

Concepts

Evolution or Revolution?

Rainer Grimm

Training, Coaching, and
Technology Consulting

www.ModernesCpp.de

Concepts

Motivation

The long, long History

Functions and Classes

Placeholder Syntax

Syntactic Sugar

Define your Concepts

πάντα ρεῖ



Concepts

Motivation

The long, long History

Functions and Classes

Placeholder Syntax

Syntactic Sugar

Define your Concepts

Two Extremes



Too Specific

- Concrete functions

➔ Type conversions

- Narrowing conversion
- Numeric promotion

```
#include <iostream>

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

int main() {

    double d{1.234};
    needInt(d);

    bool b{true};
    needInt(true);

}
```

Two Extremes



Too Generic

- Generic functions

➡ Ugly compile-time errors

```
#include <iostream>

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

int main() {

    std::cout << gcd(100, 10);
    std::cout << gcd(3.5, 4.0);

}
```

Concepts to the Rescue

- Expresses template parameter requirements as part of the interface
- Supports overloading of functions and specialisation of class templates
- Produces drastically improved error messages
- Useable as placeholders for generic programming
- Empowers definition of concepts
- Applicable for class templates, function templates, and non-template members of class templates

Concepts

Motivation

The long, long History

Functions and Classes

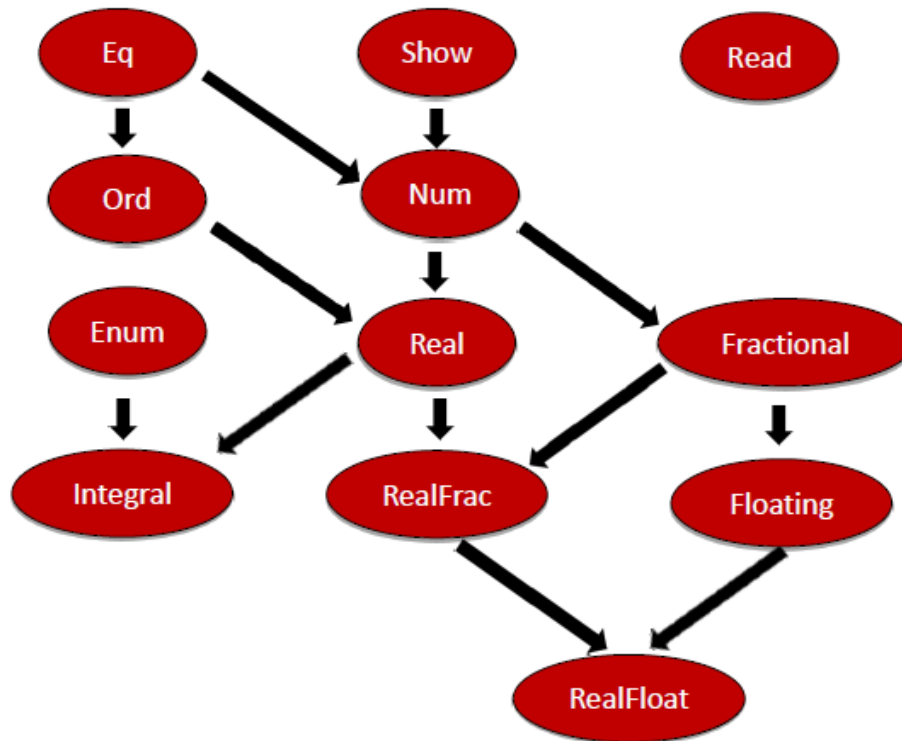
Placeholder Syntax

Syntactic Sugar

Define your Concepts

My First Impression

- Concepts are similar to Haskell's typeclasses.
- Typeclasses are interfaces for similar types.



The Long Way

- 2009: removed from the C++11 standard
"The C++0x concept design evolved into a monster of complexity."
(Bjarne Stroustrup)
- 2017: "Concept Lite" removed from the C++17 standard
- 2020: part of the C++20 standard

Concepts

Motivation

The long, long History

Functions and Classes

Placeholder Syntax

Syntactic Sugar

Define your Concepts

Functions

Using of the concept `Sortable`.

- **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 parameters**

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

Functions

- Usage:

```
std::list<int> lst = {1998, 2014, 2003, 2011};  
std::sort(lst);
```

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

- Sortable

- Has to be a constant expression and a 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.

Member-Functions

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

 The type parameter T must be copyable.

Variadic Templates

```
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 not(... || args); }
```

```
std::cout << all(true); // true  
std::cout << all(5, true, 5.5, false); // false
```

 The type parameters `Args` must be `Arithmetic`.

