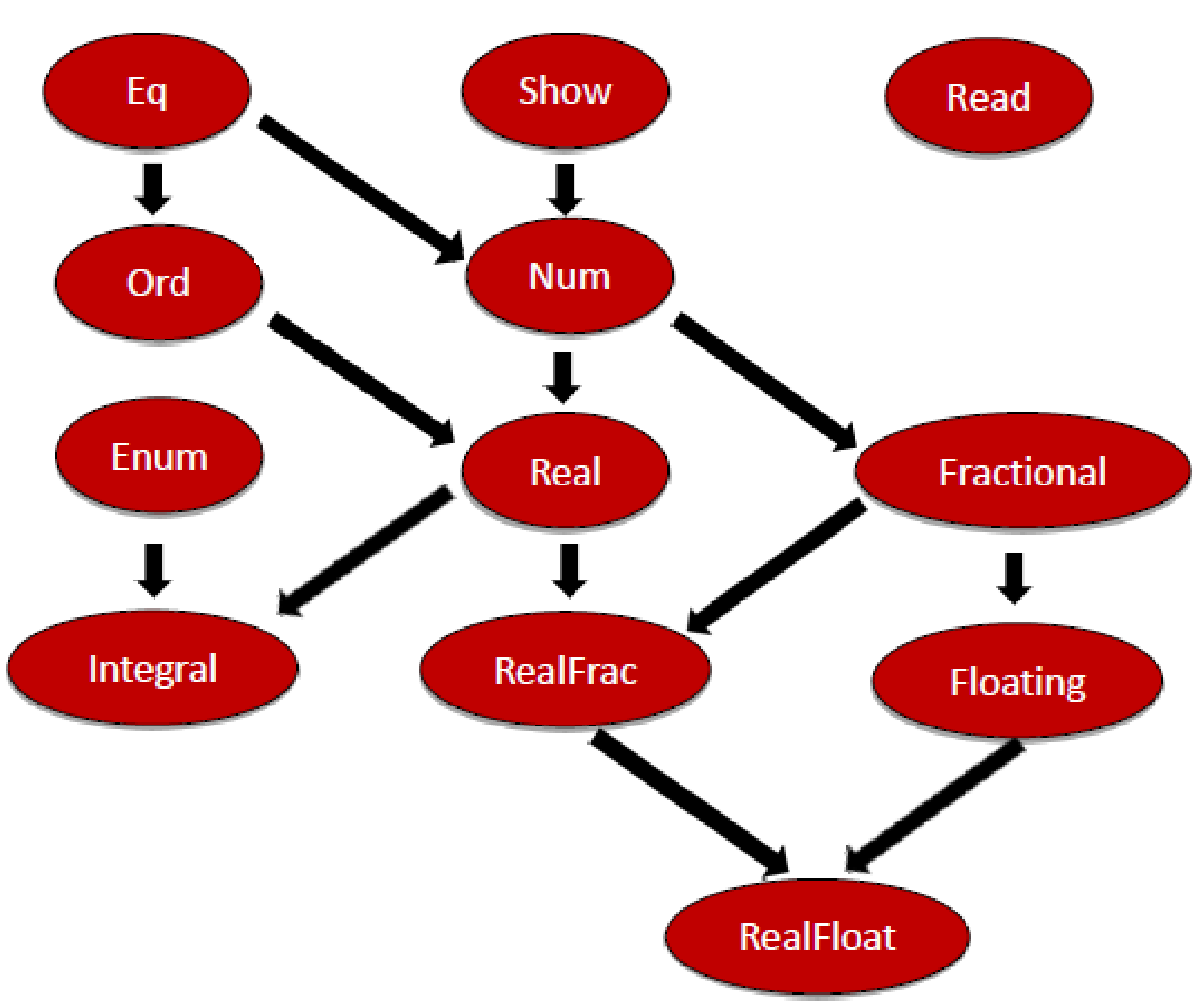
Das Concept Equal

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

Die Typklasse `Eq` (Haskell) und das `concept Equal` (C++) verlagen von ihren Typen

