Doc. no.: P1079
Date: 2018-05-6
Programming Language C++
Audience: Evolution
Reply to: Bjarne Stroustrup (bs@ms.com)

A minimal solution to the concepts syntax problems

Bjarne Stroustrup

Abstract

In Jacksonville (March 2018), the evening session on concepts notation expressed two strong views on the concepts syntax in the Concepts TS:

- Prefer independent binding of concepts over consistent binding of types to a concept name
- Somehow indicate that a function is a template function

The vote in favor of something like that was **7 | 33 | 16 | 9 | 2**. In addition, we have a problem distinguishing value concepts from type concepts. Here, I'll discuss three problems

- How do we refer to the type of a constrained variable?
- How do we indicate that a function is a template?
- How do we distinguish type template arguments from value template arguments?

For comparison, I will mostly use Herb Sutter's in-place syntax (see references) proposal that received a **29 | 21 | 11 | 4 | 4** vote. However, given that we already have alternative notations for every use case, I think that proposal is overkill, that it imposes too high a notational burden on common uses, and that we can manage better with less.

At the end, I suggest what I consider a minimal and potentially acceptable solution:

- Bind deduced types for concept names for different arguments and variables independently
- Require **template** to precede a declaration using the natural syntax, e.g.,
  template void sort(Sortable&);
- Surround a concept name introducer with **template< … >**, e,g.,
  template<Mergeable{In1,In2,Out> Out merge(In1,In1,In2,In2,Out)>;
- Use concepts defined with auto parameters to constrain the type of value parameters, e.g.,
  template<auto N> concept Arithmetic_value = Arithmetic<decltype(N)>;

After several and repeated requests, I have added an appendix being specific about what I find wrong with the in-place syntax proposal.

1. Problems and potential problems

First, consider the problems in the context of the TS syntax. Because of long history, existing text, and existing code, I would prefer us not to make dramatic changes if we can avoid it.

1.1. How do we refer to a constrained type?

My intent, as embodied in the Concepts TS, had been for a concept name used in a function declaration to denote the single type to which it was bound ("consistent binding"). For example

```
void sort(Iterator first, Iterator last)
{
        Value_type<Iterator> x;   // Iterator names the Iterator type used in an instantiation
        // …
}
```

According to the Concepts TS, **first** and **last** must have the same type.

Changing from consistent binding as in the TS to independent binding as preferred by the evening session (and by many in other discussions) this would no longer be feasible. For example:

```
void sort(Iterator first, Iterator last)
{
        Value_type<Iterator> x;         // Which Iterator?
        // …
}
```

Now, **first** and **last** can have different types.

Introducing a name for the type(s) bound to a template argument type (as suggested in various forms by many proposals) would solve that. Currently, we might write

```
template<Iterator Iter1, Iterator Iter2>
void sort(Iter1 first, Iter2 last)
{
        Value_type<Iter1> x;
        // …
}
```

Herb's proposal is

```
void sort(Iterator{Iter1} first, Iterator{Iter2} last)
{
        Value_type<Iter1> x;
        // …
}
```

For most such multi-type argument lists, we need a requirement tying the argument types together. For example:

```
template<Iterator Iter1, Iterator Iter2>
```

```
		requires  EqualityComparable<Value_type<Iter1>,Value_type<Iter2>>
	bool equal(Iter1 first, Iter1 last, Iter2)
	{
		// …
	}
```

Such patterns of constraints of template arguments types are quite repetitive, so to avoid cut&paste, we will often end up using a concept type-name introducer. In the currently implemented concepts TS notation (using a simplified version of the Ranges TS's indirect* concepts):

```
	Comparable_through_iterators{Iter1,Iter2}
	bool equal(Iter1 first, Iter1 last, Iter2)
	{
		// …
	}
```

Or if **template<…>** is required (Herb's proposal and §1.5):

```
	template<Comparable_through_iterators{Iter1,Iter2}>
	bool equal(Iter1 first, Iter1 last, Iter2)
	{
		// …
	}
```

Where **Comparable_through_iterators** is defined something like this:

```
	template<typename Iter1, typename Iter2>
	concept Comparable_though_iterators =
		 Iterator<Iter1> &&
		Iterator<Iter2> &&
		requires  EqualityComparable<Value_type<Iter1>,Value_type<Iter2>>;
```

Given that, do we need a new notation to name a bound type?