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.

Concepts

Motivation

The long, long History

Functions and Classes

Placeholder Syntax

Syntactic Sugar

Define your Concepts

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.

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.

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

template<typename T>
concept Integral =
    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;

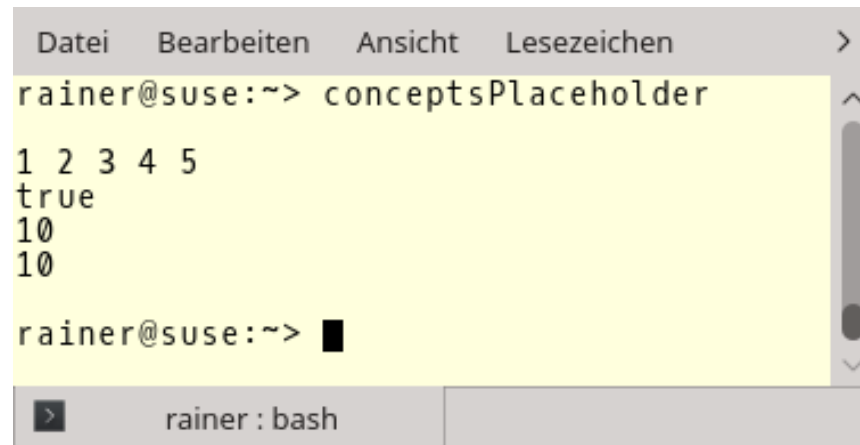
    Integral auto integ = getIntegral(10);
    std::cout << integ << std::endl;

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

}
```

Constrained and Unconstrained

Constraint and unconstrained placeholder behave as expected.



```

Datei  Bearbeiten  Ansicht  Lesezeichen  >
rainer@suse:~> conceptPlaceholder
1 2 3 4 5
true
10
10
rainer@suse:~> █
rainer : bash
```

The image shows a terminal window with a menu bar at the top containing 'Datei', 'Bearbeiten', 'Ansicht', 'Lesezeichen', and a right-pointing arrow. The main area has a yellow background and contains the following text: 'rainer@suse:~> conceptPlaceholder', '1 2 3 4 5', 'true', '10', '10', and 'rainer@suse:~> █'. A vertical scrollbar is on the right side. At the bottom, there is a status bar with a left-pointing arrow, a small square icon, and the text 'rainer : bash'.

Syntactic Sugar

Motivation

The long, long History

Functions and Classes

Placeholder Syntax

Syntactic Sugar

Define your Concepts

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 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);
}
```

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;  
  
}
```



```
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
```

Small Detour

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

gcd2's type parameters

- have to be Integral.
- ~~must have the same type.~~

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

gcd3's type parameter

- can have different types.

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:~> █
rainer : bash
```

Template Introduction

Template introduction is a simplified syntax for declaring templates

- `template <Integral T>`  `Integral{T}`
- Syntax is only available for constrained placeholders (concepts) but not for unconstrained placeholders (`auto`)
 -  Create a constrained placeholder which evaluates to `true`

Template Introduction

Constrained Placeholder

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

```
Integral{T}
class ConstrainedClass{};
```

Unconstrained Placeholder

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

```
auto{T}
class ConstrainedClass{};
```

Error

Concepts

Motivation

The long, long History

Functions and Classes

Placeholder Syntax

Syntactic Sugar

Define your Concepts

Predefined Concepts

- **Language-related**

- `same_as`
- `derived_from`
- `convertible_to`
- `common_reference_with`
- `common_with`
- `assignable_from`
- `swappable`

- **Arithmetic**

- `integral`
- `signed_integral`
- `unsigned_integral`
- `floating_point`

- **Comparison**

- `boolean`
- `equality_comparable`
- `totally_ordered`

- **Lifetime**

- `destructible`
- `constructible_from`
- `default_constructible`
- `move_constructible`
- `copy_constructible`

