A C++ Concepts Primer:
defining and applying constraints

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

July 7, 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

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

```
double f(const double* p,
         const double* const q) {
    double x {  };
    const double s = q - p;
    while (p != q)
        x += *p++;
    return x / s;
}
```

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

```
struct point2 {
    double x, y;
    point2& operator+=(const point2& p);
};

point2& point2::operator+=(const point2& p) {
    x += p.x; y += p.y;
    return *this;
}

point2 operator/(const point2& p, double s) {
    return { p.x / s, p.y / s };
}
```

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

```
point2 f(const point2* p,
         const point2* const q) {
    point2 x {  };
    const double s = q - p;
    while (p != q)
        x += *p++;
    return x / s;
}
```

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

```
template<typename T>
T mean(const T* begin,
       const T* const end) {
    T sum {  };
    const double size = end - begin;
    while (begin != end)
        sum += *begin++;
    return sum / size;
}
```

Parameter Predicates

Generic Programming

Listing 5: constraining the function template using a `requires` clause.

```
template<typename T> requires DefaultConstructible<T>
                              && SummableWith<T,T> &&
                              ScalableWith<T, double>
T mean(const T* begin,
       const T* const end) {
    T sum {  };
    const double size = end - begin;
    while (begin != end)
        sum += *begin++;
    return sum / size;
}
```

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.

Type Trait and SFINAE

Problems and Solutions

Listing 6: expressing `EqualityComparable` as a SFINAE type trait.

```
template<typename T, typename U, typename = void>
struct is_equality_comparable : std::false_type {  };

template<typename T, typename U>
struct is_equality_comparable<T, U,
    typename std::enable_if<true,
    decltype(std::declval<T&>() == std::declval<U&>()
                 ,(void)0)>::type> : std::true_type {  };
```

Listing 7: `EqualityComparable` concept which “satisfies”* Table 1.

```
template<typename T, typename U>
concept EqualityComparable = requires(T x, U y) {
    { x == y } -> bool;
    { x != y } -> bool;
    { y != x } -> bool;
    { y == x } -> bool;
};
```

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

Listing 8: overloading the constructor by using SFINAE & type traits...

```
struct Factory {
    enum { INTEGRAL, FLOATING } m_type;

    template<typename T,
             typename = std::enable_if<
             std::is_integral_v<T>>
    Factory(T) : m_type { INTEGRAL } {}
    template<typename T,
             typename = std::enable_if<
             std::is_floating_point_v<T>>
    Factory(T) : m_type { FLOATING } {}
};
```

Overloading by SFINAE

Problems and Solutions

Listing 9: ...doesn't work if we don't use a `dummy` for disambiguation.

```
struct Factory {
    enum { INTEGRAL, FLOATING } m_type;
    template<int> struct dummy { dummy(int) {  } };
    template<typename T,
             typename = std::enable_if<
             std::is_integral_v<T>>
    Factory(T, dummy<0>=0) : m_type { INTEGRAL } {}
    template<typename T,
             typename = std::enable_if<
             std::is_floating_point_v<T>>
    Factory(T, dummy<1>=0) : m_type { FLOATING } {}
};
```

Listing 10: overloading based on constraint with the `requires` clause.

```
struct Factory {
    enum { INTEGRAL, FLOATING } m_type;
    template<typename T> requires Integral<T>
    Factory(T) : m_type { INTEGRAL } {}
    template<typename T> requires Floating<T>
    Factory(T) : m_type { FLOATING } {}
};
```

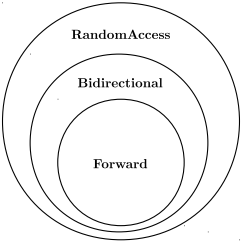
Concepts Lite

Applying Constraints

Listing 11: ...

```
template<class T> requires Add<T>
T add(T x, T y) { return x + y; }

template<auto N> requires Even<N>
int square_even() { return N*N; }
```



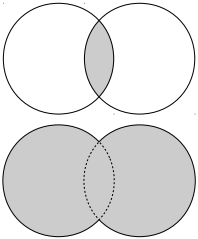
Listing 12: overload resolution for `advance` based on type constraint.

```
template<typename T> requires ForwardIterator<T>
void advance(T& iterator, std::size_t distance);

template<typename T> requires RandomAccessIterator<T>
void advance(T& iterator, std::size_t distance);
```

Logical Operations on Constraints

Applying Constraints



Defining Constraints

Simple Requirements
Requirements

Listing 13: ...

```
template<typename T>
concept ForwardIterator = requires {
    T{};
    T();
};
```

Listing 14: ...

```
template<typename T>
concept ForwardIterator = requires {
    typename iterator_traits<T>::value_type;
    typename iterator_traits<T>::difference_type;
    typename iterator_traits<T>::reference;
    typename iterator_traits<T>::pointer;
    typename iterator_traits<T>::iterator_category;
};
```

Listing 15: ...

```
template<typename T>
concept ForwardIterator = requires(T x, T y) {
    {  *x } -> iterator_traits<T>::reference;
    { ++x } -> T&;
    { x++ } -> T;
    { std::swap(x, y) } noexcept;
};
```

Listing 16: ...

```
template<typename T>
concept ForwardIterator = InputIterator<T> &&
                          DefaultConstructible<T> &&
                          EqualityComparable<T, T> &&
                          WeaklyIncrementable<T> &&
                          SwappableWith<T, T>;
template<typename T>
concept BidirectionalIterator = requires(T x) {
    { --x } -> T&;
    { x-- } -> T;
} && ForwardIterator<T>;
```

Standard Library Concepts

Terse Syntaxes

Natural Syntax
Terse Syntaxes

Concepts Summary

With post-Rapperswil Status!

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++"!