A C++ Concepts Primer:
defining and applying constraints

Erik Sven Vasconcelos Jansson
<erik.s.v.jansson@tum.de>
at Technical University of Munich

July 10, 2018

Presentation Outline

1. Generic programming in C++
   - unconstrained templates.
2. Problems and some solutions
   - read the documentation,
   - type traits plus SFINAE,
   - ... arcane "magic" code.
3. How **Concepts Lite** improve
   - ~~un~~constrained templates.
4. *Applying concept constraints*
   - using `requires` clause,
   - overload with constraint,
   - operations on constraint.
5. *Defining list of constraints*
   - `requires` expressions,
     - simple,
     - type,
     - compound,
     - nested.
   - requirement evaluation,
   - naming with `concept`,
   - defining good concepts.
6. Standard Library Concepts
7. Terse syntaxes for C++20?
8. Summary, post-Rapperswil

Generic Programming

C++ is a rich multi-paradigm language, supporting both *run time* and *compile-time polymorphism*. At compile-time, *templates* give support for *generic programming*. However, templates are fragile, unlike their run time counterpart, because they are *unconstrained*. This leads to *bad error messages*, *unclear interfaces*, and violence.

Let's build up an example of the current state of generics in C++, and see where they *succeed* & *fail*; and where *concepts* may help!

Consider this function below, what does it do? How do you know?
Note: this is not a very good implementation, e.g. `p > q` is bad.

Listing 1: a "mysterious" function; can you figure out what this code is?

```
1 double f(const double* p,
2           const double* const q) {
3     double x {  };
4     const double s = q - p;
5     while (p != q)
6         x += *p++;
7     return x / s;
8 }
```

We can make “similarly behaving things”, have the same syntaxes.

Listing 2: boilerplate for the next example; a very incomplete point class.

```
 1 struct point2 {
 2     double x, y;
 3     point2& operator+=(const point2& p);
 4 };
 5
 6 point2& point2::operator+=(const point2& p) {
 7     x += p.x; y += p.y;
 8     return *this;
 9 }
10
11 point2 operator/(const point2& p, double s) {
12     return { p.x / s, p.y / s };
13 }
```

Mysterious Functions?

Generic Programming

Consider this function below, what does it do? How do you know?
Note: this is not a very good implementation, e.g. `p > q` is bad.

Listing 3: another mysterious, yet strangely familiar function (déjà vu?).

```
1 point2 f(const point2* p,
2           const point2* const q) {
3     point2 x {  };
4     const double s = q - p;
5     while (p != q)
6         x += *p++;
7     return x / s;
8 }
```

Obviously, both functions are finding the mean/average somehow. Since both have the same syntax (thanks to operator overloading) we can "lift" the implementation. Now, what is *required* from **T**?

Listing 4: natural generalization of the function from the previous slides.

```
1 template<typename T>
2 T mean(const T* begin,
3         const T* const end) {
4     T sum {  };
5     const double size = end - begin;
6     while (begin != end)
7         sum += *begin++;
8     return sum / size;
9 }
```

Comically Bad Errors
Generic Programming

Because *requirements* of unconstrained templates aren't explicit, a user which hasn't understood the interface may get horrible errors because the syntax is checked **after** *template instantiation*. Which might be deeply nested. Can't we check **before** instantiating this?

Listing 5: classic code examples that give "bad" template error messages.

```
1 std::list l { 5, 1, 4, 3, 2 };
2 std::sort(l.begin(), l.end());
3 // ~48 lines of errors in gcc.
4
5 struct Widget {  };
6
7 std::set<Widget> w;
8 w.insert(Widget{});
9 // ~412 lines here.
```

Here we have *constrained* the template parameter list, allowing the compilers to check requirements **before** instantiating the template.

Listing 6: constraining the function template using a `requires` clause.

```
 1 template<typename T> requires DefaultConstructible<T>
 2                              && SummableWith<T,T> &&
 3                              ScalableWith<T, double>
 4 T mean(const T* begin,
 5          const T* const end) {
 6      T sum {  };
 7      const double size = end - begin;
 8      while (begin != end)
 9          sum += *begin++;
10      return sum / size;
11 }
```

Problems and Solutions

Before considering concepts, let's look at how we currently solve the problems we've discussed, and where these fall short. As you will shortly see, specifying constraints by concept is vastly better.

