Intro to C++20's Concepts

Hendrik Niemeyer (he/him)

Link to Slides:

https://github.com/hniemeyer/IntroToConcepts

Feedback and Questions

- Twitter: @hniemeye
- LinkedIn: hniemeyer87
- Xing: Hendrik_Niemeyer
- GitHub: hniemeyer

Compiler Support

- gcc 10
- clang 10 (concepts library and constraint auto not available)
- MSVC 19.23 (19.26 for constraint auto)

Motivation

Norm of a Vector

```
auto norm(const std::vector<double>& values) {
    double result = 0.0;
    for (const auto value : values) {
        result += value*value;
    }
    return std::sqrt(result);
}
```

With Templates

```
template <typename T>
auto norm(const std::vector<T>& values) {
    T result = 0.0;
    for (const auto value : values) {
        result += value*value;
    }
    return std::sqrt(result);
}
```

Constrained by Name

```
template <typename FloatingPoint>
auto norm(const std::vector<FloatingPoint>& values) {
    FloatingPoint result = 0.0;
    for (const auto value : values) {
        result += value*value;
    }
    return std::sqrt(result);
}
```

Error Messages and Templates

```
<source>:16:7: error: no matching function for call to
'std::__cxx11::basic_string<char>::basic_string(double)'
  16 |     T result = 0.0;
<source>:18:24: error: no match for 'operator*' (operand types are 'const
std::__cxx11::basic_string<char>' and 'const
std::__cxx11::basic_string<char>')
  18 |         result += value*value;
     |                   ~~~~~^~~~~~
<source>:20:21: error: no matching function for call to
'sqrt(std::__cxx11::basic_string<char>&)'
  20 |     return std::sqrt(result);
```

What Is a Concept?

- compile-time predicate on template parameters
  - `std::forward_iterator<T>` is true if T is a forward iterator
  - `std::constructible_from<T, Args...>` is true if T can be initialized with the given Args
- used together with constraints

Constraints From C++20 to the Rescue

```
template <typename T>
auto norm(const std::vector<T>& values) requires std::floating_point<T> {
    T result = 0.0;
    for (const auto value : values) {
        result += value*value;
    }
    return std::sqrt(result);
}
```

https://godbolt.org/z/Cov-tp

Error Messages and Templates With Constraints

```
<source>:27:44: error: use of function 'auto norm(const std::vector<T>&)
requires  floating_point<T> [with T = std::__cxx11::basic_string<char>]'
with unsatisfied constraints
  27 |     const auto result2 = norm(my_string_vec );
     |                                            ^
note: the expression 'is_floating_point_v<_Tp> [with _Tp =
std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >]' evaluated to 'false'
 111 |     concept floating_point = is_floating_point_v<_Tp>;
     |                              ^~~~~~~~~~~~~~~~~~~~~~~~
```

```
cc1plus: note: set '-fconcepts-diagnostics-depth=' to at least 2 for more detail
Execution build compiler returned: 1
```

Requires Clause

```
template<typename T>
T f(T t) requires MyConcept<T> {return t;}
```

```
template<typename T> requires MyConcept<T>
T f(T t) { return t;}
```

```
template<typename T> requires MyConcept<T> && MyOtherConcept<T>
T f(T t) { return t;}
```

Requires Clause and NTTP

```
template <double N, double M>
double sum() requires (N > M) {
    return N+M;
}

template <typename T, size_t N>
auto third_element(std::array<T, N> my_array) requires (N > 2) {
    return my_array[2];
}
```

https://gcc.godbolt.org/z/4Ess8c

Concepts From the Standard Library

```
#include <concepts>
#include <iterator>
#include <ranges>
```

