C++ Concepts Primer:

"defining and applying constraints"

Erik Sven Vasconcelos Jansson
<erik.s.v.jansson@tum.de>
at Technische Universität München

July 16, 2018

Contents

Warning: the author is neither an expert or a member of the standardization committee; only a fool would take everything that is written here as the truth. However, the author has tried (within his knowledge) to keep this document as accurate as possible. If you, the reader, find any mistakes in this document, please send an e-mail (or even better, create an issue on GitHub, and fix it!)

1 Introduction

Right now C++ is experiencing a *renaissance*, fueled by new language features and an extended standard library introduced with C++11, C++14, and C++17. It's a different beast than C++98 (want to feel old?: that's two decades ago!), while still keeping to its core philosophies: the *zero-overhead principle*, having a *simple & direct mapping to hardware*, and to be completely *multi-paradigm*. The language has become more powerful, but also simpler, since a lot of the rough edges in C++98 have been removed, or replaced, with modern variants. Interest in C++ has surged again, and with C++20 coming soon™, doubly so.

C++20 is expected to bring features such as *Contracts*, *Coroutines*, *Ranges*, and *Concepts* to the language. Each one of these features deserves their own version of this primer, a self-contained, textbook-style way to introduce people to these features. It should motivate why this feature is needed, what problems it's trying to solve, and present previous alternatives already in the language. After explaining where these existing methods fall short, the feature is shown, demonstrating in which cases it's better than the alternatives. Finally, the featured syntax is presented by using short (to the point!) practical examples.

In this primer we'll be looking at Concepts, a way to *constrain templates*, which leads to: *clearer template errors*, a way to *overload based on constraints*, and more *explicitly defined function template interfaces* by using requirements. The goals of this primer are to teach you how to *apply constraints* to template parameters by using the *requires clause*, and how to *define* your own set of *requirements* by using the various forms of *requires expressions*. You'll then see that we can compose requirements together to form useful *concepts*, that can be used together with *terse syntax*, for writing less verbose generic code.

The primer has the following structure: in Section 2 we present the state of *generic programming* with *unconstrained templates*, and some of its problems. We then show some solutions (e.g. type traits), where they fall short, and where concepts may help. In Section 3 we introduce concepts, and show how to use *requires clauses* and *requires expressions* when constraining templates. The three major terse syntax proposals are then presented in Section 4, and is followed by Section 5 with an overview of the *concepts and ranges library*. Finally, we wrap things up in Section 6, and see what the future might hold.

Building Examples

Alongside this document you'll find plenty of examples on how to use concepts, and you'll be happy to know *almost* all of them compile out-of-the-box when using GCC with the `-fconcepts` flag! If you are a boring person, you can clone the repository, or, if you're feeling adventurous, *unzip* this PDF with:

```
unzip concepts-primer.pdf
cd examples
make gcc-test && make -j8
```

2 Generic Programming

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

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

point2 operator+(const point2&, const point2&);
point2 operator/(const point2&, const double);
point2 operator*(const point2&, const double);
point2 operator*(const double,  const point2&);
```

```
point2 centroid(const point2* begin,
                const point2* const end) {
    point2 sum { };
    const double size = end - begin;
    while (begin != end)
        sum += *begin++;
    return sum / size;
}
```

2.1 Unconstrained Templates

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

```
std::list l { 5, 1, 2, 4, 3 };
std::sort(l.begin(), l.end());
```

```
struct Widget {  };
std::set<Widget> w;
w.insert(Widget{});
```

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

2.2 Type Traits and SFINAE

Expression	Return Type	Requirement Specification
`x == y`	`bool` convertible	`==` is an equivalence relation, that is, it has the following properties: → for all x, `x == x` → if `x == y`, then `y == x` → if `x == y`, `y == z`, then `x == z`

```
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 {  };
```

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

```
struct NumberFactory {
    enum { INTEGRAL, FLOATING } number_type;

    template<typename T,
             typename = std::enable_if<
             std::is_integral_v<T>>>
    NumberFactory(T) : number_type { INTEGRAL } {}

    int create_number() const;

    template<typename T,
             typename = std::enable_if<
             std::is_floating_point_v<T>>>
    NumberFactory(T) : number_type { FLOATING } {}
};
```

