

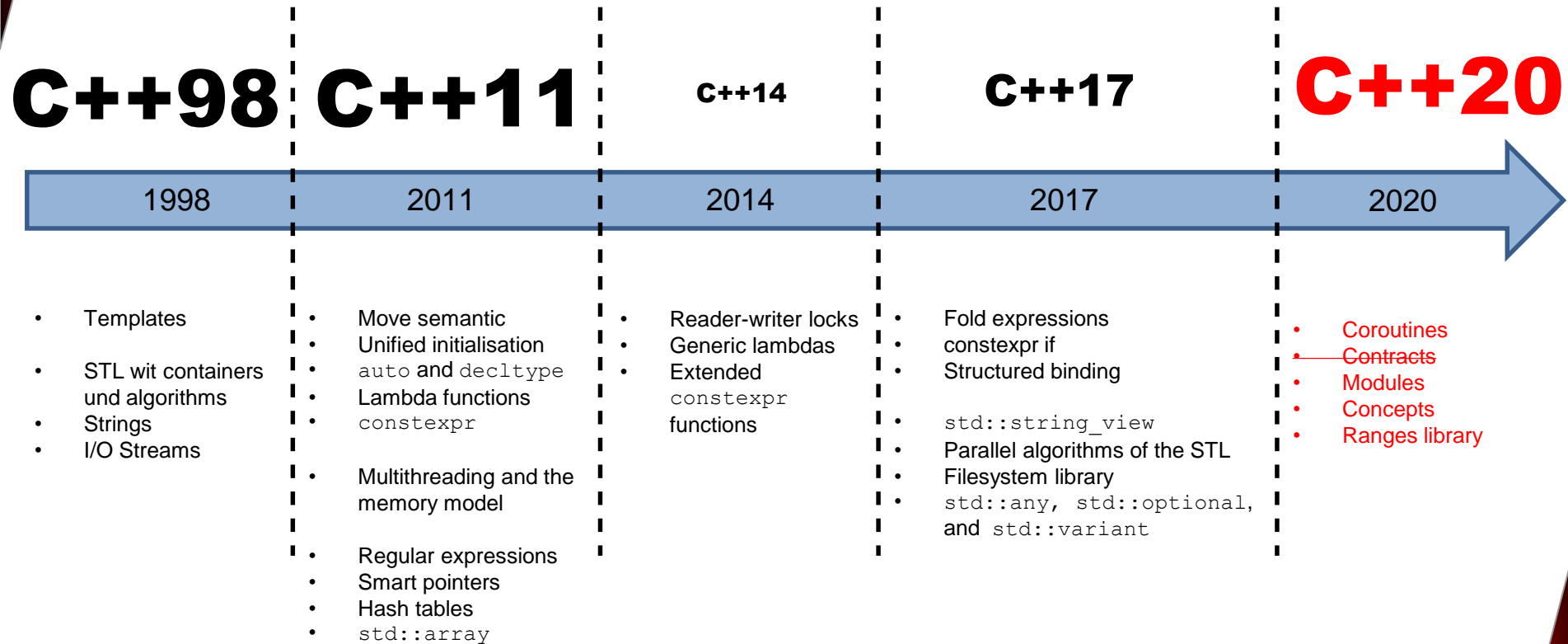
C++20

Rainer Grimm

Training, Coaching, and
Technology Consulting

www.ModernesCpp.de

History of C++



The Big Four

Coroutines

Contracts

Moduls

Concepts

Ranges Library

Coroutines

Coroutines are generalised functions that can be suspended and resumed while keeping their state.

- Typical use-case
 - Cooperative Tasks (protection from data races)
 - Event loops
 - Infinite data streams
 - Pipelines

Coroutines

Design Principles

- **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.

Coroutines

	Function	Coroutine
invoke	<code>func(args)</code>	<code>func(args)</code>
return	<code>return statement</code>	<code>co_return someValue</code>
suspend		<code>co_await someAwaitable</code> <code>co_yield someValue</code>
resume		<code>coroutine_handle<>::resume()</code>

A function is a coroutine if it has a `co_return`, `co_await`, `co_yield` call or if it has a range-based for loop with a `co_await` call.