Core language concepts

same_as (C++20)	specifies that a type is the same as another type (concept)
derived_from (C++20)	specifies that a type is derived from another type (concept)
convertible_to (C++20)	specifies that a type is implicitly convertible to another type (concept)
common_reference_with (C++20)	specifies that two types share a common reference type (concept)
common_with (C++20)	specifies that two types share a common type (concept)
integral (C++20)	specifies that a type is an integral type (concept)
signed_integral (C++20)	specifies that a type is an integral type that is signed (concept)
unsigned_integral (C++20)	specifies that a type is an integral type that is unsigned (concept)
floating_point (C++20)	specifies that a type is a floating-point type (concept)
assignable_from (C++20)	specifies that a type is assignable from another type (concept)
swappable swappable_with (C++20)	specifies that a type can be swapped or that two types can be swapped with each other (concept)
destructible (C++20)	specifies that an object of the type can be destroyed (concept)
constructible_from (C++20)	specifies that a variable of the type can be constructed from or bound to a set of argument types (concept)
default_initializable (C++20)	specifies that an object of a type can be default constructed (concept)
move_constructible (C++20)	specifies that an object of a type can be move constructed (concept)
copy_constructible (C++20)	specifies that an object of a type can be copy constructed and move constructed (concept)

Comparison concepts

boolean (C++20)	specifies that a type can be used in Boolean contexts (concept)
equality_comparable equality_comparable_with (C++20)	specifies that operator `==` is an equivalence relation (concept)
totally_ordered totally_ordered_with (C++20)	specifies that the comparison operators on the type yield a total order (concept)

Advantages of Constraints

- Bringing compile time type checking to template parameters
- clear interfaces which express intent
- Better error messages
- Selecting template overloads based on the properties of types

Are there any questions?

Back to Our Example

```
template <typename T>
auto norm(const T& values) requires
std::floating_point<typename T::value_type> {
    typename T::value_type result = 0.0;
    for (const auto value : values) {
        result += value*value;
    }
    return std::sqrt(result);
}
```

https://godbolt.org/z/XW_PYy

Even More Constraints

```
template <typename T>
auto norm(const T& values) requires std::floating_point<typename
T::value_type> && std::forward_iterator<typename T::const_iterator>
{
    typename T::value_type result = 0.0;
    for (const auto value : values) {
        result += value*value;
    }
    return std::sqrt(result);
}
```

https://godbolt.org/z/BVRPLs

FloatingPointContainer Concepts

```
template <typename T>
concept IterableWithFloats =
std::floating_point<typename T::value_type> &&
std::forward_iterator<typename T::const_iterator>;
```

FloatingPointContainer Concepts

```
template <typename T>
auto norm(const T& values) requires IterableWithFloats<T>
{
    typename T::value_type result = 0.0;
    for (const auto value : values) {
        result += value*value;
    }
    return std::sqrt(result);
}
```

https://godbolt.org/z/EVUMT6

FloatingPointContainer Concepts

```
template <IterableWithFloats T>
auto norm(const T& values) {
    typename T::value_type result = 0.0;
    for (const auto value : values) {
        result += value*value;
    }
    return std::sqrt(result);
}
```

Are there any questions?

Creating Your Own Concepts

Concepts

template < *template-parameter-list* **>**

concept *concept-name* **=** *constraint-expression***;**

```
//Constraint-expression: other concept plus type trait
template <typename T>
concept MyConcept = OtherConcept<T> || std::is_integral<T>::value
```

Pitfalls

```
template<typename T>
concept Recursion = Recursion<const T>; // Not OK: recursion

template<class T> requires C1<T>
concept C2 = ...; // Not OK: Attempting to constrain a concept definition
```

Requires Expression

requires (*parameter-list*(optional)) { *requirement-seq* }

```
template<typename T>
concept Addable =
requires (T a, T b) {
    a + b; // Meaning: "the expression a+b is a valid
expression that will compile for type T"
};
```

Type Requirements

```
template<typename T> concept HasNestedTypes =
requires {
    typename T::value_type; // Meaning: "Nested type
T::value_type exists"
    typename T::size_type; //Meaning: "Nested type
T::size_type exists"
};
```

Compound Requirements

```
template<typename T> concept AddableLikeFloats =
requires (T a, T b) {
    {a + b} noexcept -> std::convertible_to<float>;
    //Meaning: "a+b is valid, does not throw and the
result is convertible to float"
};
```

Nested Requirements

```
template <typename T>
concept Addable = requires (T a, T b) {
    requires std::convertible_to<float, decltype(a+b)>;
};
```

requires requires

```
template <typename T> requires requires (T a, T b) {a + b;}
auto add(T x, T y) {
    return x+y;
}
```

NTTPs and Concepts

```
template <int N, int M>
concept DiffOfInts = (N - M == 0);

template <int N, int M>
int sum() requires DiffOfInts<N,M> {
    return N+M;
}

int main() {
    std::cout << sum<7,7>();
    return 0;
}
```

Are there any questions?

