Chapter 3

Concepts, Constraints, and Requirements

This chapter introduces the new C++ feature *concepts* with the keywords `concept` and `requires`. It is used to constrain the availability of generic code to specific requirements.

For sure, *concepts* are a milestone for C++, because concepts provide a language feature for something we need a lot when writing generic code: specifying *requirements*. Although there have been workarounds, we now have an easy and readable way to specify the requirements for generic code, get better diagnostics when requirements are broken, disable generic code when it does not work (although it might compile), and switch between different generic code for different types.

3.1 Motivating Example of Concepts and Requirements

Consider the following function template returning the maximum of two values:

```
template<typename T>
T maxValue(T a, T b) {
  return b < a ? a : b;
}
```

This function template can be called for two arguments that have the same type, provided the operations performed for the parameters (comparing with `operator<` and copying) are valid.

However, passing two pointers would compare their addresses instead of the values they refer to.

3.1.1 Improving the Template Step-by-Step

Using a `requires` Clause

To fix that, we can equip the template with a ***constraint*** so that it is not available if raw pointers are passed:

```
template<typename T>
requires (!std::is_pointer_v<T>)
```

```
T maxValue(T a, T b)
{
  return b < a ? a : b;
}
```

Here, the constraint is formulated in a ***requires clause***, which is introduced with the keyword `requires` (there are other ways to formulate constraints).

To specify the constraint that the template cannot be used for raw pointers we use the standard *type trait* `std::is_pointer_v<>` (which yields the **v**`alue` member of the standard type trait `std::is_pointer<>`).[1] With this constraint we can no longer use the function template for raw pointers:

```
int x = 42;
int y = 77;
std::cout << maxValue(x, y) << '\n';        // OK: maximum value of ints
std::cout << maxValue(&x, &y) << '\n';      // ERROR: constraint not met
```

The requirement is a compile-time check and has no impact on the performance of the compiled code. It only means that the template simply cannot be used for raw pointers. When raw pointers are passed, the compiler behaves as if the template were not there.

Defining and Using a `concept`

It might be pretty likely that we need a constraint for pointers more than once. For this reason, we can introduce a ***concept*** for it:

```
template<typename T>
concept IsPointer = std::is_pointer_v<T>;
```

A *concept* is a template where we introduce a name for one or more requirements to passed template parameters so that we can use these requirements as constraints. After the equal sign (you cannot use braces here) we have to specify the requirements as a Boolean expression that is evaluated at compile-time. In this case, we require that the template argument used to specialize `IsPointer<>` has to be a raw pointer.

We can use this concept to constraint the `maxValue()` template as follows:

```
template<typename T>
requires (!IsPointer<T>)
T maxValue(T a, T b)
{
  return b < a ? a : b;
}
```

[1] Type traits were introduced as standard type functions with C++11 and the ability to call them with a `_v` suffix was introduced with C++17.

Overloading with Concepts

By using constraints and concepts we can even overload the `maxValue()` template to have one implementation for pointers and one for other types:

```
template<typename T>
requires (!IsPointer<T>)
T maxValue(T a, T b)          // maxValue() for non-pointers
{
  return b < a ? a : b;       // compare values
}

template<typename T>
requires IsPointer<T>
auto maxValue(T a, T b)       // maxValue() for pointers
{
  return maxValue(*a, *b);    // compare values the pointers point to
}
```

Note that a `requires` clause that just constrains a template with a concept no longer needs parentheses.

Now, we have two function templates with the same name, but only one of them is available for each type:

```
int x = 42;
int y = 77;
std::cout << maxValue(x, y) << '\n';       // calls maxValue() for non-pointers
std::cout << maxValue(&x, &y) << '\n';     // calls maxValue() for pointers
```

Because the implementation for pointers delegates the computations of the return value to the objects the pointers refer to, the second call uses both `maxValue()` templates. When passing pointers to `int`, we instantiate the template for pointers with `T` being `int*` and the basic `maxValue()` template that is for non-pointers with `T` being `int`.

This even works recursively now. We can ask for the maximum value of a pointer to a pointer to an `int`:

```
int* xp = &x;
int* yp = &y;
std::cout << maxValue(&xp, &yp) << '\n';   // calls maxValue() for int**
```

Overload Resolution with Concepts

Overload resolution considers templates with constraints as more specialized than templates without. Therefore, it is enough to only constrain the implementation for pointers:

```
template<typename T>
T maxValue(T a, T b)          // maxValue() for a value of type T
{
  return b < a ? a : b;       // compare values
}
```

```
template<typename T>
requires IsPointer<T>
auto maxValue(T a, T b)        // maxValue() for pointers (higher priority)
{
  return maxValue(*a, *b);     // compare values the pointers point to
}
```

However, be careful: overloading once using references and once using non-references might cause ambiguities.

By using concepts it is even possible to prefer some constraints over others. However, this requires that concepts that *subsume* other concepts are used.

Type Constraints

If a constraint is a concept applied to a single parameter, there are ways to shortcut the specification of the constraint. First, you can specify it directly as a ***type constraint*** when declaring the template parameter:

```
template<IsPointer T>    // only for pointers
auto maxValue(T a, T b)
{
  return maxValue(*a, *b);     // compare values the pointers point to
}
```

In addition, you can use the concept as a *type constraint* when declaring parameters with `auto`:

```
auto maxValue(IsPointer auto a, IsPointer auto b)
{
  return maxValue(*a, *b);     // compare values the pointers point to
}
```

This also works for parameters passed by reference:

```
auto maxValue3(const IsPointer auto& a, const IsPointer auto& b)
{
  return maxValue(*a, *b);     // compare values the pointers point to
}
```

Note that by constraining both parameters directly, we changed the specification of the template: we no longer require that `a` and `b` have to have the same type. We only require that both are pointer-like objects of arbitrary type.

When using the template syntax, the equivalent code looks as follows:

```
template<IsPointer T1, IsPointer T2>    // only for pointers
auto maxValue(T1 a, T2 b)
{
  return maxValue(*a, *b);     // compare values the pointers point to
}
```

We probably should also allow different types for the basic function template comparing values. One way to do that is to specify two template parameters:

```
template<typename T1, typename T2>
auto maxValue(T1 a, T2 b)       // maxValue() for values
{
  return b < a ? a : b;         // compare values
}
```

The other option would be to also use `auto` parameters:

```
auto maxValue(auto a, auto b)   // maxValue() for values
{
  return b < a ? a : b;         // compare values
}
```

Now, we could pass a pointer to an `int` and a pointer to a `double`.