- **Just read the documentation:** good luck with that, even when people read it, they might implement it incorrectly :(
- **Type traits and SFINAE:** powerful, and works in many of the cases we use concepts. **Not** easy to specify constraints.
- **Tag dispatching plus libraries:** hacky, not discussed here.

Read the Specifications

Problems and Solutions

Expression	Return Value is	Requirements Specification
`x == y`	`bool` convertible	`==` is an equivalence relation, that is, satisfies the following: → for all `x`, `x == x` is satis., → if `x == y`, then `y == x`, → if `x == y`, and `y == z`, then `x == z`, follows too.

Table 1: `EqualityComparable` requirements from the C++ standard.

Listing 7: expressing `EqualityComparable` as a SFINAE type trait.

```
1 template<typename T, typename U, typename = void>
2 struct is_equality_comparable : std::false_type {  };
3
4 template<typename T, typename U>
5 struct is_equality_comparable<T, U,
6     typename std::enable_if<true,
7     decltype(std::declval<T&>() == std::declval<U&>()
8             ,(void)0)>::type> : std::true_type {  };
```

Listing 8: `EqualityComparable` concept which "satisfies"* Table 1.

```
1 template<typename T, typename U>
2 concept EqualityComparable = requires(T x, U y) {
3     { x == y } -> bool;
4     { x != y } -> bool;
5     { y != x } -> bool;
6     { y == x } -> bool;
7 };
```

*not really; see the Ranges TS, this is `WeaklyEqualityComparable :)`

Overloading by SFINAE?

Problems and Solutions

Listing 9: overloading the constructor by using SFINAE & type traits...

```
 1 struct Factory {
 2     enum { INTEGRAL, FLOATING } m_type;
 3
 4     template<typename T,
 5              typename = std::enable_if<
 6              std::is_integral_v<T>>>
 7     Factory(T) : m_type { INTEGRAL } {}
 8     template<typename T,
 9              typename = std::enable_if<
10              std::is_floating_point_v<T>>>
11     Factory(T) : m_type { FLOATING } {}
12 };
```

Listing 10: ...doesn't work if we don't use a `dummy` for disambiguation.

```
 1 struct Factory {
 2     enum { INTEGRAL, FLOATING } m_type;
 3     template<int> struct dummy { dummy(int) {  } };
 4     template<typename T,
 5              typename = std::enable_if<
 6              std::is_integral_v<T>>
 7     Factory(T, dummy<0>=0) : m_type { INTEGRAL } {}
 8     template<typename T,
 9              typename = std::enable_if<
10              std::is_floating_point_v<T>>
11     Factory(T, dummy<1>=0) : m_type { FLOATING } {}
12 };
```

Listing 11: overloading based on constraint with the `requires` clause.

```
1 struct Factory {
2     enum { INTEGRAL, FLOATING } m_type;
3     template<typename T> requires Integral<T>
4     Factory(T) : m_type { INTEGRAL } {}
5     template<typename T> requires Floating<T>
6     Factory(T) : m_type { FLOATING } {}
7 };
```

Concepts Lite

An extension to C++ templates, allowing compile-time checking of template parameters by *constraining* them via *syntax requirements*.

- **Applying constraints:** by using the `requires` *clauses*. ← **!**
- **Defining requirements:** with an `requires` *expression*. ← **!**

Constraints can be applied more intuitively with a *terse syntax* but since it's still controversial, we'll present a subset: **Concepts Zero**.

- **History:** the first concepts proposal came out in 2003, it was merged & un-merged out of C++11 and postponed in C++17.

Applying Constraints

Constraining Template Parameters
Applying Constraints

Both *type & non-type template parameters* may be *constrained* by using a *requires clause*. The *constraint expressions* on its right can be anything that evaluates to `bool` at compile-time. *Instantiation* occurs **only** when the entire constraint expression evaluates `true`!

