Concepts in-place syntax

Document Number: **P0745 R1**
Date: 2018-04-29
Author: Herb Sutter (hsutter@microsoft.com)
Audience: EWG, CWG

Abstract. This paper proposes the ability to introduce a constrained type name in any place deduction occurs. In Jacksonville, this proposal was encouraged 29-21-11-4-4 with no polled design changes.

R1: Adds proposed wording.

Contents

1 Overview

1.1 Background and motivation

Unifying generic and ordinary programming using the terse (a.k.a. abbreviated, natural, etc.) constraint syntax has been a key goal since concepts were first presented to the committee. The original 2003 concepts paper N1536 (Stroustrup, Dos Reis) included a section on this syntax, and ever since the concepts designers have expressed the view that concepts was about more than just constraints.

Although draft C++0x concepts and later N3351 did not mention the terse syntax, the concepts designers accepted C++0x concepts as a point-in-time interim step knowing that the terse syntax could be layered on to get to "full concepts." Given that we are now some 10 years beyond C++0x concepts, and have been working on concepts within WG21 for 14 years (and longer before WG21), there is a greater sense of urgency to finally get "full concepts" including the unification of generic and ordinary programming with the terse syntax, and not decouple the terse syntax for long.

The Toronto meeting Tuesday evening session had over 70 experts present, and conducted two main polls:

- There was a strong consensus to merge the Concepts working paper (as amended by other changes approved earlier in the day) into the C++20 working draft, but excluding "for now" the introducer and terse syntaxes along with constrained variables and constrained non-type template parameters.
- There was overwhelming support to continue to pursue a terse syntax that could overcome the concerns with the Concepts TS terse syntax, especially that function templates be visually distinct and that deduction of the same concept default to independent-type semantics.

This paper presents a design for a terse syntax for applying concepts that:

- builds directly from the terse and introducer syntax in the Concepts TS and Stroustrup's P0694R0 (embraces the TS introducer syntax and allows it generally throughout the language, not just in one place);
- attempts to address the concerns about the Concepts TS terse and introducer syntaxes, especially that function templates be visually distinct and that deduction of the same concept default to independent-type semantics; and
- deliberately leaves the door open to getting exactly the Concepts TS terse parameter, introducer, and constrained variable and non-type parameter syntaxes and semantics (except only for the TS's same-type semantics default for parameters) in the future by adding a simple default if the committee is comfortable doing so once we gain further experience with the feature.

In particular, the difference between the Concepts TS and this paper's proposal is often "just `{}`", as in:

Concepts TS and P0694R0	**This paper (proposed)**
`void sort(Sortable& s);`	`void sort(Sortable{}& s);`

See §4 for many side-by-side examples, including all of the main P0694R0 examples.

This paper builds on previous papers including the original Stroustrup and Dos Reis N1536 (2003) concepts proposal which did include the terse syntax, the Concepts TS (2015), Ballo and Sutton's N3878 (Jan 2014), Smith and Dennett's P0587 (Feb 2017), Vandevoorde's [Van17] (Mar 2017 EWG presentation), Van Eerd and Ballo's P0464R2 (Mar 2017), Brown's N4434 (Apr 2015), Stroustrup's thorough treatments in P0694R0 and P0695R0 (Jun 2017), Honermann's P0696R0 (June 2017), and Spicer, Tong, and Vandevoorde's P0691R0 (June 2017).

1.2 Design principles

Note These principles apply to all design efforts and aren't specific to this paper. Please steal and reuse.

The primary design goal is conceptual integrity [Brooks 1975], which means that the design is coherent and reliably does what the user expects it to do. Conceptual integrity's major supporting principles are:

- **Be consistent:** Don't make similar things different, including in spelling, behavior, or capability. Don't make different things appear similar when they have different behavior or capability.—For example, this paper follows the principle in the Concepts TS and Stroustrup's P0694R0 that **generic programming and ordinary programming should be consistent**, especially that writing the boilerplate text "`template<>`" should not be needed for many generic algorithms, and that `auto` be the least constrained concept. It diverges from the Concepts TS where there are three different "short" constraint syntaxes (besides explicit `requires`). Also, this paper's proposed constrained type introducer can be used to **constrain any deduced type (after deduction)**, including a type parameter, auto variable, or auto return type.
- **Be orthogonal:** Avoid arbitrary coupling. Let features be used freely in combination.—For example, in this paper we follow the principle that **applying a constraint should not change the meaning of code**; applying a constraint should (only) reject a type that does not match that constraint, not change semantics in any other way. This principle is essential to preserving that `auto` is in fact the least constrained concept (otherwise replacing `auto` with a more-constrained concept would have other semantic effects than just increasing the constraint), and avoids adding surprises such as that merely adding or changing a constraint (changing `auto` to a concept name, or changing one concept name to another) could sometimes make a function be a template and sometimes not, sometimes make a two-parameter heterogenous function be usable only homogeneously, sometimes make a return type not be deduced from the return-expression, and so on. All of these would be sources of user surprise, add contextual meaning that is hostile to refactoring, and have the usual other negative consequences of language inconsistencies.
- **Be general:** Don't restrict what is inherent. Don't arbitrarily restrict a complete set of uses. Avoid special cases and partial features.—For example, this paper suggests that since the Concepts TS allows single-type concept constraints in `template<>` parameter lists, we should **allow all constraints** there, including multi-type concept constraints, as was previously suggested in N3878 (Ballo and Sutton). This paper also proposes a **general way to introduce constrained type names "on the fly"** at point of use.

These also help satisfy the principles of least surprise and of including only what is essential, and result in features that are additive and so directly minimize concept count (and therefore also redundancy and clutter).

1.3 Acknowledgments

Thanks to the following for their feedback, including all authors of recent papers in this area: Botond Ballo, Walter Brown, Chandler Carruth, Casey Carter, James Dennett, Gabriel Dos Reis, Hal Finkel, Tom Honermann, Erich Keane, Robert Klarer, Thomas Köppe, Adam Martin, Eric Niebler, Arthur O'Dwyer, Jakob Riedle, Barry Revzin, Richard Smith, John Spicer, Bjarne Stroustrup, Andrew Sutton, Daveed Vandevoorde, Tony Van Eerd, and Ville Voutilainen.

2 Proposal overview

2.1 `Concept{T1,T2}` introduces constrained type names

Today, a concept name followed by a list of type arguments in `<>` checks the constraint:

```
ConceptName < identifier-list >     // checks one or more already-existing type names
```

Note This paper neither proposes nor precludes extensions to this syntax except as noted in §2.2.3.

This paper proposes a **constrained type introducer** where a type concept name followed by a list of type names in `{}` introduces the names, as in the Concepts TS introducer syntax (see §5.2 for the alternative of using `[]`):

```
ConceptName { identifier-list_opt }   // introduces zero or more constrained type names
```

which can be used to introduce constrained type names "on the fly" within existing grammar elements (notably, in template parameter lists and in declarations of parameters, return types, and local variables).

Future simplification → Concepts TS terse syntax via a default

In the future, we could add: "where if there are no introduced names we can also omit `{}`".

It is deliberate that if we added that, we would have the current Concepts TS terse syntax. My aim is to construct a syntax that works well now and also gives us a path to adopting the Concepts TS terse syntax in the future, if the committee gains comfort with viewing empty `{}` as unnecessary once we gain experience with the feature.

Notes This proposal is for concepts whose parameters are all types.

If you see an introducer `Concept{T}`, you know `T` is introducing a name for a type that is being *deduced* and then *constrained* to `Concept`.

This proposal is not currently proposing `auto{/*name*/}` to be able to give a name to the type of a template non-type parameter, a parameter of template type parameter type, or an ordinary variable of deduced type. This could be proposed in the future as a compatible extension, and would be useful in cases like `[](auto{T}&& x) { return f(std::forward<T>(x)); }`.

This proposal is not currently proposing nesting of `Concept1{Concept2{/*...*/}}`. This could be proposed in the future as a compatible extension if it is found useful, such as to mean introducing a type name that satisfies both constraints.

This proposal is not currently proposing introducing variadic names (e.g., `Concept{Ts...}`). This could be proposed in the future as a compatible extension if it is found useful.

Introducing constrained type names has the following semantics:

- Multiple introductions of the **same type name** introduce the **same type**, whether it is constrained by the same concept or different concepts. It is the same name, therefore must be the same type. The type must satisfy all the constraints in which it is mentioned.
- Introducing **different type names** introduces **independent types**, whether they are constrained by the same concept or different concepts. They are different names, therefore need not be the same type.
- Name(s) are not required; introduce a name only if it's needed again (such as to declare another variable of the same type). When omitted, the semantics are that the compiler invents an unspecified unique name, as with other unnamed entities (e.g., lambdas). Therefore it follows that introducing **anonymous unique [therefore different] type names** introduces **independent types** (see §5.1.5).

2.2 Illustrative examples of `Concept{/*...*/}`

2.2.1 In template declarations and definitions

For class templates, variable templates, and alias templates, here is how to use the syntax to introduce constrained type names in the template parameter list:

```
template<Number{T}>                         // introduce the type name (in template arg list)
class matrix { vector<T> v; /*...*/ };      // use the type name
// means: template<class T> requires Number<T> /*etc.*/

template<FloatingNumber{N}>                 // introduce the type name (in template arg list)
constexpr N pi = N{3.14159265358979323L};   // use the type name
// means: template<class N> requires FloatingNumber<N> /*etc.*/

template<Number{N}>                         // introduce the type name (in template arg list)
using T = something<N>;                     // use the type name
// means: template<class N> requires Number<N> /*etc.*/
```

For function templates, we can introduce constrained type names in the template parameter list:

```
template<Mergeable{In1, In2, Out}>          // introduce the type names (in template arg list)
Out merge(In1, In1, In2, In2, Out);         // use the type names
// means: template<class In1, class In2, class Out> requires Mergeable<In1, In2, Out> /*etc.*/
```

and in a parameter or return type position (if the name is unneeded and omitted, the language as-if invents one):

```
void f(Number{N} n, Sortable{S}& s) {       // introduce the type names (in parameters)
    N t = n + something();                  // use the type name
    Value_type<S> val = *s.begin();         // use the type name
}
// means: template<class N, class S> requires (Number<N> && Sortable<S>) /*etc.*/

Input_iterator{I}                           // introduce the type name (in prefix return type)
find(I, I, Equality_comparable<Value_type<I>>{}); // use the type name
// means: template<class I> requires Input_iterator<I> /*etc.*/

auto multiply(Number{N}, N) -> Number{}     // constrain the return type (in trailing return)
    { /*... use the type name ... */
// means: template<class N, class _T1> requires Number<N> && Number<_T1> /*etc.*/

void sort(Sortable{}&);
// means: template<class _T> requires Sortable<_T> /*etc.*/
```

If the same type name is introduced more than once in the scope of a single template declaration or definition, the type must meet all the constraints. For example:

```
void f(Number{N}, Number{N});
void f(Number{N}, N);                       // same except conversions allowed on second param
// means: template<class N> requires Number<N> /*etc.*/
```

and in the following `N` must satisfy two constraints:

```
void g(Number{N}, Addable{N});
// means: template<class N> requires (Number<N> && Addable<N>) /*etc.*/
```

Type parameters and non-type parameters are made unambiguous by consistently requiring {} when introducing a type name, including in constrained type parameters (where the Concepts TS and C++20 working draft currently support the syntax without {}; this proposal reserves that syntax for non-type parameters). For example:

```
template<class  N,  auto     a_value> void f();            // unconstrained

template<Number{N}, Number{} a_value> void f();            // constrained
           // ^ ^  with these {} required around a type name, this is unambiguous
```

Notes This does not preclude removing empty {} in the future, which is still an unambiguous syntax (note that the parameter name must be required):

```
template<Number{TypeName}, Number Value> void f();  // if empty {} were optional
```

Also, it avoids the current visual ambiguity where the constrained template would be written:

```
template<typename T> concept Number = /*...*/;      // type concept
template<int T> concept Numeric = /*...*/;   // value concept

template<Number TypeName, Numeric Value> void f(); // current syntax ambiguous
```

where it is not possible to see that one is a type parameter and one is a value parameter without consulting the definitions of the two concepts. This paper proposes a general constraint introduction syntax for type concepts only, and the removal of the ambiguous syntax `template<Number TypeName, Numeric Value>`. For a discussion of value concepts and paths where those lead, see §5.4.

2.2.2 In function (or function template) bodies (variables)

For example, here is how to use the syntax to constrain the type of a variable:

```
Number{N} n = 42;      // n is an int, and N is int

Number{}  n = 42;      // same, except not introducing the name N
```

2.2.3 Using <> and {} together: Bound arguments via currying

Note This subsection is an independent detail on which nothing else depends.

In this proposal, using `Concept<>` with N arguments binds the first N parameters of `Concept`. For example, `ComparableTo<X><Y>` is equivalent to `ComparableTo<X,Y>`.

In this proposal, using `Concept{}` with N introduced names binds the first N parameters of `Concept`. For example, `ComparableTo{X}{Y}` is equivalent to `ComparableTo{X,Y}`.

When the end of a series of <> and {} is reached, the total number of all arguments and introduced names must match or exceed the number of non-defaulted parameters of `Concept`.

This lets us mix <> and {} more freely and consistently. For example, given this concept:

```
template <class T, class U>
concept ComparableTo = requires(T const& t, U const& u)
    {  { t == u } -> bool;   { u == t } -> bool;  };
```

we can express introducing two names `T` and `U` that satisfy `ComparableTo` like this:

```
template <ComparableTo{T,U}>     // means: template <class T, class U> requires ComparableTo<T,U>
```

and distinctly express introducing a name that is `ComparableTo` an existing type name `X` by using both <> and {}:

```
template <ComparableTo{T}<X>>    // means: template <class T> requires ComparableTo<T,X>
template <ComparableTo<X>{U}>    // means: template <class U> requires ComparableTo<X,U>
```

Notes Some of this syntax, such as A<B>{C}, just falls out of the existing grammar with the proposed {} extension, since A<B> is a concept name and we're just applying the {} syntax to that as in the base case.

When combining <> and {}, to express the constraint

```
template<class T> requires ComparableTo<T,X>     // direct <> order: T X
```

in the TS and the C++20 working paper we currently would write this as:

```
template<ComparableTo<X> T>                      // TS and WP order: X T (reversed)
```

but this proposal suggests making the order consistent while we have the chance, before casting the current syntactic inversion in stone in the standard, and instead write this as:

```
template<ComparableTo{T}<X>>                     // proposed order: T X
```

which makes ComparableTo<T,X> and Comparable{T}<X> consistent.

2.2.4 Overview comparison with Concepts TS (see §4 for detailed examples)

The Concepts TS has three "short" constraint syntaxes (besides requires) that are all different with different restrictions, shown below. The only one that is fully general is the introducer syntax in its special grammar position. This paper proposes eliminating the special grammar position, but embracing the introducer syntax and applying it generally and directly within the current grammar to constrain the places where we already write typename or auto to introduce a type.

