Generic Programming with Concepts Lite

Andrew Sutton

University of Akron

The University of Akron

Flowgrammable

Driving the Next SDN Generation

Quick recap

Concepts Lite was primarily motivated by the need for better diagnostics

Any language that defers type checking suffers the same problems

Template constraints can be specified using a **`requires`** clause

Template constraints

Declaration with concepts

```
template<typename Seq, typename Fn>
  requires Sequence<Seq>{} && Predicate<Fn>{}
bool all_of(const Seq& seq, Fn fn) {
for(const auto& x : seq);
```

We'll make this even more concise later

What kinds of constraints?

What questions are we actually asking of template arguments?

Is type T the same as type U?

Is type T in some set of types S?

Is type T a case of type U?

Is type T a subtype of T?

Can I use type T in this way?

Constraints and abstraction

Constraints define an abstraction within a generic algorithm or data structure

```
template<typename T>
  requires std::is_integral<T>{}
T gcd(T, T);
```

T is an integral type and has the usual integral operators (**+**, **-**, *****, **/**, **%**)

Separate checking

It might be nice to check the use of abstract types against their constraints

```
template<typename T>
  requires std::is_integral<T>{}
T gcd(T a, T b) {
  a.push_back(b); // error: push_back is not
                  // an admissible operation
}
```

Concepts Lite does not do this; this is why it's "lite"

Separate checking

… But we are experimenting with how to do this in a non-intrusive

What kinds of abstractions?

To what abstractions do constraints correspond?

Is type T the same as type U?

Is type T in some set of types S?

Is type T a case of type U?

Is type T a subtype of T?

Can I use type T in this way?

Same type constraints

"Is type T the same as type U?"

```
std::is_same<T, U>{}
```

Defines an abstraction as being an exact type

Can be useful for constraining sets of generic types

Same type constraints

You could do this:

```
template<typename T>
  requires is_same<T, std::list<int>>{}
void f(T& arg);
```

Same type constraints

You could do this:

```
template<typename T>
  requires is_same<T, std::list<int>>{}
void f(T& arg);
```

But this may be better:

```
void f(std::list<int>& arg);
```

Membership constraints

"Is type T a member of a set of types S?"

Defines an abstraction by enumerating all members of that set

- Presumably, all members share common syntax and semantics

Membership constraints

More type traits! Enumeration of set membership through template specialization

```
template<typename T> // General case
  struct is_integral : std::false_type {};

template<> // int is in the set integral
  struct is_integral<int> : std::true_type {};
template<> // long is in the set integral
  struct is_integral<long> : std::true_type {};
```

Membership constraints

Don't work so well for 3rd party data types

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

Big_int m = 3, n = 5;
gcd(m, n); // error: no matching call
```

Membership constraints

Defines the abstraction as a closed set

Want to add a new abstraction?

- Extend the set by modifying its definition

Specialization constraints

"Is type T a case of type U?"

Defines an abstraction based on the "shape" of the type

- Presumably, all types with the same shape share common syntax and semantics

Specialization constraints

Very, very common in C++, but not easily written using a *requires-clause*

```
template<typename T>
void sort(std::vector<T>& v) {
  sort(v.begin(), v.end());
}
```

Works for any specialization of **`vector`**, right?

Specialization constraints

"Is type T a subtype of type U?"

Defines an abstraction in terms of a related type U

- The "related type" is generally taken to be a base class
- Could also be defined in terms of conversions

Specialization constraints

Shows up in lots of languages: Java, C#, ...

Very limiting

- Imposes "OO" model on everything
- Potential arguments must be in the hierarchy of a base class
- Many interesting types don't belong to any class hierarchy
- Can still be useful in some applications

Subtype constraints

Can do this:

```
template<typename T>
  requires std::is_base_of<ISortable, T>{}
void sort(T& s);
```

Subtype constraints

Can do this:

```
template<typename T>
  requires std::is_base_of<ISortable, T>{}
void sort(T& s);
```

Contrast with:

```
void sort(ISortable*);
```

Ad hoc constraints

"Can I use type T in this way?"

Defines an abstraction in terms its required syntax

- Does type T have this member variable, function, type?
- Can an object of type T be use as in this expression?

Ad hoc constraints

In C++11, how do we ask if a type T can be used in a particular way?

With difficulty

And with type traits

Ad hoc constraints

In C++11, how do we ask if a type T can be used in a particular way?

With difficulty

And with type traits

Ask somebody else, that's the not the point of this talk

Ad hoc constraints

This is much closer to how we actually use templates

```
template<typename T>
  requires has_plus<T>{} // True iff a + b is valid
void add(T a, T b) {
  return a + b;
}
```

Note: not an example of good practice

Ad hoc constraints

Are a fundamental aspect of generic programming

> [Identify] minimal requirements on interfaces, and allow reuse by similar interfaces…

Dehnert, Stepanov

From constraints to concepts

The set of constraints on a generic algorithm is called a *concept*

Syntactic requirements – what operations can be used?

Semantic requirements – what do those operations mean?

Concepts are most useful when they appear in a broad set of related algorithms

And back to generic programming

What are the actual requirements of **`all`**?

```
template<typename Seq, typename Fn>
  requires ____________
bool all(Seq& s, Fn f) {
  for (const auto& x : s)
    if (!f(x)) return false;
  return true;
}
```

Deriving requirements

Algorithm includes this statement

```
for (const auto& x : s)
  ...
```

Requires **`std::begin(s)`**, **`std::end(s)`** are valid and return some value, let's call it **`i`**

Also that **`++i`** and **`const auto& x = *i`** are valid

Deriving requirements

Algorithm also includes

```
if (f(x))
  ...
```

Requires **`f(x)`** must be valid and that the result is convertible to **`bool`**

Required abstractions

We already have names for these things

- **Seq** is a **Range** type – has **begin/end**, return some type, let's call it **Iter**

- **Iter** is an **Input_iterator** type – has pre-increment, allows dereferencing

- **Fn** is a unary **Predicate** on the value type of **Iter** – takes an argument, returns **bool**

Constrained algorithms

Actual constraints for our algorithm

```
template<typename R, typename P>
  requires Range<R>
        && Input_iterator<Iterator<R>>
        && Predicate<P, Value_type<R>>
bool all(const R& range, P pred)
```

Constrained algorithms

Actual constraints for our algorithm

```
template<typename R, typename P>
  requires Range<R>
        && Input_iterator<Iterator<R>>
        && Predicate<P, Value_type<R>>
bool all(const R& range, P pred)
```

Still a bit verbose; we'll improve it later

Defining concepts

How do we define **`Range`**, **`Input_iterator`**, and **`Predicate`**?

Defining concepts

How do we define **Range**, **Input_iterator**, and **Predicate**?

As a type trait?

```
template<typename T>
struct Range
  : std::integral_constant<bool, /* traits */ >
{ };
```

Defining concepts

How do we define **`Range`**, **`Input_iterator`**, and **`Predicate`**?

As a type trait?

```
template<typename T>
struct Range
  : std::integral_constant<bool, /* traits */ >
{ };
```

NO

Supporting concepts

Concepts

In Concepts Lite, a *concept* is a named constraint

- A variable template whose initializer is a constraint
- Or a nullary function template with a single return statement that is a constraint

Some fairly draconian restrictions apply

Anatomy of a concept

A concept declared as a variable template

```
template<typename T>
concept bool Range = requires (T range) {
    typename Iterator_type<T>;
    {std::begin(range)} -> Iterator_type<T>;
    {std::end(range)} -> Iterator_type<T>;
  };
```

Anatomy of a concept


```
template<typename T>
concept bool Range = requires (T range) {
    typename Iterator_type<T>;
    {std::begin(range)} -> Iterator_type<T>;
    {std::end(range)} -> Iterator_type<T>;
  };
```

Anatomy of a concept

Range is a concept

```
template<typename T>
concept bool Range = requires (T range) {
    typename Iterator_type<T>;
    {std::begin(range)} -> Iterator_type<T>;
    {std::end(range)} -> Iterator_type<T>;
  };
```

Anatomy of a concept

Range is a concept

Introduces a sequence of syntactic requirements on **T**