Listing 12: constraining types & values by using the `requires` clause.

```
1 template<class T> requires Add<T>
2 T add(T x, T y) { return x + y; }
3
4 template<typename T> requires Number<T> class Matrix;
5
6
7
8 template<auto N> requires Even<N>
9 int square_even() { return N*N; }
```

Constraining Template Parameters
Applying Constraints

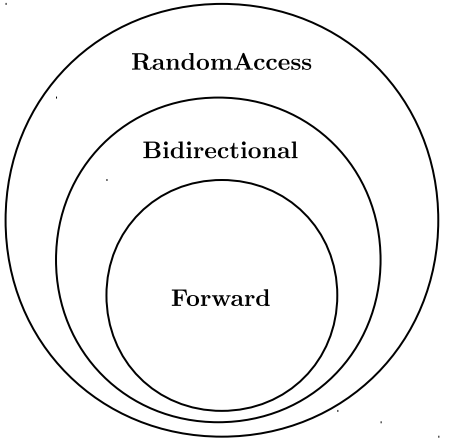
Both *type & non-type template parameters* may be *constrained* by using a *requires clause*. The *constraint expressions* on its right can be anything that evaluates to `bool` at compile-time. *Instantiation* occurs **only** when the entire constraint expression evaluates `true`!

Listing 13: which even supports constraining C++20 generic lambdas!?

```
1 template<class T> requires Add<T>
2 T add(T x, T y) { return x + y; }
3
4 template<typename T> requires Number<T> class Matrix;
5 auto add = []<typename T>(T x, T y) requires Add<T> {
6     return x + y;
7 };
8 template<auto N> requires Even<N>
9 int square_even() { return N*N; }
```

Concept Overload Resolution Rule
Applying Constraints

Intuitively, the most *constrained overload* will be chosen (i.e. with the **most** requirements). Since this is now part of "official" *overload resolution*, and isn't a ad-hoc method like SFINAE; it gives an uniform syntax with the rest of the language. The amount of edge-cases, and hacks is reduced.



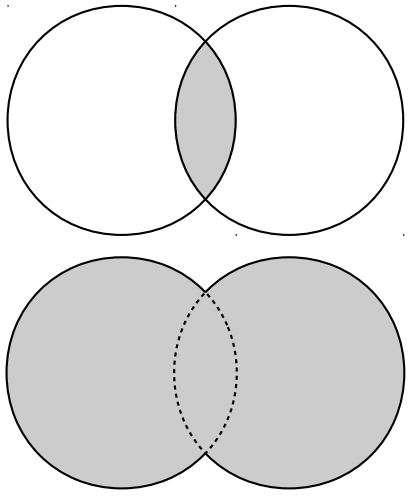
Listing 14: function overloading for `advance` based on type constraint.

```
1 template<typename T> requires ForwardIterator<T>
2 void advance(T& iterator, std::size_t distance);
3
4 template<typename T> requires RandomAccessIterator<T>
5 void advance(T& iterator, std::size_t distance);
```

Logical Operations on Constraints
Applying Constraints

The constraint expression can also be a logical summation of constraints by using `&&` and `||`. They are similar to logical operators, as should be no surprise. These evaluate to `true` when:

- **conjunctions: both** constraints satisfied,
- **disjunctions: at least one** was satisfied.



```
1 template<typename T> requires is_integral_v<T> ||
2                               is_floating_point_v<T>;
3 T add(T x, T y) requires Summable<T> { return x+y; }
4
5 template<auto N> requires Even<N> && Num<decltype(N)>
6 int square_even() { return N*N;  } // value & type N
```

Defining Constraints

A list of syntactic *requirements* for an template parameter can be checked by using an *requires expression*. This expression evaluates to `true` when **all** requirements are **satisfied**. Artificial "variables" can be introduced, which have no linkage, storage or lifetime, and are only there for writing convenience (see e.g. `std::declval`).

- **simple:** just asserts the validity of some expression `<expr>;`,
- **type:** checks the validity of some type by a `typename` prefix,
- **compound:** validates the properties of some given expression,
- **nested:** specifies **more** requirements based on local variables.