```
struct NumberFactory {
    enum { INTEGRAL, FLOATING } number_type;

    template<int> struct dummy { dummy(int) {} };

    template<typename T,
             typename = std::enable_if<
             std::is_integral_v<T>>>
    NumberFactory(T, dummy<0>=0) : number_type { INTEGRAL } {}

    int create_number() const;

    template<typename T,
             typename = std::enable_if<
             std::is_floating_point_v<T>>>
    NumberFactory(T, dummy<1>=0) : number_type { FLOATING } {}
};
```

```
struct NumberFactory {
    enum { INTEGRAL, FLOATING } number_type;

    template<typename T>
        requires std::is_integral_v<T>
    NumberFactory(T) : number_type { INTEGRAL } {}

    int create_number() const;

    template<typename T>
        requires std::is_floating_point_v<T>
    NumberFactory(T) : number_type { FLOATING } {}
};
```

2.3 Tag Dispatching

```
template<typename T, typename U>
void tagged_advance(T& iterator, U distance,
                    std::forward_iterator_tag) {
    while (distance--)
        iterator++;
}

template<typename T, typename U>
void tagged_advance(T& iterator, U distance,
                    std::bidirectional_iterator_tag) {
    std::forward_iterator_tag hack_category;
    tagged_advance(iterator, distance, hack_category);
}

template<typename T, typename U>
void tagged_advance(T& iterator, const U distance,
                    std::random_access_iterator_tag) {
    iterator += distance;
}
```

```
template<typename T, typename U>
void advance(T& iterator, U distance) {
    typename std::iterator_traits<T>::category category;
    tagged_advance(iterator, distance, category);
}
```

```
template<typename T, typename U>
    requires ForwardIterator<T> &&
             Unsigned<U> && Integral<U>
void advance(T& iterator, U distance) {
    while (distance--)
        ++iterator;
}

template<typename T, typename U>
    requires RandomAccessIterator<T> &&
             Unsigned<U> && Integral<U>
void advance(T& iterator, U distance) {
    iterator += distance;
}
```

3 Concepts

3.1 Applying Constraints

3.1.1 Requires Clauses

```
template<template<class> typename C, typename T>
    requires Summable<T>
T sum(const C<T>& container) {
    T total_sum {  };
    for (auto value : container)
        total_sum += value;
    return total_sum;
}
```

```
[]<typename T>(T x, T y) requires EqualityComparable<T> {
    return x == y;
};
```

```
template<typename T, std::size_t R,
                     std::size_t C = R>
    requires Number<T>
class Matrix {
public:
    template<typename U>
        requires ScalableWith<T, U> && Number<U>
    Matrix<T, R, C>& operator*=(U scalar) {
        std::for_each(data.begin(), data.end(),
                      [scalar](T& element) {
            element *= scalar;
        });

        return *this;
    }
};
```

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

```
// Just read The Hitchhiker's Guide!
constexpr bool the_answer(int value) {
    return value == 42;
}

```

```
template<auto N>
    requires the_answer(N)
void check_with_deep_thought() {
    std::cout << "good guess!"
              << std::endl;
}
```

3.1.2 Overload Resolution

```
template<typename T, typename U>
    requires ForwardIterator<T> &&
             Unsigned<U> && Integral<U>
void advance(T& iterator, U distance) {
    while (distance--)
        ++iterator;
}

template<typename T, typename U>
    requires RandomAccessIterator<T> &&
             Unsigned<U> && Integral<U>
void advance(T& iterator, U distance) {
    iterator += distance;
}
```

3.1.3 Logical Operations

Conjunctions

```
template<auto N>
    requires Even<N> &&
             Number<decltype(N)>
int square_only_even_numbers() {
    return N*N;
}
```

Disjunctions

```
template<template<class> typename C, typename T>
    requires Integral<T> || Floating<T>
T sum_numbers(const C<T>& container) requires Summable<T> {
    T total_sum {  };
    for (auto value : container)
        total_sum += value;
    return total_sum;
}
```

3.2 Defining Requirements

3.2.1 Requires Expressions

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

Simple Requirements

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

Type Requirements

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

Compound Requirements

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

Nested Requirements

