Chapter 4

Concepts, Constraints, and Requirements in Detail

This chapter discusses several aspects of concepts, constraints, and requirements in detail.

In addition, the following chapter lists and discusses all concepts the C++ standard library provides.

4.1 Constraints

To specify requirements for generic parameters you need ***constraints***, which are used at compile-time to decide whether to instantiate and compile a template or not.

You can constrain function templates, class templates, variable templates, and alias templates.

An ordinary *constraint* is usually specified by using a ***requires clause***. For example:

```
template<typename T>
void foo(const T& arg)
requires MyConcept<T>
...
```

In front of a template parameters or `auto` you can also directly use a concept as ***type constraint***:

```
template<MyConcept T>
void foo(const T& arg)
...
```

or:

```
void foo(const MyConcept auto& arg)
...
```

4.2 `requires` Clauses

A ***requires clause***, uses the keyword **`requires`** with some compile-time Boolean expression to restrict the availability of the template. The Boolean expression can be:

- An ad-hoc Boolean compile-time expression
- A *concept*
- A *requires expressions*

All constraints could also be used wherever a Boolean expression can be used (especially as an `if constexpr` condition).

4.2.1 Using `&&` and `||` in `requires` Clauses

To combine multiple constraints in requires clauses, we can use operator `&&`. For example:

```
template<typename T>
requires (sizeof(T) > 4)                        // ad-hoc Boolean expression
         && requires { typename T::value_type; } // requires expression
         && std::input_iterator<T>               // concept
void foo(T x) {
  ...
}
```

The order of the constraints does not matter.

It is also possible to express "alternative" constraints using operator `||`. For example:

```
template<typename T>
requires std::integral<T> || std::floating_point<T>
T power(T b, T p);
```

Specifying alternative constraints is rarely needed and should not be done too casually because excessive use of the operator `||` in requires clauses may potentially tax compilation resources (i.e., make compilation noticeably slower).

A single constraint can also involve multiple template parameters. That way, constraints can impose a relationship between type parameters. For example:

```
template<typename T, typename U>
requires std::convertible_to<T, U>
auto f(T x, U y) {
  ...
}
```

Operators `&&` and `||` are the only operators you can combine multiple constraints without the need to use parentheses. For everything else use parentheses (which formally passes an ad-hoc Boolean expression to the requires clause).

4.3 Ad hoc Boolean Expressions

The basic way to formulate constraints for templates is to use a requires clause: `requires` followed by a Boolean expressions. After `requires` the constraint can use any compile-time Boolean expressions, not only concepts or requires expressions. These expressions may especially use:

- Type predicates, such as type traits
- Compile-time variables (defined with `constexpr` or `constinit`)
- Compile-time functions (defined with `constexpr` or `consteval`)

Let us look at some examples of using ad hoc Boolean expressions to restrict the availability of a template:

- Available only if `int` and `long` have a different size:

  ```
  template<typename T>
  requires (sizeof(int) != sizeof(long))
  ...
  ```

- Available only if `sizeof(T)` is not too large:

  ```
  template<typename T>
  requires (sizeof(T) <= 64)
  ...
  ```

- Available only if the non-type template parameter `Sz` is greater than zero:

  ```
  template<typename T, std::size_t Sz>
  requires (Sz > 0)
  ...
  ```

- Available only for raw pointers and the `nullptr`:

  ```
  template<typename T>
  requires (std::is_pointer_v<T> || std::same_as<T, std::nullptr_t>)
  ...
  ```

  `std::same_as` is a new standard concept. Instead, you could also use the standard type trait `std::is_same_v<>`:

  ```
  template<typename T>
  requires (std::is_pointer_v<T> || std::is_same_v<T, std::nullptr_t>)
  ...
  ```

- Available only if the argument cannot be used as a string:

  ```
  template<typename T>
  requires (!std::convertible_to<T, std::string>)
  ...
  ```

  `std::convertible_to` is a new standard concept. You could also use the standard type trait `std::is_convertible_v<>`:

  ```
  template<typename T>
  requires (!std::is_convertible_v<T, std::string>)
  ...
  ```

- Available only if the argument is a pointer (or pointer-like object) to an integral value:

```
template<typename T>
requires std::integral<std::remove_reference_t<decltype(*std::declval<T>())>>
...
```

Note that `operator*` usually yields a reference, which is not an integral type. Therefore, we do the following:

– Assume we have an object of type `T`: `std::declval<T>()`
– Call `operator*` for it: `*`
– Ask for its type: `decltype()`
– Remove referenceness: `std::remove_reference_v<>`
– Check of integral type: `std::integral<>`

This constraint would also be satisfied by a `std::optional<int>`.