```
template<typename T>
concept bool Range = requires (T range) {
    typename Iterator_type<T>;
    {std::begin(range)} -> Iterator_type<T>;
    {std::end(range)} -> Iterator_type<T>;
  };
```

Anatomy of a concept

Range is a concept

Introduces a sequence of syntactic requirements on **T**

Requirement for an associated type

```
template<typename T>
concept bool Range = requires (T range) {
    typename Iterator_type<T>;
    {std::begin(range)} -> Iterator_type<T>;
    {std::end(range)} -> Iterator_type<T>;
  };
```

Anatomy of a concept

NEW!

Range is a concept

Introduces a sequence of syntactic requirements on **T**

Requirement for an associated type

```
template<typename T>
concept bool Range = requires (T range) {
    typename Iterator_type<T>;
    {std::begin(range)} -> Iterator_type<T>;
    {std::end(range)} -> Iterator_type<T>;
  };
```

Requires a valid expression

Anatomy of a concept

NEW!

Range is a concept

Introduces a sequence of syntactic requirements on **T**

Requirement for an associated type

```
template<typename T>
concept bool Range = requires (T range) {
    typename Iterator_type<T>;
    {std::begin(range)} -> Iterator_type<T>;
    {std::end(range)} -> Iterator_type<T>;
  };
```

Requires a valid expression

Result type is convertible to **`Iterator_type<T>`**

Anatomy of a concept

A concept declared as a function template

```
template<typename T>
concept bool Range() {
  return requires (T range) {
    typename Iterator_type<T>;
    {std::begin(range)} -> Iterator_type<T>;
    {std::end(range)} -> Iterator_type<T>;
  };
}
```

Why two kinds of concepts?

Concepts Lite always had function concepts

Supports overloading on arity, kind of template parameters

Variable concepts came in with variable templates

People didn't want to write `()`s

More concepts

A **`Predicate`** is a callable type that accepts arguments and returns (something convertible to) **`bool`**

```
template<typename P, typename... Args>
concept bool Predicate() {
  return requires (P pred, Args... args) {
    {pred(args...)} -> bool;
  };
}
```

Using constrained algorithms

So what happens now?

```
all(0, true); // error
```

Using constrained algorithms

```
all.cpp:19:11: error: cannot call function ‘bool all
  (const R&, P) [with R = int; P = int]’
   all(0, 2);
           ^
all.cpp:11:6: note:   constraints not satisfied
  [with R = int; P = int]
 bool all(const R& range, P pred) {
      ^
```

Diagnostics with concepts

Enumerate reasons why the constraints fail

```
all.cpp:11:6: note:
  concept ‘Range<int>’ not satisfied because
all.cpp:11:6: note:
  requiring syntax with values (int range)
all.cpp:11:6: note:
  ‘std::end(range)’ is not a valid expression
all.cpp:11:6: note:
  ‘std::begin(range)’ is not a valid expression
all.cpp:11:6: note:
  ‘origin::Iterator_type<R>’ does not name a type
```

Diagnostics with concepts

Enumerate reasons why the constraints fail

```
all.cpp:11:6: note:
  concept ‘Range<int>’ not satisfied because
all.cpp:11:6: note:
  requiring syntax with values (int range)
all.cpp:11:6: note:
  ‘std::end(range)’ is not a valid expression
all.cpp:11:6: note:
  ‘std::begin(range)’ is not a valid expression
all.cpp:11:6: note:
  ‘origin::Iterator_type<R>’ does not name a type
```

Generic programming simplified

Constrained algorithms

Our current algorithm:

```
template<typename R, typename P>
  requires Range<R>
        && Input_iterator<Iterator<R>>
        && Predicate<P, Value_type<R>>
bool all(const R& range, P pred)
```

As promised, we're going to make it simpler

Template parameters

The concept keyword lets us simplify our constraint notation

```
template<Sortable_container C>
bool all(C& c);
```

Concept names can be used to declare template parameters

Template parameters

The concept keyword lets us simplify our constraint notation

```
template<Sortable_container C>
bool all(C& c);
```

Template parameters

The concept keyword lets us simplify our constraint notation

```
template<typename C>
  requires Sortable_container<C>
bool all(C& c);
```

Shorthand declaration is equivalent to this one

Constrained algorithms

Our current algorithm:

```
template<typename R, typename P>
  requires Range<R>
        && Input_iterator<Iterator<R>>
        && Predicate<P, Value_type<R>>
bool all(const R& range, P pred);
```

Constrained algorithms

Our current algorithm:

```
template<Range R, typename P>
  requires Input_iterator<Iterator<R>>
        && Predicate<P, Value_type<R>>
bool all(const R& range, P pred);
```

Constrained algorithms

Our current algorithm:

```
template<Range R, Predicate P>
  requires Input_iterator<Iterator<R>>
bool all(const R& range, P pred);
```

Constrained algorithms

Our current algorithm:

```
template<Range R, Predicate<Value_type<R>> P>
  requires Input_iterator<Iterator<R>>
bool all(const R& range, P pred)
```

Building abstraction

Defining new concepts when requirements frequently occur is a good idea

```
template<typename R>
concept bool In_range =
  Range<R> && Input_iterator<Iterator_type<R>>;
```

Constrained algorithms

Our current algorithm:

```
template<In_Range R, Predicate<Value_type<R>> P>
bool all(const R& range, P pred);
```

Families of algorithms

Many declarations have the same template parameters and constraints

```
Query{R, P} bool all(const R&, P);
Query{R, P} bool some(const R&, P);
Query{R, P} bool none(const R&, P);
```

A concept introduction declares a template

Family concepts

Are just concepts:

```
template<typename R, typename P>
concept bool Query() {
  return In_range<R>()
         && Predicate<P, Value_type<R>>();
}
```

Concept introductions

This declaration:

```
Query{R, P} bool all(const R&, P);
```

Is the same as this:

```
template<typename R, typename P>
  requires Query<R, P>
bool all(const R&, P);
```

Type specifiers

Still too verbose? Concept names can be used as type-specifiers in parameter declarations

```
void sort(Sortable_container& c);
```

Type specifiers

Still too verbose? Concept names can be used as type-specifiers in parameter declarations

```
template<Sortable_container C>
void sort(C& c);
```

Type specifiers

Still too verbose? Concept names can be used as type-specifiers in parameter declarations

```
template<typename C>
  requires Sortable_container<C>
void sort(C& c);
```

Type specifiers

Still too verbose? Concept names can be used as type-specifiers in parameter declarations

```
void sort(Sortable_container& c);
```

Type specifiers

Just like auto, concept names can be used in different contexts

```
void f(Some_concept);

void f(vector<Some_concept>&);

void f(Some_concept (*)(Other_concept));
```

Type specifiers

Just like auto, concept names can be used in different contexts

```
template<Some_concept T>
void f(T);
```

```
void f(vector<Some_concept>&);
```

```
void f(Some_concept (*)(Other_concept));
```

Type specifiers

Just like auto, concept names can be used in different contexts

```
template<Some_concept T>
void f(T);

template<Some_concept T>
void f(vector<T>&);


void f(Some_concept (*)(Other_concept));
```

Type specifiers

Just like auto, concept names can be used in different contexts

```
template<Some_concept T>
void f(T);

template<Some_concept T>
void f(vector<T>&);

template<Some_concept T, Other_concept U>
void f(T (*)(U));
```

Effective shorthand

Shorthand notations are not a replacement for existing template syntax

- The goal is to make simple things simple

Many templates are not simple

- Using shorthand can result in uglier code

Effective shorthand

This is not an effective use of terse notation

```
bool all(const In_range& r,
         Predicate<decltype(r)> p);
```

Same concept? Same type

All concept names used as a type specifier name the same type