- **Object**

- `movable`
- `copyable`
- `semiregular`
- `regular`

- **Callable**

- `invocable`
- `regular_invocable`
- `predicate`

Direct Definition

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

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

Requires-Expressions

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

- T fulfils the function concept if == and != are overloaded and return something convertible to 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
```

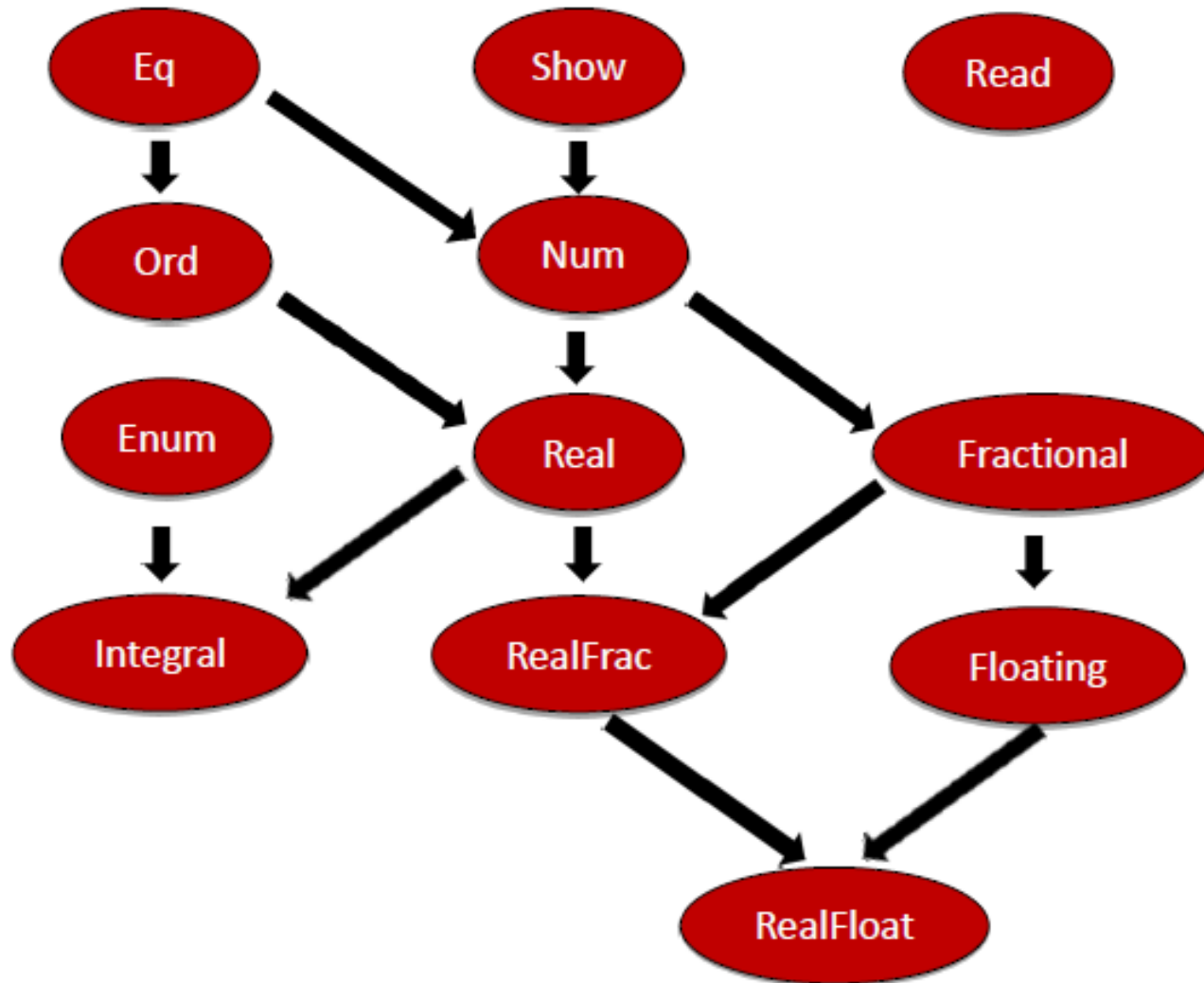
The Concept Equal

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

```
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>;
    };
```

 Each type supporting Ord must support Equal.

The Concept Ord

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

Ord auto 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

SemiRegular and Regular

SemiRegular

- Default constructor: `X ()`
- Copy constructor: `X (const X&)`
- Copy assignment: `X& operator=(const X&)`
- Move constructor: `X (X&&)`
- Move assignment: `X& operator=(X&&)`
- Destructor: `~X ()`

- Swappable: `swap (X&, Y&)`

Regular

- SemiRegular
- Equality comparable

SemiRegular and Regular

```
template<typename T>
struct isSemiRegular:
    std::integral_constant<bool,
        std::is_default_constructible<T>::value &&
        std::is_copy_constructible<T>::value &&
        std::is_copy_assignable<T>::value &&
        std::is_move_constructible<T>::value &&
        std::is_move_assignable<T>::value &&
        std::is_destructible<T>::value &&
        std::is_swappable<T>::value
    >{};
```