Trailing `requires` Clauses

Consider the pointer version of `maxValue()`:

```
auto maxValue(IsPointer auto a, IsPointer auto b)
{
  return maxValue(*a, *b);      // compare values the pointers point to
}
```

There is still an implicit requirement that is not obvious: after dereferencing, the values have to be comparable.

Compilers detect that requirement when (recursively) instantiating the `maxValue()` templates. However, the error message might be a problem, because the error occurs late and the requirement is not visible in the declaration of `maxValue()` for pointers.

To let the pointer version directly require in its declaration that the values the pointers point to have to be comparable, we can add another constraint to the function:

```
auto maxValue(IsPointer auto a, IsPointer auto b)
requires std::three_way_comparable_with<decltype(*a), decltype(*b)>
{
  return maxValue(*a, *b);
}
```

Here we use a ***trailing requires clause***, which can be specified ***after*** the parameter list. It has the benefit that it can use the parameters to formulate constraints.

Standard Concepts

The trailing requires clause used here uses a standard concept defined by the C++20 standard library: `three_way_comparable_with`. The standard library has many standard concepts for common requirements. They are provided in namespace `std` (sometimes a subnamespace is used).

`three_way_comparable_with` takes two template parameters to check whether the values of the passed types are comparable with operators `==`, `!=`, `<`, `<=`, `>`, `>=`, and the new operator `<=>` (which gives the concept the name).

`requires` Expressions

So far, the `maxValue()` templates do not work for pointer-like types that are not raw pointers, such as smart pointers. If the code should also compile for those types, we'd better define that pointers are objects, for which we can call `operator*`.

Such a requirement is easy to specify since C++20:

```
template<typename T>
concept IsPointer = requires(T p) { *p; };   // calling operator * has to be valid
```

Instead of using the type trait for raw pointers, the concept formulates a simple requirement: The expression `*p` has to be valid for an object `p` of the passed type `T`.

Here, we are using the `requires` keyword again to introduce a ***requires expression***, which can define one or more ***requirements*** for types and parameters. By declaring a parameter `p` of type `T`, we can simply specify which operations for such an object have to be supported.

We can also require multiple operations, type members, and that expressions yield constrained types. For example:

```
template<typename T>
concept IsPointer = requires(T p) {
                     *p;                              // operator * has to be valid
                     p == nullptr;                    // can compare with nullptr
                     {p < p} -> std::convertible_to<bool>; // operator < yields bool
                    };
```

We specify three requirements, which all apply to a parameter `p` of the type `T` we define this concept for

- The type has to support operator `*`.
- The type has to support operator `<`, which has to yield type usable as Boolean value.
- Objects of that type have to be comparable with `nullptr`.

Note that we do not need two parameters of type `T` to check whether `<` can be called. The runtime value does not matter. However, note that there are some restrictions for how to specify what an expression yields (e.g., you cannot specify just `bool` there).

Also note that we do ***not*** require that `p` *is* `nullptr` here. We require only that we ***can*** compare `p` with `nullptr`. However, that rules out iterators, because in general they cannot be compared with `nullptr` (except when they happen to be implemented as raw pointers as for `std::array<>`s).

Again, this is a compile-time constraint that has no impact on the generated code; we only decide whether and for which types the code compiles. Therefore, it does not matter whether we declare parameter `p` as a value or as a reference.

You could also use the *requires expression* as an ad-hoc constraint directly in the *requires clause* (which looks a bit funny but makes total sense once you understand the difference between a requires clause and a requires expression and that both need the keyword `requires`):

```
template<typename T>
requires requires(T p) { *p; }   // constrain template with ad-hoc requirement
auto maxValue(T a, T b)
{
  return maxValue(*a, *b);
```

```
}
```

3.1.2 A Complete Example with Concepts

We have introduced everything now necessary to look at a complete example program for concepts that computes the maximum value for plain values and pointer-like objects:

lang/maxvalue.cpp

```
#include <iostream>

// concept for pointer-like objects:
template<typename T>
concept IsPointer = requires(T p) {
                      *p;              // operator * has to be valid
                      p == nullptr;    // can compare with nullptr
                      {p < p} -> std::convertible_to<bool>;   // < yields bool
                    };

// maxValue() for plain values:
auto maxValue(auto a, auto b)
{
  return b < a ? a : b;
}

// maxValue() for pointers:
auto maxValue(IsPointer auto a, IsPointer auto b)
requires std::three_way_comparable_with<decltype(*a), decltype(*b)>
{
  return maxValue(*a, *b);   // return maximum value of where the pointers refer to
}

int main()
{
  int x = 42;
  int y = 77;
  std::cout << maxValue(x, y) << '\n';       // maximum value of ints
  std::cout << maxValue(&x, &y) << '\n';     // maximum value of where the pointers point to

  int* xp = &x;
  int* yp = &y;
  std::cout << maxValue(&xp, &yp) << '\n';   // maximum value of pointer to pointer

  double d = 49.9;
  std::cout << maxValue(xp, &d) << '\n';     // maximum value of int and double pointer
```

```
}
```

Note that we cannot use `maxValue()` to check for the maximum of two iterator values:

```
std::vector coll{0, 8, 15, 11, 47};
auto pos = std::find(coll.begin(), coll.end(), 11);    // find specific value
if (pos != coll.end()) {
  // maximum of first and found value:
  auto val = maxValue(coll.begin(), pos);              // ERROR
}
```

The reason is that we require the parameters to be comparable with `nullptr`, which is not supported by iterators. Whether this is what you want, is a design question. However, this example demonstrates that it is important to think carefully about the definition of general concepts.

3.2 Where Constraints and Concepts Can be Used

You can use requires clauses or concepts to constrain almost all forms of generic code:

- Functions templates:

```
template<typename T>
requires  ...
void print(const T&) {
  ...
}
```

- Class templates

```
template<typename T>
requires  ...
class MyType {
  ...
}
```

- This even applies to member functions.
- Alias templates
- Variable templates

For these templates you can constrain both type and value parameters.