```
int distance(Input_iterator i, Input_iterator j)
```

Here, **`i`** and **`j`** have the same type

Because a type name doesn't change types mid-declaration

Meaning of names

The keyword **`auto`** is magic, it is allowed to have whatever meaning we assign it

When a user-defined name is used as a type we have an expectation that it names a single type

Open abstractions

Composing concepts

Concepts Lite supports two mechanism for extending definitions

- Refinement
- Generalization

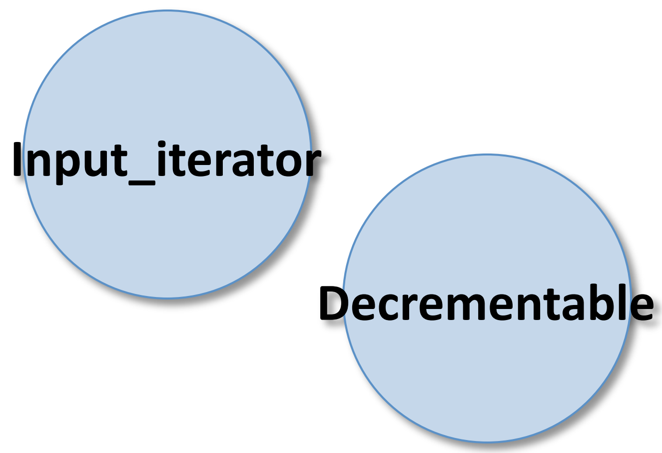
Refinement

Concept refinement strengthens a concept by adding constraints using logical conjunction

```
template<typename T>
concept bool Bidirectional_iterator =
  Input_iterator<T> && Decrementable<T>;
```

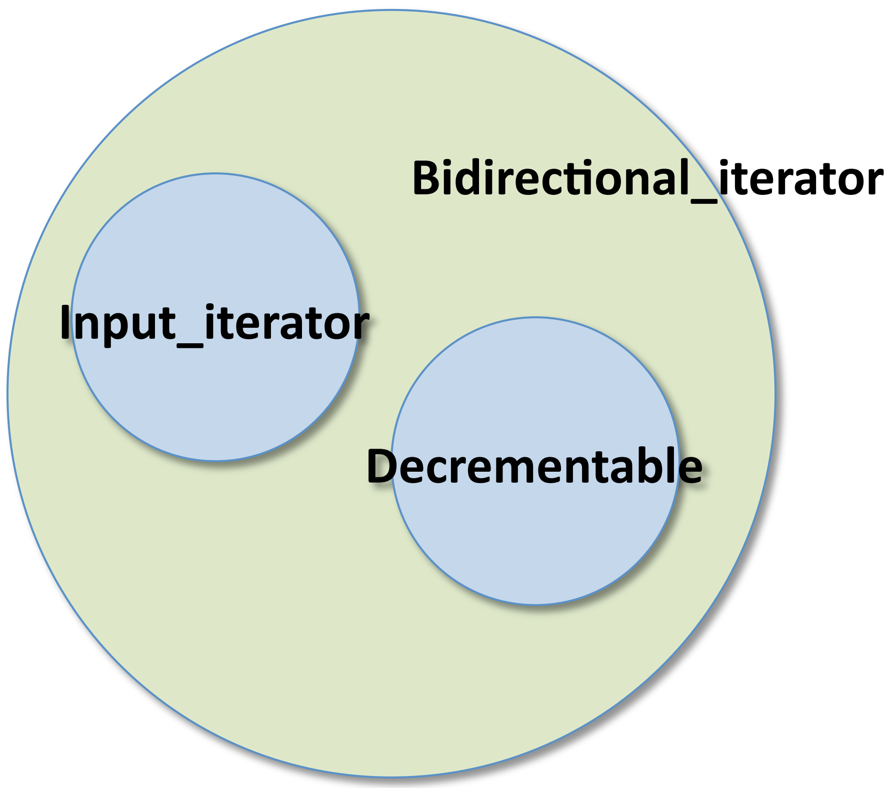
Concepts as sets

Concepts can be viewed as sets of individual (or atomic) constraints



Concepts as sets

Refinement is analogous to the union of sets



Refinement

Happens frequently… is by far the predominant method for extending definitions

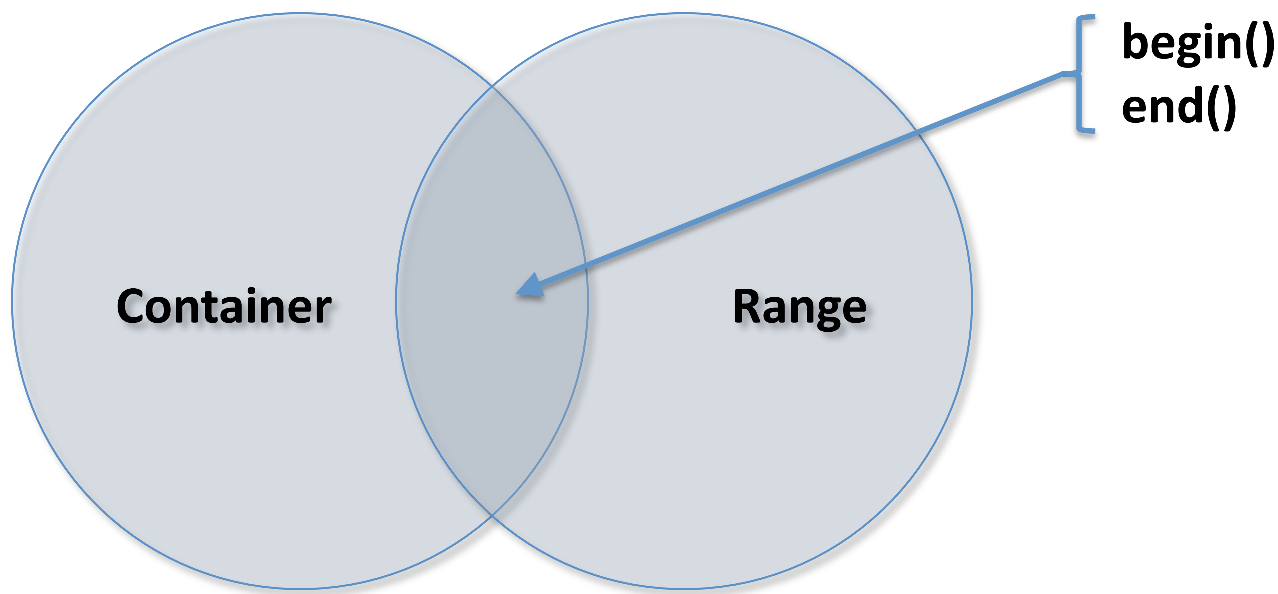
Generalization

Concept generalization weakens a concept by removing constraints using logical disjunction