`std::integral` is a new standard concept. You could also use the standard type trait `std::is_integral_v<>`.

- Available only if the non-type template parameters `Min` and `Max` have a greatest common divisor that is greater than one:

```
template<typename T>
constexpr bool gcd(T a, T b);   // greatest common divisor (forward declaration)

template<typename T, int Min, int Max>
requires (gcd(Min, Max) > 1)    // available if there is a GCD greater than 1
...
```

- Disable a template (temporarily):

```
template<typename T>
requires false                  // disable the template
...
```

Usually, you need parentheses around the whole expression. The only exception are constraints using identifiers, `::`, and `<...>` (optionally combined with `&&` and `||`):

```
requires std::convertible_to<T, int>      // no parentheses needed here
         &&
         (!std::convertible_to<int, T>)   // ! forces the need of parentheses
```

4.4 `requires` Expressions

Requires expressions (which are distinct from *requires clauses*) provide a simple but flexible syntax to specify multiple requirements on one or multiple template parameters. You can specify:

- Required type definitions
- Expressions that have to be valid
- Requirements on the types that expressions yield

A *requires expression* starts with `requires` followed by an optional parameter list and then a block of requirements (all ending with semicolons). For example:

```
template<typename Coll>
```

```
... requires {
      typename Coll::value_type::first_type;    // elements/values have first_type
      typename Coll::value_type::second_type;   // elements/values have second_type
    }
```

The optional parameter list allows you to introduce a set of "dummy variables" usable to express requirements in the body of the requires expression:

```
template<typename T>
... requires(T x, T y) {
      x + y;    // supports +
      x - y;    // supports -
    }
```

These parameters are never replaced by arguments. Therefore, it usually does not matter whether you declare them by value or by reference.

The parameters also allow us to introduce (parameters of) sub-types:

```
template<typename Coll>
... requires(Coll::value_type v) {
      std::cout << v;    // supports output operator
    }
```

This requirements checks whether `Coll::value_type` is valid and whether objects of this type support the output operator.

Note that type members in this parameter list do not have to be qualified with `typename`.

When using this to check only whether `Coll::value_type` is valid, you do not need anything in the body of the block of the requirements. However, the block cannot be empty. So, you might simply use `true` then:

```
template<typename Coll>
... requires(Coll::value_type v) {
      true;                  // dummy requirement because the block cannot be empty
    }
```

4.4.1 Simple Requirements

Simple requirements are just expressions that have to be well-formed. That means the calls have to compile. The calls are not performed, so it does not matter whether the operations have defined behavior or yield `true`.

For example:

```
template<typename T1, typename T2>
... requires(T1 val, T2 p) {
      *p;                 // operator* has to be supported for T2
      p[0];               // operator[] has to be supported for int as index
      p->value();         // calling a member function value() without arguments has to be possible
      *p > val;           // support that we can compare the result of operator* with T1
      p == nullptr;       // support that we can compare a T2 with a nullptr
    }
```

```
    }
```

The last call does not require that `p` ***is*** the `nullptr` (to require that, you have to check whether `T2` is type `std::nullptr_t`). Instead, we require that we ***can*** compare an object of type `T2` with an object of type `std::nullptr_t` (the type of `nullptr`).

It usually does not make sense to use operator `||`. A simple requirement such as

```
*p > val || p == nullptr;
```

does ***not*** require that either the left or the right sub-expression is possible. It formulates the requirement that we can combine the results of both sub-expressions with `operator||`.