However, note that you cannot constrain concepts:

```
template<std::ranges::sized_range T>                    // ERROR
concept IntegralValType = std::integral<std::ranges::range_value_t<T>>;
```

Instead, you have to specify this as follows:

```
template<typename T>
concept IntegralValType = std::ranges::sized_range<T> &&
                          std::integral<std::ranges::range_value_t<T>>;
```

3.2.1 Constraining Alias Templates

Here is an example of constraining alias templates (generic using declarations):

```
template<std::ranges::range T>
using ValueType = std::ranges::range_value_t<T>;
```

The declaration is equivalent to the following:

```
template<typename T>
requires std::ranges::range<T>
using ValueType = std::ranges::range_value_t<T>;
```

Type `ValueType<>` is now only defined for types that are ranges:

```
ValueType<int> vt1;                          // ERROR
ValueType<std::vector<int>> vt2;             // int
ValueType<std::list<double>> vt3;            // double
```

3.2.2 Constraining Variable Templates

Here is an example of constraining variable templates:

```
template<std::ranges::range T>
constexpr bool IntegralValType = std::integral<std::ranges::range_value_t<T>>;
```

Again, this is equivalent to the following:

```
template<typename T>
requires std::ranges::range<T>
constexpr bool IntegralValType = std::integral<std::ranges::range_value_t<T>>;
```

The Boolean variable template is now only defined for ranges:

```
bool b1 = IntegralValType<int>;              // ERROR
bool b2 = IntegralValType<std::vector<int>>; // true
bool b3 = IntegralValType<std::list<double>>; // false
```

3.2.3 Constraining Member Functions

Requires clauses can also be part of the declaration of member functions. That way, programmers can specify different APIs based on requirements and concepts.

Consider the following example:

lang/valorcoll.hpp

```
#include <iostream>
#include <ranges>

template<typename T>
class ValOrColl {
  T value;
 public:
```

```
  ValOrColl(const T& val)
   : value{val} {
  }

  void print() const {
    std::cout << value << '\n';
  }

  void print() const requires std::ranges::range<T> {
    for (const auto& elem : value) {
      std::cout << elem << ' ';
    }
    std::cout << '\n';
  }
};
```

Here we define a class `ValOrColl` that can hold a single value or a collection as `value` of type `T`. Depending on whether type `T` is a collection (checked with the concept `std::ranges::range`) different `print()` member functions are called.

For example, you can use the class as follows:

lang/valorcoll.cpp

```
#include "valorcoll.hpp"
#include <vector>

int main()
{
  ValOrColl o1 = 42;
  o1.print();

  ValOrColl o2 = std::vector{1, 2, 3, 4};
  o2.print();
}
```

The program has the following output:

```
42
1 2 3 4
```

Note that you can only constrain templates this way. You cannot use `requires` to constrain ordinary functions:

```
void foo() requires std::numeric_limits<char>::is_signed    // ERROR
{
  ...
}
```

One example, where this feature is used inside the C++standard library is the conditional availability of `begin()` for `const` views.

3.2.4 Constraining Non-Type Template Parameters

You cannot only constrain types. You can also constrain values that are template parameters (non-type templates parameters, *NTTP*).

For example:

```
template<int Val>
concept LessThan10 = Val < 10;
```

Or more generic:

```
template<auto Val>
concept LessThan10 = Val < 10;
```

This concept can be used as follows:

```
template<typename T, int Size>
requires LessThan10<Size>
class MyType {
  ...
};
```

Later we discuss more examples.

3.3 Typical Applications of Concepts and Constraints in Practice

There are multiple reasons why using requirements as constraints can be useful:

- Constraints help to *understand* the restrictions on templates and to get more understandable error messages when requirements are broken.
- Constraints can be used to *disable generic code* for cases when it does not make sense:
  - For some types, generic code might compile but not do the right thing.
  - We might have to fix overload resolution, which decides which operation to call if there are multiple valid options.
- Constraints can be used to *overload* or *specialize* generic code so that different code is compiled for different types.

Let us have a closer look at these reasons by developing another example step by step. This way we can also introduce a couple of additional details about constraints, requirements, and concepts.

3.3.1 Using Concepts to Understand Code and Error Messages

Assume we want to write generic code that inserts the value of an object into a collection. Thanks to templates we can implement it once as generic code that gets compiled for the types of passed objects once we know them:

```
template<typename Coll, typename T>
```

```
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

This code does not always compile. There is an implicit requirement for the types of the passed arguments: the call to `push_back()` for the value of type `T` has to be supported for the container of type `Coll`.

You could also argue that this is a combination of multiple basic requirements:

- Type `Coll` has to support `push_back()`.
- There has to be a conversion from type `T` to the type of the elements of `Coll`.
- If the passed argument has the elements type of `Coll`, that type has to support copying (because a new element is created initialized with the passed value).

If any of these requirements is broken, the code does not compile. For example:

```
std::vector<int> vec;
add(vec, 42);        // OK
add(vec, "hello");   // ERROR: no conversion from string literal to int

std::set<int> coll;
add(coll, 42);       // ERROR: no push_back() supported by std::set<>

std::vector<std::atomic<int>> aiVec;
std::atomic<int> ai{42};
add(aiVec, ai);      // ERROR: cannot copy/move atomics
```

When compilation fails, error messages can be very clear, such as when on the top level of a template the member `push_back()` is not found:

```
prog.cpp: In instantiation of 'void add(Coll&, const T&)
          [with Coll = std::__debug::set<int>; T = int]':
prog.cpp:17:18:   required from here
prog.cpp:11:8: error: 'class std::set<int>' has no member named 'push_back'
   11 |   coll.push_back(val);
      |   ~~~~~~~~~~~~~~
```

However, generic error messages can also be very tough to read and understand. For example, when the compiler has to deal with the broken requirement that copying is not supported, the problem is detected in the deep darkness of the implementation of `std::vector<>`. We get between 40 and 90 lines of error messages, where you have to carefully look for the broken requirement:

```
...
prog.cpp:11:17:   required from 'void add(Coll&, const T&)
                  [with Coll = std::vector<std::atomic<int> >; T = std::atomic<int>]'
prog.cpp:25:18:   required from here
.../include/bits/stl_construct.h:96:17:
  error: use of deleted function
         'std::atomic<int>::atomic(const std::atomic<int>&)'
   96 |     -> decltype(::new((void*)0) _Tp(std::declval<_Args>()...))
      |                 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
...
```

You might think you improve the situation by defining and using a concept that checks whether you can perform the `push_back()` call:

```
template<typename Coll, typename T>
concept SupportsPushBack = requires(Coll c, T v) {
    c.push_back(v);
};

template<typename Coll, typename T>
requires SupportsPushBack<Coll, T>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

The error message for not finding `push_back()` now might be as follows:

```
prog.cpp:27:4: error: no matching function for call to 'add(std::set<int>&, int)'
   27 | add(coll, 42);
      | ~~~~~~~~~~~~~
prog.cpp:14:6: note: candidate: 'template<class Coll, class T>  requires  …'
   14 | void add(Coll& coll, const T& val)
      |      ^~~
prog.cpp:14:6: note:   template argument deduction/substitution failed:
prog.cpp:14:6: note: constraints not satisfied
prog.cpp: In substitution of 'template<class Coll, class T>  requires  …
          [with Coll = std::set<int>; T = int]':
prog.cpp:27:4:   required from here
prog.cpp:8:9:   required for the satisfaction of 'SupportsPushBack<Coll, T>'
                [with Coll = std::set<int, std::less<int>, std::allocator<int> >; T = int]
prog.cpp:8:28:   in requirements with 'Coll c', 'T v'
                 [with T = int; Coll = std::set<int, std::less<int>, std::allocator<int> >]
prog.cpp:9:16: note: the required expression 'c.push_back(v)' is invalid
    9 |     c.push_back(v);
      |     ~~~~~~~~~~~~~~
```

However, when passing atomics, the check for being copyable still is detected in the deep darkness of the code for `std::vector<>` (this time when the concept is checked instead of when the code is compiled).

In that case, the situation gets better when we specify the basic requirement that is used by `push_back()` as requirement instead:

```
template<typename Coll, typename T>
requires std::convertible_to<T, typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

Here, we use the standard concept `std::convertible_to` to require that the type of the passed argument `T` can be (implicitly and explicitly) converted to the element type of the collection.

Now, if the requirement is broken, we can get an error message with the broken concept and the location where it is broken. For example:[2]

```
...
prog.cpp:11:17:   In substitution of 'template<class Coll, class T>
                  requires  convertible_to<T, typename Coll::value_type>
                  void add(Coll&, const T&)
                  [with Coll = std::vector<std::atomic<int> >; T = std::atomic<int>]':
prog.cpp:25:18:   required from here
.../include/concepts:72:13: required for the satisfaction of
                  'convertible_to<T, typename Coll::value_type>
                  [with T = std::atomic<int>;
                        Coll = std::vector<std::atomic<int>,
                                           std::allocator<std::atomic<int> > >]'
.../include/concepts:72:30: note: the expression 'is_convertible_v<_From, _To>
                  [with _From = std::atomic<int>; _To = std::atomic<int>]'
                  evaluated to 'false'
   72 |     concept convertible_to = is_convertible_v<_From, _To>
      |                              ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
...
```

See rangessort.cpp for another example of algorithms checking the constraints of their parameters.

3.3.2 Using Concepts to Disable Generic Code

Assume we want to provide a special implementation for an `add()` function template like the one introduced above. When dealing with floating-point values, something different or in addition should happen.

A naive approach might be to overload the function template for `double`:

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val)     // for generic value types
{
  coll.push_back(val);
}

template<typename Coll>
void add(Coll& coll, double val)       // for floating-point value types
{
  ...   // special stuff for floating-point values
  coll.push_back(val);
}
```

As expected, when we pass a `double` as a second argument, the second function is called; otherwise, the generic argument is used:

```
std::vector<int> iVec;
add(iVec, 42);        // OK: calls add() for T being int
```

[2] This is just one example of a possible message, not necessarily matching the situation of a specific compiler.

```
std::vector<double> dVec;
add(dVec, 0.7);        // OK: calls 2nd add() for double
```

When passing a `double`, both function overloads match. The second overload is preferred, because it is a perfect match for the second argument.

However, if we pass a `float`, we have the following effect:

```
float f = 0.7;
add(dVec, f);          // OOPS: calls 1st add() for T being float
```

The reason lies in the sometimes subtle details of overload resolution. Again, both function could be called. Overload resolution has some general rules, such as:

- No type conversion is better than a type conversion.
- No template parameter is better than a template parameter.

Here, however, overload resolution has to decide between a type conversion and using a template parameter. Unfortunately, the version with the template parameter is preferred.

Fixing Overload Resolution

The fix for the wrong overload resolution is pretty simple. Instead of declaring the second parameter with a specific type, we should just require that the value to insert has a floating-point type. For this, we can constrain the function template for floating-point values by using the new standard concept `std::floating_point`:

```
template<typename Coll, typename T>
requires std::floating_point<T>
void add(Coll& coll, const T& val)
{
  ...   // special stuff for floating-point values
  coll.push_back(val);
}
```

Because we use a concept that applies to a single template parameters, we can also use the shorthand notation:

```
template<typename Coll, std::floating_point T>
void add(Coll& coll, const T& val)
{
  ...   // special stuff for floating-point values
  coll.push_back(val);
}
```

Or we can use `auto` parameters:

```
void add(auto& coll, const std::floating_point auto& val)
{
  ...   // special stuff for floating-point values
  coll.push_back(val);
}
```

For `add()`, we now have two function templates that can be called, one without and one with a specific requirement:

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val)     // for generic value types
{
  coll.push_back(val);
}

template<typename Coll, std::floating_point T>
void add(Coll& coll, const T& val)     // for floating-point value types
{
  ...   // special stuff for floating-point values
  coll.push_back(val);
}
```

This is enough because overload resolution also prefers overloads or specializations that have constraints over those that have less or no constraints:

```
std::vector<int> iVec;
add(iVec, 42);       // OK: calls add() for generic value types

std::vector<double> dVec;
add(dVec, 0.7);      // OK: calls add() for floating-point types
```

Do not Differ More than Necessary

If two overloads or specializations have constraints, it is important that overload resolution can decide which one is better. For that, the signatures should not differ more than necessary.

If the signatures differ too much, the more constrained overload might not be preferred. For example, if we declare the overload for floating-point values to take the argument by value, passing a floating-point value becomes ambiguous:

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val)  // note: pass by const reference
{
  coll.push_back(val);
}

template<typename Coll, std::floating_point T>
void add(Coll& coll, T val)         // note: pass by value
{
  ...   // special stuff for floating-point values
  coll.push_back(val);
}

std::vector<double> dVec;
```

```
add(dVec, 0.7);        // ERROR: both templates match and no preference
```

The latter declaration is no longer a special case of the former declaration. We have just two different function templates that both could be called.