```cpp
template<typename T>
concept Allocatable = requires(T x, std::size_t n) {
    requires Same<T*, decltype(&x)>;
    { x.~T } noexcept;
    requires Same<T*, decltype(new T)>;
    requires Same<T*, decltype(new T[n])>;
    { delete new T[n] };
    { delete new T };
};
```

3.3 Giving Names to Concepts

```cpp
template<typename T>
concept ForwardIterator = DefaultConstructible<T> &&
                          IteratorTraits<T> &&
                          ReadableIterator<T> &&
                          WeaklyIncrementable<T> &&
                          Swappable<T>;
```

```cpp
template<typename T>
concept BidirectionalIterator = ForwardIterator<T> &&
                                WeaklyDecrementable<T>;
template<typename T>
concept RandomAccessIterator = BidirectionalIterator<T> &&
                               WeaklyRandomAccess<T>;
```

3.3.1 “Good” Concepts

4 Terse Syntax

4.1 Natural Syntax

4.2 Concepts In-Place Syntax

4.3 Constrained `auto` Syntax

5 Standard Library Concepts

5.1 The Ranges Library

```
std::vector v { 10, 2, 6, 10, 4, 1, 9, 5, 8, 3 };
v = std::move(v) | action::sort | action::unique;
```

```
auto r = v | view::remove_if([](int i){ return (i % 2) == 1; })
           | view::transform([](int i){ return to_string(i); })
           | view::take(4);
```

6 Summary

References

[Bja17] Bjarne Stroustrup. *Concepts: The Future of Generic Programming*. *Document P0557 R1*, 31/01/2017. http://wg21.link/p0557r1.

[Cas18] Casey Carter and Eric Niebler. *Standard Library Concepts* (SLC). *Document P0898 R3*, 08/06/2018. http://wg21.link/p0898r3.

[Eri18] Eric Niebler, Casey Carter, C. Di Bella. *The One Ranges Proposal*. *Document P0896 R2*, 25/06/2018. http://wg21.link/p0896r2.

[Kö18] Köppe et al. *Yet Another Approach For Constrained Declarations*. *Document P1141 R0*, 23/06/2018. http://wg21.link/p1141r0.

[Str18] Stroustrup B. *A Minimal Solution to the Concepts Syntax Problems*. *Document P1079 R0*, 06/05/2018. http://wg21.link/p1079r0.

[Sut18] Sutter H. *Concepts In-Place Syntax* (the post-Jacksonville variant!). *Document P0745 R1*, 29/04/2018. http://wg21.link/p0745r1.

[TS15] *Working Draft, C++ Extension for Concepts* (Concepts TS Draft). *Technical Spec. D4553*, 02/10/2015. http://wg21.link/d4553.

[wd17] *Wording Paper, C++ Extension for Concepts* (C++20WD Syntax). *Document P0734 R0*, 14/07/2017. http://wg21.link/p0734r0.

Acknowledgements

- **Roger Orr:** for his awesome "*Concepts Lite in Practice*" talk and for the article from ACCU 2016. Some motivating examples (e.g bad errors, and type traits) in this primer are borrowed (with consent) from there.
- **Andrew Sutton:** for another great talk: "*Generic Programming with Concepts*" at C++Now 2015. Many of the early examples in this primer are based from that talk (with some modifications); consent acquired.
- **Peter Sommerlad:** and the rest of the committee, for allowing me to participate in the discussion on Concepts, Modules and Contracts in the *Rapperswil* 2018 EWG meeting. It was great fun, and it was very interesting to get a perspective on how the commitee operates; while the work was tiring (reading so many papers...), it was very rewarding!
- **Tobias Lasser:** and the rest of the teachers/participants in the course seminar "*Discovering and Teaching Modern C++*", for very interesting talks and for providing a platform where like-minded people can discuss and teach modern C++ to each other. It forced made (being very lazy), to write something that might (?) be useful (I hope?) to others as well.

You may find the short presentation for the seminar above, useful as well.