To require either one of the two sub-expressions, you have to use:

```
template<typename T1, typename T2>
... requires(T1 val, T2 p) {
      *p > val;             // support that we can compare the result of operator* with T1
    }
    || requires(T2 p) {   // OR
      p == nullptr;         // support that we can compare a T2 with a nullptr
    }
```

Also note that this concept does ***not*** require that `T` to be an integral type:

```
template<typename T>
... requires {
      std::integral<T>;              // OOPS: does not require T to be integral
      ...
    };
```

It only requires that the expression `std::integral<T>` is valid, which is the case for all types. Instead, you have to formulate it as follows:

```
template<typename T>
... std::integral<T> &&              // OK, does require T to be integral
    requires {
      ...
    };
```

or as follows:

```
template<typename T>
... requires {
      requires std::integral<T>;     // OK, does require T to be integral
      ...
    };
```

4.4.2 Type Requirements

Type requirements are expressions that have to be well-formed when using a name of a type. That means the specified name has to be defined as a valid type.

For example:

```
template<typename T1, typename T2>
... requires {
  typename T1::value_type;                // type member value_type required for T1
  typename std::ranges::iterator_t<T1>;   // iterator type required for T1
  typename std::ranges::iterator_t<std::vector<T1>>;
  typename std::common_type_t<T1, T2>;    // T1 and T2 have to have a common type
}
```

For all type requirements, if the type exists but is `void` then the requirement is met.

Note that you can only check for names given to types (names of classes, enumeration types, from `typedef` or `using`). You cannot check for other type declarations using the type:

```
template<typename T>
... requires {
  typename int;           // ERROR: invalid type requirement
  typename T&;            // ERROR: invalid type requirement
}
```

The way to test the latter is to declare a corresponding parameter:

```
template<typename T>
... requires(T&) {
  true;        // some dummy requirement
};
```

Again, the requirements checks whether using the passed type(s) to define another type is valid. For example:

```
template<std::integral T>
class MyType1 {
  ...
};

template<typename T>
requires requires {
  typename MyType1<T>;      // instantiation of MyType1 for T would be valid
}
void mytype1(T) {
  ...
}

mytype1(42);      // OK
mytype1(7.7);     // ERROR
```

Therefore, the following requirement does ***not*** check whether there is a standard hash function for type `T`:

```
template<typename T>
concept StdHash = requires {
  typename std::hash<T>;   // does not check whether std::hash<> is defined for T
};
```

The way to do that is to try to create or use it:

```
template<typename T>
concept StdHash = requires {
  std::hash<T>{};   // OK, checks whether we can create a standard hasher for T
};
```

Note that simple requirements only check whether a requirement is **valid**, not whether it is fulfilled. For this reason:

- It does not make sense to use type functions that always yield a value:

  ```
  template<typename T>
  ... requires {
    std::is_const_v<T>;                // not useful: always valid (doesn't matter what it yields)
  }
  ```

  To check for constness, use:

  ```
  template<typename T>
  ... std::is_const_v<T>   // ensure that T is const
  ```

  Inside an `requires` expression, you can use a nested requirement (see below).

- It does not make sense to use type functions that always yield a type:

  ```
  template<typename T>
  ... requires {
    typename std::remove_const_t<T>;     // not useful: always valid (yields a type)
  }
  ```

  The requirement only checks whether the type expression yields a type, which is always the case.

It also does not make sense to use type functions may have undefined behavior. For example, the type trait `std::make_unsigned<>` requires that the passed argument is an integral type other than `bool`. If this is not the case, you get undefined behavior. If you used it as a requirement:

```
std::make_unsigned_r<T>::type     // not useful as type requirement (valid or undefined behavior)
```

the requirement can only be fulfilled or results in undefined behavior (which might mean that the requirement is still fulfilled).