Simple Requirements
Requirements

Listing 15: simple requirements in an incomplete `ForwardIterator`.

```
1 template<typename T>
2 concept ForwardIterator = requires {
3     T{};
4     T();
5 };
```

Type Requirements
Requirements

Listing 16: type requirements in our incomplete `ForwardIterator`.

```
1 template<typename T>
2 concept ForwardIterator = requires {
3     typename iterator_traits<T>::value_type;
4     typename iterator_traits<T>::difference_type;
5     typename iterator_traits<T>::reference;
6     typename iterator_traits<T>::pointer;
7     typename iterator_traits<T>::iterator_category;
8 };
```

Compound Requirements

Requirements

Listing 17: compound requirements found in an `ForwardIterator`.

```
1 template<typename T>
2 concept ForwardIterator = requires(T x) {
3     {  *x } -> iterator_traits<T>::reference;
4     { ++x } -> T&;
5     { x++ } -> T;
6 } && requires(T x, T y) {
7     { std::swap(x, y) } noexcept;
8 };
```

Listing 18: usage of nested requirement in an `Allocatable` concept.

```
1 template<typename T>
2 concept Allocatable = requires(T x, std::size_t n) {
3     requires Same<T*, decltype(&x)>;
4     { x.~T() } noexcept;
5     requires Same<T*, decltype(new T)>;
6     requires Same<T*, decltype(new T[n])>;
7     { delete new T[n] };
8     { delete new T };
9 };
```

Naming Constraints
Defining Constraints

One can name *complex constraints* into a *concept* with `concept`.
It's a glorified `constexpr bool` without Turing completeness :)

Listing 19: giving names to constraints by using the `concept` keyword.

```
1  template<typename T>
2  concept ForwardIterator = InputIterator<T> &&
3                            DefaultConstructible<T> &&
4                            EqualityComparable<T, T> &&
5                            WeaklyIncrementable<T> &&
6                            SwappableWith<T, T>;
7  template<typename T>
8  concept BidirectionalIterator = requires(T x) {
9      { --x } -> T&;
10     { x-- } -> T;
11 } && ForwardIterator<T>;
12 template<auto N> concept Even = (N % 2) == 0;
```

Defining "Good" Concepts
Defining Constraints

Some concepts are better than others. Many of the concepts here, are not "good" concepts. The core idea is that concepts should be defined to make types and algorithms 'plug compatible'. Which is:

- to write *algorithms* that can be used for a *variety of types*, and
- to define *types* that can be used with a *variety of algorithms*.

"the ideal is not minimal requirements, but requirements expressed in terms of fundamental and complete concepts." – B. Stroustrup

Standard Library Concepts

Writing many of the "boilerplate" concepts isn't fun, and it's easy to get them wrong. Luckily, C++20 will be receiving a bunch from the *Ranges TS*. Many of these we've defined in this presentation!!

Core	**Comparison**	**Object**	**Callable**
Same	Boolean	Copyable	Invocable
Integral	EqualityComparable	Movable	Predicate
Swappable	StrictTotallyOrdered	Regular	Relation
Constructible	$+ \cdots$ With variants!	Semiregular	WeakOrder

Table 2: excerpt of the concept groups from the `<concepts>` header.

Many of the remaining concepts are found in the *ranges library*, by *Eric Niebler*, under the `<ranges>` header. e.g `InputIterator`. Along with them, we'll get a concepts-ready STL for C++20, which enable cool things like *lazy evaluation* by using *range views/actions*.

Listing 20: example of the composability possibilities of range adaptors.

```
1 std::vector v { 10, 2, 6, 10, 4, 1, 9, 5, 8, 3 };
2 v = std::move(v) | action::sort | action::unique;
3 //  ---> v = { 1, 2, 3, 4, 5, 6, 8, 9, 10 } <---
4 auto range_of_v = v | view::remove_if([](int i) {
5                           return i % 2 == 1; })
6                     | view::transform([](int i) {
7                           return to_string(i); })
8                     | view::take(4);
9 // ---> range_of_v = { "2", "4", "6", "8" } <---
```

Terse Syntaxes

- **Natural syntax** by *Bjarne Stroustrup et al.*, which was part of the Concepts TS, and is implemented in `gcc`'s `-fconcepts`. Issues was related to ambiguous syntax and introducer syntax.
- **Concepts in-place syntax** by *Herb Sutter*. In order to "gain" more consensus, removed ambiguity and dependent binding. It is forward-compatible with Bjarne syntax, but is a bit verbose.
- **Adjective syntax variants** by *Thomas Köppe et al.*, after the Rapperswil *"Bjarne / Herb stand-off"*, a new syntax: YAACD. It's essentially a constrained `auto`, and handles simpler cases.

