Intro to C++20's Concepts

Hendrik Niemeyer (@hniemeye)

Link to Slides:

Feedback and Questions

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

Compiler Support

- gcc 10
- clang 10 (concepts library not available)
- MSVC 19.23 (at least partial)

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);
```

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>;
      |                                 ^~~~~~~~~~~~~~~~~~~~~~~~
```

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;}

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

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
- Selecting template overloads based on the properties of types
- Better error messages

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 ContainerWithFloats = 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 ContainerWithFloats<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 <ContainerWithFloats 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);
}
```

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> concept C1 = ...;

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;
}
```

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

Better 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/wkzcSm

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

More Details

Concepts and auto

```
int main() {
    std::floating_point auto my_float = 5.0;

    std::vector<double> myVec({1.0, 2.0, 3.0});
    for (std::floating_point auto v: myVec) std::cout << v << '\n';
}

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 T>
auto add(T a, T b) {
    puts("add_1");
    return a+b;
}

template <typename T> requires Addable<T>
auto add(T a, T b) {
    puts("add_2");
    return a+b;
}
```

https://godbolt.org/z/b65StN

Overload Resolution

```
template <typename T>
auto add(T a, T b) {
    puts("add_1");
    return a+b;
}

template <typename T> requires Addable<T>
auto add(T a, T b) {
    puts("add_2");
    return a+b;
}

template <typename T> requires Addable<T> && Substractable<T>
auto add(T a, T b) {
    puts("add_3");
    return a+b;
}
```

https://godbolt.org/z/WufGQq

Overload Resolution

```
template <typename T>
auto add(T a, T b) {
    puts("add_1");
    return a+b;
}

template <typename T> requires true
auto add(T a, T b) {
    puts("add_2");
    return a+b;
}
```

https://godbolt.org/z/_Wgrbt

Concepts vs. SFINAE

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

Concepts vs. SFINAE

```
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

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 (Better Readable)

```
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

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
- Replacing auto with a concept at a place where a function call occurs gives us clarity about the result
- **My opinion: Use constraints for all template arguments**

When to Write Your Own Concepts?

- If possible use concepts and logical conjunctions 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

Conclusion

- Concepts are more than beautified type_traits
- Concepts are not an expert-only feature
- In an ideal world we would have had them for a long time already
- Concepts eliminate unreadable workarounds and complicated boilerplate code

Feedback and Questions

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

User Group

- Twitter: @osnacpp
- Meetup: Osnacpp