Doc. no. P0694r0
Date: 2017-06-18
Programming Language C++
Audience: All
Reply to: Bjarne Stroustrup (bs@ms.com)

Function declarations using concepts

Bjarne Stroustrup

Morgan Stanley and Columba University

www.stroustrup.com

Abstract

This paper considers the use of concept names in function declarations. In particular, it considers the rules for binding of types to concept names (§3), counts potential uses of different function declaration notations in the STL and similar libraries (§4), and explores alternatives for consistent notation (§5).

The results do not support changes to concepts, except that we should explore whether the requirement for consistent binding of types to concepts should extend to all scopes as in the original concept design. If so, we should consider if we need a way of defining concepts in a local scope (§7.2).

If we insist on using a single notation style, we must use concept type-name introducers, rather than explicit **requires**-clauses (§5).

The natural declaration syntax is still the shortest and simplest for many generic functions; especially in the simplest and most common cases. For example, it can reasonably be used for about 2/3rd of the STL algorithms and the STL is not the only use case. It is a valuable addition to the language as well as the ideal [Str17]. It emphasizes the fundamental similarity between concepts and types (§6).

1. Introduction and Summary

The use of the natural syntax (conventional syntax, usual syntax, common syntax, terse syntax, abbreviated syntax, functional syntax, simple syntax, whatever; see §8) for function declarations using concepts has unfortunately become controversial. For example:

void sort(Sortable&); **//** *Sortable is a concept*

The rationale for this feature and its place in generic programming can be found in §2, §7, and §8.6 of [Str17]. Users generally love it, but various people have raised technical issues, deemed this syntax potentially confusing, inconsistent with uses of **auto**, and/or unnecessary.

The natural syntax is a necessity: to make generic programming (i.e., programming with templates and concepts) mainstream, we must make it approachable to ordinary programmers. For most programmers and most examples, the fully general notation using explicit **requires**-clauses is a verbose and "heavy" nuisance. By necessity and design, the natural syntax cannot express everything that the fully general notations can. The natural syntax is the notation we use to define "ordinary" functions – and in generic programming generic functions are ordinary. For the multi-parameter concepts with mutual dependencies among the arguments, we need type-name introducers.

This note does not discuss the syntax itself (see [Str17]), but addresses common objections:

- It is wrong that repeated occurrences of a concept in a declaration must bind to the same type.
- It is confusing to have multiple notations for using concepts.
- It is confusing that the meaning of a declaration can differ depending on whether an argument is a type or a concept.
- It is confusing that a concept can be used where we are used to see types.
- The natural syntax is useful for toy examples only.

This note discusses technical issues, addresses concerns about consistency of style, and considers how notation reflects and encourages good programming style. It presents a few numbers from the STL, and points to the concept-based type-introducer notation as an essential part of any solution.

My major conclusions are

1. Repeated uses of the same concept in a scope must refer to the same type. Otherwise, the basic equivalences among the notations are broken, adding complexity, harming comprehension, and complicating maintenance.
2. Independent resolution of concepts in a scope leads to technical problems, including breaking the basic equivalences among notations and complicating the specification of return values.
3. If you want a unique notation for all declarations, you must use concepts as type-name introducers throughout, rather than explicit **requires**-clauses.
4. The likelihood of confusion and errors from the natural declaration syntax is minimal and is caused by irregularities elsewhere in the language.
5. For the STL algorithms, you can use the natural declaration syntax as the simplest notation for about 2/3rd of the algorithms, and the STL style is by no means the only use of the natural notation.
6. Independent resolution of concepts severely limits the utility of the natural syntax.
7. Concepts and types are fundamentally similar and not allowing the natural syntax obscures this similarity, making generic programming and "ordinary programming" unnecessarily dissimilar.

Point [1] suggests a minor change to status quo (reverting to the original concept design). The other conclusions do not suggest changes to status quo. See also §8. My basic conclusion is that status quo is pretty good compared to suggested alternatives. I am not against improvements (see §7), but the suggestions examined here seem not to be improvements.

This paper is longer than what I would have preferred, but initial readers raised a lot of questions that I felt obliged to answer directly. It is easy to raise a concern, but much harder to argue that a potential problem isn't significant or that a suggested alternative has its own problems.