This supports strictly more uses of a non-requires syntax (it appears to eliminate the need to resort to requires), with consistent syntax and no novel grammar position. For example, it avoids the Concepts TS and P0694R0 limitation that you can have multiple constraints (in the template<> parameter list, #1 below), or you can have a multi-type constraint (in the special introducer syntax, #3 below), but you cannot have both (without resorting to low-level requires which is undesirable), which violate both consistency and generality.

Concepts TS and P0694R0	**This paper (proposed)**

`// extended syntax 1: no multi-type constraints`
`template</*Mergeable ???*/>`
`void merge(In1, In1, In2, In2, Out);`
	`// extended syntax 1: multi-type constraints ok`
	`template<Mergeable{In1, In2, Out}>`
	`void merge(In1, In1, In2, In2, Out);`

`// extended syntax 2: no ability to name the type`
`void sort(Sortable& s) {`
`  // using S = remove_reference_t<decltype(s)>;`
`}`
	`// extended syntax 2: optionally naming type ok`
	`void sort(Sortable{S}& s) {`
	
	`}`

`// novel syntax 3: only one constraint allowed`
`Sortable{S} /*Number{N}*/`
`void sort(S& s, N pivot);`
	`// n/a, use 1 or 2 above`

`// (TS only) syntax 1 ambiguous and inconsistent`
`// for constrained non-type parameters`
`template<Number Type, Number Value>`
`void f(Number parameter) {`
`  Number variable;`
`}`
	`// introduced type names are always inside {}`
	`// nontype/parm/var names are always outside {}`
	`template<Number{Type}, Number{} Value>`
	`void f(Number{} parameter) {`
	`  Number{} variable;`
	`}`

3 Proposal

3.1 Part 1: Allow all constraints in template parameter lists

3.1.1 Proposal

For template parameter lists, allow all constraints (including multi-type constraints) where the semantics for type parameters are identical to today's `class` or `typename`, and for non-type parameters are identical to today's `auto`, except that additionally a deduced type can be constrained and (optionally) be named.

In a template parameter list (including of a template template parameter or generic lambda), permit the constrained type introducer `Concept{T1 /*, ... Tn*/}` as an entry in the list. The following rules apply:

- The list of introduced names must be nonempty (`n>0`).
- The semantics are the same as:
  - replace `Concept{T1 /*, ... Tn*/}` with `typename T1 /*, ... typename Tn*/` for each `Ti` that does not already appear as a template type parameter in the same list (possibly via another constrained type introducer); and
  - add (or extend) the requires-clause with `requires Concept<T1 /*, ... Tn*/>`.

In a template parameter list (including of a template template parameter or generic lambda), permit the constrained type introducer `Concept{/*T*/}` as the type of a non-type (value) parameter in the list. The following rules apply:

- The list of introduced names must have 0 or 1 entries (`n<=1`).
- The name of the parameter is required.
- If the list is empty, the compiler invents an unspecified unique name for `T`.
- The semantics are the same as:
  - replace `Concept{T}` with `auto`;
  - make `T` an alias for the deduced type of the non-type parameter (if that name appears elsewhere in the list, the deduction must agree with the other uses); and
  - add (or extend) the requires-clause with `requires Concept<T>`.

For example, to illustrate the main cases, writing this template parameter list:

```
template<
    Sortable{S},
    Mergeable{In1, In2, Out},
    template<Iterator{I}> class X,
    Number{N} MaxSize
    >
```

is equivalent to writing this:

```
template<
    typename S,
    typename In1, typename In2, typename Out,
    template<typename I> requires Iterator<I> class X,
    auto MaxSize  /*, typename N = decltype(MaxSize)*/ /* exposition only */
    >
```

```
requires
    Sortable<S> &&
    Mergeable<In1, In2, Out> &&
    Number<N>
```

And writing this:

```
template<
    Mergeable{In1, In2, Out},
    RandomAccessIterator{Out},
    Number{} MaxSize
    >
```

is equivalent to writing this:

```
template<
    typename In1, typename In2, typename Out,
    auto MaxSize  /*, typename _T = decltype(MaxSize)*/ /* exposition only */
    >
requires
    Mergeable<In1, In2, Out> &&
    RandomAccessIterator<Out> &&
    Number<_T>
```

Writing this:

```
template<class Out, Mergeable{In1, In2, Out}>
void f(In1, In2, Out);
```

is equivalent to writing this:

```
template<class Out, class In1, class In2>
    requires Mergeable<In1, In2, Out>
void f(In1, In2, Out);
```

Similarly for generic lambdas, writing this:

```
auto f = []<Sortable{S}, Mergeable{In1, In2, Out}> (S, In1, In2, Out) { };
```

is equivalent to writing this:

```
auto f = []<typename S, typename In1, typename In2, typename Out>
              requires Sortable<S> && Mergeable<In1, In2, Out>
         (S, In1, In2, Out) { };
```

3.1.2 Discussion

The Concepts TS and the current working paper allow single-type constraints in the `template<>` list as follows:

```
template<Sortable S>                // same meaning in Concepts TS and C++20 WP
void sort(S&);

// means: template<class S> requires Sortable<S> /*etc.*/
```

However, allowing only single-type constraints is an arbitrary restriction that violates the principle of generality.

Also, the lack of any decoration creates a visual ambiguity with non-type parameters; for example,

```
template<Number N> void f();        // different meanings in Concepts TS and C++20 WP

  // means this:  template<typename N> requires Number<N>           void f();
  //    or this:  template<auto N>     requires Number<decltype(N)> void f();
```

where the TS assigns the first meaning if `Number` takes a type parameter, and the second if it takes a non-type parameter, in addition to the ordinary non-type parameter meaning if `Number` happens to be a type rather than a concept. This is a three-way visual ambiguity at the call site. Furthermore, in the second meaning it does not permit a direct way to name the type (the user must resort to a later `decltype` as today). This proposal avoids these ambiguities and inconsistencies.

Like N3878, this paper proposes that the `template<>` parameter list allow all constraints including multi-type constraints using the same syntax as the Concepts TS introducer syntax, which requires `{}`. For example:

```
template<Mergeable{In1, In2, Out}>        // proposed (not allowed in Concepts TS)
Out merge(In1, In1, In2, In2, Out);

// means: template<class In1, class In2, class Out> requires Mergeable<In1, In2, Out> /*etc.*/
```

Future simplification → Concepts TS introducer syntax via a default

In the future, we could add: "where if there is only one item in `<>` we can also omit `template<>`".

It is deliberate that if we added that, we would have the current Concepts TS introducer syntax. Because the EWG Toronto polls were to progress concepts without the introducer and terse syntaxes, and continue to work on the terse syntax, I am focusing on the terse syntax again. But this design deliberately aims to be able to achieve both introducer and terse syntaxes as they appear in the Concepts TS if we want them in the future, and as special cases of a natural generalization (reducing from the four distinct syntaxes allowed by the Concepts TS).

Template template parameters are handled recursively:

```
template< template<Mergeable{In1, In2, Out}> class X >
Out do_merge(X);

// means: template< template<class In1, class In2, class Out> requires Mergeable<In1, In2, Out> class X > /*etc.*/
```

The proposed generalization encourages users to write better code by removing reasons to resort to `requires`-clauses and needless concepts proliferation. As described in N3878, the Concepts TS currently has no syntax for expressing a constrained template that has a multi-type constraint and another constraint, without resorting to a low-level `requires` clause, or inventing a one-off function-specific named concept as an ugly workaround that essentially nobody will actually do. For example, we should be able to write:

```
// 1: illegal in Concepts TS, proposed herein
template <Mergeable{In1, In2, Out}, SortableIterator{Out}>
void merge_then_sort(In1 first1, In1 last1, In2 first2, In2 last2, Out out);
```

but this cannot be expressed today without forcing the user to resort to a `requires`-clause (which I argue is lower-level and less disciplined; it also makes conjunctions explicit which opens the door for the user to as easily write a disjunction and I'd rather remove that temptation):

```
// 2: drop to using requires-clauses (lower-level; Concepts TS style with requires)
template <typename In1, typename In2, typename Out>
```

```
    requires Mergeable<In1, In2, Out> && SortableIterator<Out>
void merge_then_sort(In1 first1, In1 last1, In2 first2, In2 last2, Out out);
```

or alternatively resort to writing an algorithm-specific concept (which I argue is undesirable and leads to verbosity and name littering) and using introducer syntax:

```
// 3: proliferate one-off concepts (Concepts TS style with introducer syntax)
template <typename In1, typename In2, typename Out>
concept MergeableAndOutIsSortable
    = Mergeable<In1, In2, Out> && SortableIterator<Out>;

MergeableAndOutIsSortable{In1, In2, Out}
void merge_then_sort(In1 first1, In1 last1, In2 first2, In2 last2, Out out);
```

3.2 Part 2: Constrained type names in parameter/return values

3.2.1 Proposal

For parameter and return values, the semantics are identical to today's template type parameters and `auto` return types except that additionally a deduced type can be constrained and (optionally) be named. Adding a constraint to an `auto` return type does not affect whether or not the return type is deduced from `return` statements, and does not affect whether or not the function is a template.

Replacement for a constrained return type is performed before replacement for a constrained parameter. This ensures that the choice of leading or trailing return type syntax does not change the semantics of the function template, in particular to allow deduction on constrained parameter types whether the constrained return type is declared lexically first or last (see example below).

In the declaration of a parameter type for a function template (which could be a lambda) or a template argument of such a type (e.g., `vector<Number{N}>`), permit the constrained type introducer `Concept{/*T*/}` as the type. The following rules apply:

- The list of introduced names must have 0 or 1 entries (`n<=1`).
- If the list is empty, the compiler invents an unspecified unique name for `T`.
- The semantics are the same as moving the constraint into the template parameter list:
    - replace `Concept{T}` with `T`; and
    - add (or extend) the function template's template parameter list with `Concept{T}` (see §3.1).

In the declaration of a trailing return type for a function or function template (which could be a lambda), permit the constrained type introducer `Concept{}` as the type. The following rules apply:

- The list of introduced names must be empty (`n==0`).
- The compiler invents an unspecified unique name for `T`.
- The semantics are the same as normal return type deduction and asserting the concept at each return:
    - replace `Concept{}` with `auto`;
    - before each `return` statement, insert `static_assert(Concept<R>);` where `R` is the deduced return type.

Notes The `static_assert` intentionally implies that repeating a function or function template definition with different return type constraints is an ODR violation, even if the repeated signature and body are otherwise identical.

> Since C++14 we already have a precedent for the `static_assert`-like (hard-error) behavior of constrained return types, in contrast to the SFINAE behavior of constrained parameter types, in the semantics for `auto*`:
>
> ```
> auto* f(/*...*/) { /*...*/ }            // C++14
> ```
>
> In C++14, this constrains `f` to return a pointer, and is very similar to constraining a return type with an actual concept for `Pointer`. In C++14, this produces the same `static_assert`-like behavior, namely a hard error (not SFINAE) if violated.

In the declaration of a leading return type for a function template, permit the constrained type introducer `Concept{/*T*/}` as the type. The following rules apply:

- The list of introduced names must have 0 or 1 entries (`n<=1`).
- If the list is empty, the semantics are the same as moving the constraint to the trailing position:
  - change `Concept{} function_name(/*...*/)`
    to `auto function_name() -> Concept{}`.
- If the list is nonempty, then `T` must be used as a parameter type, and the semantics are the same as moving the constraint to the first deduced parameter and then referring to it in the trailing return type:
  - change `Concept{T} function_name(/*...*/, cv-qual T first_t_param, /*...*/)`
    to `auto function_name(/*...*/, cv-qual Concept{T} first_t_param, /*...*/) -> T`.

For example, writing this:

```
void f(Concept{T} a, Concept{U} b) { /* can use the names T and U here */ }
void g(Concept{}  a)               { /* if we don't need the name, can omit it */ }
void h(Concept{T} a, T b)          { /* same type */ }
```

is equivalent to writing this:

```
template<Concept{T}, Concept{U}> void f(T a, U b)   { /* can use the names T and U here */ }
template<Concept{_T}>            void g(_T a)       { /* name _T not available to the program */ }
template<Concept{T}>             void h(T a, T b)   { /* same type */ }
```

Similarly for generic lambdas, writing this:

```
auto f = [](Concept{T} a, Concept{U} b) { /* can use the names T and U here */ };
auto g = [](Concept{}  a)               { /* if we don't need the name, can omit it */ };
auto h = [](Concept{T} a, T b)          { /* same type */ };
```

is equivalent to writing this:

```
auto f = []<Concept{T}, Concept{U}> (T a, U b) { /* can use names T and U here */ };
auto g = []<Concept{_T}>            (_T a)     { /* name _T not available to the program*/ };
auto h = []<Concept{T}>             (T a, T b) { /* same type */ };
```

3.2.2 Discussion

Note that the syntax is now clear about what is a template parameter type in a signature. For example, consider:

```
void constrained_forwarder(X{T}&&);
```

In this proposal, we do not need to go look up `X` to see whether it is a type or a concept to know that this `&&` is a forwarding reference, not an rvalue reference.

Notes Another way of thinking about it is: If you see an introducer `{T}`, you know `T` is introducing a name for a type that is being deduced and then constrained.

In the future if we allow omitting empty `{}`, then we would have the syntactic ambiguity of the Concepts TS terse syntax regarding whether the parameter is a forwarding reference or an rvalue reference. Some people are concerned about that, some are not. The point of this proposal is that we no longer need to worry about that now, and can defer the decision regarding whether to allow omitting empty `{}` until a future time when we have more experience and familiarity with the syntax and may feel comfortable doing so.

Multiple redefinitions flirt of the same function or function template with ODR in the usual ways, with the addition that the `static_assert` creates an ODR violation if the same function or function template is defined twice with different constraints even if it is otherwise identical, because this:

```
Concept1{} foo(T x) {
    return x;
}
Concept2{} foo(T x) {                           // ODR violation
    return x;
}
```

is equivalent to this:

```
auto foo(T x) {
    static_assert(Concept1</*type of x*/>);
    return x;
}
auto foo(T x) {
    static_assert(Concept2</*type of x*/>);     // ODR violation
    return x;
}
```

This is an ODR violation even if all deduced return types always satisfied both constraints, which they might not.

3.3 Part 3: Constrained type names in function bodies

3.3.1 Proposal

For local variable declarations, the semantics are identical to today's `auto` except that additionally a deduced type can be constrained and (optionally) be named.

For a local variable declaration, permit the constrained type introducer `Concept{/*T*/}` as the type in positions where `auto` would be allowed today. The following rules apply:

- The list of introduced names must have 0 or 1 entries (`n<=1`).
- If the list is empty, the compiler invents an unspecified unique name for `T`.
- The semantics are the same as:
  - replace `Concept{T}` with `auto`;
  - add `using T =` the deduced type of the variable minus top-level cv- and ref-qualifiers; and
  - add the statement `static_assert<Concept{T}>;` following the variable declaration.

Note C++11 allows redeclaring the same alias in the same scope:

```
{
    using N = int;
    using N = signed int;        // ok
}
```

So for consistency and simplicity, the design above naturally and intentionally permits this:

```
Number{N} var1 = /*...*/;
// ...
Number{N} var2 = /*...*/;

Iterator{It} var1 = /*...*/;
// ...
RandomAccessIterator{It} var2 = /*...*/;
```

in the same cases `using` allows, which is when the second introducer re-deduces the same type.