4.4.3 Compound Requirements

Compound requirements allow us to combine the abilities of simple and type requirements. In this case, you can specify an expression (inside a block of braces) and then add one or both of the following:

- `noexcept` to require that the expression guarantees not to throw
- `->` *type-constraint* to apply a concept on what the expression evaluates to

Here are some examples:

```
template<typename T>
... requires(T x) {
  { &x } -> std::input_or_output_iterator;
  { x == x }
  { x == x } -> std::convertible_to<bool>;
```

```
    { x == x }noexcept
    { x == x }noexcept -> std::convertible_to<bool>;
  }
```

Note that the type constraint after the `->` takes the resulting type as its first template argument. That means:

- In the first requirement we require that the concept `std::input_or_output_iterator` is satisfied when using `operator&` for an object of type `T` (`std::input_or_output_iterator<decltype(&x)>` yields `true`).

  You could also specify this as follows:

  ```
  { &x } -> std::is_pointer_v<>;
  ```

- In the last requirement we require that we can use the result of `operator==` for two objects of type `T` as `bool` (the concept `std::convertible_to` is satisfied when passing the result of `operator==` for two objects of type `T` and `bool` as arguments).

Requires expressions can also express the need for associated types. For example:

```
template<typename T>
... requires(T coll) {
  { *coll.begin() } -> std::convertible_to<T::value_type>;
}
```

However, you cannot specify type requirement using nested types. For example, you cannot use it to require that the return value of `operator*` yields an integral value. The problem is that the return value is a reference you have to dereference first:

```
std::integral<std::remove_reference_t<T>>
```

and you cannot use such a nested expression with a type trait in a result of a requires expression:

```
template<typename T>
concept Check = requires(T p) {
  { *p } -> std::integral<std::remove_reference_t<>>;  // ERROR
  { *p } -> std::integral<std::remove_reference_t>;    // ERROR
};
```

You either have to define a corresponding concept first:

```
template<typename T>
concept UnrefIntegral = std::integral<std::remove_reference_t<T>>;

template<typename T>
concept Check = requires(T p) {
  { *p } -> UnrefIntegral;  // OK
};
```

Or you have to use a nested requirement.

4.4.4 Nested Requirements

Nested requirements allow us to specify additional constraints by using local parameters. They start with `requires` followed by a compile-time Boolean expression, which might itself again be or use a requires expression. The benefit os this feature is that we can ensure that an expression using objects declared using the passed type is `true` instead of only ensuring that it is valid.

For example, consider a concept has to ensure that both operator `*` and operator `[]` yield the same type for a given type. By using nested requirements, we can specify this as follows:

```
template<typename T>
concept DerefAndIndexMatch = requires (T p) {
                                requires std::same_as<decltype(*p), decltype(p[0])>;
                              };
```

The good thing is that we have an easy syntax here for "*assume we have an object of type* `T`." We do not have to use a requires expressions here; however, then the code has to use `std::declval<>()`:

```
template<typename T>
concept DerefAndIndexMatch = std::same_as<decltype(*std::declval<T>()),
                                          decltype(std::declval<T>()[0])>;
```

As another example, we can use a nested requirement to solve the problem just introduced to specify a complex type requirement on an expression:[1]

```
template<typename T>
concept Check = requires(T p) {
  requires std::integral<std::remove_cvref_t<decltype(*p)>>;
};
```

Note the following difference inside a requires expression:

```
template<typename T>
... requires {
  !std::is_const_v<T>;                    // OOPS: checks whether we call call is_const_v<>
  requires !std::is_const_v<T>;           // OK: checks whether T is not const
}
```

Here we use the type trait `is_const_v<>` without and with `requires`. However, only the second requirement is probably what was meant:

- The first expression requires only that ***checking for constness is valid*** and negating the result. This requirement is always met (even if `T` is `const int`) because doing this check is always valid. This requirement is worthless.
- The second expression with `requires` has to be ***fulfilled***. The requirement is met if `T` is `int`, but not if `T` is `const int`.

[1] Thanks to Hannes Hauswedell for pointing this out.

4.5 Concepts in Detail

By defining a `concept` you can introduce a name for one or more *constraints*.

Templates (function, class, variable templates, and alias templates) can use concepts to constrain their ability (via a requires clause or as a direct type constraint for a template parameter). However, concepts are also Boolean compile-time expressions (type predicates) you can use wherever you have to check something for a type (such as in an `if constexpr` condition).