```
template<typename T>
concept bool Iterable =
  Container<T> || Range<T>;
```

Concepts as sets

Generalization can be seen as the intersection of sets



Concept generalization

Not used very commonly (yet)

- Useful as a means of bridging domains without creating a name (might resist naming)

Beware: heavy use of disjunction in constraints can result in longer compile times

Why does this matter?

Constraints can be partially ordered by their "strength"

- If we can determine that a constraint C logically implies another constraint D, then C is stronger than D

Lets us to extend overloading, partial specialization include the ordering of constraints

Overloading

Canonical example of **advance**

```
void advance(Input_iterator&, int);
void advance(Bidirectional_iterator&, int);
void advance(Random_access_iterator&, int);
```

Each definition provides weaker preconditions or stronger guarantees

Overloading

Compiler selects the best viable candidate

- Most specialized by type
- Most constrained by requirements

```
std::list<int> lst { 0 , 1, 2 };
auto iter = lst.begin();
advance(iter); // Calls bidirectional overload
```

Partial Specialization

Also extended to support constraints

```
template<typename T>
class Complex; // Undefined primary template

template<Real T>
class Complex<T> { ... }; // Complex number

template<Integer T>
class Complex<T> { ... }; // Gaussian integer
```

Understanding specialization

Effective use of this feature requires understanding the interplay between types and constraints

Concepts help flesh out a hierarchy of specializations

- Already exists in C++, we're making it better

Type specialization hierarchy

More general

Any type

Specialized type

Concrete type

More specific

Type specialization hierarchy

More general

`typename T, auto`

Specialized type

Concrete type

More specific

Type specialization hierarchy

More general

`typename T, auto`

`T*, vector<T>`

Concrete type

More specific

Type specialization hierarchy

More general