Natural Syntax
Terse Syntaxes

```
 1 template<typename T> requires Sortable<T>
 2 void sort(T& range);
 3
 4 template<Sortable T>
 5 void sort(T& range);
 6 void sort(Sortable& range);
 7 void sort(RandomAccessIterator begin,
 8           RandomAccessIterator end);
 9 auto sort = []<Sortable T>(T& r) { };
10 auto sort = [](Sortable& r) { };
11
12 BidirectionalIterator it = l.begin();
13
14 Mergeable{I1, I2, O}
15 O merge(I1 f1, I1 l1, I2 f2, I2 l2, O d);
```

Concepts In-Place Syntax

Terse Syntaxes

```
 1 template<typename T> requires Sortable<T>
 2 void sort(T& range);
 3
 4 template<Sortable{T}>
 5 void sort(T& range);
 6 void sort(Sortable{}& range);
 7 void sort(RandomAccessIterator{T} begin,
 8            T end);
 9 auto sort = []<Sortable{T}>(T& r) { };
10 auto sort = [](Sortable{}& r) { };
11
12 BidirectionalIterator{T} it = l.begin();
13
14 template<Mergeable{I1, I2, O}>
15 O merge(I1 f1, I1 l1, I2 f2, I2 l2, O d);
```

Constrained **auto** Syntax

Terse Syntaxes

```
 1 template<typename T> requires Sortable<T>
 2 void sort(T& range);
 3
 4 template<Sortable T>
 5 void sort(T& range);
 6 void sort(Sortable auto& range);
 7 template<RandomAccessIterator T>
 8 void sort(T begin, T end);
 9 auto sort = []<Sortable T>(T& r) { };
10 auto sort = [](Sortable auto& r) { };
11
12 BidirectionalIterator auto it = l.begin();
13 template<class I1, class I2, typename O>
14     requires Mergeable<I1, I2, O>
15 O merge(I1 f1, I1 l1, I2 f2, I2 l2, O d);
```

Concepts Summary
With post-Rapperswil Status!

- Generic programming for C++ uses *unconstrained templates*, which leads to *horrible error messages*, and *fragile interfaces*.
- Existing techniques (like SFINAE) are either *insufficient*, *not easy to define*, or have laughably *obscure edge-cases* when it comes to defining, and using *template parameter constraints*.
- With *concepts* we can *constrain template parameters* but not have to suffer from the problems above, by using its features:
  - `requires` *clauses*, `requires` *expressions* & *terse syntaxes*.
- Generic programming in C++20 is a lot nicer using concepts!
- **Status:** Concepts, SLC/Ranges, Contracts, likely for C++20!

Questions?


WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
from xkcd 208: Randall Munroe
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

References

Bjarne Stroustrup.
Concepts: The Future of Generic Programming.
Technical report, P00557R1, 2017-01-31.
https://wg21.link/p00557r1.

Bjarne Stroustrup.
A Minimal Solution to Concepts Syntax Problems.
Technical report, P1079R0, 2018-05-06.
https://wg21.link/p1079R0.

Herb Sutter.
Concepts In-Place Syntax.
Technical report, P0745R1, 2018-04-29.
https://wg21.link/p0745r1.

Working Draft, C++ Extension for Concepts.
Technical report, N4553, 2015-10-02.
https://wg21.link/n4553.

Wording Paper, C++ Extension for Concepts.
Technical report, P0734R0, 2017-07-14.
https://wg21.link/p0734r0.

Voutilainen, Köppe, Sutton, Sutter, Stroustrup etal.
Yet Another Approach for Constrained Declarations.
Technical report, P1141R0, 2018-06-23.
https://wg21.link/p1141R0.

Acknowledgements

- **Concepts Lite in Practice** by *R. Orr* (2016) for giving a nice and intuitive introduction to Concepts Lite TS at ACCU 2016. Some of the examples are taken from his slides and the article.
- **Generic Programming with Concepts** by *A. Sutton* (2015), for presenting Concepts Lite from another angle. Many of the motivating example are based on those in his presentation too.
- I would like to thank *P. Sommerlad*, for hosting the wonderful meeting in Rapperswil (2018), and allowing me to participate in the discussion on Concepts along with other topics in EWG.
- Finally, I would like to thank *T. Lasser* and the other teachers and participants of "Discovering and Teaching Modern C++"!