For example, writing this:

```
Concept{T} n = init;         // initializer is required iff T is being newly introduced
```

is equivalent to writing this when the name `T` has not already been introduced to be visible at this point:

```
auto n = init;
using T = decltype(n);
static_assert(Concept<T>);
```

Note Each use is treated individually. A use either may be the first to introduce the name `T`, or may use an already-introduced name `T` and possibly add new constraints at that point.

If the name is omitted, an invented name is generated, and so writing this:

```
Concept{} n = init;          // initializer is required
```

is equivalent to writing this:

```
auto n = init;
using _T = decltype(n);
static_assert(Concept<_T>);
```

For example, with local variables, introducing a new name to get a new type or else reusing a type by its name:

```
Number{N} n = 42.2;                    // N is double, n is a double
Number{M} m = 18;                      // M is int, m is an int

Number{} n = 42.2;                     // n is a double
Number{} m = 18;                       // m is an int

Number{N} n = 42.2;                    // N is double, n is a double
N         m = 18;                      // N is double, m is a double
```

Future simplification → Concepts TS terse syntax via a default

As before, in the future we could add: “if there are no introduced names we can also omit `{}`”.

It is deliberate that if we added that, we would have the current Concepts TS terse syntax for constrained variables. As before, my aim is to construct a syntax that works well now and also gives us a

path to adopting the Concepts TS terse syntax in the future, if the committee gains comfort with viewing empty `{}` as unnecessary once we gain experience with the feature.

Note Even though C++ already supports `new auto(init)` and `new auto{init}`, it does not currently support `make_unique<auto>(init)` which should be preferred. Therefore this paper is not proposing allowing a constrained new-expression such as `new Concept{/*T*/}(init)` or `new Concept{/*T*/}{init}` until we have a way to both deduce and constrain general template arguments including the argument of `make_unique`. We don't want to create new reasons to use raw `new` by giving it capabilities that aren't supported by the safer smart pointers we want to recommend.

3.3.2 Discussion

Constraints that appear in the bodies of `if constexpr` blocks that evaluate `false` are not enforced. In this example, at most one of the nested constraints is enforced:

```
template<class C>
void f(const C& c) {
    Iterator{It} a = c.begin();
    /* use a */
    if constexpr (x) {
        RandomAccessIterator{It} b = c.begin();
        /* ... */
    } else {
        ForwardIterator{It} b = c.begin();
        It b = c.begin();
        /* ... */
    }
}
```

is equivalent to writing this:

```
template<class C>
void f(const C& c) {
    auto n = init;
    using _T = decltype(n);
    static_assert(Concept<_T>);

    auto a = c.begin();  using It = decltype(a);  static_assert(Iterator{It});
    /* use a */
    if constexpr (x) {
        auto b = c.begin();  using It = decltype(b);  static_assert(RandomAccessIterator{It});
        /* ... */
    } else {
        auto b = c.begin();  using It = decltype(b);  static_assert(ForwardIterator{It});
        /* ... */
    }
}
```

Also, this:

```
Integral{T} a = int16_t(0);

int main() {
    Integral{T} b = int32_t(0);
    if (true) {
        Integral{T} c = int64_t(0);
    }
}
```

is equivalent to this (i.e., the local `T` is unaffected by the global `::T` which it shadows as if the body of main contained a plain `using T = /*...*/;`):

```
auto a = int16_t(0);  using T = decltype(a);  static_assert(Integral<T>);

int main() {
    auto b = int32_t(0);  using T = decltype(b);  static_assert(Integral<T>); // shadows ::T
    if (true) {
        auto c = int64_t(0);  using T = decltype(c);  static_assert(Integral<T>);
                                                      // shadows ::T and (main)::T
    }
}
```

4 P0694R0 examples side by side

To illustrate, consider the P0694R0 examples in order, showing the examples side by side with the preferred expression of the example in P0694R0 on the left and the proposed version on the right. In some cases the latter uses the terse syntax, and in others the traditional (extended) template syntax.

Future simplification → Concepts TS terse, introducer, and variable syntaxes via defaults

Note that in this paper's proposal, if in the future we additionally allow dropping empty `{}` and `template<>` as noted earlier, all Concepts TS examples would be supported identically as in the TS except only for the same-type default on parameters.

4.1 sort(range)

From P0694R0 section 1:

P0694R0 style	This paper (proposed)
`void sort(Sortable&);`	`void sort(Sortable{}&);`

4.2 sort(iter, iter)

From P0694R0 section 3.1:

P0694R0 style	This paper (proposed)
`void sort(Random_access_iterator,` `  Random_access_iterator);`	`void sort(Random_access_iterator{I}, I);`

4.3 Variables in a block

From P0694R0 section 3.1, where the Concepts TS has independent-type but following P0694R0 which recommends same-type:

P0694R0 style	This paper (proposed)

`void use2(auto x, auto y) {`
`  Number xx = f(x);`
`  Number yy = g(y);`
`}`
	`void use2(auto x, auto y) {`
	`  Number{N} xx = f(x);`
	`  N yy = g(y);`
	`}`

4.4 Reusing the concept name in the body

From P0694R0 section 3.1, where the Concepts TS does not support the use of `Forward_iterator` in the body, and P0694R0 recommends same-type:

P0694R0 style	This paper (proposed)

`void sort(Forward_iterator b, Forward_iterator e,`
`  Relation<Value_type<Forward_iterator>> r)`
`{`
`  vector<Value_type<Forward_iterator>> v {b,e};`
`  sort(v);`
	`void sort(Forward_iterator{I} b, I e,`
	`  Relation{}<Value_type<I>> r)`
	`{`
	`  vector<Value_type<I>> v {b,e};`
	`  sort(v);`

```
    copy(v.begin(),v.end(),b);
}
```

```
    copy(v.begin(),v.end(),b);
}
```

4.5 Return types

From P0694R0 section 3.3:

P0694R0 style	This paper (proposed)
`Output_iterator copy(Input_iterator, Input_iterator, Output_iterator);`	`Output_iterator{O} copy(Input_iterator{I}, I, O);`

Alternatively, using the trailing return syntax:

P0694R0 style	This paper (proposed)
`auto copy(Input_iterator, Input_iterator, Output_iterator) -> Output_iterator;`	`auto copy(Input_iterator{I}, I, Output_iterator{O}) -> O;`

4.6 Same-type resolution cases

From P0694R0 section 3.3:

P0694R0 style	This paper (proposed)

`Value next(Value);`
`Value compute1(Value,Value);  // homogeneous op`
`Value compute2(Value,Other_value); // heterogeneous op`
`vector<Value> compute3(vector<Value>);`
	`Value{V} next(V);`
	`Value{V} compute1(V,V);  // homogeneous op`
	`Value{V} compute2(V,Other_value{}); // heterogeneous op`
	`vector<Value{V}> compute3(vector<V>);`

Alternatively, using the trailing return syntax:

P0694R0 style	This paper (proposed)

`auto next(Value) -> Value;`
`auto compute1(Value,Value) -> Value;`
`auto compute2(Value,Other_value) -> Value;`
`auto compute3(vector<Value>) -> vector<Value>;`
	`auto next(Value{V}) -> V;`
	`auto compute1(Value{V},V) -> V;`
	`auto compute2(Value{V},Other_value{}) -> V;`
	`auto compute3(vector<Value{V}>) -> vector<V>;`

4.7 find()

From P0694R0 section 3.4:

P0694R0 style	This paper (proposed)
`Input_iterator find(Input_iterator, Input_iterator, Equality_comparable<Value_type<Input_iterator>>);`	`Input_iterator{I} find(I, I, Equality_comparable<Value_type<I>>{});`

Alternatively, using the trailing return syntax:

P0694R0 style	This paper (proposed)
`auto find(Input_iterator, Input_iterator, Equality_comparable<Value_type<Input_iterator>>) -> Input_iterator;`	`auto find(Input_iterator{I}, I, Equality_comparable<Value_type<I>>{}) -> I;`

4.8 merge()

Only single-type concepts fit in the terse syntax, because the terse syntax is inherently for constraining a single type. This proposal allows multi-type concepts to work fine in `template<>` parameter lists.

From P0694R0 section 3.4:

P0694R0 style (using introducer syntax)	This paper (proposed)
`Mergeable{In1,In2,Out}` `Out merge(In1, In1, In2, In2, Out);`	`template<Mergeable{In1,In2,Out}>` `Out merge(In1, In1, In2, In2, Out);`

4.9 Different-type resolution: operator+() for Numbers

Here P0694R0 cannot use its terse syntax because it would use same-type resolution, and must resort to either a verbose traditional `template<>` syntax (which is not semantics-preserving because it changes deducibility) or invent duplicate concept names (which feels like a hack).

From P0694R0 section 3.6:

P0694R0 style	This paper (proposed)
`template<Number N1, Number N2 = N1, Number N3 = N1>` `N3 operator+(N1,N2);`	`Number{} operator+(Number{}, Number{});`

Note that in this case the left and right versions are not quite equivalent. The left-hand side does not deduce for `operator+({},x)` but does deduce for `operator+(y,{})`, so it is not actually a general method to get independent-type resolution if the default is same-type.

Alternatively, P0694R0 suggests giving the concept a different name, which proliferates one-off names as a workaround:

P0694R0 style	This paper (proposed)

`template<typename T> concept Number2 = requires Number<T>;`
`template<typename T> concept Number3 = requires Number<T>;`
`Number3 operator+(Number,Number2);`
	`Number{} operator+(Number{}, Number{});`

4.10 Multiple constraints: merge_then_sort (from §3.1.2)

Here P0694R0 cannot use its terse syntax, and must resort to either a verbose traditional `template<>` syntax or invent duplicate concept names.

From this paper, §3.1.2:

P0694R0 style, using requires	This paper (proposed)

`template <typename In1, typename In2, typename Out>`
`  requires Mergeable<In1, In2, Out> && SortableIterator<Out>`
`void merge_then_sort`
`  (In1 first1, In1 last1, In2 first2, In2 last2, Out out);`
	`template <Mergeable{In1, In2, Out}, SortableIterator{Out}>`
	`void merge_then_sort`
	`  (In1 first1, In1 last1, In2 first2, In2 last2, Out out);`

Alternatively, P0694R0 suggests giving the concept a different name, which proliferates one-off names as a workaround; this pollutes namespaces and is otherwise undesirable:

P0694R0 style, using introducer syntax	This paper (proposed)

`template <typename In1, typename In2, typename Out>`
`concept MergeableAndOutIsSortable`
`  = Mergeable<In1, In2, Out> && SortableIterator<Out>;`
`MergeableAndOutIsSortable{In1, In2, Out}`
`void merge_then_sort`
`  (In1 first1, In1 last1, In2 first2, In2 last2, Out out);`
	`template <Mergeable{In1, In2, Out}, SortableIterator{Out}>`
	`void merge_then_sort`
	`  (In1 first1, In1 last1, In2 first2, In2 last2, Out out);`

4.11 Constraints on both type and non-type parameters

In the Concepts, but not the current C++20 draft, a constraint can be applied to a non-type parameter and the meaning depends on the implementation of the concept. Constraining a non-type parameter uses an inconsistent special meaning that is different than constraining a parameter or local variable.

Concepts TS	This paper (proposed)

`template<Number Type, Numeric Value>`
`void f(Number parameter) {`
`  Number variable;`
`}`
	`template<Number{Type}, Number{} Value>`
	`void f(Number{} parameter) {`
	`  Number{} variable;`
	`}`

5 Discussion / Q&A

5.1 Why are unnamed constrained types independent?

This section addresses the question: If we introduce multiple unnamed constrained type names, should their types be the same, or be independent?

5.1.1 Proposal recap

As already noted, in this proposal, when an introduced type name is unnamed, the compiler invents an unspecified unique name. This is consistent with what we do in the rest of the language for unnamed entities (e.g., lambdas). Therefore, writing this:

```
void g(Concept{} a, Concept{} b) { /* if we don't need the names, can omit them */ }
```

is equivalent to writing this, where `_T1` and `_T2` are unique names invented by the compiler:

```
void g(Concept{_T1} a, Concept{_T2} b) { /* ... */ }
```

Similarly for generic lambdas, writing this:

```
auto g = [](Concept{} a, Concept{} b) { /* if we don't need the names, can omit them */ };
```

is equivalent to writing this:

```
auto g = [](Concept{_T1} a, Concept{_T2} b) { /* ... */ };
```

Because introducing different (user-written) type names introduces independent types, anonymous types with unique invented names are naturally independent too. I believe this consistency, together with the resulting natural expressiveness in the list of side-by-side examples in §4, is sufficient to justify independent-type semantics for unnamed constrained types. However, additional reasons for independent-type resolution appear below.

5.1.2 General problems with Concepts TS same-type default

The Concepts TS terse syntax requires same-type semantics for some cases, and P0694R0 argues for changing the TS semantics to add more cases. For example:

```
// Using syntax of the Concepts TS and P0694
auto f(Concept a, Concept b) -> Concept   // Concepts TS and P0694: same types
    requires Something<Concept>           // Concepts TS: not allowed
{                                         // P0694 suggestion: same type
    some_template<Concept> x = /*...*/;   // Concepts TS: independent type
}                                         // P0694 proposal: same type

auto g() {
    Concept xx = f();
    Concept yy = g();                     // Concepts TS: independent type
}                                         // N3701 and P0694 proposal: same type
```

This has several serious problems:

- **Consistency:** Whenever we do require same-type, we change semantics. For example, if the parameter types were `auto` or variadic `Concept...` then they would be independent in the Concepts TS; changing

semantic meaning between `(Concept,Concept)` and `(Concept...)`, or between `(Concept,Concept)` and `(auto,auto)`, is inconsistent. Making the list variadic should not change its basic semantics, and using `auto` as the least constrained "satisfied by any type" concept should not change the constraint's basic semantics.—As another example, if the return type were unconstrained, or did not use the same constraint as a parameter, the return type would be deduced from the return-expression; but if it happens to be constrained, and then only with a constraint also used to constrain a parameter, it is no longer deduced; I feel that too is inconsistent.

- **Incomplete / slippery slope:** Once we begin to implicitly say "some" uses of the same concept name must be the same type, but "not others," we will continue to find reasons to add more cases, as shown above. The Concepts TS added some same-type cases; P0694's main text proposes adding new cases, and its final section suggests exploring still more. Even if we want to pursue same-type, we're clearly not done specifying when we want it. This is a moving target where each addition is an ad-hoc patch.
- **Teachability / usability:** Teaching "sometimes using a concept name twice means same-type, sometimes it doesn't" is subtle—you "just have to know" and teach the rules by rote.

This paper advocates the core principle that **adding a constraint should not change the meaning of code**—there should be no change in semantics, only rejecting uses of a type that does not match the constraints. This principle is at the root of most of the (good) technical concerns raised above and in P0464R2: Violating this principle (by additionally saying that "in some cases" multiple uses of the concept are also the same type) is what leads to inconsistency with `auto`, inconsistency with dynamic polymorphism, inconsistency with variadic templates, inconsistency with return type deduction, and other problems.