1.2. When is a function a template function?

Consider

```
	void sort(Sortable& s);
```

People worry (IMO far too much) that it is not obvious that this is a template and would like some "template indicator." The obvious syntax is

```
	template void sort(Sortable& s);
```

I have tried this out of several people and it really is the obvious notation for saying "sort is a template." Unfortunately, unadorned **template** has been taken for explicit instantiation. However, it is not taken for concept arguments, so a compiler can check: if there is a prefix **template** and a concept argument, it's a constrained template.

Naturally, I looked for alternatives

```
template<> void sort(Sortable& s);      //  taken for specialization
```

And

```
template: void sort(Sortable& s);       // no, template is not a label
```

and

```
template! void sort(Sortable& s);       // but why? Why !
```

and

```
requires void sort(Sortable& s);        // recycling of keyword. Yuck
```

and

```
template{} void sort(Sortable& s);      // but why? People complain about overuse of {}
```

and

```
void sort(Sortable& s);                 // strongly opposed in EWG and elsewhere
```

and

```
concept void sort(Sortable& s);         // recycling of keyword. Yuck
```

and

```
* void sort(Sortable& s);               // obscure (as would be every non-keyword notation)
```

and more. Not using plain **template** would allow an unfortunate repurposing of a keyword for an obscure use case prevent an obvious (and demanded) use.

Herb's proposal expresses that a function is a template indirectly by requiring **{}** on at least one parameter:

```
void sort(Sortable{}& s);
```

Currently, we can write

```
template<Sortable S>  void sort(S& s);
```

Is this use sufficiently verbose to warrant a new syntax (assuming that we cannot get the plain natural/terse notation)?

This is not the place for me to repeat the arguments in favor of the natural syntax, see references.

It has been noted that just stating that a function is a function template doesn't say which argument is a concept:

```
template void sort(Sortable&, Predicate);
```

Is **Sortable** or **Predicate** or both concepts? Herb's proposal would make answer obvious. On the other hand,

```
void sort(Sortable&, Predicate{});
```

makes it possible to overlook that **sort()** is a template. I consider this an obscure corner case that shouldn't be allowed to distract from the main issues.

1.3. How do we distinguish value template arguments from type template arguments?

The shorthand syntax has been used in every description of concepts since 2003 and every program using concepts. It is in the WP. Consider

```
template<Sortable S> void sort(S& s);
```

I have never (in 15 years) heard a complaint about that notation from a user.

Unfortunately, when we introduced **auto** to designate a generic template value type argument, we introduced a problem. We now have

```
template<auto n> decltype(n) f();         // unconstrained value
auto x1 = f<1>();          // x1 is an int
auto x2 = f<'1'>();        // x2 is an char

template<typename T> T g(T);              // unconstrained type
auto y1 = g(1);            // y1 is an int
auto y2 = g('1');          // y2 is an char
```

To clarify intent and improve error messages, we would like for the unconstrained **auto** and **typename** to become rare, so we would define numeric concepts to replace **auto** and type concepts to replace **typename**. For example:

```
template<auto N> concept Number1 = Integer<decltype(N)>;

template<typename T> concept Number2 = requires(T a, T b) { {a+b }-> T; };
```

This already compiles with GCC when you add the **bool** required by the TS. Note that it is the type of the value argument that we are constraining (in analogy to **auto**) rather than the value of the value argument. We might need a name for this kind of concept, taking an **auto** parameter. Now our example becomes:

```
template<Number1 n> decltype(n) ff(); // Number1 is a value concept
auto xx1 = ff<1>();        // x1 is an int
auto xx2 = ff<'1'>();      // x2 is an char

template<Number2 T> T gg(T);              // Number2 is a type concept
auto yy1 = gg(1);          // y1 is an int
auto yy2 = gg('1');        // y2 is an char
```

The distinction between the value concept **Number1** and the type concept **Number2** is obvious to a compiler. Could it be too subtle for a programmer? Had I used descriptive names there wouldn't have been a problem for the human reader.

I suspect the real problem is that we would like values of types that match the concepts we otherwise use. That is, we might need pairs of concepts, such as **Arithmetic** and **Arithmetic_value**.

```
template<typename T> concept Arithmetic = requires { /* … */  };

template<auto N> concept Arithmetic_value = Arithmetic<decltype(N)>;
```

We can now write:

```
template<Arithmetic N,  Arithmetic_value n> void f(N);  // type concept and value concept
```

I consider this simple and elegant, and it works today. If that is considered unacceptable and only if that is considered unacceptable, do we need a syntactic marker. Incidentally, this technique mirrors the **_v** and **_t** naming convention for type traits. That is, we already use naming conventions to help people where compilers have no problems.