1.1 Overview

The rest of the paper is organized as follows:

2. Uses of concepts

There are three uses of concepts in the current TS:

- As *a name for a type in a declaration*, indicating a constraint on what types can bind to it; in this role, **auto** is the (unnamed) least constrained concept. A concept can be used as a template parameter, a return type, or the type of a named object. For example:

```
void sort(Sortable&);

Output_iterator copy(Input_iterator, Input_iterator, Output_iterator);

void sort(Random_access_iterator, Random_access_iterator,
                Relation<Value_type<Random_access_iterator>>);

void my_sort(Sortable& c)
{
        Random_access_iterator middle = c.end()-c.begin();
        // …
}
```

- As *the introducer of one or more type names*, indicating a constraint on the types that can bind to a type name; in this role **typename** is the least constrained type name introducer. A concept can be used to introduce named types for use in function declarations. For example:

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

template<Sortable Seq> void sort(Seq&);
```

- As *an explicit predicate* for a class declaration or function declaration (an explicit **requires**-clause) or a type in a concept definition. For example:

```
template<typename Seq>
        requires Sortable<Seq>
void sort(Seq&);
```

```
template<typename In1, typename In2, typename Out>
concept bool Mergeable = Input_iterator<In1>
                        && Input_Iterator<In2>
                        && Output_iterator<Out>
                        && Lessthan_comparable<Value_type<In1>,Value_type<In2>>
                        && Assignable<Value_type<Out>,Value_type<In1>>
                        && Assignable<Value_type<Out>,Value_type<In2>>;
```

There have been lots of (mutually incompatible) suggestions for improving concepts by changing or removing one or more of these uses or by changing the meaning of one of those. This note examines some of those suggestions.

2.1 Questions

The questions I'm trying to answer (rationally):

- Is the natural syntax sufficiently useful to deserve support? (yes)
- If so, should we support independent or consistent type-name binding? (consistent)
- What notations could be used exclusively? (concept type-name introducers)
- What notations encourage the kind of generic programming that concepts were invented to support? (the natural declaration syntax and concept type-name introducers).

First, I'll look into the independent vs. consistent binding issue, then look at how the notations can be used in the STL and the Ranges proposal, and finally look at implications for programming style.

2.2 Terminology

I call a construct that introduces a name for a template argument type a *type-name introducer*. There are two forms type-name introducer

- A *template type-name introducer*: **template<typename Iter>** or **template<Input_iterator Iter>**. Here, **Iter** is introduced as a name of the type used as the template argument; **typename** indicates that the type is unconstrained and **Input_iterator** indicates that the type is constrained by the concept **Input_iterator**.
- A *concept type-name introducer*: **Sortable{Seq}**. Here, the name **Seq** is introduced as the name of a template argument that must be Sortable. This form is particularly useful for introducing a set of names for types that together must match a concept. For example, **Mergeable{In,In2,Out}** introduces three names **In**, **In2**, and **Out** for three types that must match the somewhat complicated requirements, **Mergeable**, for the arguments to a merge algorithm.

We have (depending on exactly how you count) four declaration notations:

- The *natural notation*:
    - **void sort(Sortable&);**
- The *original template notation* with explicit **requires**-clauses:

    - **template<typename Seq> requires Sortable<Seq> void sort(Seq&);**
- The *shorthand notation*:
    - **template<Sortable Seq> void sort(Seq&);**
- The *concept introducer notation*:
    - **Sortable{Seq} void sort(Seq&);**

The natural notation can be used only for functions (algorithms). The others apply to classes also.

When two arguments must match the same concept, we must distinguish two kinds of type-name resolution:

- *Independent binding*: Each occurrence of a concept can be bound to a different type
    - **Number operator+(Number a, Number b);** **//** 20+12.23 is allowed

  The types bound to **Number** have no names and there are no connections between the types used for the two Numbers. The concept is used only to constrain the argument. If we must, we can refer to one of those types with **decltype**, e.g., **decltype(b)**.

- *Consistent binding*: All occurrences of a concept must be bound to the same type
    - **Number operator+(Number a, Number b);** **//** 20+12.23 is an error; 20+12 is OK

  The name of a concept acts as a name of the type it matched.

In both cases, **auto** is the least constrained concept.

Status quo is consistent resolution, so given a concept **In**, we can write

In find(In b, In e, Value_type<In> val);

The return type for **In** is the type deduced from the argument for **In**. So is the type used for the third parameter.

Note that the original model of concepts as presented in [Sut13] and in numerous papers and talks before and after the formal proposal species the notations as equivalent:

C f(C,C);

means

template<C __name> __name f(__name, __name);

where **__name** is some name invented by the compiler; this in turn means

template<typename __name> requires C<__name> __name f(__name, __name);

The concept introducer notation for a single type also fits into this pattern:

C{T} T f(T,T);

The language-technical arguments below basically boil down to "don't mess with these equivalences." If we break that fundamental equivalence, we will pay for that in specification complexity, implementation complexity, and user confusion.

In particular, this equivalence means that we can change from one notation to another – for aesthetic or practical reasons – without changes to uses. This is a convenience during initial development and close to essential during maintenance.

3. Multiple occurrences of a concept name

Many algorithms use two arguments constrained by the same concept. For example:

