C++20 Concepts – Complete Guide

© 2021 RNDr. Šimon Tóth[1]

@HappyCerberus @SimonToth

@Happy.Cerberus @SimonToth

@SimonToth83

One of the well deserved common complaints about C++ is the low quality of compiler errors. Both GCC and Clang made a lot of progress to improve the situation in the past 10 years, but templated code is one area where they can't really help.

This article was originally published at:
`https://itnext.io/c-20-concepts-complete-guide-42c9e009c6bf`

Also available in video form: `https://www.youtube.com/watch?v=1So7onMFxJM`

For all the examples and bash scripts, visit:
`https://github.com/HappyCerberus/article-cpp20-concepts`

A function that doesn't like an int.

The difficulty with templated code is that compilation errors physically occur inside the implementation code. This can result in confounding error messages.

Take this straightforward example:

function_that_doesnt_like_int.cc

```cpp
#include <string>

void function_without_concept(const auto& x) {
    std::string v = x;
}

int main() {
    function_without_concept(1);
}
```

The error is reported inside of the function because the invalid operation here is the assignment of `int` to a `string`.

[1]Redistribution and use of this document without modification is permitted. Extended free licenses for educational purposes are available upon request.

```
function_that_doesnt_like_int.cc:4:14: error: no viable conversion from 'const
↪  int' to 'std::string' (aka 'basic_string<char>')
        std::string v = x;
                    ^   ~
function_that_doesnt_like_int.cc:8:2: note: in instantiation of function template
↪  specialization 'function_without_concept<int>' requested here
        function_without_concept(1);

- shortened -
```

We don't have to dig too deep with a shallow error like this one, but we still need to look at the implementation to fully understand that this function only accepts types that can be assigned to a string.

Using concepts

So let's have a look at how we can improve the situation by using concepts. We will rely on concepts that are defined in the standard library.

For a complete overview of predefined concepts in the standard library, have a look at:

- `<concepts>`
  - https://en.cppreference.com/w/cpp/concepts
- `<iterator>`
  - https://en.cppreference.com/w/cpp/iterator#C.2B.2B20_iterator_concepts
  - https://en.cppreference.com/w/cpp/iterator#Algorithm_concepts_and_utilities
- `<ranges>`
  - https://en.cppreference.com/w/cpp/ranges#Range_concepts

Let's start with our simple function that expects a parameter convertible to a string.

function_with_concept.cc

```cpp
#include <string>
#include <concepts>

void func_with_auto_inline(const std::convertible_to<std::string> auto& x) {
    std::string v = x;
}

int main() {
    func_with_auto_inline(1);
}
```

This time we get a longer but also more descriptive error.

```
function_with_concept.cc:9:5: error: no matching function for call to
↪  'func_with_auto_inline'
    func_with_auto_inline(1);
    ^~~~~~~~~~~~~~~~~~~~~

function_with_concept.cc:4:6: note: candidate template ignored: constraints not
↪  satisfied [with x:auto = int]
void func_with_auto_inline(const std::convertible_to<std::string> auto& x) {
     ^
function_with_concept.cc:4:39: note: because 'std::convertible_to<int,
↪  std::string>' evaluated to false
void func_with_auto_inline(const std::convertible_to<std::string> auto& x) {

- shortened -
```

The most important part is that we are presented with an `no matching function for call` error that explains why overload resolution did not consider our function.

You might notice in the output that `std::convertible_to` actually takes two parameters. With inline syntax, the first parameter is automatically filled in from context. We can see that more clearly when considering all the other syntax variants:

builtin_concepts.cc

```cpp
void func_with_auto_inline(const std::convertible_to<std::string> auto& x) {
    std::string v = x;
}

void func_with_auto_postfix(const auto& x)
    requires std::convertible_to<decltype(x), std::string> {
    std::string v = x;
}

template <std::convertible_to<std::string> T>
void func_with_template_inline(const T& x) {
    std::string v = x;
}

template <typename T>
    requires std::convertible_to<T, std::string>
void func_with_template_postfix(const T& x) {
    std::string v = x;
}
```

We can omit the first parameter when using the inline syntax (either with auto or a template). When using the postfix requires-based syntax, we need to supply it either through `decltype` or by passing in the template parameter.

Concepts can also be combined using logical operators. Here is a convoluted example of a function template that accepts integrals or invocable that returns an integral. Notice that we also use a concept inside of an `constexpr` conditional statement.

combine_concepts.cc

```cpp
#include <string>
#include <concepts>
#include <iostream>

template <typename T>
    requires std::integral<T> ||
    (std::invocable<T> &&
     std::integral<typename std::invoke_result<T>::type>)
void function(const T& x) {
    if constexpr (std::invocable<T>) {
        std::cout << "Result of call is " << x() << "\n";
    } else {
        std::cout << "Value is " << x << "\n";
    }
}

int main() {
    function(1); // OK, integral
    function([]() { return 2; }); // OK, invocable, returns integral
    function(2.0); // Fails
}
```

We are starting to compromise readability with complex formulas like this one, so let's look at how to write our own concepts.