Herb's proposal uses the position of **{}** to distinguish:

```
template<Arithmetic{N}, Arithmetic {} n> void f(N);         // different uses of type concepts
```

Here **N** is introduced within **{}** and therefore a type name and **n** is introduced outside **{}** and therefore a value name.

I find that **{}**-notation disturbing for three reasons

- It is a unique syntax and "odd looking"
- We now have to redundantly mark every use of a concept with **{}** or **{N}**
- The type name case is the most common (probably the 99% case), yet it requires redundant **{}**s.

The construct seems unique in the language. Having a unique syntactic construct for a rare use is dicey. Its success depends on people liking it and getting used to it. That's part aesthetic but should be considered. I am not the only person who has found it odd.

Having to redundantly mark all uses of a name by its syntactic category comes close to the "abomination level" of language design. This is beyond simple disambiguation because it requires use even for examples where there is no ambiguity. Programmers invariably object, finding it objectionable to have to add redundant and verbose annotation. Remember **struct S** from C and the complaints that led to the recent elimination of redundant **typename** uses? In particular, I have had a constant stream of (unprovoked) complaints about "too many **{}**s" from non-language experts who has read Herb's proposal or (more commonly) reports thereof.

The third reason bothers me a lot.

```
template<Number{N}> void f(N);          // N is a type name
```

This is "odd" and redundant. We don't usually use clustering syntax for a single element. At least, we could make the common, single-type argument concept case default such cases to "type name":

```
template<Number N> void f(N);           // N is a type name
```

That would make most common uses still valid independently of what other design decisions we make.

If we need to have a syntactic marker, it should go on value concepts (only). For example

```
template<Number{} N> void f();          // use the {}: N is a value
```

Alternatively, we could use **auto** (as also proposed under the label "adjective syntax"):

```
template<Number auto N> void f();       // require auto: N is a value syntax
```

or

```
template<auto Number N> void f();       // require auto: N is a value
```

I consider that ugly and it is redundant (as shown above).

Note that in contrast to

```
template<Arithmetic N, Arithmetic_value n> void f(N);  // type concept and value concept
```

Herb's variant

```
template<Arithmetic{N}, Arithmetic {} n> void f(N);     // different uses of type concepts
```

doesn't use a value concept at all. In fact, he seems to argue against value concepts in general (§5.4). I suspect that is based on the mistaken assumption that value concepts simply constrain values.

I expect the type concepts far to outnumber the value concepts, most likely on the order of 100:1, so we shouldn't put a notational burden on type concepts purely for the benefit of value concepts.

1.4. Constrained local variables

In addition to constrained parameters, we have constrained local variables. For example

```
template<Number N> void compute(N x, N y)
{
        N diff = x-y;           // use the argument type
        Number sum = x+y;       // not necessarily an N
        Number mul = x*y;       // not necessarily an N or a decltype(sum)
        // …
}
```

Some see the lack of indication that **Number** is a concept in the body of a function as a problem. I have not seen such problems in actual practice. This use is in the TS but has not (yet) been moved into the WP. Practical experience shows that we need concepts to constrain variables. Code using "plain **auto**" is too hard to maintain (and people do introduce variables in generic code where using specific types would overconstrain).

This minimal syntax doesn't offer a specific way of naming the bound type.However, we can use **decltype** or **auto** when we want to name a type:

```
template<Number N> void compute(N x, N y)
{
        N diff = x-y;                  // use the argument type
        Number sum = x+y;              // possibly a different type from N
        decltype(sum) s2 = sum;        // preserve reference
        auto s3 = sum;                 // dereference
        Number mul = x*y;
        // …
}
```

Herb's proposal allows the optional introduction of a name for a constrained type

```
template<Number {N}> void compute(N x, N y)
{
        N diff = x-y;                  // use the argument type

        Number{N2} sum = x+y;
        N2 s2 = sum;
        auto s3 = sum;
        Number{} mul = x*y;
        // …
}
```

The **{}** is compulsory to indicate that Number is a concept (even if no name for the type is needed).

I have found the need for a name for a deduced type of a local variable rare, so I conjecture that **decltype** will be sufficient; **decltype** can also be used to introduce a name:

```
        Number sum = x+y;
        using N2 = decltype(sum);
```

I suspect that functions where complicated naming structures are needed are too long and complex to be deemed good style anyway.