- sie müssen Gleicheit und Ungleichheit unterstützen
- beide Operationen müssen einen Wahrheitswert zurückgeben
- beide Typen müssen identisch sein

Haskells Typklasse



Haskells Typklasse `Ord`

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
```

➡ Jeder Typ, der `Ord` unterstützt, muss auch `Eq` untersützten.

Das Concept `Ord`

Das Concept Equal

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

Das Concept Ord

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

The Concept `Ord`

```
bool areEqual(Equal auto a,
              Equal auto b){
  return a == b;
}

Ord getSmaller(Ord auto a,
               Ord auto b){
  return (a < b) ? a : b;
}
```

```
int main(){

  std::cout << areEqual(1, 5);

  std::cout << getSmaller(1, 5);

  std::unordered_set<int> firSet{1, 2, 3, 4, 5};
  std::unordered_set<int> secSet{5, 4, 3, 2, 1};

  std::cout << areEqual(firSet, secSet);

  // auto smallerSet= getSmaller(firSet, secSet);

}
```

The Concept `Ord`

```
File   Edit   View   Bookmarks   Settings   Help
rainer@suse:~> conceptsDefinitionOrd

areEqual(1, 5): false
getSmaller(1, 5): 1
areEqual(firSet, secSet): true

rainer@suse:~>
rainer : bash
```

```
File   Edit   View   Bookmarks   Settings   Help
rainer@suse:~> g++ -fconcepts conceptsDefinitionOrd.cpp -o conceptsDefinitionOrd
conceptsDefinitionOrd.cpp: In function 'int main()':
conceptsDefinitionOrd.cpp:44:45: error: cannot call function 'auto getSmaller(auto:2, auto:2)
 [with auto:2 = std::unordered_set<int>]'
   auto smallerSet= getSmaller(firSet, secSet);
                                             ^
conceptsDefinitionOrd.cpp:27:5: note:   constraints not satisfied
 Ord getSmaller(Ord a, Ord b){
     ^~~~~~~~~~
conceptsDefinitionOrd.cpp:13:14: note: within 'template<class T> concept bool Ord() [with T =
 std::unordered_set<int>]'
 concept bool Ord(){
              ^~~
conceptsDefinitionOrd.cpp:13:14: note:     with 'std::unordered_set<int> a'
conceptsDefinitionOrd.cpp:13:14: note:     with 'std::unordered_set<int> b'
conceptsDefinitionOrd.cpp:13:14: note: the required expression '(a <= b)' would be ill-formed
conceptsDefinitionOrd.cpp:13:14: note: the required expression '(a < b)' would be ill-formed
conceptsDefinitionOrd.cpp:13:14: note: the required expression '(a > b)' would be ill-formed
conceptsDefinitionOrd.cpp:13:14: note: the required expression '(a >= b)' would be ill-formed
rainer@suse:~>
rainer : bash
```

Die großen Fünf

- Coroutinen
- Contracts
- Module
- Concepts
- **Ranges Bibliothek**

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.

Funktionskomposition

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

int main(){
  for (int i : std::view::iota{1, 5})
    std::cout << i << ' ';          // 1 2 3 4 5

  std::cout << '\n';

  for (int i : std::view::iota(1) | std::view::take(5))
    std::cout << i << ' ';           // 1 2 3 4 5
}
```

Bedarfsauswertung

```
#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 Fünf

- **Coroutinen**
  - Asynchrone Programmierung
- **Contracts**
  - Design by Contract
- **Module**
  - Software-komponenten
- **Concepts**
  - Semantische Kategorien
- **Ranges Bibliothek**
  - Bedarfsaus-wertung und Funktions-komposition

Blogs

[www.grimm-jaud.de [De]](http://www.grimm-jaud.de)

[www.ModernesCpp.com [En]](http://www.ModernesCpp.com)

Rainer Grimm

Training, Coaching und Technologieberatung

www.ModernesCpp.de