5.1.3 Stroustrup's empirical STL analysis strongly favors independent-type default

For P0694R0 Stroustrup did extensive and important research work, multiple times, to carefully and thoroughly compile data and statistics from analyzing the current STL; aside from a similar analysis of the Ranges TS by its authors, no other paper authors, myself included, have made such a rigorous effort, and that effort and data is much appreciated—and should be given more weight than design papers with less data and therefore necessarily based more on personal design sensibility and conjecture.

Consider that research data from analyzing the STL. The same-type terse default was motivated by the observation that the current STL algorithms' parameters are dominated by `[first,last)` iterator pairs. However, iterator pairs are known to be a design flaw—any time we have two variables that share an invariant, we know we are missing an abstraction (by definition, because an invariant should be encapsulated within a single object). Thankfully that is being addressed with ranges; and, in part at my request during draft review of D0694R0 (and with great thanks), in P0694R0 Stroustrup additionally took time to perform a careful analysis of what the STL data would look like if we made just the single change of removing iterator pairs with a sequence (range) abstraction, and no other change. The results reported in P0694R0 are as follows (highlights added):

> 4.2 Using a sequence abstraction
>
> The pair-of-iterators idiom is largely responsible for the high number of repeated concepts. A roughly equivalent library based on a sequence idiom (where a sequence would be an object holding a pair of iterators or equivalent), the numbers would be:
>
> - 173 with no repeated concepts
> - 0 where all repeated concepts must be the same
> - 78 where all repeated concepts can be different
> - 0 where some repeated concepts can differ and some duplicate concepts must be the same

The initial conclusion from this data, as reported in P0694R0, was that 69% of the algorithms (category #1) are easily expressed using the terse syntax with same-type semantics. However, those same 69% would equally have been expressed without same-type semantics because by definition they contain no repeated concepts. A better characterization is that "*only* 69%" can be expressed in the terse syntax using the same-type default.

I believe the stronger conclusions from this data set are that:

- **100%** can be easily expressed using the terse syntax with independent-type semantics;
- **31%** (78) **require independent-type** semantics to be easily expressed using terse syntax; and
- **None** actually benefit from same-type semantics as the default.

> **Note** The "real" sequence-ized STL is the Ranges TS, which as reported in P0694R0 cannot use the terse syntax alone with same-type semantics (and likely not even with independent-type semantics).

Of course, everyone agrees that that there are examples that want to express same-type and examples that want to express independent-type; both need to be expressible (see especially §5.1.4 below). However, in the context of the empirical analysis of the STL, it is worth emphasizing that the above is the result after STL iterator pairs are removed—that is, the *only* class of example in our empirical data so far presented to argue in favor of the Concepts TS same-type as a default is STL iterator pairs. And even that example is simpler to express in this proposal (even with independent-type default) than in the Concepts TS with its same-type default:

Concepts TS (using same-type default)	**This paper (proposed)**
`void sort(Iterator first, Iterator last);`	`void sort(Iterator{I} first, I last);`

The TS design (left) has several problems, including that it is:

- more implicit and therefore opaque, because programmers must learn and remember that this is one of the cases where mentioning the same concept name twice imposes a same-type requirement so that `first` and `last` must be the same type;
- inconsistent, because mentioning the same concept name again does not have same-type semantics in other local uses; and
- slightly more verbose, because of the implicit redundancy in meaning.

I feel that this paper's proposed design (right) is better because it is:

- explicit and clear, because it is obvious that `first` and `last` have the same type for the simple and usual reason that they use the same type name;
- consistent, because in the function template the same name always implies the same type and a different name always implies an independent type; and
- less verbose, because it removes the implicit redundancy in meaning.

5.1.4 Complexity of opting out favors independent-type default

Further, even if the examples for same-type and independent-type were otherwise equally desirable (which I believe is not the case), whichever we choose as the default will require us to opt out when we want the other. With same-type as the default, to opt out of N uses of the concept to make them independent-type requires writing N names, whereas with independent-type as the default, to opt out of N uses of the concept to make them the same requires writing one name:

```
// Using Concepts TS syntax

// given this, with some default meaning (either same-type or independent-type)
Concept f(Concept, Concept, Concept);

// if same-type is the default and we want to opt to independent-type
template<Concept C1, Concept C2, Concept C3, Concept C4> C1 f(C2, C3, C4);

// if independent-type is the default and we want to opt to same-type
Concept{C} f(C, C, C);
```

Especially given that we have examples that benefit from same-type and examples that benefit from independent-type with no conclusive data that either is dominant in general use, optimizing the design to 'give the nice syntax to the common case' should include choosing the default that minimizes the overall opt-out burden.

5.1.5 This proposal: Convenient choice of either same-type or independent-type

In this proposal, returning to the earlier example, if you want same type then give the type a name and use its name:

```
// Proposed semantics: If you want same type, name the type and use the name

auto f(Concept{T} a, T b) -> T                  // same types
    requires Something<T>                       // same type
{
    some_template<T> x = /*...*/;               // same type
}
```

Note This seems about as concise a way to apply a concept constraint as possible over ordinary syntax; it simply names a type in-place and keeps going. Not that terseness is a virtue in itself, but this is by far the tersest way to say this in any of the alternatives proposed in papers so far including the Concepts TS terse syntax (and similar in terseness to [Van17]).

```
auto g() {
    Number{N} xx = f();
    N yy = g();                                 // same type
}
```

And if you want independent types, either give them different names, or don't name them if no name is needed:

```
// Proposed semantics: If you don't want same type, give them all different names
//                     or omit the name(s) that don't need to be referred to again

auto f(Concept{} a, Concept{} b) -> Concept{}   // independent types
    // requires Something<Concept>              // (n/a for independent type)
{
    some_template<Concept> x = /*...*/;         // independent type
}

auto g() {
    Number{} xx = f();
    Number{} yy = g();                          // independent type
}
```

Notes It is intentional that these rules work the same as if we had used the least constrained concept, `auto`.

The delta from today's terse syntax is just `{}`, and doubtlessly sooner or later someone will propose dropping the "unnecessary empty `{}`"—it is unclear whether the committee will ever be comfortable with this, but if it did adopt this we would be back to exactly the Concepts TS terse syntax except with independent-type resolution. Leaving this option available, while not using it immediately, is an explicit design goal—because of this, I believe that this paper's approach may be our best chance for (a) giving those people who are uncomfortable with zero syntactic differences what they want for now (i.e., a syntactic marker) while (b) keeping that difference very lightweight and (c) also leaving the door wide open to erasing that tersely later if and when we as a group gain familiarity and comfort with unifying generic and ordinary programming.

One design question is whether `Concept{T}` should allow default type arguments, e.g., `Concept{T = vector<int>}`. In parameters, the main use case that we have been able to find so far is if we want the caller to be able to pass `{}` as an argument, or if the parameter has a defaulted argument of `{}` or `T{}`—but the latter is predominant in the Ranges TS for projection and comparison operators. If we supported this, then the Ranges TS could change this

```
template <RandomAccessIterator I, Sentinel<I> S, class Comp = less<>, class Proj = identity>
  requires Sortable<I, Comp, Proj>
I sort(I first, S last, Comp comp = Comp{}, Proj proj = Proj{});
```

to this:

```
template <Sortable{I, Comp = less<>, Proj = identity}>
I sort(RandomAccessIterator{I} first, Sentinel<I>{} last, Comp comp = Comp{}, Proj proj = Proj{});
```

Summarizing side by side with the P0694R0 preference:

P0694R0 style	**This paper (proposed)**

`auto f(Concept a, Concept b) -> Concept  // implicitly same types`
`  requires Something<Concept>  // P0694: same type`
`{`
`  Something<Concept> x = /*...*/;  // P0694: same type`
`}`
`auto g() {`
`  Number xx = f();`
`  Number yy = g();  // P0694: same type`
`}`
	`auto f(Concept{T} a, T b) -> T  // visibly same type`
	`  requires Something<T>  // visibly same type`
	`{`
	`  Something<T> x = /*...*/;  // visibly same type`
	`}`
	`auto g() {`
	`  Number{N} xx = f();`
	`  N yy = g();  // visibly same type`
	`}`

5.2 Why no multi-type constraints in single-type contexts?

5.2.1 Parameters, return values, and local variables

This proposal deliberately does not propose allowing multi-type constraints on parameters, return values, or local variables, where the language by construction is talking about only a single type.

There is a temptation to try to extend declarations of multiple actual parameters/variables to allow multi-type constraints, but I could not find a way to do it naturally that did not make readability suffer. For example, I considered allowing a generalized multi-type parameter syntax that allows something like this:

```
void f(Concept{X x, Y y});                  // not proposed
```

as a synonym for this:

```
template<Concept{X, Y}> void f(X x, Y y);  // hypothetical meaning
```

However, for both parameters and true local variables we would then have to support at least default arguments/initializers and possibly cv-qualifiers too, such as:

```
void f(Concept{X x, Y y = y_init})
```

and the variable case where the initializer is mandatory since otherwise we have no type to deduce:

```
Concept{X x = x_init, Y y = y_init};
```

I worry that this last is quickly losing readability, and no longer looks like two local variable declarations.

Furthermore, to make this broadly useful, the order and number of type constraints in a concept and parameters or variables constrained by that concept would need to commonly agree, but examples where they do are scarce and uncompelling; instead, those orders and numbers are generally unrelated.

I do not view the restriction of using only single-type constraints in single-type contexts as a loss of generality in this proposal. In this proposal, multi-type constraints are universally allowed in all places where multiple types can exist or be introduced; and if the language in the future allows more places where multiple types can be named, such as multi-type variable introductions, then multi-type constraints can naturally be applied there too.

5.2.2 Structured bindings

This proposal also deliberately does not propose allowing multi-type constraints on structured bindings.

There is a temptation to try to allow N-ary multi-type constraints on N-ary structured bindings, such as:

```
Concept{X, Y, Z} [x, y, z] = init();
```

However, this apparent symmetry seems illusory, because again in general the order and number of type constraints is unrelated to the order and number of components of an aggregate, and the cases that "work" are usually forced instead of expressing something that programmers actually want to express.

However, as a future pure extension of this proposal, we could consider allowing single-type constraints on individual elements of a destructuring, for example:

```
auto [Number{X} x, y, Iterator{} z] = init();
```

which does not cause the problems that allowing a concrete type in that position would cause (i.e., the type is still deduced, there are no conversions, there is no pressure to additionally allow other things like cv-qualifiers).

5.3 Can we have a `template` prefix on function templates?

The proposed syntax already makes function templates visually distinct via `{}`, which addresses one of the objections to the original Concepts TS syntax.

A small number of reviewers of this paper have suggested additionally requiring the unadorned keyword `template`, without a parameter list, in front of function template declarations to make them more visually distinct. I worry about this for several reasons: One is that it treads very near the existing syntax we have for explicit instantiations. Another is that I don't think it has the desired benefits. The basic argument I've heard about why it may be desirable for the word "template" to be required throughout the language when writing a template is

twofold: (a) for consistency, and (b) to make it clearer still that function template rules apply in the body rather than normal function body rules. I do not find this argument for additionally requiring the word "template" convincing: for (a), the consistency argument fails because "template" is already not required for generic lambdas, which are (or generate) function templates; and for (b), not only do the `{}` in this proposal make templates visible, but additionally field experience over the past half-decade with C++14 generic lambdas (which also are visually distinct, with the keyword `auto`) has not brought any feedback that I am aware of to the effect that the absence of the word "template" somehow makes generic lambdas more difficult or brittle in practice. With generic lambdas, the presence of `auto` is sufficient unambiguous indicator that we are writing a template, and in this proposal the presence of `{}` is also a sufficient unambiguous indicator.

However, if and only if this proposal as written does not manage to get consensus, and just adding `template` would by itself make the difference in this proposal getting consensus, then we could tack on a "required `template`" wart that, like empty `{}`, we can consider removing later once we gain more familiarity with the feature. The result would be that all examples in this proposal are identical, with the exception of adding the boilerplate word, such as:

```
template void sort(Sortable{}&);
template void sort(Random_access_iterator{I}, I);
```

If we did add this keyword requirement, we could already anticipate the arguments in the future paper to remove the redundant `template`: that it adds no information (the code is unambiguous without it), that it is tedious boilerplate (a case of the compiler saying "I know exactly what you mean but I'm going to force you to write an extra word," one of the kinds of error messages most resented by C++ developers), and that removing it increases consistency in the language (with generic lambdas).

5.4 What about "value concepts" and "adjective syntax"?

Recall from §2.2.4 that in this proposal, unlike the Concepts TS and the C++20 working draft, we naturally support using the same concept to constrain both a type parameter and non-type parameter (the latter with the same syntax and semantics as value parameters and variables).

Some experts, including in Issaquah 2016 (Richard Smith) and papers P0791R0 (Riedle) and P0807R0 (Köppe), have suggested a generalization whereby we might use concepts to express both type constraints and value constraints, and because types and values can lexically appear in similar positions to have a syntax that requires `typename` for constrained types and `auto` for constrained variables, with concepts allowed as adjective prefixes to both. That is, to have `Concept typename T` constrain a type `T`, and `Concept auto v` constrain a variable `v` and be termed a "value concept." It is an interesting idea, though as the papers note it has not been deeply explored.

The immediate problem with such an approach is that is requires more tedious boilerplate (`typename` and `auto`) that is redundant in all current examples compared to this paper, in the name of a generalization that is unproven so that we can't evaluate whether it would really deliver more consistency throughout the language or the ability to express things that are harder or impossible to express today.

The more fundamental problem I see with this approach, and why I doubt the generalization can ever be a good fit for concepts, is that an object's type and its values are fundamentally distinct things:

- An object's type is invariant for the lifetime of the object. Therefore a type constraint naturally should appear on the type (i.e., on the object's declaration), and is a good match for a constraint that is logically

and lexically/visually applied as a type modifier, such as to state that variable `n` is-always-a `Number` for its entire existence.

- An object's value is a dynamic property that can change over the lifetime of the object, even for purely compile-time (and so "static" in a different sense) objects such as local variables in `constexpr` functions used to initialize `constexpr` values. Therefore a value constrain naturally should appear on the value (i.e., on expressions and statements that manipulate the value, including but not limited to the initial value at time of initialization which could happens to usually appear near the object declaration), and is not a good match for a constraint that is logically or lexically/visually attached to the object's type (e.g., a numeric variable `n` generally is-not-always-an `Even` number at all points of its lifetime, even if its initial values is even). Fortunately, we are already progressing the proposal that does provide a match, which is constraint via general contracts.

In short, concepts are to constrain types and naturally should annotate the type (i.e., object declarations), whereas contracts are to constrain values and naturally should annotate the values (i.e., expressions and statements). Because those two things have fundamentally different properties, I suspect any attempt to conflate them will make code less clear, not more clear.