If you really want to have different signatures, you have to constrain the first function template not to be available for floating-point values.

Restrictions on Narrowing

We have another interesting issue in this example: Both function templates allow us to pass a `double` to add it into a collection of `int`:

```
std::vector<int> iVec;

add(iVec, 1.9);        // OOPS: add 1
```

The reason is that we have implicit type conversions from `double` to `int` (due to being compatible to the programming language C). Such an implicit conversion where we might lose parts of the value is called *narrowing*. It means that the code above compiles and converts the value `1.9` to `1` before it gets inserted.

If you do not want to support narrowing, you have multiple options: One option is to disable type conversions at all by requiring that the passed value type matches the element type of the collection:

```
requires std::same_as<typename Coll::value_type, T>
```

However, this would also disable useful and safe type conversions.

For that reason, it is better to define a concept that yields whether a type can be converted to another type without narrowing, which is possible in a short tricky requirement:[3]

```
template<typename From, typename To>
concept ConvertsWithoutNarrowing =
    std::convertible_to<From, To> &&
    requires (From&& x) {
      { std::type_identity_t<To[]>{std::forward<From>(x)} }
         -> std::same_as<To[1]>;
    };
```

Then we can use this concept to formulate a corresponding constraint:

```
template<typename Coll, typename T>
requires ConvertsWithoutNarrowing<T, typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  ...
}
```

[3] Thanks to Giuseppe D'Angelo and Zhihao Yuan for the trick to check for narrowing conversions introduced in http://wg21.link/p0870.

Subsuming Constraints

It would be probably enough to define the concept for narrowing conversions above without requiring the concept `std::convertible_to`, because the rest checks this implicitly:

```
template<typename From, typename To>
concept ConvertsWithoutNarrowing = requires (From&& x) {
    { std::type_identity_t<To[]>{std::forward<From>(x)} } -> std::same_as<To[1]>;
};
```

However, there is an important benefit if the concept `ConvertsWithoutNarrowing` also checks for the concept `std::convertible_to`. In that case, the compiler can detect that `ConvertsWithoutNarrowing` is more constrained than `std::convertible_to`. The terminology is that `ConvertsWithoutNarrowing` ***subsumes*** `std::convertible_to`.

That allows a programmer to do the following:

```
template<typename F, typename T>
requires std::convertible_to<F, T>
void foo(F, T)
{
   std::cout << "may be narrowing\n";
}

template<typename F, typename T>
requires ConvertsWithoutNarrowing<F, T>
void foo(F, T)
{
   std::cout << "without narrowing\n";
}
```

Without specifying that `ConvertsWithoutNarrowing` ***subsumes*** `std::convertible_to`, the compiler would raise an ambiguity error here when calling `foo()` with two parameters that convert to each other without narrowing.

The same way, concepts can subsume other concepts, so that they count for overload resolution as more specialized. In fact, the C++ standard concepts build a pretty complex subsumption graph.

We discuss details of subsumption later.

3.3.3 Using Requirements to Call Different Functions

Finally, we should make our `add()` function template more flexible:

- We might also want to support collections that only provide `insert()` instead of `push_back()` to insert new elements.
- We might want to support passing a collection (container or range) to insert multiple values.

Yes, you can argue that these are different functions and they should have different names and if that works it is often better to use different names. However, the C++ standard library is a good example of the benefits you can get if different API's are harmonized. For example, you can use the same generic code to iterate

over all containers, although internally containers use very different ways to go to the next element and access its value.

Using Concepts to Call Different Functions

Having just introduced concepts, the "obvious" approach might be to introduce a concept to find out whether a certain function call is supported:

```
template<typename Coll, typename T>
concept SupportsPushBack = requires(Coll c, T v) {
    c.push_back(v);
};
```

Note that we can also define a concept that only needs the collection as template parameter:

```
template<typename Coll>
concept SupportsPushBack = requires(Coll coll, Coll::value_type val) {
  coll.push_back(val);
};
```

Note that we do not have to use `typename` here to use `T::value_type`. Since C++20, `typename` is no longer required when it is clear that it has to be a type.

There are various other ways to declare this concept:

- You can use `std::declval<>()` to get a value of the element type:

  ```
  template<typename Coll>
  concept SupportsPushBack = requires(Coll coll) {
    coll.push_back(std::declval<typename Coll::value_type>());
  };
  ```

  Here, you can see that the definition of concepts and requirements does not create code. It is an unevaluated context where we can use `std::declval<>()` for "assume we would have an object of this type" and it does not matter whether we declare `coll` as a value or non-`const` reference.

- You could use `std::ranges::range_value_t` instead of the `value_type` member:

  ```
  template<typename Coll>
  concept SupportsPushBack = requires(Coll coll) {
    coll.push_back(std::declval<std::ranges::range_value_t<Coll>>());
  };
  ```

  In general, using `std::ranges::range_value_t<>` makes code more generic when the element type of a collection is needed (e.g., it also works with raw array). However, because we require the member `push_back()` here, also requiring the member `value_type` does not hurt.

With the concept `SupportPushBack` for one parameter, we can provide two implementations:

```
template<typename Coll, typename T>
requires SupportsPushBack<Coll>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
```

```
}

template<typename Coll, typename T>
void add(Coll& coll, const T& val)
{
  coll.insert(val);
}
```

Note that we do not need a named requirement `SupportsInsert` here because the `add()` with the additional requirement is more special so that overload resolution prefers it.