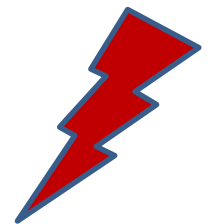
Coroutines

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

```
int main(){  
    auto gen = genForNumbers(-10);  
    for (int i = 1; i <= 20; ++i) std::cout << gen.get() << " ";  
  
    auto gen2 = genForNumbers(0, 5);  
    for (int i = 1; ; ++i) std::cout << gen2.get() << " ";  
}
```

➡ **-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



Coroutines

Blocking

```
Acceptor accept{443};

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

Waiting

```
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);
}
```


Couroutines - Waiting

```
Event event1{};
auto senderThread1 = std::thread([&event1]{ event1.notify(); }); // Notification first
auto receiverThread1 = std::thread(receiver, std::ref(event1));
```

```
receiverThread1.join(), senderThread1.join();
```

```
Event event2{};
auto receiverThread2 = std::thread(receiver, std::ref(event2));
auto senderThread2 = std::thread([&event2]{
    std::this_thread::sleep_for(2s);
    event2.notify(); // Notification after 2 seconds
});
```

```
receiverThread2.join(), senderThread2.join();
```

The Big Four

Coroutines

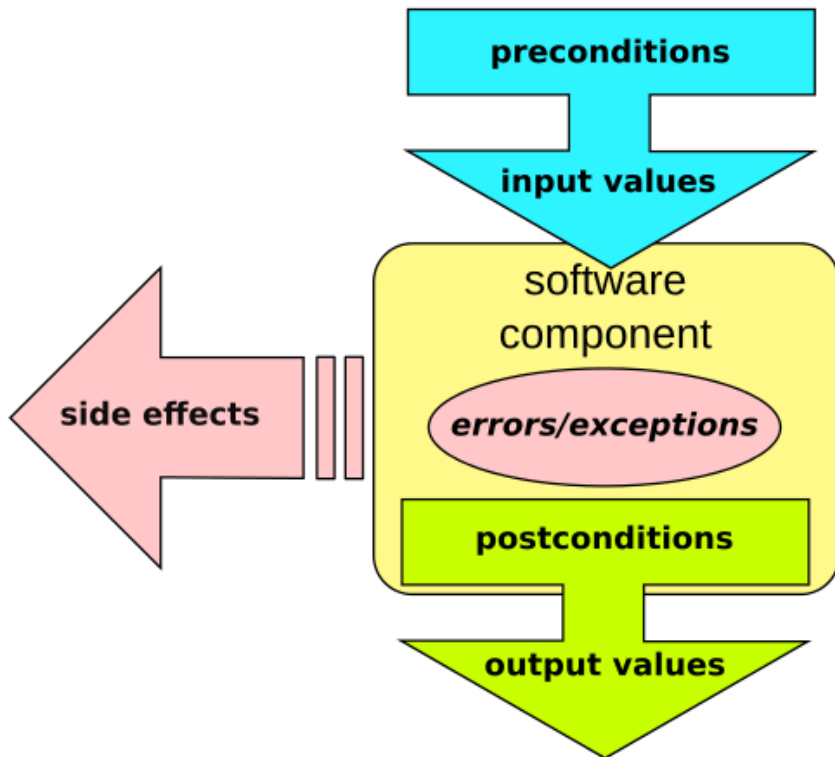
Contracts

Moduls

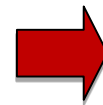
Concepts

Ranges Library

Contracts



A contract is a verifiable interface for a software component. It consists of preconditions, postconditions, and invariants.



Design By Contract

The Big Four

Coroutines

Contracts

Moduls

Concepts

Ranges Library

Modules

```
// math.cppm
```

```
// Moduls declaration  
export Moduls math;
```

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

```
// main.cpp
```

```
// imported Moduls  
import math;
```

```
int main(){  
    add(2000, 20);  
}
```

Advantages of Modules

- Faster compile times
 - A module is only imported once
- Isolation of macros
 - Name collisions and dependencies to macro disappear
- Express the logical structure of the code
 - Explicit exporting of names
 - Packing of Modules
- Header files superfluous
 - No separations of interface and implementation files
 - Amount of source files may be reduced
- Get rid of an ugly workaround
 - No include guards with `LONG_UPPERCASE_NAMES` are necessary any more

Module Interface Unit