For example, consider this example which appears in both P0791 and P0807 (updated to current WP syntax):

```
template<int N> concept Even = N%2 == 0;
```

and the proposed use as shown in those papers, contrasted what we would write with contracts (P0542):

```
// P0791/P0807 proposal: 'Even' is a type modifier (invariant property of i)
void f(Even int  i);
void g(Even auto i);

// P0542 contracts: 'is_even' is a check of i's value on entry
void f(int  i) [[expects: is_even(i)]] ;
void g(auto i) [[expects: is_even(i)]] ;
```

So far both may look plausible, though the contract syntax has the important property that it is logically and visually distinct from `i`'s type. This is makes it already more clear, more correct, and more general, because we see that `is_even` is a property of the value at a point in time only, not an invariant statement about `i`'s nature at all times.

But next, look into the function body: What if we wrote `++i` in `f`'s body—should that be allowed? The answer in P0791 is no, through dynamic checking (the answer in P0807 appears to be that this was not considered because the constraint was intended to be limited to checking an immutable value such as a template non-type parameter):

```
// P0791 proposal: 'Even' is a type modifier (invariant property of i)
void f(Even int i) {
    i += 2;     // P0791 intends this be allowed
    i += 3;     // P0791 intends this be a constraint violation
}

// P0542 contracts: 'is_even' is a check of i's value on entry
void f(int i) [[expects: is_even(i)]] {
    i += 2;     // ok
    i += 3;     // ok
```

```
}
```

Finally, let's rename `f` to `make_odd` (a function that takes an even `int` and returns an odd one):

```
// P0791 proposal: 'Even' and 'Odd' look like type modifiers (invariant properties of i/retval)
Odd int make_odd(Even int i) {   // ?
    ++i;                         // P0791 intends that this should be illegal, but that
    return i;                    // "return i+1;" be allowed, which feels like a programming
}                                // model that is more difficult to teach and refactor
// call site: Odd int i = f(2); ++i; // disallowed, same issue again here

// P0542 contracts: 'is_even' and 'is_odd' look like a check of i's value on entry
int make_odd(int i) [[expects: is_even(i)]] [[ensures: is_odd(i)]] {
    ++i;                         // ok (abd "return i+1;" would be ok too)
    [[assert: is_odd(i)]];       // alternative position for check
    return i;
}
```

With the second contract form, it's natural and logically clear that `is_even` is a property of the initial argument value on entry, and `is_odd` is a property of the returned value. I do not see a way that trying to wedge this information into a type modifier makes code clearer; though of course we could contort the language to force it to 'work,' I think any attempt to paper over this impedance mismatch will necessarily interfere with readability.

So I strongly support value constraints, but I think the generalization is to contracts (which already support this and so the generalization is already complete and done) rather than concepts (by inventing a novel attempted generalization that makes it appear to be an invariant property of the type when it is not).

Note I have corresponded at length with the author of P0791R0 (Riedle) and he understands this. It turns out the motivation for P0791R0 is not about concepts per se, but rather that the author objects to the *contracts* proposal P0542 on the grounds that the contracts are too low-level (granular code in attributes) and can be disabled (by changing contract checking levels), whereas he feels it is essential that contracts be able to be pushed up to the call site.

In competition with P0542 contracts, the intention of P0791R0 actually is to try to force contracts instead into the type system as type qualifiers so that they are part of the signature of a function, that callers cannot turn them off, and that they be pushed to the call site. The intention is also to enable powerful Haskell-like advanced overloading such as this:

```
// Example provided by J. Riedle: The goal is that overload resolution
// would see that an input value is Empty and if so call the no-op

auto quick_sort( Empty Range auto& rng ){ return std::move(rng); }
auto quick_sort( NonEmpty Range auto rng ){ /**/; }
```

and enable optimization based on run-time values such as this:

```
// Example provided by J. Riedle: The goal is that if the input matrix
// is found to be already Symmetric, this function can become a no-op

concept_cast<Symmetric matrix>( my_matrix );
```

These ideas may be interesting, though I have concerns about the viability of this approach (for example, it would require introducing a dynamic component to overload resolution which is certainly possible but that we do not have today and would require careful thought), but I don't think they

should influence our concept constraint design because I don't believe anything in this proposal rules out pursuing such a direction in the future, while on the other hand I don't think this direction is sufficiently developed to justify pursuing a more verbose concept constraint design now in anticipation of this approach maturing soon.

One reviewer asked: "I'm having trouble squaring these thoughts with section 2.2.1 of your proposal in which you retain the ability to constrain a non-type template parameter via use of a concept as a type specifier."

The answer is that that's totally different—in 2.2.1 it's just a normal type concept and type constraint. Consider:

```
// unconstrained
template<auto Size>
void f() {
    auto num = /*...*/;
}

// constrained (with: template<typename T> concept Number)
template<Number{} Size>             // type concept
void f() {
    Number{} num = /*...*/;         // type concept
}
```

In this proposal, those both do the same thing, they're the identical *type* concept being applied with identical semantics to constrain the deduced *types* of `Size` and `num`.

In contrast, a *value* concept is not the same thing at all and leads instead to this:

```
// constrained (with: template<typename T> concept Number, template<int T> concept Even)
template<Even Size>                 // value concept
void f() {
    Number num = /*...*/;           // type concept
}
```

which is very different because the first constraint tries to constrain the *value*, not the type, of `Size` (and we want, and should have, a way to constrain the type of `Size`). I think that the discussion in this section applies equally to value concepts as being not a useful direction, because concepts are not the right tool to constrain values (and even if they did exist they especially should not visually appear to be applied to type as in the TS) for the same reasons as (and that appear to have led directly to) the adjective concepts proposals discussed in this section. Separately from this proposal, we should consider removing value concepts as an undesirable direction.

5.5 `Concept{T1, T2}` vs. `Concept[T1, T2]`

This paper shows `{}` syntax because it's the least delta from the Concepts TS syntax.

It has been suggested that `{}` is getting overused, and that we could instead follow structured bindings' precedent for introducing names using `[]`. Example: `Mergeable[X, Y, Z]` for symmetry with `auto [x, y, z] = f();`.

Notes Structured bindings was also initially proposed with `{}` syntax, but EWG preferred `[]` syntax.

I'm not aware of any current ambiguity that would be caused by using `[]`. For example, array parameters would be `Concept[T] a[]` where the `[]` do not collide, although they have different meanings. And a function returning an array of constrained type would be `Concept[X][] f();` which appears unambiguous.

> However, two different reviewers opined that "`Concept[]` looks like how array declarations *should* look, but don't (e.g., `int[10] x;`)" so that `[]` is more likely to interfere with making the language more regular in the future, and may already create visual confusion for some readers.—Also, if in the future we want to explore constraining structured bindings in the position where `auto` goes today to constrain the whole unnamed composite entity (despite the rationale given in §5.2.2), then using `[]` (e.g., `Concept[X] x;` for a variable vs. `Concept[X,Y,Z][x,y,z]` for a destructuring) means parsing starts to need more lookahead (or lookup to disambiguate, to determine `Concept` is a concept) whereas `{}` would not.

If we followed the same here, the examples in §4 would look as follows:

```
void sort(Sortable[]&);

void sort(Random_access_iterator[I], I);

void use2(auto x, auto y) {
    Number[N] xx = f(x);
    N yy = g(y);
}

void sort(Forward_iterator[I] b, I e, Relation[]<Value_type<I>> r) {
    vector<Value_type<I>> v {b,e};
    sort(v);
    copy(v.begin(),v.end(),b);
}

Output_iterator[O] copy(Input_iterator[I], I, O);

Value[V] next(V);
Value[V] compute1(V,V);             // homogeneous op
Value[V] compute2(V,Other_value[]); // heterogeneous op
vector<Value[V]> compute3(vector<V>);

auto next(Value[V]) -> V;
auto compute1(Value[V],V) -> V;
auto compute2(Value[V],Other_value[]) -> V;
auto compute3(vector<Value[V]>) -> vector<V>;

Input_iterator[I] find(I, I, Equality_comparable<Value_type<I>>[]);

auto find(Input_iterator[I], I, Equality_comparable<Value_type<I>>[]) -> I;

template <Mergeable[In1,In2,Out]>
Out merge(In1, In1, In2, In2, Out);

Number[] operator+(Number[], Number[]);

template <Mergeable[In1, In2, Out], SortableIterator[Out]>
void merge_then_sort(In1 first1, In1 last1, In2 first2, In2 last2, Out out);

template<Number[Type], Number[] Value>

void f(Number[] parameter) {
    Number[] variable;
}
```

5.6 Vexing parse

Note this corner case where the empty braces can be visually ambiguous:

```
X f(Concept{});        // a function template
X f(int{});            // a variable (ill-formed if X not constructible from int)
```

6 Proposed wording

6.1 Clause 3: Terms and definitions [intro.defs]

Modify the definitions of "signature" to include associated constraints (17.4.2). This allows different translation units to contain definitions of functions with the same signature, excluding associated constraints, without violating the one definition rule (6.2). That is, without incorporating the constraints in the signature, such functions would have the same mangled name, thus appearing as multiple definitions of the same function.

3.19 [defns.signature]
signature
<function> name, parameter type list (11.3.5), enclosing namespace (if any), any associated constraints (17.4.2), and trailing *requires-clause* (Clause 11) (if any)

3.20 [defns.signature.templ]
signature
<function template> name, parameter type list (11.3.5), enclosing namespace (if any), return type, template-head, any associated constraints (17.4.2), and trailing *requires-clause* (Clause 11) (if any)

3.22 [defns.signature.member]
signature
<class member function> name, parameter type list (11.3.5), class of which the function is a member, cv-qualifiers (if any), ref-qualifier (if any), any associated constraints (17.4.2), and trailing *requires-clause* (Clause 11) (if any)

3.23 [defns.signature.member.templ]
signature
<class member function template> name, parameter type list (11.3.5), class of which the function is a member, cv-qualifiers (if any), ref-qualifier (if any), return type (if any), template-head, any associated constraints (17.4.2), and trailing *requires-clause* (Clause 11) (if any)

6.2 Clause 8: Expressions [expr]

Add `auto` and *constrained-type-name* to the nested-name-specifier grammar.

8.4.4.2 Qualified names [expr.prim.id.qual]

qualified-id:
nested-name-specifier `template`$_{opt}$ *unqualified-id*

nested-name-specifier:
`::`
type-name `::`
namespace-name `::`
decltype-specifier `::`
`auto ::`
constrained-type-name `::`
nested-name-specifier identifier `::`
nested-name-specifier `template`$_{opt}$ *simple-template-id* `::`

Add a new paragraph at the end of this section.

In a *nested-name-specifier* of the form `auto::` or `C::`, where `C` is a *constrained-type-name*, that *nested-name-specifier* designates a placeholder that will be replaced later according to the rules for placeholder deduction in 10.1.7.4. If a placeholder designated by a *constrained-type-name* is not a placeholder type, the program is ill-formed. [Note: A *constrained-type-name* can designate a placeholder for a non-type or template (10.1.7.4.2). — end note] The replacement type deduced for a placeholder shall be a class or enumeration type. [*Example*:

```
template<typename T> concept C = sizeof(T) == sizeof(int);

struct S1 { int n; };
struct S2 { char c; };
struct S3 { struct X { using Y = int; }; };

int auto::* p1 = &S1::n; // auto deduced as S1
int C{}::* p3 = &S1::n;  // OK: C deduced as S1
char C{}::* p4 = &S2::c;        // error: deduction fails because constraints are not satisfied

void f(typename auto::X::Y);
f(S1());                  // error: auto cannot be deduced from S1()
f<S3>(0);                 // OK
```

In the declaration of `f`, the placeholder appears in a non-deduced context (17.8.2.5). It may be replaced later through the explicit specification of template arguments. — *end example*]

8.4.5 Lambda expressions [expr.prim.lambdas]

Modify paragraph 5 as follows.

5 A lambda is a *generic lambda* if the `auto` *type-specifier* appears as one of the *decl-specifier*s in the *decl-specifier-seq* of a *parameter-declaration* of the *lambda-expression*, or if the lambda has a *template-parameter-list*, or one or more placeholders (10.1.7.4) appear in the parameter-type-list of the *lambda-declarator*.

8.4.5.1 Closure types [expr.prim.lambda.closure]

Modify paragraph 3 so that the meaning of a generic lambda is defined in terms of its abbreviated member function template call operator.

3 The closure type for a non-generic *lambda-expression* has a public inline function call operator (16.5.4) whose parameters and return type are described by the *lambda-expression*'s *parameter-declaration-clause* and *trailing-return-type* respectively. ~~For a generic lambda, the closure type has a public inline function call operator member template (17.6.2) whose *template-parameter-list* consists of the specified *template-parameter-list*, if any, to which is appended one invented type *template-parameter* for each occurrence of `auto` in the lambda's *parameter-declaration-clause*, in order of appearance. The invented type *template-parameter* is a parameter pack if the corresponding *parameter-declaration* declares a function parameter pack (11.3.5). The return type and function parameters of the function call~~

~~operator template are derived from the *lambda-expression*'s *trailing-return-type* and *parameter-declaration-clause* by replacing each occurrence of auto in the *decl-specifiers* of the *parameter-declaration-clause* with the name of the corresponding invented *template-parameter*.~~ The closure type for a generic lambda has a public inline function call operator member template that is an abbreviated function template (11.3.5) whose parameters and return type are derived from the *lambda-expression*'s *parameter-declaration-clause* and *trailing-return-type* according to the rules in (11.3.5).

Add the following example after those in paragraph 3 in the C++ Standard.

[*Example*:

```
template<typename T> concept C = true;

auto g1 = [](C{}& a, C{}* b) { a = *b; };     // OK: denotes a generic lambda

struct Fun {
  auto operator()(C{}& a, C{}* b) const { a = *b; }
} fun;
```

`C` is a *constrained-type-specifier*, signifying that the lambda is generic. The generic lambda `g1` and the function object `fun` have equivalent behavior when called with the same arguments. — *end example*]

6.3 Clause 10: Declarations [dcl.dcl]

10.1.7.2 Simple type specifiers [dcl.type.simple]

Add *constrained-type-specifier* to the grammar for *simple-type-specifier*s.

simple-type-specifier:
nested-name-specifier$_{opt}$ *type-name*
nested-name-specifier `template` *simple-template-id*
nested-name-specifier$_{opt}$ *template-name*
`char`
`char16_t`
`char32_t`
`wchar_t`
`bool`
`short`
`int`
`long`
`signed`
`unsigned`
`float`
`double`
`void`
`auto`
decltype-specifier
constrained-type-specifier

Modify paragraph 2 to begin:

2 ~~The *simple-type-specifier* `auto` is a placeholder for a type to be deduced (10.1.7.4).~~ The simple-type-specifier `auto` and *constrained-type-name*s introduced by *constrained-type-specifier*s are placeholders for values (type, non-type, template) to be deduced (10.1.7.4).

Add *constrained-type-specifier*s to the table of *simple-type-specifier*s in Table 11.

Table 11 — *simple-type-specifier*s and the types they specify

Specifier(s)	Type
type-name	the type named
simple-template-id	the type as defined in 17.2
⋮	⋮
`auto`	placeholder for a type to be deduced
`decltype(auto)`	placeholder for a type to be deduced
`decltype(`*expression*`)`	the type as defined below
constrained-type-specifier	placeholder for a type to be deduced

10.1.7.4 auto specifier [dcl.spec.auto]

Extend this section to allow for *constrained-type-specifier*s as a new syntax for designating placeholders. The meaning of *constrained-type-specifier*s is described in 10.1.7.4.2.