Because we define the requirement as a concept, we can use it even as a *type constraint* for the template parameter:

```
template<SupportsPushBack Coll, typename T>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

As a concept we can also use it as type constraint for `auto` as parameter types:

```
void add(SupportsPushBack auto& coll, const auto& val)
{
  coll.push_back(val);
}

template<typename Coll, typename T>
void add(auto& coll, const auto& val)
{
  coll.insert(val);
}
```

Concepts for `if constexpr`

We can also use the concept `SupportsPushBack` directly in an `if constexpr` condition:

```
if constexpr (SupportsPushBack<decltype(coll)>) {
  coll.push_back(val);
}
else {
  coll.insert(val);
}
```

Combining `requires` with `if constexpr`

We can even skip to introduce a concept and pass the `requires` expression directly as condition to the compile-time `if`:[4]

```
if constexpr (requires { coll.push_back(val); }) {
  coll.push_back(val);
}
else {
  coll.insert(val);
}
```

This is a remarkable nice way to switch between different two different function calls in generic code. It is especially recommended, when introducing a concept is not worth it.

Concepts versus Variable Templates

You might wonder, why using concepts is better than using a variable template of type `bool` (like type traits do) such as the following:

```
template<typename T>
constexpr bool SupportsPushBack = requires(T coll) {
  coll.push_back(std::declval<typename T::value_type>());
};
```

However, concepts have a couple of benefits:

- They subsume.
- They can be used as type constraints directly in front of template parameters or `auto`.
- They can be used with the compile-time `if` (`if constexpr`).

If you do not need any of these benefits, the question whether to prefer defining a concept or a variable template of type `bool` becomes interesting and is discussed later in detail.

Inserting Single and Multiple Values

To provide an overload that deals with multiple values passed as one collection, we can simply add constraints for them. The standard concept `std::ranges::input_range` can be used for that:

```
template<SupportsPushBack Coll, std::ranges::input_range T>
void add(Coll& coll, const T& val)
{
  coll.insert(coll.end(), val.begin(), val.end());
}

template<typename Coll, std::ranges::input_range T>
void add(Coll& coll, const T& val)
{
```

[4] Thanks to Arthur O'Dwyer for pointing this out.

```
  coll.insert(val.begin(), val.end());
}
```

Again, as long as the overload has this as an additional constraint, these functions will be preferred.

The concept `std::ranges::input_range` is a concept introduced to deal with ranges, which are collections you can iterate over with `begin()` and `end()`. However, ranges are not required to have `begin()` and `end()` as member functions. Code dealing with ranges should therefore use the helpers `std::ranges::begin()` and `std::ranges::end()` that the ranges library provides:

```
template<SupportsPushBack Coll, std::ranges::input_range T>
void add(Coll& coll, const T& val)
{
  coll.insert(coll.end(), std::ranges::begin(val), std::ranges::end(val));
}

template<typename Coll, std::ranges::input_range T>
void add(Coll& coll, const T& val)
{
  coll.insert(std::ranges::begin(val), std::ranges::end(val));
}
```

These helpers are in fact function objects, which avoids ADL problems.

Dealing with Multiple Constraints

By bringing together all useful concepts and requirements, we could place them all in one function at different locations.

```
template<SupportsPushBack Coll, std::ranges::input_range T>
requires ConvertsWithoutNarrowing<std::ranges::range_value_t<T>,
                                  typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  coll.insert(coll.end(),
              std::ranges::begin(val), std::ranges::end(val));
}
```

To disable narrowing conversions, we use `std::ranges::range_value_t` to pass the element type of the ranges to `ConvertsWithoutNarrowing`. `std::ranges::range_value_t` is another ranges utility to get the element type of ranges when iterating over them.

We could also formulate them together in the requires clause:

```
template<typename Coll, typename T>
requires SupportsPushBack<Coll> &&
         std::ranges::input_range<T> &&
         ConvertsWithoutNarrowing<std::ranges::range_value_t<T>,
                                  typename Coll::value_type>
void add(Coll& coll, const T& val)
{
```

```
    coll.insert(coll.end(),
                std::ranges::begin(val), std::ranges::end(val));
  }
```

Both ways to declare the function template are equivalent.

3.3.4 The Example as a Whole

The past subsections did provide a huge amount of flexibility. So, let us bring all options together to have at least one full example as a whole:

lang/add.cpp

```
#include <iostream>
#include <vector>
#include <set>
#include <ranges>
#include <atomic>

// concept for container with push_back():
template<typename Coll>
concept SupportsPushBack = requires(Coll coll, Coll::value_type val) {
  coll.push_back(val);
};

// concept to disable narrowing conversions:
template<typename From, typename To>
concept ConvertsWithoutNarrowing =
    std::convertible_to<From, To> &&
    requires (From&& x) {
      { std::type_identity_t<To[]>{std::forward<From>(x)} }
          -> std::same_as<To[1]>;
    };


// add() for single value:
template<typename Coll, typename T>
requires ConvertsWithoutNarrowing<T, typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  if constexpr (SupportsPushBack<Coll>) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
```

```
}

// add() for multiple values:
template<typename Coll, std::ranges::input_range T>
requires ConvertsWithoutNarrowing<std::ranges::range_value_t<T>,
                                  typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  if constexpr (SupportsPushBack<Coll>) {
    coll.insert(coll.end(),
                std::ranges::begin(val), std::ranges::end(val));
  }
  else {
    coll.insert(std::ranges::begin(val), std::ranges::end(val));
  }
}

int main()
{
  std::vector<int> iVec;
  add(iVec, 42);       // OK: calls push_back() for T being int

  std::set<int> iSet;
  add(iSet, 42);       // OK: calls insert() for T being int

  short s = 42;
  add(iVec, s);        // OK: calls push_back() for T being short

  long long ll = 42;
  //add(iVec, ll);     // ERROR: narrowing
  //add(iVec, 7.7);    // ERROR: narrowing

  std::vector<double> dVec;
  add(dVec, 0.7);       // OK: calls push_back() for floating-point types
  add(dVec, 0.7f);      // OK: calls push_back() for floating-point types
  //add(dVec, 7);       // ERROR: narrowing

  // insert collections:
  add(iVec, iSet);      // OK: insert set elements into a vector
  add(iSet, iVec);      // OK: insert vector elements into a set

  // can even insert raw array:
  int vals[] = {0, 8, 18};
  add(iVec, vals);      // OK
  //add(dVec, vals);    // ERROR: narrowing
```

```
}
```

As you can see, I decided to use concept `SupportsPushBack` for one template parameter inside the various functions templates with `if constexpr`.

3.3.5 Former Workarounds

Constraining templates was already possible before C++20. However, the way this was possible often was not easy to use and could provide significant drawbacks.

SFINAE

The major approach to disable the availability of templates before C++20 was SFINAE. The term "SFINAE" (pronounced like *sfee-nay*) stands for "*substitution failure is not an error*" and means the rule that we simply ignore generic code if its declaration is not well-formed instead of raising a compile-time error.

For example, to switch between `push_back()` and `insert()`, we could declare the function templates before C++20 as follows:

```
template<typename Coll, typename T>
auto add(Coll& coll, const T& val) -> decltype(coll.push_back(val))
{
  return coll.push_back(val);
}

template<typename Coll, typename T>
auto add(Coll& coll, const T& val) -> decltype(coll.insert(val))
{
  return coll.insert(val);
}
```

Because we make the call to `push_back()` here part of the first template **declaration**, this template is ignored if `push_back()` is not supported. The corresponding declaration is necessary in the second template for `insert()` (overload resolution does not understand that one template is more constrained than the other).

However, that is a very subtle way to place a requirement and programmers might overlook it easily.

`std::enable_if<>`

For more complicated cases of disabling generic code the C++ standard library provides since C++11 `std::enable_if<>`.

This is a type trait that takes a Boolean condition which yields invalid code if `false`. By using the `std::enable_if<>` somewhere inside a declaration, this could also "SFINAE out" generic code.

For example, you could use the `std::enable_if<>` type trait to exclude types from calling the `add()` function template in the following way:

```
// disable the template for floating-point values:
template<typename Coll, typename T,
```

```
          typename = std::enable_if_t<!std::is_floating_point_v<T>>>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

The trick is to insert an additional template parameter to be able to use the trait `std::enable_if<>`. The trait yields `void` if it does not disable the template (you can specify that it yields another type as optional second template parameter).

Code like this is also hard to write and read and has subtle drawbacks. For example, to provide another overload of `add()` with a different constraint need yet another template parameter because for the compiler `std::enable_if<>` does not change the signature and you would have two different overloads of the same function. In addition, you have to be very careful that with each call exactly one of the different overloads is available.

Concepts provide a far more readable way of formulating constraints. This includes that overloads of a template with different constraints do not violate the one definiton rule as long as only one constraint is met and that even multiple constraints can be met provided one constraint subsumes another constraint.

3.4 Semantic Constraints

Concepts might check both syntactical and semantic constraints:

- **Syntactic constraints** mean that we can check at compile-time whether certain functional requirements are satisfied ("*Is a specific operation supported?*" or "*Does a specific operation yield a specific type?*").
- **Semantic constraints** mean that certain requirements are satisfied, which can only be checked at runtime ("*Does an operation have the same effect?*" or "*Does the same operation performed for a specific value always yield the same result?*").

Sometimes, concepts allow programmers to convert semantic constraints into syntactic constraints, by providing an interface to specify that a semantic constraint is fulfilled or is not fulfilled.

3.4.1 Examples of Semantic Constraints

Let us look at some examples of semantic constraints.

`std::ranges::sized_range`

An example of a semantic constraint is the concept `std::ranges::sized_range`. It guarantees that the number of elements in a range can be computed in constant time (either by calling the member `size()` or by computing the difference between the begin and the end).

If a range type provides `size()` (provided as member of free-standing function), this concept is fulfilled by default. To opt-out from this concept (e.g., because it iterates over all elements to yield its result), you can and should set `std::disable_sized_range<`*Rg*`>` to `true`:

```
class MyCont {
  ...
  std::size_t size() const;   // assume this is expensive, so that this is not a sized range
```

```
};
// opt-out from concept std::ranges::sized_range:
constexpr bool std::ranges::disable_sized_range<MyCont> = true;
```

std::ranges::range vs. std::ranges::view

A similar example of a semantic constraint is the concept `std::ranges::view`. Besides some syntactic constraints it also guarantees that move constructor/assignment, copy constructor/assignment (if available), and destructor have constant complexity (the time they take does not depend on the number of elements).

An implementor can provide the corresponding guarantee by deriving publicly from either `std::ranges::view_base` or `std::ranges::view_interface<>` or by setting the specialization `std::ranges::enable_view<`*Rg*`>` to `true`.

std::invocable vs. std::regular_invocable

A simple example of a semantic constraint is the difference between the concepts `std::invocable` and `std::regular_invocable`. The latter guarantees not to modify the state of the passed operation and the passed arguments. We cannot check that with a compiler, so the difference document the ***intention*** of the specified API. Often, for simplicity, just `std::invocable` is used.

std::weakly_incrementable vs. std::incrementable

Besides certain syntactic differences there are also semantic differences between the concepts `incrementable` and `weakly_incrementable`:

- `incrementable` requires each increment of the same value gives the same result.
- `weakly_incrementable` only requires that a type supports the increment operators. Incrementing the same value may yield different results.

Therefore:

- When `incrementable` is satisfied, you can iterate multiple times from a starting value over a range.
- When only `weakly_incrementable` is satisfied, you can iterate over a range only once. A second iteration with the same staring value might yield different results.

This difference matters for iterators: Input stream iterators (iterators that read values from a stream) can only iterate once because the next iteration yields different values. Consequently, input stream iterators satisfy the `weakly_incrementable` concept but not the `incrementable` concept. However, the concepts cannot be used to check for this difference:

```
std::weakly_incrementable<std::istream_iterator<int>>   // yields true
std::incrementable<std::istream_iterator<int>>          // OOPS: also yields true
```

The reason is that the difference is a semantic constraint that cannot be checked at compile time. So the concepts can be used to *document* the constraints:

```
template<std::weakly_incrementable T>
void algo1(T beg, T end);                    // single-pass algorithm

template<std::incrementable T>
```

```
void algo2(T beg, T end);                    // multi-pass algorithm
```

Note that we use different names for the algorithms here. Due to the fact that we cannot check the semantic difference of the constraints, it is up to the programmer to not pass an input stream iterator:

```
algo1(std::istream_iterator<int>{std::cin},  // OK
      std::istream_iterator<int>{});

algo2(std::istream_iterator<int>{std::cin},  // OOPS: violates constraint
      std::istream_iterator<int>{});
```

However, you cannot dispatch between two different implementations based on this difference:

```
template<std::weakly_incrementable T>
void algo(T beg, T end);                     // single-pass implementation

template<std::incrementable T>
void algo(T beg, T end);                     // multi-pass implementation
```

If you pass an input stream iterator here, the compiler will wrongly use the multi-pass implementation:

```
algo(std::istream_iterator<int>{std::cin},  // OOPS: calls the wrong overload
     std::istream_iterator<int>{});
```

Fortunately, there is a solution here, because for this semantic difference C++98 already had introduced iterator traits, which are used by the iterator concepts. If you use these concepts (or the corresponding range concepts), everything works fine:

```
template<std::input_iterator T>
void algo(T beg, T end);                     // single-pass implementation

template<std::forward_iterator T>
void algo(T beg, T end);                     // multi-pass implementation

algo(std::istream_iterator<int>{std::cin},  // OK: calls the right overload
     std::istream_iterator<int>{});
```

You should prefer the more specific concepts for iterators and ranges, which also honor the new and slightly modified iterator categories.