1.5. Concept type-name introducers

Using a concept to introduce a set of constrained type names is arguably the most elegant and general way of expressing constraints. For example

```
Mergeable{In1, In2, Out}
Out merge(In1,In1,In2,In2,Out);
```

I see no technical problems here, and there has been little discussion, but I have heard the notation referred to as “weird.” The notation is supported by the TS, but not the WP. Herb (and others) have suggested adding an otherwise redundant **template< … >**:

```
template<Mergeable{In1, In2, Out}>
```

```
Out merge(In1,In1,In2,In2,Out);
```

This redundancy may be worthwhile. This also shows how the **{}** fits into the general scheme. The **{}** is only redundant for single type concepts. Note that the multi-argument concept name introducer notion is the only fully general principled notation. Every other syntax become repetitive (and therefore error prone).

The number of repetitive patterns of relations among template arguments is very high. In some form or other, this concept type name introducer is necessary to avoid cut&paste and macros. It has proven itself in real use.

If not using concept type introducers (as available in the Concepts TS and GCC), the but current alternative for **merge() is**:

```
template<typename In1, typename In2, typename Out>
        requires Mergeable<In1, In2, Out>
Out merge(In1,In1,In2,In2,Out);
```

1.6. Lambdas and functions

As far as possible, a generic lambda and a function template should be identical syntactically and semantically. That was one of the aims of the "natural syntax":

```
[](auto x) { … }          // unconstrained

void f(auto x) { … }
```

Currently, only the lambda is allowed, and of course it is a template. Moving to use concepts, we get

```
[](Number x) { … }        // constrained

void g(Number x) { … }
```

Note that I use the type concept, **Number**, rather than a value concept, e.g., **Number_value**. Here, there is no ambiguity to resolve and ***_value** concepts are only for template arguments.

Using **auto** as a function argument was first proposed in 2002. One reason the natural syntax was for years called the terse syntax was because of examples like these and because it was seen as essential to keep lambdas terse. Curiously, **auto** is accepted (in C++17) for lambdas but not for functions.

Currently, we can write:

```
[]<typename N>(N x) { … }

template<typename N> void g(N x) { … }
```

Using concepts, we get:

```
[]<Number N>(N x) { … }
```

```
template<Number N> void g(N x) { … }
```

Herb's proposal would require:

```
[](Number{} x) { … }

void g(Number{} x) { … }
```

2. Discussion of possible solutions

So, can these problems be solved without inventing a whole new syntax? Any new notation will have problems yet to be discovered (the law of unintended consequence). There are several such suggested notations.

An improved notation, should be

- Acceptable: Must resolve the concerns that has been expressed about the current notation
- Simplify common use cases: The common use cases must not be bloated

In addition, improved notation, should offer

- Compatibility: Should not break successful code, except to achieve the goals above
- Easy repair: Where existing code must be broken, ideally there should be an easy rewrite to the new notation.

Note that we already have a fully general notation in the WP, the explicit **requires**-clause:

```
template<typename In1, typename In2, typename Out>
    requires Mergeable<In1, In2, Out>
Out merge(In1,In1,In2,In2,Out);
```

In other words, the only problem is to decide if we want a less verbose notation, and if so what it should be. I think we have two alternatives:

- Go for a comprehensive scheme with heavy syntax everywhere
- Go for a set of modifications aiming for just enough notation to simplify common cases and to disambiguate (to everyone's satisfaction)

IMO, Herb's proposal is the best of breed of the first kind, but I think it is far more than we need and we have yet to explore its implications. Instead, I suggest minimal changes to status quo:

1. Keep what we have in the WP unchanged, in particular, that implies that we keep the most common ("99%") case:

   ```
   template<Number N> void f(N);          // N is a type name
   ```

2. Don't offer any new facilities for distinguishing value concepts. Use naming to help humans distinguish value and type concepts (compilers do not have a problem), e.g.:

```
template<typename T> concept Arithmetic = requires { /* … */ };
template<auto N> concept Arithmetic_value = Arithmetic<decltype(N)>;

 template<Arithmetic T> void f(T);                  // T is a constrained type

template<Arithmetic_value N> void f();              // N is a value with its type constrained
```

If that is not acceptable, use **auto** to designate value parameters (it's a rare "1% or less" case):

```
template<Arithmetic T> void f();           // T is a constrained type

template<Arithmetic auto N> void f();   // N is a value with its type constrained
```

or

```
template<auto Arithmetic N> void f();   // N is a value with its type constrained
```

3. Don't offer any new facilities for naming types in local scopes, use **decltype** and **auto** where needed:

```
template<Number N> void compute(N x, N y)
{
        Number sum = x+y;
        decltype(sum) s2 = sum;
        auto s3 = sum;
        Number mul = x*y;
        // …
}
```

The current language mechanisms are already sufficient.