4.5.1 Defining Concepts

Concepts are defined as follows:

```
template<...>
concept name = ... ;
```

The equal sign is required (you cannot declare a concept without defining it and you cannot use braces here). Behind the equal sign you can specify any compile-time expression that converts to `true` or `false`.

Concepts are much like `constexpr` variable templates of type `bool`, but the type is not explicitly specified:

```
template<typename T>
concept MyConcept = ... ;

std::is_same<MyConcept<...>, bool>    // yields true
```

That means, at compile-time or runtime you can always use a concept where the value of a Boolean expression is needed. However, you cannot take the address because there is no object behind it (it is a prvalue).

The template parameters may not have constraints (you cannot use a concept to define a concept).

You cannot define concepts inside a function (as is the case for all templates).

4.5.2 Special Abilities of Concepts

Concepts have special abilities.

Consider, for example, the following concept:

```
template<typename T>
concept IsOrHasThisOrThat = ... ;
```

Compared to a definition of a Boolean variable template (which is the usual way type traits are defined):

```
template<typename T>
inline constexpr bool IsOrHasThisOrThat = ... ;
```

we have the following differences:

- Concepts do not represent code. They have no type, storage, lifetime, or any other properties associated with objects.

  By instantiating them at compile time for specific template parameters, their instantiation just becomes `true` or `false`. Therefore, you can use them wherever you can use `true` or `false` and you get all properties of these literals.

- Concepts do not have to be declared as `inline`, they implicitly are.
- Concepts can be used as type constraints:

  ```
  template<IsOrHasThisOrThat T>
  ...
  ```

  Variable templates cannot be used that way.
- Concepts are the only way to give *constraints* a name, which means that you need them to decide whether a constraint is a special case of another constraint.
- Concepts subsume. To let the compiler decide whether a constraint implies another constraint (and is therefore special), the constraints have to be formulated as concepts.

4.5.3 Concepts for Non-Type Template Parameters

Concepts can also be applied to non-type template parameters (NTTP). For example:

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

template<int Val>
requires LessThan10<Val>
class MyType {
  ...
};
```

As a more useful example, we can use a concept to constrain the value of a non-type template parameter to be a power of two:

lang/conceptnttp.cpp

```
#include <bit>

template<auto Val>
concept PowerOf2 = std::has_single_bit(static_cast<unsigned>(Val));

template<typename T, auto Val>
requires PowerOf2<Val>
class Memory {
  ...
};

int main()
{
   Memory<int, 8> m1;     // OK
   Memory<int, 9> m2;     // ERROR
   Memory<int, 32> m3;    // OK
   Memory<int, true> m4;  // OK
   ...
```

```
}
```

The concept `PowerOf2` takes a value instead of a type as template parameter (here using `auto` to not require a specific type):

```
template<auto Val>
concept PowerOf2 = std::has_single_bit(static_cast<unsigned>(Val));
```

The concept is satisfied when the new standard function `std::has_single_bit()` yields `true` for the passed value (having only one bit set means that a value is a power of 2). Note that `std::has_single_bit()` requires that we have an `unsigned` integral value. By casting to `unsigend` programmers can pass signed integral values and reject types that cannot be converted to an unsigend integral value.

The concept is then used to require that a class `Memory`, taking a type and a size, only accepts sizes that are a power of 2:

```
template<typename T, auto Val>
requires PowerOf2<Val>
class Memory {
  ...
};
```

Note that you cannot write this:

```
template<typename T, PowerOf2 auto Val>
class Memory {
  ...
};
```

This puts the requirement on the ***type*** of `Val`; however, the concept `PowerOf2` does not constrain a type; it constrains the value.

4.6 Using Concepts as Type Constraints

As introduced, concepts can be used as type constraints. There are different places where type constraints can be used:

- In the declaration of a template type parameter
- In the declaration of a call parameter declared with `auto`
- As requirement in a compound requirements

For example:

```
template<std::integral T>                          // type constraint for a template parameter
class MyClass {
  ...
};

auto myFunc(const std::integral auto& val) {       // type constraint for an auto parameter
  ...
};
```