```
template<typename T>
concept SemiRegular = isSemiRegular<T>::value;
```

```
template<typename T>
concept Regular = Equal<T> && SemiRegular<T>;
```

semiregular and regular

```
template<class T>
concept movable = is_object_v<T> && move_constructible<T> &&
assignable_from<T&, T> && swappable<T>;
```

```
template<class T>
concept copyable = copy_constructible<T> && movable<T> &&
assignable_from<T&, const T&>;
```

```
template<class T>
concept semiregular = copyable<T> && default_constructible<T>;
```

```
template<class T>
concept regular = semiregular<T> && equality_comparable<T>;
```

Regular and `std::regular`

```
#include <concept>

template <Regular T>
void behavesLikeAnInt(T) {}

template <std::regular T>
void behavesLikeAnInt2(T) {}

struct EqualityComparable { };
bool operator ==
    (EqualityComparable const&,
     EqualityComparable const&) {
    return true;
}

struct NotEqualityComparable { };
```

```
int myInt{};
behavesLikeAnInt(myInt);
behavesLikeAnInt2(myInt);

std::vector<int> myVec{};
behavesLikeAnInt(myVec);
behavesLikeAnInt2(myVec);

EqualityComparable equComp;
behavesLikeAnInt(equComp);
behavesLikeAnInt2(equComp);

NotEqualityComparable notEquComp;
behavesLikeAnInt(notEquComp);
behavesLikeAnInt2(notEquComp);
```

Regular and `std::regular`

- GCC (latest version, <https://godbolt.org/z/XAJ2w3>)
 - The concept `Regular`

```
<source>:24:13: note: the required expression '(a == b)' is invalid
24 |         { a == b } -> std::convertible_to<bool>;
    |         ~~~^~~~
<source>:25:13: note: the required expression '(a != b)' is invalid
25 |         { a != b } -> std::convertible_to<bool>;
    |         ~~~^~~~
```

- The concept `std::regular`

```
/opt/compiler-explorer/gcc-trunk-20200131/include/c++/10.0.0/concepts:290:10: note: the required expression '(__t == __u)' is invalid
290 |     { __t == __u } -> boolean;
    |     ~~~~^~~~~~
/opt/compiler-explorer/gcc-trunk-20200131/include/c++/10.0.0/concepts:291:10: note: the required expression '(__t != __u)' is invalid
291 |     { __t != __u } -> boolean;
    |     ~~~~^~~~~~
/opt/compiler-explorer/gcc-trunk-20200131/include/c++/10.0.0/concepts:292:10: note: the required expression '(__u == __t)' is invalid
292 |     { __u == __t } -> boolean;
    |     ~~~~^~~~~~
/opt/compiler-explorer/gcc-trunk-20200131/include/c++/10.0.0/concepts:293:10: note: the required expression '(__u != __t)' is invalid
293 |     { __u != __t } -> boolean;
    |     ~~~~^~~~~~
```


Regular and `std::regular`

- MSVC (Microsoft Visual C++ 2019, 16.4.3)
 - The concepts `Regular` and `std::regular`

```
x64 Native Tools Command Prompt for VS 2019
regularSemiRegular.cpp
regularSemiRegular.cpp(62): error C2672: 'behavesLikeAnInt': no matching overloaded function found
regularSemiRegular.cpp(62): error C7602: 'behavesLikeAnInt': the associated constraints are not satisfied
regularSemiRegular.cpp(33): note: see declaration of 'behavesLikeAnInt'
regularSemiRegular.cpp(63): error C2672: 'behavesLikeAnInt2': no matching overloaded function found
regularSemiRegular.cpp(63): error C7602: 'behavesLikeAnInt2': the associated constraints are not satisfied
regularSemiRegular.cpp(38): note: see declaration of 'behavesLikeAnInt2'
C:\Users\rainer>
```

Concepts

Motivation

The long, long History

Functions and Classes

Placeholder Syntax

Syntactic Sugar

Define your Concepts

Evolution or Revolution?



Evolution or Revolution?

Evolution

- `auto` as unconstrained placeholders
- Generic lambdas as new way to define templates

```
auto add = [](auto a, auto b) {  
    return a + b;  
}
```

Revolution

- Template requirements are verified by the compiler
- Declaration and definition of templates radically improve
- **Concepts define semantic categories and not syntactic requirements**

Blogs

www.grimm-jaud.de [De]

www.ModernesCpp.com [En]

Rainer Grimm

Training, Coaching, and
Technology Consulting

www.ModernesCpp.de