4. Require **template** before a declaration using the "natural syntax":

```
template void sort(Sortable& s);

template [](Sortable& s) { sort(s); }
```

I am not sure that this **template** is necessary, but I have heard claims that it is and would accept it for consensus.

If the **template** prefix is deemed necessary, the venerable shorthand notation might still be sufficient:

```
template<Sortable S> void sort(S& s);

template<Sortable S> [](S& s) { sort(s); }
```

If you think in terms of terseness, the latter "wastes" only 3 characters.

5. Require **template< … >** for multi-type-name concept introducers:

```
template<Mergeable{In1, In2, Out}>
        Out merge(In1,In1,In2,In2,Out);
```

I am not sure that this **template<** … **>** is necessary, but I have heard claims that it is and would accept this for consensus.

The **<** … **>** around **Mergeable{In1, In2, Out}** is redundant so we might prefer plain

```
template Mergeable{In1, In2, Out}
Out merge(In1,In1,In2,In2,Out);
```

This could be considered in line with the use of plain **template** to indicate a template function.

6. Accept auto for function arguments just as it is for lambda arguments

```
void f(auto x) { … }
```

Having it for lambdas and not functions is plain weird. With or without a (technically redundant) required **template** to help human readers.

This is a suggested solution based on the opinions expressed in evening sessions. Personally, I consider the use of **template** in (4) and (5) redundant.

Potential votes

After reading this, Herb suggested that I made some concrete proposals for people to vote on in Rapperswil. I consider it the WG chair's prerogative to propose and order votes, so this is merely as suggestion as requested by our convener.

Votes

1. Don't change the WP's rules for using **template<** … **>**.
2. Move the "natural syntax" from the concepts TS into the WP, but require the keyword **template** to precede all declarations using a concept as a function parameter type, e.g., **template void sort(Sortable&);** rather than plain **void sort(Sortable&);**, and make the deduction of different constrained arguments independent (as they already are for local variables in the Concept TS). This applies to both functions and templates.
3. Move the constrained local variables from the Concepts TS into the WP.
4. Move the concept name introducers from the Concepts TS into the WP, but require **template<** … **>** around concept name introducers, e.g., **template<Mergeable{In1,In2,Out}>** rather than plain **Mergeable{In1,In2,Out}**.
5. Allow **auto** as a function parameter type exactly as it is allowed for a lambda.
6. Allow the plain **template** prefix (from vote 2) to be used for functions and lambdas with **auto** parameters.

All the tricky wording is already in the Concepts TS, so wording can be generated quite quickly.

References

- Thomas Köppe: *An Adjective Syntax for Concepts*. P0807r0. 2017-10-12.
- Jakob Riedle: *Concepts are Adjectives, not Nouns*. P0791R0 Date: 2017-10-10
- Bjarne Stroustrup: *Concepts: The Future of Generic Programming*. P00557r1. 2017-01-31.
- Bjarne Stroustrup: *Answers to concept syntax suggestions*. P0956r0. 2018-02-11
- Herb Sutter: *Concepts in-place syntax*. P0745R0. 2018-02-11.
- Andrew Sutton: *C++ extensions for Concepts*. P0734R0. 2017-07-14.

Appendix: What's wrong with the in-place syntax proposal

I have had several and repeated requests to be less polite and more specific about what I think is wrong with Herb's in-place syntax proposal. I think it

- solves problems that don't need solving and
- fails to solve problems that do need solving.

My opposition is unlikely to go away even if EWG approves of that proposal and could become a public embarrassment to the WG21 and the C++ community. As always, I'm willing to be proven wrong, so I guess it is fair to be more specific, so we might be able to deal with the problems head on.

First, let me say that IMO, Herb brought forward his proposal in good faith to solve what looked like a deadlock in the committee that could develop into exactly the festering problem I am worried about. Technically, the in-place syntax proposal is not a bad solution to the problems it sets out to solve. However, I think it is a proposal that only a language lawyer (or wannabe language lawyers) could love. Others, will need a long explanation to get the point.