```
// math1.cppm  
  
export module math1;  
  
export int add(int fir, int sec);
```

The Module Interface Unit

- Contains the exported names: `export module math1.`
 - Can only export names
 - Names that are not exported are not visible outside the module
-
- A module can only have one Module Interface Unit.

Module Implementation Unit

```
// math1.cpp  
  
module math1;  
  
int add(int fir, int sec) {  
    return fir + sec;  
}
```

The Module Implementation Unit

- contains the non-exported module definition: `module math1;`
- A module can have more than one Module Implementation Unit.

Predefined Modules

- **std.regex:** `<regex>`
- **std.filesystem:** `<experimental/filesystem>`
- **std.memory:** `<memory>`
- **std.threading:** `<atomic>`,
`<condition_variable>`, `<future>`, `<mutex>`,
`<shared_mutex>`, `<thread>`
- **std.core:** Remaining parts of the STL

The Big Four

Coroutines

Contracts

Moduls

Concepts

Ranges Library

The Advantages of Concepts

- Express the template parameter requirements as part of the interface
- Support the overloading of functions and the specialisation of class templates
- Produce drastically improved error messages by comparing the requirements of the template parameter with the template arguments
- Use them as placeholders for generic programming
- Empower you to define your concepts
- Can be used for all kinds of templates

Functions

Using of the concept `Sortable`.

implicit

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

explicit

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

■ Usage:

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

■ Sortable

- has to be a constant expression and predicate

Classes

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

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

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

 A reference is not an object.

Methods

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

- The type parameter T must be copyable.
- The concepts has to be placed before the method declaration.

More Requirements

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

- `find` requires that the elements of the container must
 - build a sequence
 - be equality comparable


Overloading

```
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` puts its iterator `n` positions further
- depending on the iterator, another function template is used

```
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
```


Specialisation

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

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

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

`MyVector<int&>` goes to the unconstrained template parameter.

`MyVector<int>` goes to the constrained template parameter.

Placeholder Syntax: auto

Detour: Asymmetry 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;  
}
```

 Generic lambdas introduced a new way to define templates.

Placeholder Syntax: `auto`

C++20 unifies this asymmetry.

- `auto`: Unconstrained placeholder
- `Concept`: Constrained placeholder

➔ Usage of a placeholder generates a function template.

Constrained and Unconstrained

Constrained concepts can be used where `auto` is usable.

```
int main(){

#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;
}

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;

}
```

Syntactic Sugar

Classical

```
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 gcd1(b, a % b); }
}
```

Abbreviated Function Templates

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

```
auto gcd3(auto a, auto b){
    if( b == 0 ) return a;
    else return gcd3(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 << std::endl;  
  
}
```

Compiled with GCC 6.3 and the
Flag `-fconcepts`

A terminal window screenshot showing the execution of a C++ program. The terminal title is 'rainer@suse:~> conceptsIntegralVariations'. The output of the program is displayed on a yellow background: 'gcd(100, 10)= 10', 'gcd1(100, 10)= 10', 'gcd2(100, 10)= 10', and 'gcd3(100, 10)= 10'. The prompt 'rainer@suse:~>' is shown again at the bottom. The terminal window has a menu bar with 'Datei', 'Bearbeiten', 'Ansicht', 'Lesezeichen', 'Einstellungen', and 'Hilfe'. The status bar at the bottom shows 'rainer : bash'.

Syntactic Sugar: Overloading

```
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(20201);