Example: std::function as a Concept

A Function Which Takes Another Callable

```
template <typename Callable>
int call_twice(Callable callable, int argument) {
    return callable(argument) + callable(argument);
}
```

std::function for Constraining

```
int call_twice(std::function<int(int)> callable, int argument) {
    return callable(argument) + callable(argument);
}
```

https://godbolt.org/z/vTqmxH

Doing the Same With Concepts

```
template<typename Func, typename Arg, typename Ret> concept
FuncOneArg =
requires (Arg a, Func func) {
    {func(a)} -> std::same_as<Ret>;

};

template <typename Callable> requires FuncOneArg<Callable, int, int>
int call_twice(Callable callable, int argument) {
    return callable(argument) + callable(argument);
}
```

https://godbolt.org/z/sDQ3wR

Error Message

```
<source>:10:5: note: constraints not satisfied
<source>: In instantiation of 'int call_twice(Callable, int) [with Callable = main()::<lambda(auto:1)>]':
<source>:17:36:   required from here
<source>:3:61:   required for the satisfaction of 'FuncOneArg<Callable, int, int>' [with Callable = main::._anon_2]
<source>:4:1:   in requirements with 'Arg a', 'Func func' [with Ret = int; Func = main::._anon_2; Arg = int]
<source>:5:9: note: 'func(a)' does not satisfy return-type-requirement, because
    5 |     {func(a)} -> std::same_as<Ret>;
      |      ~~~~^~~
<source>:5:5: error: deduced expression type does not satisfy placeholder constraints
    5 |     {func(a)} -> std::same_as<Ret>;
      |     ~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
:   required for the satisfaction of '__same_as<_Tp, _Up>' [with _Tp = double; _Up = int]
:   required for the satisfaction of 'same_as<double, int>'
: note: the expression 'is_same_v<_Tp, _Up> [with _Tp = double; _Up = int]' evaluated to 'false'
```

Other Implementation

```
template<typename Func, typename Arg, typename Ret> concept
FuncOneArg =
requires (Arg&& a, Func&& func) {
   {std::invoke(std::forward<Func>(func), std::forward<Arg>(a))} ->
std::same_as<Ret>;

};

template <typename Callable> requires FuncOneArg<Callable, int, int>
int call_twice(Callable callable, int argument) {
    return callable(argument) + callable(argument);
}
```

https://godbolt.org/z/gh3zmG

There Is Also Something in std

```
template<typename Func, typename Arg, typename Ret> concept
FuncWithStd =
std::regular_invocable<Func, Arg> &&
std::same_as<std::invoke_result_t<Func, Arg>, Ret>;


template <typename Callable> requires FuncWithStd<Callable, int,
int>
int call_twice(Callable callable, int argument) {
    return callable(argument) + callable(argument);
}
```

https://godbolt.org/z/ZCE2j4

Are there any questions?

More Details

Concepts and auto

```
std::floating_point auto x = 5.0;

std::floating_point auto divide(std::floating_point auto first, std::floating_point auto
second) {
    return first / second;
}

std::floating_point auto my_result = divide(x,y)
```

Overload Resolution

```
template <typename Callable, typename Arg>
void print(Callable func, Arg a) {
  puts("Base case without constraints!");
}

template <typename Callable, typename Arg>
requires std::predicate<Callable, Arg> void print(Callable func, Arg a) {
  puts("This is a predicate!");
}

template <typename Callable, typename Arg>
requires std::invocable<Callable, Arg> void print(Callable func, Arg a) {
  puts("This is a general callable!");
}
```

https://godbolt.org/z/aRLHNF

Overload Resolution

```
template <typename T>
requires std::convertible_to<T, float> void print(T x) {
  puts("This is convertible to float!");
}

template <typename T>
requires std::convertible_to<T, double> void print(T x) {
  puts("This is convertible to double!");
}

int main() {
    print(5.0);
}
```

https://godbolt.org/z/8aWKc9

Are there any questions?