```
typename T, auto
```

```
T*, vector<T>
```

```
int*, vector<int>
```

More specific

Type specialization hierarchy

More general

Any type

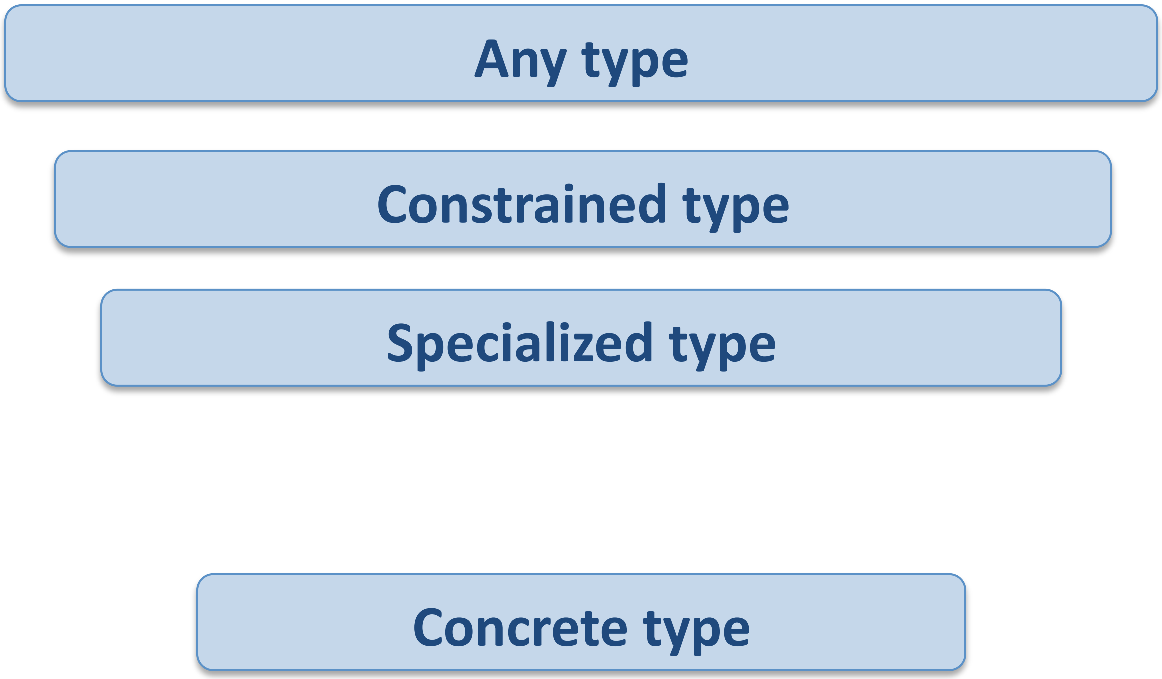
Specialized type

Concrete type

More specific

Type specialization hierarchy

More general



More specific

Type specialization hierarchy

More general

Any type

Constrained type

Specialized type

Constrained specialized type

Concrete type

More specific

Type specialization hierarchy

More general

`typename T, auto`

Constrained type

Specialized type

Constrained specialized type

Concrete type

More specific

Type specialization hierarchy

More general

typename T, auto

Con T, Con, requires Con<T>

Specialized type

Constrained specialized type

Concrete type

More specific

Type specialization hierarchy

More general

```
typename T, auto
```

```
Con T, Con, requires Con<T>
```

```
T*, vector<T>
```

Constrained specialized type

Concrete type

More specific

Type specialization hierarchy

More general

typename T, auto

Con T, Con, requires Con<T>

T*, vector<T>

Con*, vector<Con>

Concrete type

More specific

Type specialization hierarchy

More general

```
typename T, auto
Con T, Con, requires Con<T>
T*, vector<T>
Con*, vector<Con>
int*, vector<int>
```

More specific

Defining Value Type

Can use this hierarchy to help define type functions

```
template<typename T>
using Value_type = value_type_trait<T>::type;
```

Defining Value Type

```
template<typename T> // Any type
struct value_type_trait;

template<Member_value_type T> // Constrained type
struct value_type_trait<T> {
 using type = T;
};

template<typename T>
struct value_type_trait<T*> { // Specialized type
  using type = T;
};
```

Summary

Concepts Lite: set out to solve a simple problem, end up with so much more

Improved diagnostics

Principled open extension mechanism

Simplified template declarations

Still no runtime overhead for templates

Improved compiled times*

Missing material

What didn't I include in this talk?

- What makes a good concept?
- Building generic libraries
- Programming gotchas
- New and clever programming techniques
- And so much more…

Maybe next year

Questions?