Replace paragraph 1 with the text below.

1 ~~The `auto` and `decltype(auto)` *type-specifiers* are used to designate a placeholder type that will be replaced later by deduction from an initializer. The `auto` *type-specifier* is also used to introduce a function type having a *trailing-return-type* or to signify that a lambda is a generic lambda (8.4.5). The `auto` *type-specifier* is also used to introduce a structured binding declaration (11.5).~~ The type-specifiers `auto` and `decltype(auto)` and *constrained-type-name*s introduced by *constrained-type-specifier*s designate a placeholder (type, non-type, or template) that will be replaced later, either through deduction or an explicit specification. The `auto` and `decltype(auto)` *type-specifier*s designate placeholder types; a *constrained-type-specifier* can also designate placeholders for values and templates. [*Note*: The deduction of placeholders is done through the invention of template parameters as described in 10.1.7.4.1 and 11.3.5. — *end note*] Placeholders are also used to signify that a lambda is a generic lambda (8.4.5), that a function declaration is an abbreviated function template (11.3.5), or that a *trailing-return-type* in a *compound-requirement* (8.1.7.3) introduces an argument deduction constraint (17.4.1.6). The `auto` *type-specifier* is also used to introduce a function type having a *trailing-return-type* or to introduce a structured binding declaration 11.5. [Note: A *nested-name-specifier* can also include placeholders (8.4.4.2). Replacements for those placeholders are determined according to the rules in this section. — *end note*]

Modify paragraph 2 to allow *constrained-type-specifier*s with function declarators.

2 ~~The placeholder type~~Placeholders can appear with a function declarator in the *decl-specifier-seq*, *type-specifier-seq*, *conversion-function-id*, or *trailing-return-type*, in any context where such a declarator is valid. If the function declarator includes a *trailing-return-type* (11.3.5), that *trailing-return-type* specifies the declared return type of the function. Otherwise, the function declarator shall declare a function. If the declared return type of the function contains a placeholder type, the return type of the function is deduced from non-discarded return statements, if any, in the body of the function (9.4.1). In a function declarator of the form `auto D -> T` where `T` contains placeholders, the initial `auto` does not designate a placeholder. If a placeholder appears in a parameter type of a function declaration, the function declaration declares an abbreviated function template (11.3.5). [*Example*:

```
void f(const auto&, int); // OK: an abbreviated function template
```

— end example]

Add the following after paragraph 3 to allow the use of `auto` in the *trailing-return-type* of a *compound-requirement*. Also, disallow the use of `decltype(auto)` with function parameters and deduction constraints.

3.# If a placeholder appears in the *trailing-return-type* of a *compound-requirement* in a *requires-expression* (8.1.7.3), that return type introduces an argument deduction constraint (17.4.1.6). [*Example*:

```
template<typename T> concept C() {
  return requires (T i) {
    {*i} -> const auto&; // OK: introduces an argument deduction constraint
  };
}
```

— end example]

3.# The `decltype(auto)` type-specifier shall not appear in the declared type of a *parameter-declaration* or the *trailing-return-type* of a *compound-requirement*.

Modify paragraph 3 to allow multiple placeholders within a variable declaration.

3 The type of a variable declared using ~~`auto` or `decltype(auto)`~~ a placeholder is deduced from its initializer. This use is allowed in an initializing declaration (11.6) of a variable. ~~`auto` or `decltype(auto)` shall appear as one of the *decl-specifiers* in the *decl-specifier-seq* and the~~ A placeholder can appear anywhere in the declared type of the variable, but `decltype(auto)` shall appear only as one of the *decl-specifiers* of the *decl-specifier-seq*. The *decl-specifier-seq* of such a variable shall be followed by one or more *declarator*s, each of which shall be followed by a non-empty *initializer*.

Add the following declarations to the example in the previous paragraph.

```
struct N {
  template<typename T> struct Wrap;
  template<typename T> static Wrap<T> make_wrap(T);
};
```

```
template<typename T, typename U> struct Pair;
template<typename T, typename U> Pair<T, U> make_pair(T, U);
template<int N> struct Size { void f(int) { } };

void (auto::* p1)(auto) = &Size<0>::f;            // OK: p1 has type
void(Size<0>::*)(int)
Pair<auto, auto> p2 = make_pair(0, 'a');   // OK: p2 has type Pair<int, char>
N::Wrap<auto> a = N::make_wrap(0.0);              // OK: a has type Wrap<double>
auto::Wrap<int> x = N::make_wrap(0);              // error: failed to deduce value
for auto
Size<sizeof(auto)> y = Size<0>{};          // error: failed to deduce value for
auto

template<typename T> concept C = true;
template<typename T> concept D = false;

C z1 = 0; // OK: z1 has type int
D z2 = 0; // error: constraints not satisfied
C cf1() { return 0.0; }; // OK: cf1 returns double
D cf2() { return 0.0; }; // error: constraints not satisfied
auto cf3() -> C; // OK: cf3's return type will be deduced when it is defined
```

Update paragraph 5 to disallow placeholders in other contexts.

5 A program that uses ~~auto or decltype(auto)~~ placeholders in a context not explicitly allowed in this subclause is ill-formed.

Modify paragraph 6 as follows.

6 If the *init-declarator-list* contains more than one *init-declarator*, they shall all form declarations of variables. The type of each declared variable is determined by placeholder type deduction (10.1.7.4.1), and if the type that replaces ~~the placeholder type~~ the declared variable type or return type is not the same in each deduction, the program is ill-formed.

Add the following examples to that paragraph.

[*Example*:

```
Pair<auto, auto> p1 = make_pair(0, 0),
                 *p2 = &p1;                 // OK: replacement type is Pair<int,
int>

Pair<auto, auto> p3 = make_pair(0, 'a'),
                 p4 = make_pair('a', 0);   // error: different replacement types
```

— end example]

Modify paragraphs 7, 8, and 9 as follows.

7 If a function with a declared return type that contains ~~a placeholder type~~ placeholders has multiple non-discarded `return` statements, the return type is deduced for each such `return` statement. If the type deduced is not the same in each deduction, the program is ill-formed.

8 If a function with a declared return type that uses ~~a placeholder type~~ placeholders has no non-discarded `return` statements, the `return` type is deduced as though from a `return` statement with no operand at the closing brace of the function body. […]

9 If the name of an entity with an undeduced placeholder ~~type~~ appears in an expression, the program is ill-formed. […]

Modify paragraph 11 as follows.

11 Redeclarations or specializations of a function or function template with a declared return type that uses a placeholder ~~type~~ shall also use that placeholder, not a deduced type. If a placeholder is designated by a *constrained-type-specifier*, redeclarations or specializations shall use the same *constrained-type-specifier*. Similarly, redeclarations or specializations of a function or function template with a declared return type that does not use a placeholder ~~type~~ shall not use a placeholder.

Add the following examples to that paragraph.

```
template<typename T> concept C1 = true;
template<typename T> concept C2 = true;

template<typename T> auto cf(T) -> C1{}; // #1
template<typename T> C1{} cf(T);        // #2, redeclaration of #1
template<typename T> C2{} cf(T);        // error: redeclared with different
placeholder
```

Modify paragraphs 13 and 14 as follows.

12 A function declared with a return type that uses a placeholder ~~type~~ shall not be `virtual` (13.3).

13 An explicit instantiation declaration (17.8.2) does not cause the instantiation of an entity declared using ~~a placeholder type~~ placeholders, but it also does not prevent that entity from being instantiated as needed to determine its type.

10.1.7.4.1 Placeholder type deduction [dcl.type.auto.deduct]

Modify this section as follows.

1 *Placeholder type deduction* is the process by which a type containing ~~a placeholder type~~ placeholders is replaced by a deduced type.

2 A type T containing ~~a placeholder type~~ placeholders, and a corresponding initializer e, are determined as follows:

(2.1) — for a non-discarded `return` statement that occurs in a function declared with a return type that contains ~~a placeholder type~~ placeholders, T is the declared return type and e is the operand of the `return` statement. If the `return` statement has no operand, then e is `void()`;

(2.2) — for a variable declared with a type that contains ~~a placeholder type~~ placeholders, T is the declared type of the variable and e is the initializer. If the initialization is direct-list-initialization, the initializer shall be a *braced-init-list* containing only a single *assignment-expression* and e is the *assignment-expression*;

(2.3) — for a non-type template parameter declared with a type that contains ~~a placeholder type~~ placeholders, T is the declared type of the non-type template parameter and e is the corresponding template argument.

In the case of a `return` statement with no operand or with an operand of type `void`, T shall be either `decltype(auto)` ~~or~~ , *cv* `auto`, or a *constrained-type-name*.

3 If the deduction is for a `return` statement and e is a *braced-init-list* (11.6.4), the program is ill-formed.

4 If the placeholder is the `auto` *type-specifier* or a *constrained-type-name*, the deduced type `T'` replacing T is determined using the rules for template argument deduction. ~~Obtain P from T by replacing the occurrences of `auto` with either a new invented type template parameter U or, if the initialization is copy-list-initialization, with `std::initializer_list<U>`.~~ Otherwise, obtain a type P from T as follows:

(4.1) — if the initialization is a copy-list-initialization and a placeholder is a *decl-specifier* of the *decl-specifier-seq* of the variable declaration, replace that occurrence of the placeholder with `std::initializer_list<U>` where U is an invented type template parameter;

(4.2) — otherwise, replace each occurrence of a placeholder in the variable or return type with a new invented type template parameter according to the rules for inventing template parameters for placeholders in 11.3.5.

Deduce a value for ~~U~~ each invented type template parameter using the rules of template argument deduction from a function call (17.9.2.1), where P is a function template parameter type and the corresponding argument is e. If the deduction fails, the declaration is ill-formed. If any placeholders in the declared type were introduced by a *constrained-type-specifier*, then define C to be a *constraint-expression* as follows:

(4.3) — if there is single *constrained-type-specifier*, then C is the *constraint-expression* introduced by the invented template constrained-parameter (17.1) corresponding to that *constrained-type-specifier*;

(4.4) — otherwise, C is the *logical-and-expression* (8.5.14) whose operands are the *constraint-expressions* introduced by the invented template *constrained-parameters* corresponding to each *constrained-type-specifier*, in order of appearance.

If the normalized constraint for C (17.4.2) is not satisfied by the deduced values, the declaration is ill-formed. Otherwise, `T'` is obtained by substituting the deduced ~~U~~ values for the invented type template parameters into P. [*Example:*

```
auto x1 = { 1, 2 }; // decltype(x1) is std::initializer_list<int>
```

```
auto x2 = { 1, 2.0 }; // error: cannot deduce element type
auto x3{ 1, 2 }; // error: not a single element
auto x4 = { 3 }; // decltype(x4) is std::initializer_list<int>
auto x5{ 3 }; // decltype(x5) is int
```

—end example]

[*Example:*

```
const auto &i = expr;
```

The type of `i` is the deduced type of the parameter `u` in the call `f(expr)` of the following invented function template:

```
template <class U> void f(const U& u);
```

—end example]

Add the following to the first example in paragraph 5.

[*Example*:

```
template<typename T> struct Vec { };
template<typename T> Vec<T> make_vec(std::initializer_list<T>) { return Vec<T>{}; }

template<typename... Ts> struct Tuple { };
template<typename... Ts> auto make_tup(Ts... args) { return Tuple<Ts...>{}; }

auto& x3 = *x1.begin();                    // OK: decltype(x3) is int&
const auto* p = &x3;                       // OK: decltype(p) is const int*
Vec<auto> v1 = make_vec({1, 2, 3});        // OK: decltype(v1) is Vec<int>
Vec<auto> v2 = {1, 2, 3};                  // error: type deduction fails
Tuple<auto...> v3 = make_tup(0, 'a');      // OK: decltype(v3) is Tuple<int, char>
```

— end example]

Add the following after the second example in paragraph 5.

[*Example*:

```
template<typename F, typename S> struct Pair;
template<typename T, typename U> Pair<T, U> make_pair(T, U);

struct S { void mfn(bool); } s;
int fn(char, double);

Pair<auto (*)(auto, auto), auto (auto::*)(auto)> p = make_pair(fn, &S::mfn);
```

The declared type of `p` is the deduced type of the parameter `x` in the call of `g(make_pair(fn, &S::mfn))` of the following invented function template:

```
template<class T1, class T2, class T3, class T4, class T5, class T6>
```

```
void g(Pair< T1(*)(T2, T3), T4 (T5::*)(T6)> x);
```

— end example]

[*Example*:

```
template<typename T> concept C = true;

const C* cv = expr;
```

The type of `cv` is deduced from the parameter `p1` in the call `f1(expr)` of the following invented function:

```
template<C{T}> void f1(const T* p1);
```

— end example]

10.1.7.4.2 Constrained type specifiers [dcl.spec.auto.constr]

Add this section to 10.1.7.4.

1 A *constrained-type-specifier* designates zero or more placeholder *constrained-type-name*s (type, non-type, or template) and introduces an associated constraint (17.4.2).

variadic-constrained-type-specifier:
qualified-concept-name { ... *constrained-type-name*$_{opt}$ }

constrained-type-specifier:
qualified-concept-name { *constrained-type-name-list*$_{opt}$ }

qualified-concept-name:
nested-name-specifier$_{opt}$ *concept-name*

concept-name:
identifier
partial-concept-id

partial-concept-id:
concept-name < *template-argument-list*$_{opt}$ >

constrained-type-name-list:
constrained-type-name
constrained-type-name , *constrained-type-name*

constrained-type-name:
identifier

[*Example*:

```
template<typename T> concept C1 = false;
template<int N> concept C2 = false;
template<template<typename> class X> concept C3 = false;
```

```
template<typename T, int N> class Array { };
template<typename T, template<typename> class A> class Stack { };
template<typename T> class Alloc { };

void f1(C1{});                // C1 designates a placeholder type
void f2(Array<auto, C2>);     // C2 designates a placeholder for an integer value
void f3(Stack<auto, C3{}>);   // C3 designates a placeholder for a class template
```

— *end example*]

2 An identifier is a *concept-name* if it refers to a set of concept definitions (10.1.8). [*Note*: The set of concepts has multiple members only when referring to a set of overloaded function concepts. There is at most one member of this set when a *concept-name* refers to a variable concept. — *end note*] [*Example*:

```
template<typename T> concept C() { return true; }                    // #1
template<typename T, typename U> concept C() { return true; }        // #2
template<typename T> concept D = true;                               // #3

void f(C);       // OK: the set of concepts referred to by C includes both #1 and
#2;
          // concept resolution (17.4.5) selects #1.

void g(D);       // OK: the concept-name D refers only to #3
```

— *end example*]

3 A *partial-concept-id* is a *concept-name* followed by a sequence of *template-arguments*. [Example:

```
template<typename T, int N = 0> concept Seq = true;

void f1(Seq<3>); // OK
void f2(Seq<>);  // OK
```

— *end example*]

4 The concept designated by a *constrained-type-specifier* is the one selected according to the rules for concept resolution in 17.4.5.