}
```



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

Predefined Concepts (Ranges)

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

Concept Definition: Variable Concept

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

- T fulfils the variable concept if `std::integral<T>::value` evaluates to true

Concept Definition: Function Concept

Concepts TS

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

Draft C++20 standard

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

- T fulfils the function concept if `==` and `!=` are overloaded and return a boolean

The 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

The Typeclass Eq

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

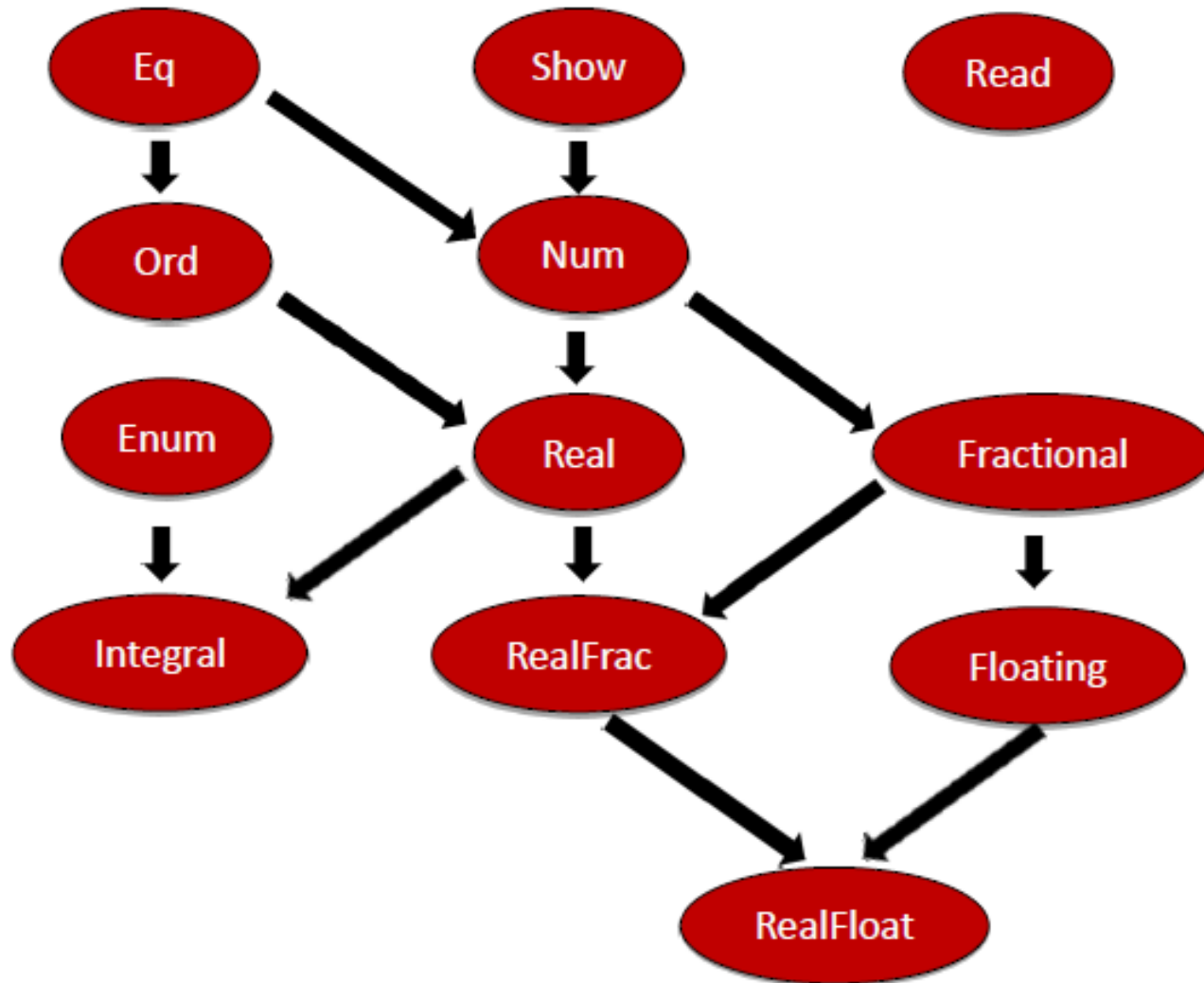
The Concept Equal

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

The typeclass `Eq` (Haskell) and the concept `Equal` (C++) require for the concrete types

- they have to support equal and the unequal operations
- the operations have to return a boolean
- both types have to be the same

Haskells Typeclasses



Haskell's Typeclass `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
```

➔ Each type supporting `Ord` must support `Eq`.

The Concept Ord

The concept Equal

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

The 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

The Big Four

Coroutines

Contracts

Moduls

Concepts

Ranges Library

Die Ranges Library

The ranges library support algorithms which operate

- direct on the container
- can be evaluated lazily
- can be composed

 The ranges library extend C++20 with functional pattern.

