



What Has C++20 Ever Done For Templates?

Hendrik Niemeyer



What Has C++20 Ever Done for Templates

Hendrik Niemeyer (he/him)





Link to Slides:

<https://tinyurl.com/cppsonea2022>



Feedback and Questions

- Twitter: [@hniemeye](#)
- LinkedIn: [hniemeyer87](#)
- GitHub: [hniemeyer](#)



Concepts

NTTP

What Has C++20 Ever Done for Templates?

A Lot



Templated
Lambdas

Relaxed
mandatory
usage of
typename



Concepts



Use Case

- A function or a class template
- but we want to limit what the template parameters can be
- this is already possible in C++17 (using `enable_if`, `static_assert`, type traits, `void_t`)
- but concepts improve readability, error messages and the overall design of type constraints
- not necessarily compile time improvements

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

Concepts From the Standard Library

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

Core language concepts

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

Comparison concepts

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

Constraints

```
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;}
```

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>

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)>;
};
```

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);
}
```

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>

Implementation With Standard Concepts

```
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>

Concepts and Auto

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

Overload Resolution

- compiler starts looking at the most constrained version of the template
- and ends with the least constrained or unconstrained version of the template
- we do not need a negated concept in overload resolution because of this
- a named concept can restrict other named concepts via subsumption
- overload resolution prefers the concept that subsumes another
- subsumption only works with named concepts and boolean combinations of concepts
- concepts of the standard library are carefully designed to use this

Examples for Subsumption

<https://gcc.godbolt.org/z/Too7x3Kb3>

<https://gcc.godbolt.org/z/1hGcWoq7E>

<https://gcc.godbolt.org/z/dxEo3Kbvs>

<https://gcc.godbolt.org/z/hxnjbh4h4>

Overload Resolution With Standard Concepts

```
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 With Standard Concepts

```
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>

Other Places for Concepts

- concepts are booleans known at compile time
- can be stored in variable, used in if constexpr, ...

<https://gcc.godbolt.org/z/8n5ffW3fs>

Summary

- Concepts provide an easy to use functionality to constrain templates
- overload resolution mechanism based on subsumption for fined grained overload control
- concepts from the standard library provide a well designed hierarchy of concepts



Are there any questions?





NTTP



What Can Be Used As NTTP?

- floats and doubles
- classes
 - with constexpr constructor and destructor or only default ctor/dtor (= literal class type)
 - with only public, non-mutable members and base classes (= structural types)
 - all types of members and base classes fulfil these requirements
- lambdas (without captures) because they are structural types

Examples For Structural Types

```
struct A {}; // OK: no user defined constructor
```

```
struct A {  
    constexpr A(){};  
    A(int x){};  
}; // OK or Not OK: a user defined constructor which is not  
constexpr
```

Examples For Structural Types

```
struct A {  
    constexpr A(){};  
}; // OK: only constexpr constructor
```

```
struct A {  
    private:  
    int y; }; // not OK: private member
```

Examples For Structural Types

```
struct A {  
    constexpr A(int i) {};  
    ~A() {};  
}; //not OK: destructor is defined and not constexpr
```

Floats and Doubles as NTTP

- Floating point numbers are considered equivalent as NTTP if their underlying representation is the same
- this can lead to problems if floating point math is involved

<https://gcc.godbolt.org/z/56T45qvh6>

Reminder: auto and NTTP

- Since C++17 we can declare non-type template parameters with auto
- very useful for C++20 if we want to pass lambdas as NTTP

Lambdas as NTTP

<https://gcc.godbolt.org/z/sGE8abGs5>

<https://gcc.godbolt.org/z/j7TqdcWbh>

Application: Strong Types

<https://gcc.gnu.org/z/nxKjs9oT9>

<https://gcc.gnu.org/z/vqaa53P56>

Application: Ratio as NTTP

- `std::ratio` is designed as a type template parameter
- uses template metaprogramming for operations
- with new NTTP rules we can do it differently
- can be useful if you are writing a units library (look at [mp-units](#))

<https://gcc.godbolt.org/z/fjMe9f7PG>



Are there any questions?





Templated Lambdas



Generic Lambdas

```
auto comparator = [](auto x, auto y) {return x<y;};
```

Generic Lambdas With Concepts

```
auto comparator = [](std::integral auto x,  
std::integral auto y) {  
    return x < y;  
};
```

Templated Lambdas

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

```
auto first_elem = []<typename T>(std::vector<T> vec)  
{ return vec[0]; };
```


Templated Lambdas And Concepts

```
auto a = []<typename T>(T a, T b)requires std::floating_point<T>{};
```

```
auto b = []<std::integral T>(T a, T b){};
```



Are there any questions?



Fewer Places Which Require Typename

- fewer place which require `typename X<T> :: name`

The End

- use concepts
- use the new possibilities for NTTP
- use templated lambdas
- dont write typename if you dont have to