```
template<typename T>
concept MyConcept = requires(T x) {
    { x + x } -> std::integral;              // type constraint for return type
  };
```

Here, we use unary constraints that are called for a single parameter or type returned by an expression.

Type Constraints with Multiple Parameters

You can also use constraints with multiple parameters, for which the parameter type or return value is then used as the first argument:

```
template<std::convertible_to<int> T>                     // conversion to int required
class MyClass {
  ...
};

auto myFunc(const std::convertible_to<int> auto& val) {  // conversion to int required
  ...
};

template<typename T>
concept MyConcept = requires(T x) {
    { x + x } -> std::convertible_to<int>;              // conversion to int required
  };
```

Another example often used is to constrain the type of a callable (function, function object, lambda) to require that you can pass a certain number of arguments of certain types using the concepts std::invocable or std::regular_invocable: For example, to require to pass an operation that takes an `int` and a `std::string`, you have to declare:

```
template<std::invocable<int, std::string> Callable>
void call(Callable op);
```

or:

```
void call(std::invocable<int, std::string> auto op);
```

The difference between `std::invocable` and `std::regular_invocable` is that the latter guarantees not to modify the passed operation and arguments. That is a semantic difference which only helps to document the intention. Often just `std::invocable` is used.

Type Constraints and `auto`

Type constraints can be used in all places where `auto` can be used. The major application of this feature is to use them for the abbreviated function template syntax. For example:

```
void foo(const std::integral auto& val)
{
  ...
```

```
}
```

However, type constraints can be used everywhere where `auto` is used:

- To constrain declarations:

  ```
  std::floating_point auto val1 = f();              // valid, if f() yields floating-point value

  for (const std::integral auto& elem : coll) {  // valid, if elements are integral values
    ...
  }
  ```

- To constrain return types:

  ```
  std::copyable auto foo(auto) {                    // valid if foo() returns copyable value
    ...
  }
  ```

- to constrain non-type template parameters:

  ```
  template<typename T, std::integral auto Max>
  class SizedColl {
    ...
  };
  ```

This also works with concepts taking multiple parameters:

```
template<typename T, std::convertible_to<T> auto DefaultValue>
class MyType {
  ...
};
```

For another example, see the support for lambdas as non-type template parameters.

4.7 Subsuming Constraints with Concepts

Two concepts can have a subsuming relation. That is, one concept can be specified that it restricts one or more other concepts. The benefit is that overload resolution then prefers the more constrained generic code over the less constrained generic code when both constraints are satisfied.

For example, consider we introduce the following two concepts:

```
template<typename T>
concept GeoObject = requires(T obj) {
    { obj.width() } -> std::integral;
    { obj.height() } -> std::integral;
    obj.draw();
  };

template<typename T>
concept ColoredGeoObject =
  GeoObject<T> &&              // subsumes concept GeoObject
  requires(T obj) {            // additional constraints
    obj.setColor(Color{});
    { obj.getColor() } -> std::convertible_to<Color>;
  };
```

The concept `ColoredGeoObject` explicitly *subsumes* the concept `GeoObject` because it explicitly formulates the constraint that type `T` also has to satisfy the concept `GeoObject`.

As a consequence, when overloading templates for both concepts and both are satisfied, we do not get an ambiguity. Overload resolution prefers the concept that subsumes the other(s):

```
template<GeoObject T>
void process(T)          // called for objects not providing setColor() and getColor()
{
  ...
}

template<ColoredGeoObject T>
void process(T)          // called for objects providing setColor() and getColor()
{
  ...
}
```

Constraint subsumption only works when concepts are used. There is no automatic subsumption when one concept/constraint is more special than the other.

Constraints and concepts do *not* subsume based only on requirements. Consider the following example:[2]

```
// declared in some header:
```

[2] That was in fact the example discussed regarding this feature during standardization. Thanks to Ville Voutilainen for pointing that out.