Function Composition

```
#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
}
```

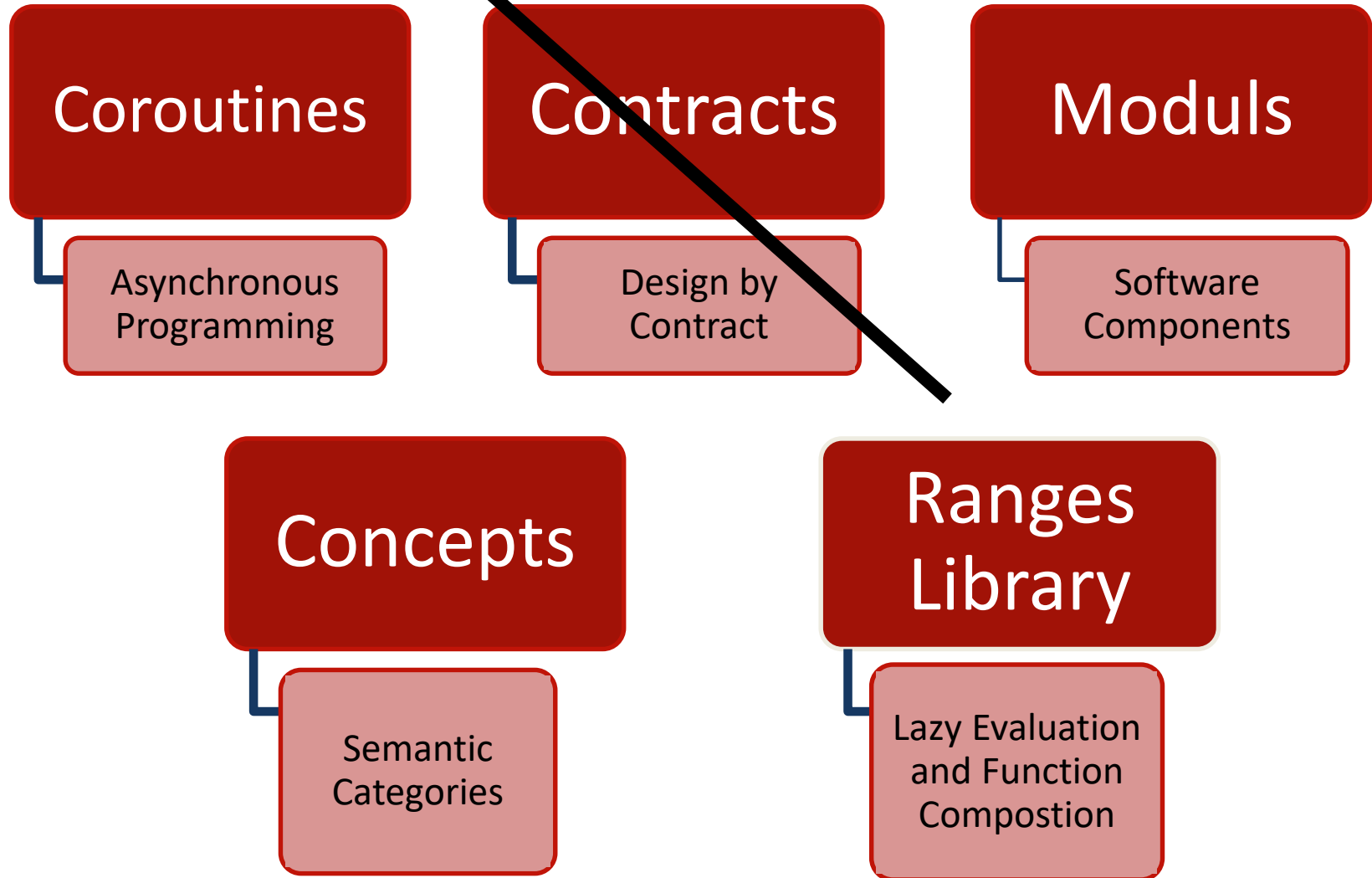
Lazy Evaluation

```
#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
    }
}
```

The Big Four



Blogs

www.grimm-jaud.de [De]

www.ModernesCpp.com [En]

Rainer Grimm

Training, Coaching, and
Technology Consulting

www.ModernesCpp.de