5 [*Note*: The *constraint-expression* introduced by a *constrained-type-name* is the one introduced by the invention of a *constrained-parameter* (17.1). The rules for inventing template parameters corresponding to placeholders in the type of a variable or the declared return type of a function are described in 10.1.7.4.1. The rules for inventing template parameters corresponding to placeholders in the *parameter-declaration-clause* of a *lambda-expression* (8.4.5) or function declaration (11.3.5) are described in 11.3.5. The rules for inventing a template parameter corresponding to placeholders in the *trailing-return-type* of a *compound-requirement* are described in 17.4.1.6. — *end note*]

6.4 Clause 11: Declarators [dcl.decl]

11.3.5 Functions [dcl.fct]

Modify paragraph 10 as follows.

10 A single name can be used for several different functions in a single scope; this is function overloading (Clause 16). All declarations for a function shall have equivalent return types, parameter-type-lists, associated constraints, and *requires-clause*s (17.6.6.1).

Modify paragraph 8 to exclude constraints from the type of a function.

8 The return type, the parameter-type-list, the *ref-qualifier*, the *cv-qualifier-seq*, and the exception specification, but not the default arguments (11.3.6), associated constraints (17.4.2), or the trailing *requires-clause* (Clause 11), are part of the function type. [*Note:* Function types are checked during the assignments and initializations of pointers to functions, references to functions, and pointers to member functions. —*end note*]

Modify paragraph 18 as follows.

18 There is a syntactic ambiguity when an ellipsis occurs at the end of a *parameter-declaration-clause* without a preceding comma. In this case, the ellipsis is parsed as part of the *abstract-declarator* if the type of the parameter either names a template parameter pack that has not been expanded or contains ~~`auto`~~ a placeholder (10.1.7.4); otherwise, it is parsed as part of the *parameter-declaration-clause*.[103]

Add the following paragraphs after paragraph 18.

19 An *abbreviated function template* is a function declaration whose *parameter-type-list* includes one or more placeholders that introduce zero or one type name(s) (10.1.7.4). An abbreviated function template is equivalent to a function template (17.6.5) whose *template-parameter-list* includes one invented *template-parameter* for each occurrence of a placeholder in the *parameter-declaration-clause*, in order of appearance, according to the rules below. [*Note*: Template parameters are also invented to deduce the type of a variable or the return type of a function when the declared type contains placeholders (10.1.7.4.1). — *end note*]

20 Each template parameter is invented as follows.

(20.1) — If the placeholder is designated by the `auto` type-specifier, then the invented template parameter is a type template-parameter.

(20.2) — Otherwise, the placeholder is designated by a *constrained-type-specifier*, and the invented parameter is a *constrained-parameter* (17.1) whose *qualified-concept-name* is that of the *constrained-type-specifier*.

(20.3) — If the placeholder appears in the *decl-specifier-seq* of a function parameter pack (??), or the *type-specifier-seq* of a *type-id* that is a pack expansion, the invented template parameter is a template parameter pack.

21 The adjusted function parameters of an abbreviated function template are derived from the *parameter-declaration-clause* by replacing each occurrence of a placeholder with the name of the corresponding invented *template-parameter*. If the replacement of a placeholder with the name of a template parameter results in an invalid parameter declaration, the program is ill-formed. [*Note*: Equivalent function template declarations declare the same function template (17.6.5.1). — *end note*] [*Example*:

```
template<typename T> class Vec { };
template<typename T, typename U> class Pair { };
template<typename... Args> class Tuple { };

void f1(const auto&, auto);
void f2(Vec<auto*>...);
void f3(Tuple<auto...>);
void f4(auto (auto::*)(auto));

template<typename T, typename U> void f1(const T&, U);                    // redeclaration of f1
template<typename... T> void f2(Vec<T*>...);                              // redeclaration of f2
template<typename... Ts> void f3(Tuple<Ts...>);                           // redeclaration of f3
template<typename T, typename U, typename V> void f4(T (U::*)(V)); // redeclaration of f4

template<typename T> concept C1 = true;
template<typename T> concept C2 = true;
template<typename T, typename U> concept C3 = true;
template<typename... Ts> concept C4 = true;

void g1(const C1{}*, C2{}&);
void g2(Vec<C1{}>&);
void g3(C1{}&...);
void g4(Vec<C3<int>>);
void g5(C4{}...);
void g6(Tuple<C4{}...>);
void g7(C4{} p);
void g8(Tuple<C4{}>);

template<C1 T, C2 U> void g1(const T*, U&);                               // redeclaration of g1
template<C1 T> void g2(Vec<T>&);                                          // redeclaration of g2
template<C1... Ts> void g3(Ts&...);                                       // redeclaration of g3
template<C3<int> T> void g4(Vec<T>);                                      // redeclaration of g4
template<C4... Ts> void g5(Ts...);                                        // redeclaration of g5
template<C4... Ts> void g6(Tuple<Ts...>);                                 // redeclaration of g6
template<C4 T> void g7(T);                                                // redeclaration of g7
template<C4 T> void g8(Tuple<T>);                                         // redeclaration of g8
```

— end example] [*Example*:

```
template<int N> concept Num = true;
```

```
void h(Num{}*); // error: invalid type in parameter declaration
```

The equivalent declaration would have this form:

```
template<int N> void h(N*); // error: invalid type
```

— end example]

22 A function template can be an abbreviated function template. The invented *template-parameters* are appended to the *template-parameter-list* after the explicitly declared *template-parameters*. [*Example*:

```
template<typename T, int N> class Array { };

template<int N> void f(Array<auto, N>*);

template<int N, typename T> void f(Array<T, N>*); // OK: redeclaration of f(Array<auto, N>*)
```

— end example]

6.5 Clause 16: Overloading [over]

Modify paragraph 2 to allow overload selection based on constraints.

2 When a function name is used in a call, which function declaration is being referenced and the validity of the call are determined by comparing the types of the arguments at the point of use with the types or associated constraints (17.4.2) of the parameters in the declarations that are visible at the point of use. This function selection process is called *overload resolution* and is defined in 16.3. […]

16.1 Overloadable declarations [over.load]

Update paragraph 3 to mention a function's overloaded constraints.

3 [*Note:* As specified in 11.3.5, function declarations that have equivalent parameter declarations, associated constraints if any (17.4.2), and *requires-clause*s, if any (17.4.2), declare the same function and therefore cannot be overloaded: […]

16.2 Declaration matching [over.dcl]

Modify paragraph 1 to extend the notion of declaration matching to also include a function's associated constraints.

1 Two function declarations of the same name refer to the same function if they are in the same scope and have equivalent parameter declarations (16.1), equivalent associated constraints if any (17.4.2), and equivalent trailing *requires-clause*s, if any (Clause 11).

6.6 Clause 17: Templates [temp]

17.1 Template parameters [temp.param]

In paragraph 1, update the grammar for constrained template parameters.

constrained-parameter:
~~*qualified-concept-name ... identifier*$_{opt}$~~
~~*qualified-concept-name identifier*$_{opt}$ ~~*default-template-argument*$_{opt}$~~
variadic-constrained-type-specifier
constrained-type-specifier default-template-argument$_{opt}$

Modify paragraph 10 as follows.

10 A *constrained-parameter* declares a template parameter whose kind (type, non-type, template) and type match that of the prototype parameter (17.6.8) of the concept designated by the *qualified-concept-name* in the *constrained-parameter*. The designated concept is selected by the rules for concept resolution described in 17.4.5. Let X be the prototype parameter of the designated concept. The declared template parameter is determined by the kind of X (type, non-type, template) and the optional ellipsis in the *constrained-parameter* as follows.

(10.1) — If X is a type *template-parameter*, the declared parameter is a type *template-parameter*.

(10.2) — If X is a non-type *template-parameter*, the declared parameter is a non-type *template-parameter* having the same type as X.

(10.3) — If X is a template *template-parameter*, the declared parameter is a template *template-parameter* having the same *template-parameter-list* as X, excluding default template arguments.

(10.4) — If the *qualified-concept-name* is followed by an ellipsis, then the declared parameter is a template parameter pack (17.6.3).

[*Example:*

```
template<typename T> concept C1 = true;
template<template<typename> class X> concept C2 = true;
template<int N> concept C3 = true;
template<typename... Ts> concept C4 = true;
template<char... Cs> concept C5 = true;

template<C1 {T}> void f1();         // OK, T is a type template-parameter
template<C2 {X}> void f2();         // OK, X is a template with one type-parameter
template<C3 N> void f3();           // OK, N has type int
template<C4{... Ts}> void f4();     // OK, Ts is a template parameter pack of
types
template<C4 {T}> void f5();         // OK, T is a type template-parameter
template<C5... Cs> void f6();               // OK, Cs is a template parameter pack
of chars
```

—end example]

Modify the example in paragraph 11 as follows.

```
template<typename T> concept C1 = true;
template<typename... Ts> concept C2 = true;
template<typename T, typename U> concept C3 = true;

template<C1<T>> struct s1;          // associates C1<T>
template<C1<T...>> struct s2;             // associates (C1<T> && ...)
template<C2<T...>> struct s3;             // associates C2<T...>
template<C3<T><int>> struct s4;           // associates C3<T, int>
```

17.2 Names of template specializations [temp.names]

Modify the example in paragraph 8 as follows.

```
template<typename T> concept C1 = sizeof(T) != sizeof(int);

template<C1<T>> struct S1 { };
template<C1<T>> using Ptr = T*;

S1<int>* p;                         // error: constraints not satisfied
Ptr<int> p;                         // error: constraints not satisfied

template<typename T>
struct S2 { Ptr<int> x; };          // error, no diagnostic required

template<typename T>
struct S3 { Ptr<T> x; };            // OK, satisfaction is not required

S3<int> x;                          // error: constraints not satisfied

template<template<C1<T>> class X>
struct S4 {
   X<int> x;                        // error, no diagnostic required
};

template<typename T> concept C2 = sizeof(T) == 1;

template<C2<T>> struct S { };

template struct S<char[2]>;         // error: constraints not satisfied
template<> struct S<char[2]> { };   // error: constraints not satisfied
```

Modify paragraph 3 as follows.

3 A *template-argument* matches a template *template-parameter* P when P is at least as specialized as the *template-argument* A, and P is at least as constrained as A according to the rules in 17.4.3. If P contains a template parameter pack, then A also matches P if each of A's template parameters matches the corresponding template parameter in the *template-head* of P, and P is at least as constrained as A according to the rules in 17.4.3. […]

Later in the same paragraph, modify the third example as follows:

```
template<typename T> concept C = requires (T t) { t.f(); };
template<typename T> concept D = C<T> && requires (T t) { t.g(); };

template<template<C{}> class P> struct S { };

template<C{}> struct X { };
template<D{}> struct Y { };
template<typename T> struct Z { };

S<X> s1; // OK, X and P have equivalent constraints
S<Y> s2; // error: P is not at least as specialized as Y
S<Z> s3; // OK, P is at least as specialized as Z
```

17.6 Template declarations [temp.decls]

Modify paragraph 2 to indicate that associated constraints are instantiated separately from the template they are associated with.

2 For purposes of name lookup and instantiation, default arguments, associated constraints (17.4.2), *partial-concept-id*s, *requires-clause*s (Clause 17), and *noexcept-specifier*s of function templates and of member functions of class templates are considered definitions; each default argument, *partial-concept-ids*, associated constraint, *requires-clause*, or *noexcept-specifier* is a separate definition which is unrelated to the templated function definition or to any other default arguments *partial-concept-id*s, *requires-clause*s, or *noexcept-specifier*s. For the purpose of instantiation, the substatements of a constexpr if statement (9.4.1) are considered definitions.

Modify paragraph 3 as follows.

3 When a member function, a member class, a member enumeration, a static data member or a member template of a class template is defined outside of the class template definition, the member definition is defined as a template definition in which the *template-head* ~~is~~ and associated constraints (17.4.2) are is equivalent to that of the class template (17.6.6.1). The names of the template parameters used in the definition of the member may be different from the template parameter names used in the class template definition. The template argument list following the class template name in the member definition shall name the parameters in the same order as the one used in the template parameter list of the member. Each template parameter pack shall be expanded with an ellipsis in the template argument list. [*Example:*

```
template<class T1, class T2> struct A {
   void f1();
   void f2();
};

template<class T2, class T1> void A<T2,T1>::f1() { }       // OK
template<class T2, class T1> void A<T1,T2>::f2() { }       // error

template<class ... Types> struct B {
   void f3();
```

```
   void f4();
};

template<class ... Types> void B<Types ...>::f3() { }      // OK
template<class ... Types> void B<Types>::f4() { }   // error

template<typename T> concept C = true;
template<typename T> concept D = true;

template<C<T>> struct S {
   void f();
   void g();
   void h();
   template<D<U>> struct Inner;

};

template<C<A>> void S<A>::f() { }     // OK: template-heads match
template<typename T> void S<T>::g() { }      // error: no matching declaration for S<T>

template<typename T> requires C<T>     // error (no diagnostic required): template-heads
are
void S<T>::h() { }                           // functionally equivalent but not
equivalent

template<C<X>> template<D<Y>>
struct S<X>::Inner { };               // OK
```

—end example]

17.6.2 Member templates [temp.mem]

Modify paragraph 1 as follows.

1 A template can be declared within a class or class template; such a template is called a member template. A member template can be defined within or outside its class definition or class template definition. A member template of a class template that is defined outside of its class template definition shall be specified with a *template-head* and associated constraints equivalent to that of the class template followed by a *template-head* and associated constraints equivalent to that of the member template (17.6.6.1). [*Example:*

```
template<class T> struct string {
   template<class T2> int compare(const T2&);
   template<class T2> string(const string<T2>& s) { /* ... */ }
};

template<class T> template<class T2> int string<T>::compare(const T2& s) {
}
```

—end example] [*Example:*

```
template<typename T> concept C1 = true;
template<typename T> concept C2 = sizeof(T) <= 4;
```

```
template<C1–{T}> struct S {
   template<C2–{U}> void f(U);
   template<C2–{U}> void g(U);
};

template<C1–{T}> template<C2–{U}>
void S<T>::f(U) { }          // OK

template<C1–{T}> template<typename U>
void S<T>::g(U) { }          // error: no matching function in S<T>
```

—*end example*]

17.6.4 Friends [temp.friend]

Modify paragraph 8 to restrict constrained friend declarations and add examples.

8 When a friend declaration refers to a specialization of a function template, the function parameter declarations shall not include default arguments, the declaration shall not have associated constraints (17.4.2), nor shall the inline specifier be used in such a declaration.