Writing concepts

Writing a new concept follows the following syntax:

```cpp
template <typename T>
concept Name = constraint_expression;
```

Constraint expression can contain `constexpr` boolean expressions, conjunction/disjunction of other concepts and `requires` blocks. So for our previous example, we could construct a boolean expression using C++11 type traits or construct a compound concept from C++20 standard library concepts.

maybe_invocable.cc

```cpp
template <typename T>
concept maybe_invokable_integral_v1 = std::is_integral<T>::value ||
    (std::is_invocable<T>::value &&
     std::is_integral<typename std::invoke_result<T>::type>::value);

template <typename T>
concept maybe_invokable_integral_v2 = std::integral<T> ||
    (std::invocable<T> &&
     std::integral<typename std::invoke_result<T>::type>);
```

Requires expression

Let's dig a bit deeper and go over the different elements used inside an `requires` expression.

The simplest test we can do is to test the validity of an expression. In the following example, we also use the optional parameter list to generate arguments. We can then use these arguments inside the expression. Here we test whether the type supports the binary plus operator.

simple.cc

```
template <typename T>
concept addable = requires (T a, T b) {
    a+b;
};

void function(addable auto x) {}
struct X {};

int main() {
    function(1); // OK
    function(X{}); // Fails
}
```

We can also test for type validity. The referred type can be a template and therefore used to check for substitution failures.

type.cc

```
#include <concepts>

template <typename T>
concept type_test = requires {
    typename T::ElementType; // ElementType member type must exist
};

template <std::integral T>
struct S;

template <typename T>
concept template_test = requires {
    typename S<T>; // checks whether S<T>
                   // is a valid template substitution
};

void function1(type_test auto x) {}
void function2(template_test auto x) {}

struct X { using ElementType = int; };

int main() {
    function1(X{}); // OK
    function1(1); // Fails
    function2(1); // OK
    function2(X{}); // Fails
}
```

Compound expressions allow us to constraint the result of an expression.

compound_return_type.cc

```cpp
#include <concepts>

template <typename T>
concept invoke_integral = requires (T a) {
    { a() } -> std::integral;
};

template <invoke_integral T>
void function(const T& f) {};

int main() {
    function([](){ return 1; }); // OK
    function([](){ return 1.0; }); // Fail: doesn't return integral
    function(1); // Fail: 1() is not a valid expression
}
```

With compound expressions, we can also require the expression to be non-throwing.

compound_noexcept.cc

```cpp
template <typename T>
concept assignment_cant_throw = requires (T a, T b) {
    { a = b } noexcept;
};

struct X {
    X& operator = (const X& lhs) noexcept { return *this; }
};

struct Y {
    Y& operator = (const Y& lhs) { return *this; }
};

template <assignment_cant_throw T> struct Test {};

int main() {
    Test<X> a; // OK
    Test<Y> b; // Fails
}
```

Lastly, we can nest `requires` expressions. This is useful for referring to other concepts or to construct nested `constexpr` boolean expressions.

Note that if you list a concept inside a `requires` block without prefixing it with requires, it will be treated as an expression and only checked for validity (GCC already warns about this).

```cpp
template <typename T>
concept x = requires (T a) {
    requires sizeof(a) >= 4;
    requires std::integral<T>;
    std::integral<T>; // probably not what you meant
};
```

How a specialization is selected

In all the previous examples, we always had a single specialization, which either matched or didn't. Concepts allow us to provide a set of specializations. The compiler will first determine feasible specialisations and then the specialization with the most specific constraint.

To determine which constraint is the most specific, each will be expanded until it is a list of atoms in a conjunction/disjunction formula. Note that a `requires` expression is considered an atom here, no matter how complex, which has implications on organising and writing your concepts if you want to take advantage of this behaviour.

In the following example, while the `has_x` concept may seem to be a subset of the `coord` concept, however, based on the expansion rules, both of these constraints have are a singular atom, and because these atoms are not identical, these two concepts are unordered.

atom.cc

```cpp
template <typename T>
concept has_x = requires (T v) {
    v.x;
};

template <typename T>
concept coord = requires (T v) {
    v.x;
    v.y;
};

void function(has_x auto x) {}
void function(coord auto x) {}

struct X {
    int x;
};

struct Y {
    int x;
    int y;
};

int main() {
    function(X{}); // OK, only one viable candidate
    function(Y{}); // Fails, ambiguous
}
```

However, it is possible to rewrite these concepts so they are comparable and the `coord` concept subsumes the `has_x` concept.

ordered.cc

```cpp
template <typename T>
concept has_x = requires (T v) {
    v.x;
};

template <typename T>
concept coord = has_x<T> && requires (T v) {
    v.y;
};

void function(has_x auto x) {}
void function(coord auto x) {}

struct X {
    int x;
};

struct Y {
    int x;
    int y;
};

int main() {
    function(X{}); // OK, only one viable candidate
    function(Y{}); // OK, coord is more specific
}
```

So if you are designing a library that needs to take advantage of this behaviour, take care when designing your concepts.

A good rule of thumb for two concepts A and B is:

- `A = (X1 ... XN); B = Y && (X1 ... XN);` B is considered to be more specific than A
- `A = (X1 ... XN); B = Y || (X1 ... XN);` A is considered to be more specific than B

Thank you for reading

Thank you for reading this article. Did you enjoy it? If yes, pass it along to coleagues or classmates that might benefit from reading it.