```
template<typename T>
concept GeoObject = requires(T obj) {
                       obj.draw();
                     };

// declared in another header:
template<typename T>
concept Cowboy = requires(T obj) {
                    obj.draw();
                    obj = obj;
                  };
```

Consider we overload a function template for both, `GeoObject`'s and `Cowboys`'s:

```
template<GeoObject T>
void print(T) {
  ...
}

template<Cowboy T>
void print(T) {
  ...
}
```

We do not want that for a `Circle` or `Rectangle`, which have a `draw()` member function, the call to `print()` prefers the `print()` for cowboys, just because the `Cowboy` concept is more special. We want to see that there are two possible `print()` functions that in this case collide.

The effort to check for subsumptions is only evaluated for concepts. Overloading with different constraints is ambiguous if no concepts are used:

```
template<typename T>
requires std::is_convertible_v<T, int>
void print(T) {
  ...
}

template<typename T>
requires (std::is_convertible_v<T, int> && sizeof(int) >= 4)
void print(T) {
  ...
}

print(42);   // ERROR: ambiguous (if both constraints are true)
```

When using concepts instead, this code works:

```
template<typename T>
requires std::convertible_to<T, int>
void print(T) {
```

```
  ...
}

template<typename T>
requires (std::convertible_to<T, int> && sizeof(int) >= 4)
void print(T) {
  ...
}

print(42);    // OK
```

One reason for this behavior is that it takes compile time to process dependencies between concepts in detail.

The concepts provided by the C++ standard library are carefully designed to subsume other concepts when it makes sense. In fact, they build a pretty complex subsumption graph. For example:

- `std::random_access_range` subsumes `std::bidirectional_range`, both subsume the concept `std::forward_range`, all three subsume `std::input_range`, and all of them subsume `std::range`. However, `std::sized_range` does only subsume `std::range` and none of the others.
- `std::regular` subsumes `std::semiregular` and both subsume both `std::copyable` and `std::default_initializable` (which subsume several other concepts such as `std::movable`, `std::copy_constructible`, and `std::destructible`).
- `std::sortable` subsumes `std::permutable` and both subsume `std::indirectly_swappable` for both parameters being the same type.

4.7.1 Indirect Subsumptions

Constraints can even subsume indirectly.[3] That means, overload resolution can still prefer one overload or specialization over the other although their constraints are not defined in terms of each other.

For example, consider you have defined the following two concepts:

```
template<typename T>
concept RgSwap = std::ranges::input_range<T> && std::swappable<T>;

template<typename T>
concept ContCopy = std::ranges::contiguous_range<T> && std::copyable<T>;
```

Now when we overload two functions for these two concepts and pass an object that fits both concepts, this is no ambiguity:

```
template<RgSwap T>
void foo1(T) {
  std::cout << "foo1(RgSwap)\n";
}

template<ContCopy T>
```

[3] Thanks to Arthur O’Dwyer for pointing this out.

```
void foo1(T) {
  std::cout << "foo1(ContCopy)\n";
}

foo1(std::vector<int>{});    // OK: both fit, ContCopy is more constrained
```

The reason is that ContCopy subsumes RgSwap because

- Concept contiguous_range is defined in terms of concept input_range (it implies random_access_range, which implies bidirectional_range, which implies forward_range, which implies input_range).
- Concept copyable is defined in terms of concept swappable (it implies movable, which implies swappable).

However, with the following declarations we get an ambiguity when both concepts fit:

```
template<typename T>
concept RgSwap = std::ranges::sized_range<T> && std::swappable<T>;

template<typename T>
concept ContCopy = std::ranges::contiguous_range<T> && std::copyable<T>;
```

The reason is that neither the concept contiguous_range implies sized_range nor the concept sized_range implies contiguous_range.

Also, for the following declarations, no concept subsumes the other:

```
template<typename T>
concept RgCopy = std::ranges::input_range<T> && std::copyable<T>;

template<typename T>
concept ContMove = std::ranges::contiguous_range<T> && std::movable<T>;
```

On one hand, ContMove is more constrained because contiguous_range implies input_range; however, on the other hand, RgCopy is more constrained because copyable implies movable.

To avoid confusion, do not make too many assumptions about concepts subsuming each other. When in doubt, specify all the concepts you require.