“Concepts” before C++20

- Concepts and constraints do not bring new “functionality” to C++
- The same “functionality” can be achieved with type traits, static asserts/enable_if and SFINAE

Excursion: enable_if

```
template<bool B, class T = void>
struct enable_if {};
```

```
template<class T>
struct enable_if<true, T> { typedef T type; };
```

- If B is true the public member type exists otherwise not
- Can be used to conditionally remove functions from overload resolution
- Put type traits into the boolean argument

“Concepts” before C++20

```
template<typename T,
         typename std::enable_if<std::is_integral_v<T>, int>::type = 0>
T add(T a, T b) {
    return a+b;
}


template<typename T>
T add(T a, T b) {
    static_assert(std::is_integral_v<T>, "Use only with integral
types!");
    return a+b;
}
```

https://godbolt.org/z/AiYRVG

Excursion: void_t

```
template< class... >
using void_t = void;


// primary template handles types that have no nested ::type member:
template< class, class =  void >
struct has_type_member : std::false_type { };

// specialization recognizes types that do have a nested ::type member:
template< class T >
struct has_type_member<T, std::void_t<typename T::type>> : std::true_type { };
```

Defining Addable Without Concepts

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

template <typename T>
struct is_addable<T,std::void_t<decltype(std::declval<T>()+std::declval<T>())>> :
    std::is_convertible<decltype(std::declval<T>()+std::declval<T>()), float>::type {};
```

https://godbolt.org/z/EJEALX

Addable Without Concepts

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

template <typename T>
struct is_addable<T,std::void_t<decltype(std::declval<float&>() =
std::declval<T>()+std::declval<T>())>> :
    std::true_type {};
```

https://godbolt.org/z/t8XxCj

Error Message

```
<source>: In function 'int main()':
<source>:19:23: error: no matching function for call to 'add(X, X)'
   19 |     return add(X{},X{});
      |                       ^
<source>:14:3: note: candidate: 'template<class T, typename std::enable_if<is_addable<T>::value, int>::type <anonymous> > T add(T, T)'
   14 | T add(T a, T b) {
      |   ^~~
<source>:14:3: note:   template argument deduction/substitution failed:
<source>:13:69: error: no type named 'type' in 'struct std::enable_if<false, int>'
   13 |          typename std::enable_if<is_addable<T>::value, int>::type = 0>
      |                                                                     ^
```

Are there any questions?

Concepts as Compile Time Booleans

```
template <typename T>
concept Addable = requires(T a, T b) {
    {a+b} -> std::convertible_to<float>;
};

template <typename T>
void print_something(T a)
{
    if constexpr(Addable<T>) std::cout << "Addable\n";
    else std::cout << " Not Addable\n";
}

int main() {
    constexpr bool floaty_add = Addable<float>;
    return floaty_add;
}
```

https://godbolt.org/z/eaAi3o

Why Use Concepts?

- without concepts requirements are hidden
  - in the body of a function/class
  - in the documentation
  - in complex boilerplate code using enable_if and void_t
- increase readability (clear interfaces) of code at zero costs
- better error messages
- easy syntax
- Selecting template overloads based on the properties of types
- Replacing auto with a concept at a place where a function call occurs increases readability of code
- **My opinion: Use constraints for all template arguments**

When to Write Your Own Concepts?

- If possible use concepts and logical combinations of concepts from the standard library
- avoid writing single property concepts (Addable is a bad concept, Number is a good one)
- Packing unrelated operations and types into a concept is a bad idea

std::semiregular

Defined in header <concepts>

```
template <class T>
concept semiregular = std::copyable<T> && std::default_initializable<T>;
```

(since C++20)

std::regular

Defined in header <concepts>

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

(since C++20)

Conclusion

Angehefteter Tweet

Eric Niebler
@ericniebler

Generic Programming pro tip: Although Concepts are constraints on types, you don't find them by looking at the types in your system. You find them by studying the algorithms.

#cpp

Tweet übersetzen

2:39 vorm. · 29. Apr. 2018 · Twitter Web Client

Feedback and Questions

- Twitter: @hniemeye
- LinkedIn: hniemeyer87
- Xing: Hendrik_Niemeyer
- GitHub: hniemeyer