[*Note*: Other friend declarations can be constrained. In a constrained friend declaration that is not a definition, the constraints are used for declaration. — *end note*] [*Example*:

```
template<typename T> concept bool C1 = true;
template<typename T> concept bool C2 = false;

template<C1{T}> void g0(T);
template<C1{T}> void g1(T);
template<C2{T}> void g2(T);

template<typename T>
   struct S {
      friend void f1() requires true;       // OK
      friend void f2() requires C1<T>;      // OK
      friend void g0<T>(T) requires C1<T>; // error: constrained friend
specialization
      friend void g1<T>(T);          // OK
      friend void g2<T>(T);          // error: constraint can never be satisfied,
                                     // no diagnostic required
  };

void f1() requires true;             // friend of all S<T>
void f2() requires C1<int>;          // friend of only S<int>
```

The friend declaration of `g2` is ill-formed, no diagnostic required, because no valid specialization of `S` can be generated: the constraint on `g2` can never be satisfied, so template argument deduction (17.9.2.6) will always fail. — *end example*]

[*Note*: Within a class template, a friend may define a non-template function whose constraints specify requirements on template arguments. [*Example*:

```
template<typename T> concept bool Eq = requires (T t) { t == t; };
```

```
template<typename T>
  struct S {
    friend bool operator==(S a, S b) requires Eq<T> { return a == b; } // OK
  };
```

— *end example*] In the instantiation of such a class template (17.8), the template arguments are substituted into the constraints but not evaluated. Constraints are checked (17.4) only when that function is considered as a viable candidate for overload resolution (16.3.2). If substitution fails, the program is ill-formed. — end note]

17.6.5 Class template partial specialization [temp.class.spec]

Modify paragraph 4 as follows.

4 A class template partial specialization may be constrained (Clause 17). [*Example:*

```
template<typename T> concept C = true;

template<typename T> struct X { };
template<typename T> struct X<T*> { };     // #1
template<C T> struct X<T> { };             // #2
```

Both partial specializations are more specialized than the primary template. #1 is more specialized because the deduction of its template arguments from the template argument list of the class template specialization succeeds, while the reverse does not. #2 is more specialized because the template arguments are equivalent, but the partial specialization is more constrained (17.4.4). —*end example*]

17.6.5.1 Matching of class template partial specializations [temp.class.spec.match]

Modify the second example in paragraph 2 as follows.

```
template<typename T> concept C = requires (T t) { t.f(); };

template<typename T> struct S { };          // #1
template<C T> struct S<T> { };             // #2

struct Arg { void f(); };

S<int> s1; // uses #1; the constraints of #2 are not satisfied
S<Arg> s2; // uses #2; both constraints are satisfied but #2 is more specialized
```

17.6.5.2 Partial ordering of class template specializations [temp.class.order]

Modify the second example in paragraph 2 as follows.

```
template<typename T> concept C = requires (T t) { t.f(); };
template<typename T> concept D = C<T> && requires (T t) { t.f(); };

template<typename T> class S { };
template<C T> class S<T> { };      // #1
template<D T> class S<T> { };      // #2
```

```
template<C-{T}> void f(S<T>);        // A
template<D-{T}> void f(S<T>);        // B
```

17.6.6.1 Function template overloading [temp.over.link]

Modify paragraph 7 as follows.

7 Two function templates are *equivalent* if they are declared in the same scope, have the same name, have equivalent *template-head*s, and have return types, parameter lists, associated constraints (if any), and trailing *requires-clause*s (if any) that are equivalent using the rules described above to compare expressions involving template parameters. Two function templates are *functionally equivalent* if they are declared in the same scope, have the same name, accept and are satisfied by the same set of template argument lists, and have return types, ~~and~~ parameter lists, associated constraints (if any), and trailing *requires-clause*s (if any) that are functionally equivalent using the rules described above to compare expressions involving template parameters. If the validity or meaning of the program depends on whether two constructs are equivalent, and they are functionally equivalent but not equivalent, the program is ill-formed, no diagnostic required.

17.7 Name resolution [temp.res]

Add a new bullet to paragraph 8 as follows:

8 The validity of a template may be checked prior to any instantiation. [*Note:* Knowing which names are type names allows the syntax of every template to be checked in this way. —*end note*] The program is ill-formed, no diagnostic required, if:

(8.1) — no valid specialization can be generated for a template or a substatement of a constexpr if statement (9.4.1) within a template and the template is not instantiated, or

(8.2) — no substitution of template arguments into a *partial-concept-id* or *requires-clause* would result in a valid expression, or

(8.3) — every valid specialization of a variadic template requires an empty template parameter pack, or

(8.x) — no instantiation of the associated constraints (17.4.2) of a template would result in a valid expression, or

(8.4) — a hypothetical instantiation of a template immediately following its definition would be ill-formed due to a construct that does not depend on a template parameter, or

[…]

17.8.1 Implicit Instantiation [temp.inst]

Modify the final Note in paragraph 1 as follows.

[…] [*Note:* Within a template declaration, a local class (12.4) or enumeration and the members of a local class are never considered to be entities that can be separately instantiated (this includes their default

arguments, associated constraints (17.4.2), *noexcept-specifier*s, and non-static data member initializers, if any, but not their *partial-concept-id*s or *requires-clause*s). As a result, the dependent names are looked up, the semantic constraints are checked, and any templates used are instantiated as part of the instantiation of the entity within which the local class or enumeration is declared. —*end note*]

17.9.2.6 Deducing template arguments from a function declaration [temp.deduct.decl]

Expand paragraph 2 as follows in order to require the satisfaction of constraints when matching a specialization to a template.

2 Remove from the set of function templates considered all those whose associated constraints (if any) are not satisfied by the deduced template arguments (17.4.2).

3 If, for the ~~set of function templates so considered~~ remaining function templates, there is either no match or more than one match after partial ordering has been considered (17.6.6.2), deduction fails and, in the declaration cases, the program is ill-formed. [*Example*:

```
template<typename T> concept bool C = requires (T t) { -t; };

template<C{T}> void f(T) { }         // #1
template<typename T> void g(T) { }   // #2
template<C{T}> void g(T) { }         // #3

template void f(int);                // OK: refers to #1
template void f(void*);                      // error: no matching template
template void g(int);                // OK: refers to #3
template void g(void*);                      // OK: refers to #2
```

— *end example*]

17.4.2 Constrained declarations [temp.constr.decl]

Modify paragraphs 2 and 3 as follows.

2 Constraints can also be associated with a declaration through the use of *constrained-parameter*s in a *template-parameter-list*, and *constrained-type-specifiers* in the *parameter-type-list* of a function template. Each of these forms introduces additional *constraint-expression*s that are used to constrain the declaration.

3 A template's *associated constraints* are defined as follows:

(3.1) — If there are no introduced *constraint-expression*s, the declaration has no associated constraints.

(3.2) — Otherwise, if there is a single introduced *constraint-expression*, the associated constraints are the normal form (17.4.3) of that expression.

(3.3) — Otherwise, the associated constraints are the normal form of a logical AND expression (8.5.14) whose operands are in the following order:

(3.3.1) — the *constraint-expression* introduced by each *constrained-parameter* (17.1) in the declaration's *template-parameter-list*, in order of appearance, and

(3.3.2) — the *constraint-expression* introduced by a *requires-clause* following a *template-parameter-list* (Clause 17), ~~and~~

(3.3.#) — the *constraint-expression* introduced by each *constrained-type-specifier* (10.1.7.4.2) in the type of a parameter-declaration in a function declaration (11.3.5), in order of appearance, and

(3.3.3) — the *constraint-expression* introduced by a trailing *requires-clause* (Clause 11) of a function declaration (11.3.5).

The formation of the associated constraints establishes the order in which constraints are instantiated when checking for satisfaction (17.4.1). [*Example:*

```
template<typename T> concept C = true;

void f0(C{T});
template<C{T}> void f1(T);
template<typename T> requires C<T> void f2(T);
template<typename T> void f3(T) requires C<T>;
```

The functions `f0`, `f1`, `f2`, and `f3` have the associated constraint `C<T>`.

```
template<typename T> concept C1 = true;
template<typename T> concept C2 = sizeof(T) > 0;

template<C1{T}> void f4(T) requires C2<T>;
template<typename T> requires C1<T> && C2<T> void f5(T);
```

The associated constraints of `f4` and `f5` are `C1<T>` ∧ `C2<T>`.

```
template<C1{T}> requires C2<T> void f6();
template<C2{T}> requires C1<T> void f7();
```

The associated constraints of `f6` are `C1<T>` ∧ `C2<T>`, and those of `f7` are `C2<T>` ∧ `C1<T>`. —*end example*]

17.4.3 Constraint normalization [temp.constr.normal]

In paragraph 3, modify the example as follows.

```
template<typename T> concept C1 = sizeof(T) == 1;
template<typename T> concept C2 = C1<T>() && 1 == 2;
template<typename T> concept C3 = requires { typename T::type; };
template<typename T> concept C4 = requires (T x) { ++x; }

template<C2{U}> void f1(U);         // #1
template<C3{U}> void f2(U);         // #2
template<C4{U}> void f3(U);         // #3
```

17.4.4 Partial ordering by constraints [temp.constr.order]

In paragraph 4, modify the example as follows.

```
template<typename T> concept C1 = requires(T t) { --t; };
template<typename T> concept C2 = C1<T> && requires(T t) { *t; };

template<C1~~<~~{T}> void f(T);          // #1
~~template<C2 T> ~~void f(C2{T});              // #2
template<typename T> void g(T);      // #3
template<C1~~<~~{T}> void g(T);          // #4

f(0);           // selects #1
f((int*)0);     // selects #2
g(true);        // selects #3 because C1<bool> is not satisfied
g(0);           // selects #4
```

Before 17.5, insert the following new subclause:

17.4.5 Resolution of qualified references to concepts [temp.constr.resolve]

1 *Concept resolution* is the process of selecting a concept from a set of concept definitions referred to by a *qualified-concept-name*, or from a set of declarations including one or more concept definitions referred to by a *simple-template-id* or a *qualified-id* whose *unqualified-id* is a *simple-template-id*. Concept resolution is performed when such a name appears

(1.1) — as a constrained-type-specifier (10.1.7.4.2),

(1.2) — in a constrained-parameter (17.1),

(1.3) — in a template-introduction (17.2), or

(1.4) — within a constraint-expression (17.10.2).

Within such a name, let C be the *concept-name* or *template-name* that refers to the set of concept definitions.

2 The selection of a concept from this set is done by matching the template parameters of each concept in that set to a sequence of template arguments and *wildcards*. This sequence is called the *concept argument list*, and its elements are called *concept arguments*. For the purpose this matching, a wildcard can match a template parameter of any kind (type, non-type, template) as described below.

3 The method for determining the concept argument list depends on the context in which C appears.

(3.1) — If C is part of a *constrained-type-specifier* or *constrained-parameter*, then

(3.1.1) — if C is a *constrained-type-name*, the concept argument list is a sequence of wildcards of the same length as the *constrained-type-name-list*, or

(3.1.2) — if C is the *concept-name* of a *partial-concept-id*, the concept argument list comprises a single wildcard followed by the *template-arguments* of that *partial-concept-id*.

(3.2) — If C appears as a *template-name* of a *simple-template-id*, the concept argument list is the sequence of *template-arguments* of that *simple-template-id*.

4 The selection of a concept from the set referred to by C is done by matching the concept argument list against the template parameter lists of each concept in that set. For a concept CC in that set to be a viable selection, each argument in the concept argument list is matched against the corresponding template parameters of CC. Default template arguments of CC (if any) are instantiated for each template parameter that does not correspond to a concept argument. Instantiated default arguments are appended to the concept argument list. If the last declared template parameter of CC is not a parameter pack and the number of template parameters of CC is greater than the number of concept arguments, CC is not a viable selection. Otherwise, concept arguments are matched to template parameters using the following rules:

(4.1) — a template argument matches a template parameter if and only if it matches in kind (type, non-type, template) and type according to the rules in 17.4;

(4.2) — a wildcard matches a template parameter of any kind;

(4.3) — a template parameter pack matches zero or more concept arguments, provided that each of those arguments matches the pattern of the template parameter pack using the rules above for matching concept arguments and template parameters.

If any concept arguments do not match a corresponding template parameter, the concept CC is not a viable selection. The concept selected by concept resolution shall be the single viable selection in the set of concepts referred by C. [*Example*:

```
template<typename T> concept bool C1() { return true; } // #1
template<typename T, typename U> concept bool C1() { return true; } // #2
template<typename T> concept bool C2() { return true; }
template<int T> concept bool C2() { return true; }
template<typename... Ts> concept bool C3 = true;
void f1(const C1*); // OK: C1 selects #1
void f2(C1<char>); // OK: C1<char> selects #2
template<C2<0> T> struct S1; // error: no matching concept for C2<0>,
// mismatched template arguments
template<C2 T> struct S2; // error: resolution of C2 is ambiguous,
// both concepts are viable
C3{...Ts} void q1(); // OK: selects C3
C3{T} void q2(); // OK: selects C3
```

— *end example*]

7 Bibliography

The following are key "landmark" historical concepts papers, and most current concepts papers, in chronological order. Note: **P0557R0** and **P0694R0** are considered prerequisites to reading this paper.

[N1536] B. Stroustrup and G. Dos Reis. "Concepts—syntax and composition" (WG21 paper, 2003-10-22).

[N3351] B. Stroustrup and A. Sutton (eds.). "A Concept Design for the STL" (WG21 paper, 2012-01-13). Also known as the original "Concepts Lite design paper."

[N3878] B. Ballo and A. Sutton. "Extensions to the Concept Introduction Syntax in Concepts Lite" (WG21 paper, 2014-01-13).

[N4434] W. Brown. "Tweaks to Streamline Concepts Lite Syntax" (WG21 paper, 2015-04-10).

[P0542R2] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup. "Support for contract based programming in C++" (WG21 paper, 2017-11-26).

[P0557R0] B. Stroustrup. "Concepts: The Future of Generic Programming (or, How to design good concepts and use them well)" (WG21 paper, 2017-01-31). Also known as the "good concepts" paper.

[P0587R0] R. Smith and J. Dennett. "Concepts TS revisited" (WG21 paper, 2017-02-05).

[Van17] D. Vandevoorde, EWG presentation (unpublished, 2017-03 Kona meeting).

[P0464R2] T. Van Eerd, B. Ballo. "Revisiting the meaning of 'foo(ConceptName,ConceptName)'" (WG21 paper, 2017-03-12).

[P0691R0] J. Spicer, H. Tong, D. Vandevoorde. "Integrating Concepts: 'Open' items for consideration" (WG21 paper, 2017-06-17).

[P0694R0] B. Stroustrup. "Function declarations using concepts" (WG21 paper, 2017-06-18). A comprehensive and thorough treatment (and not the first, but the most recent)—prerequisite reading assumed to be understood before attempting this paper.

[P0695R0] B. Stroustrup. "Alternative concepts" (WG21 paper, 2017-06-18).

[P0696R0] T. Honermann. "Remove abbreviated functions and template-introduction syntax from the Concepts TS" (WG21 paper, 2017-06-19).

[P0782R0] E. Keane, A. D. A. Martin, D. Deutsch. "A Case for Simplifying/Improving Natural Syntax Concepts" (WG21 paper, 2017-09-25).

[P0791R0] J. Riedle. "Concepts are Adjectives, not Nouns" (WG21 paper, 2017-10-10).

[P0807R0] T. Köppe. "An Adjective Syntax for Concepts" (WG21 paper, 2017-10-12).