3.5 Design Guidelines for Concepts

Let us look at some guidelines of how to use concepts. Note that concepts are new and we still learn how to use them best. In addition, improved support of concepts might change some guidelines over time.

3.5.1 Concepts Should Group Requirements

Introducing a concept for each attribute or functionality of types is certainly too fine-grained. As a consequence we would get far too much concepts, which a compiler has to deal with and we all would have to specify as constraints.

Therefore, concepts should provide common and typical aspects that separate different categories of requirements or types. However, there are corner cases.

The C++ standard library provides a good example of a design that might follow this approach. Most concepts are provided to categorize types such as ranges, iterators, functions, and so on as a whole. However, to support subsumption and ensure that concepts are consistent, several basic concepts (such as std::movable) are provided.

The consequence is a pretty complex subsumption graph. The chapter that describes the C++ standard concepts groups the concepts accordingly.

3.5.2 Define Concepts with Care

Code using concepts can be equivalent although it syntactically differs. To enable compilers to find out whether various applications of concepts are equivalent or subsume, concepts need careful design.

For example, you have to specify carefully of the order of two template arguments does not matter. Tu understand what that means, let us assume we define our own concept `SameAs` and use it as follows:

```
template<typename T, typename U>
concept SameAs = std::is_same_v<T, U>;          // define concept SameAs

template<typename T, typename U>
requires SameAs<T, U>                           // use concept SameAs
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires SameAs<T, U> && std::integral<T>       // use concept SameAs again
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2);    // OK: second foo() preferred
```

Here, the second definition of `foo()` subsumes the first one so that it is used when two arguments of the same type are passed.

However, note what happens if we slightly modify the parameters when we call `SameAs<>` in the constraints of the second `foo()`:

```
template<typename T, typename U>
concept SameAs = std::is_same_v<T, U>;          // define concept SameAs

template<typename T, typename U>
requires SameAs<T, U>                           // use concept SameAs
void foo(T, U)
```

```
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires SameAs<U, T> && std::integral<T>          // use concept SameAs with other order
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2);    // ERROR: ambiguity: both constraints are satisfied without one subsuming the other
```

The problem is that the compiler cannot detect that `SameAs<>` is commutative. For the compiler, the order matters and therefore the first requirement is not necessarily a subset of the second requirement.

To solve this problem we have to design the concept `SameAs` in a way that the order of the arguments does not matter. This requires a helper concept:

```
template<typename T, typename U>
concept SameAsHelper = std::is_same_v<T, U>;

template<typename T, typename U>
concept SameAs = SameAsHelper<T, U> && SameAsHelper<U, T>;     // commutative
```

Now, the order of the parameters no longer matters for `IsSame<>`:

```
template<typename T, typename U>
requires SameAs<T, U>                              // use SameAs
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires SameAs<U, T> && std::integral<T>          // use SameAs with other order
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2);    // OK: second foo() preferred
```

The compiler can find out that the first building block `SameAs<U,T>` is part of a sub-concept of the definition of `SameAs` so that the other building blocks `SameAs<T,U>` and `std::integral<T>` are an extension. Therefore, the second `foo()` now has preference.

Tricky details like this are designed into the concepts the C++20 standard library provides (see `std::same_as`). Therefore, you should use them when they fit your needs instead of providing your own concepts.

```
template<typename T, typename U>
```

```
requires std::same_as<T, U>                    // standard same_as<> is commutative
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires std::same_as<U, T> && std::integral<T>   // so different order does not matter
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2);    // OK: second foo() preferred
```

For your own concepts, take as much as possible use (and abuse) into account. The more the better. As any good piece of software, concepts also require good design and test cases.

3.5.3 Concepts versus Traits and Boolean Expressions

Concepts are more than just expressions that evaluate Boolean results at compile time. You should usually prefer them over type traits and other compile-time expressions.

However, concepts have a couple of benefits:

- They subsume.
- They can be used as type constraints directly in front of template parameters or `auto`.
- They can be used with the compile-time `if` (`if constexpr`) as introduced before.

Benefit From Subsumption

The main benefit of concepts is that they subsume. Type traits do not subsume.

Consider the following example, where we overload a function `foo()` with two requirements defined as **type traits**:

```
template<typename T, typename U>
requires std::is_same_v<T, U>              // using traits
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires std::is_same_v<T, U> && std::is_integral_v<T>
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
```

```
}

foo(1, 2);    // ERROR: ambiguity: both requirements are true
```

The problem is that if both requirements evaluate to `true` both overloads fit and there is no rule that one of them has priority over the other. Therefore, the compiler stops compiling with an ambiguity error.

If we use the corresponding **concepts** instead, the compiler finds out that the second requirement is a specialization and prefers it if both requirements are met:

```
template<typename T, typename U>
requires std::same_as<T, U>                // using concepts
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires std::same_as<T, U> && std::integral<T>
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2);    // OK: second foo() preferred
```

Benefit From Using Concepts with `if constexpr`

C++17 introduced a compile-time if which allows us to switch between code depending on certain compile-time conditions.

For example (as introduced before):

```
template<typename Coll, std::floating_point T>
void add(Coll& coll, const T& val)     // for floating-point value types
{
  if constexpr(std::is_floating_point_v<T>) {
    ...    // special stuff for floating-point values
  }
  coll.push_back(val);
}
```

To behave different when using template parameters, this can be more readable than providing overloaded or specialized templates.

However, you cannot use `if constexpr` to provide different API's, to allow others to add other overloads or specializations later on, or to disable template at all, However, remember that you can constrain member functions to enable or disable parts of an API based on requirements.