What is wrong with the in-place syntax proposal:

- It adds a way of introducing names for constrained types. But we do not need another one (see §1.1).
- It adds a syntactic distinction between value template parameters and type template. But we do not need one (see §1.3).
- It breaks the most popular notation for the use of concepts (see §1.3),
- It adds verbosity to the most common case (see §1.3).
- It does not increase the similarity between "ordinary programming" and "generic programming" (relates to §1.2).
- The syntax is unique, novel, and untried. Remember the law of unintended consequences.

Uses of constrained type names

What do we need names of template parameter types for?

- To use them in further constraints (i.e., as concept arguments)
- To use them in code

We already have a mechanism for getting the type of a parameter (or a variable) in code (**decltype**), a way of using a type without naming it (**auto**), and a way of naming it (**using**). I see no benefits for adding an alternative to those, and there is to obvious added cost of learning and tooling.

By “to use them in further constraints” I think of examples like this (1.1):

```
template<Iterator Iter1, Iterator Iter2>
        requires  EqualityComparable<Value_type<Iter1>,Value_type<Iter2>>
bool equal(Iter1 first, Iter1 last, Iter2)
{
        // …
}
```

We obviously need to name the types of the **Iterator**s to be able to write the requires clause. As we can see, Concepts TS (and the WP) offers a quite reasonable way of doing that. I see no benefits to this code from writing **template<Iterator{iter1}, Iterator{Iter2}>. F**urthermore, this is quite low-level code: Such patterns of constraints of template arguments types are often quite repetitive, so we should use concept type-name introducers to express commonality

```
template<Comparable_through_iterators{Iter1,Iter2}>
bool equal(Iter1 first, Iter1 last, Iter2)
{
        // …
}
```

Doing that further reduces the need for novel notation for explicit naming in the initializer list itself. We need to gain maturity in our use of concepts. Much early experimentation is (necessarily) primitive – a bit like when novices write their first code in long linear stretches because they have not yet grasped the value of functions.

“Natural syntax”

The “natural syntax” has two declared purposes:

- To help make generic programming more like “ordinary programming”
- To provide a simple and familiar notation for simple common cases

The prefix **template** syntax does that:

template void sort(Sortable&); // *ordinary function notation, bur explicitly for a template*

It is minimally different. Whereas

void sort(Sortable{}&); // *function with an odd argument syntax that makes is a template*

offers a new syntax for templates only.

Yes, people insist on a notation because **&&** works different for ordinary functions and function templates, but that is arguably an unfortunate design and certainly doesn’t deserve to be honored with its own notation. It seems to me as “the tail wagging the dog.” To prevent unlikely confusion from **C&&**, we get **C{}**, **C{}***, **C{}&**, **C{}&&**, **C{T}**, **C{T}***, **C{T}&**, **C{T}&&**, and most likely more.

The shorthand notation

Consider a simple (and classical example):

```
template<Forward_iterator Iter, typename Val>
Iter find(Iter first, Iter last, Val v);
```

I don't think it is improved by the in-place proposal. On the contrary, it adds variations, opportunity for confusion, and adds (minor) verbosity (where it hurts the most). Let's try:

```
template<Forward_iterator{Iter}, typename{Val}>
Iter find(Iter first, Iter last, Val v);
```

Should I use **{}** around **Val**, or is typename a special case:

```
template<Forward_iterator{Iter}, typename Val>
Iter find(Iter first, Iter last, Val v);
```

Maybe, it would be better to name within the function declaration list

```
template<typename Val>
Iter find(Forward_iterator {Iter} first, Iter last, Val v);
```

Or maybe

```
Iter find(Forward_iterator {Iter} first, Iter last, auto v);
```

But since we don't name the type of **v**, must we use **{}** (or can we)?

```
Iter find(Forward_iterator {Iter} first, Iter last, auto{} v);
```

The flexibility (also) yields new ways of confusing ourselves.

Please note that I "forgot" to require **EqualityComparable<Value_type<Iter>,Val>** or something like that. I don't actually think this is a good example of the in-place syntax at all. I prefer either the well-established shorthand notation:

```
template<Forward_iterator Iter, typename Val>
        requires EqualityComparable<Value_type<Iter>,Val>
Iter find(Iter first, Iter last, Val v);
```

Or (better still) if there is a common pattern to exploit, a concept type name introducer:

```
template<Forward_value{Iter,Val}>
Iter find(Iter first, Iter last, Val v);
```

Which (not accidentally) happens to be identical in both proposals.