```
void sort(Random_access_iterator, Random_access_iterator);
```

In such cases, must the concept be matched by the same type (twice) or can the two occurrences be independently matched by different types? The answer to this question affects how often the natural syntax can be used and how.

3.1 Current resolution

The original concepts proposal [Sut13] was clear:

> "We can use concepts to restrict that definition.
> ```
> Integer gcd(Integer a, Integer b);
> ```
> This is equivalent to writing:
> ```
> template<Integer T>
> T gcd(T a, T b);
> ```
> Note that repeated uses of the same concept name with a scope bind to the type name, whereas the unconstrained **auto** version allows those types to vary. This makes the use of concept names consistent with the usual rules for type names (i.e., they don't change between uses)."

That is, the original proposal explicitly supports consistent binding for arguments and return types. A slightly less approachable, but presumably more precise, formulation can be found in latest working paper [Sut17] (7.1.6.4 [dcl.spec.auto]. First paragraph on top of page 13):

```
C const& f1(C); // has one template parameter and no deduced return type
```

Note that the return type is consistent with the argument type, rather than being independently deduced.

Consistent binding is status quo, so that

```
void sort(Random_access_iterator, Random_access_iterator);
```

implies that the same type must be used for both arguments

```
void use(vector<int>& v1, vector<double>& v2)
{
        sort(v1.begin(),v1.end());          // OK
```

```
        sort(v1.begin(),v2.end());          // error
}
```

Had the second call passed the compiler, the result would have been a run-time error.

The consistent resolution is ideal for STL iterator pairs and many other uses, but not for every use.

We can also use a concept twice in a block:

```
void use2(auto x, auto y)
{
        Number xx = f(x);
        Number yy = g(y);         // must yy be the same type as xx?
        // …
}
```

The original proposal [Sut13] required consistency:

> “Also, 7.1.6.5 Constrained type specifiers [dcl.spec.constrained]: The first use of concept-name or partial-concept-id within a scope binds that name to the placeholder type so that subsequent uses of the same name refer to the same type.”

However, this was not emphasized and the original designers agreed at the time that independent resolution might be reasonable. Later, for reasons I don’t exactly know, that requirement was reversed, so that the TS supports independent resolution for variables in a block.

So currently, **xx** and **yy** can be of different types, just as **x** and **y** can. This is appropriate for many uses, but not for all.

That implies that whether two occurrences of a concept must bind to the same type is currently context dependent. That’s technically inconsistent and potentially confusing (§3.6).

The use of a concept as a template type argument is defined in a declaration. For example:

```
void sort(Random_access_iterator, Random_access_iterator,
                        Relation<Value_type<Random_access_iterator>>);
```

However, that use in a function body is not described in the TS, so this is unspecified:

```
void sort(Forward_iterator& b, Forward_iterator& e,
                        Relation<Value_type<Forward_iterator>>)       // OK
{
        vector<Value_type<Forward_iterator>> v {b,e};          // currently unspecified
        sort(v);
        copy(v.begin(),v.end(),b);
}
```

Note that ***not*** having this facility breaks the fundamental equivalence of the notations (§2.2). For simplicity and consistency, we should support this usage. Alternatively, we would need to use workarounds (e.g., using **decltype(*b)**).

3.2 Simple alternatives

There are exactly three fundamentally simple alternative rules for binding to a concept

1. All occurrences of a named concept a declaration (incl. definitions) are independent
2. All occurrences of a named concept in a declaration (incl. definitions) must refer to the same type ("be consistent")
3. A concept can occur in a declaration at most once

Unfortunately, none of these is a complete solution. That is, for each, there are important use cases that require a more flexible approach. The next sections explore solutions based on these three alternatives.

3.3 Alternative 1: Independent resolution

For this section, assume that different occurrences of a named concept can bind to different types. That implies that we can no longer just write the **std::sort** like this:

```
void sort(Random_access_iterator, Random_access_iterator);
```

That would imply that we accepted **sort(v1.begin(),v2.end())** where the two iterators are of different types, which would be an error for **std::sort**. To get consistent resolution as required for **std::sort**, we would have to write something like

```
template<Random_access_iterator Ran> void sort(Ran, Ran);
```

Ran is a type, so the two arguments must (modulo implicit conversions) be of the same type. As a notation, that's not too bad.

If we considered explicit use of **requires** important , we could write

```
template<typename Ran1, typename Ran2>
        requires Random_access_iterator<Ran1> &&
                 Random_access_iterator<Ran2> &&
                 Same_type<Ran1,Ran2>
 void sort(Ran1, Ran2);
```

That's so verbose that I don't consider that realistic for real-world code.

The independent resolution doesn't handle return types well. Consider:

```
Output_iterator copy(Input_iterator, Input_iterator, Output_iterator);
```

If the concept used to constrain the return type is independent of the concept used in the arguments specifications, we can only deduce its type from a return statement in a function definition (see §6.1). That renders declarations that are not also definitions useless.

Independent resolution is insufficient to express the STL pair-of-iterators idiom. With generalization of the STL to accommodate ranges, this idiom will no longer be as dominant as it is today. However, the STL style is used very widely and will remain important for many years ("forever") and it is not the only idiom that relies on pairs of arguments of the same type. In addition, this problem with independent resolution appears whenever we use the idiom of returning an object of an argument type. For example:

```
Value next(Value);
Value compute1(Value,Value);                // homogeneous operation
Value compute2(Value,Other_value);          // heterogeneous operation
vector<Value> compute3(vector<Value>);
```

Thus, independent resolution would dramatically reduce the usefulness of the natural syntax.

Note that independent resolution breaks the fundamental equivalence of the notations (§2.2).

3.4 Alternative 2: Consistent resolution

Currently, we require the same type to match all occurrences of a concept in a declaration, simple examples, like **std::sort**, work as written:

```
void sort(Random_access_iterator, Random_access_iterator);

Input_iterator find(Input_iterator, Input_iterator,

        Equality_comparable<Value_type<Input_iterator>>);
```

The problem comes when we need to match with multiple types; **std::merge** is the classic example:

```
Output_iterator merge(Input_iterator, Input_iterator,  // Wrong!
                      Input_iterator, Input_iterator,
                      Output_iterator);
```

Here, the first pair of **Input_iterator**s must bind to the same type, the second pair of **Input_iterator**s must bind to the same type, but the two types bound to the two pairs of **Input_iterator**s may differ. Thus, neither independent resolution nor consistent resolution is sufficient to express **merge()**.

Again, the conventional solution is to use **template** to introduce type names:

```
template<Input_iterator In1, Input_iterator In2,  Output_iterator Out>
Out merge(In1, In1 In2, In2, Out);
```

Again, that's not too bad as a notation and the shorthand notation is essential for avoiding bloat. Unfortunately, this is insufficient in most such cases because when there are different concepts involved, the relationships among the iterators typically need to be specified. For example:

```
template< Input_iterator In1, Input_iterator In2,  Output_iterator Out>
        requires Equality_comparable<Value_type<In1>, Value_type<In2>> &&
                Assignable<Value_type<Out>,Value_type<In1>> &&
                Assignable<Value_type<Out>,Value_type<In2>>
Out merge(In1, In1, In2, In2, Out);
```

This is pretty verbose and gets repetitive because complicated functions tend to come in clusters with similar requirements. For example, the STL offers 7 merge functions and 20 set algorithms.

Repetitive use of the same or very similar code is usually considered bad style and is a source of errors. The solution (the currently deployed solution) is the use concepts as type-name introducers:

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

Here, **Mergeable** is the concept that requires the three types and **{In1,In2,Out}** names the types. This solution works nicely independently of whether the natural declaration syntax resolves types independently or uniformly.

The concept **Mergeable** expresses the requirements and encapsulates the complexity:

```
template<typename In1, typename In2, typename Out>
concept bool Mergeable = requires Input_iterator<In1> &&
                                Input_iterator<In2> &&
                                Output_iterator<Out> &&
                                Equality_comparable<Value_type<In1>, Value_type<In2>> &&
                                Assignable<Value_type<Out>,Value_type<In1>> &&
                                Assignable<Value_type<Out>,Value_type<In2>>;
```

3.5 Alternative 3: Ban repeated use of a concept

If a declaration can mention a concept only once, we can't get the problems involved with resolving the types bound to multiple uses of a concept. Unfortunately, this dramatically limits the expressiveness of the natural syntax and eliminates many non-problematic uses (e.g., 161 algorithms out of 271 in the STL; see §4.1). I mention this alternative just for completeness and I will not explore it further.

3.6 Concepts in function bodies

In addition to using concepts for parameters and for the return type of a function declaration, they can be used to constrain the result of an expression (usually a function call). Assume that we have a **Number** concept:

```
Number operator+(Number,Number);

Number  x = 1+2;        // OK
Number  y = 1+2.0;      // OK?
```

For some applications, we want just one number type; in others, we want a set of numbers with implicit conversions. Replicated concepts are here to stay and some need independent resolution and some need consistent resolution. Whatever resolution we decide on will be a default and we need a way to express the alternative. We can express this using type-name introducers (see §5). For example:

```
template<Number N1, Number N2 = N1, Number N3 = N1>
N3 operator+(N1,N2);

Number  x = 1+2;        // OK
Number  y = 1+2.0;        // OK
```

Alternatively, we can give a concept another name. For example:

```
template<typename T> concept bool Number2 = requires Number<T>;
template<typename T> concept bool Number3 = requires Number<T>;

Number3 operator+(Number,Number2);

Number  x = 1+2;          // OK
Number2  y = 1+2.0;       // OK
```

Unfortunately, such extra names cannot be introduced inside a body (we don't have local templates; see §7.2).

Language support for such numbering has been suggested (§7), but we should only adopt such an extension if the number of uses warrants it. I doubt they do.

For independent resolution, we would need some language extension to express consistent resolution where needed (§7.1).

Whichever resolution we choose, we must support some form of this common idiom:

```
void sort(Forward_iterator b, Forward_iterator e)
{
        vector<Value_type<Forward_iterator>> v {b,e};
        sort(v);
        copy(v.begin(),v.end(),b);
}
```

That is, we need to get the value type of an argument (e.g., an iterator, container, or smart pointer). Since there is no initializer for the **Forward_iterator** in the **Value_type<Forward_iterator>** we must choose consistent resolution or use a workaround for expressing **Value_type<Forward_iterator>**, such as **decltype(*b)**. Not all type expressions have simple workarounds.

For consistency, this should probably be extended to repeated uses of a concept in a scope (§3.4):

```
Number operator+(Number,Number);

template<typename T> concept bool Number2 = requires Number<T>;

void f()
{
        Number  x = 1+2;          // OK
        Number  y = 1.0+2.0;      // error: different type bound to same concept in a single scope
        Number2  y = 1.0+2.0;     // OK
}
```

This use of a second named concept to solve the problem of expressing identical constraints while still allowing different types to be bound it simple, general, and works today. It does lead to a proliferation of names, but must be considered if/when we consider language extensions related to name introduction (§7.2).

Note that this issue reappears in classes. Consider:

```
class X {
        // …
        Number n1;
        Number n2;
        void f1(Number,Number);
        Number f2(Number,Number);
        // …
};
```

The use of concepts as data member constraints is currently not supported, but it probably should be. The two **Number**s in **f1** must refer to the same type and the three **Number**s in **f2** must refer to the same type, but not necessarily to the **Number**s in **f1**.

I do not know how serious the issue of consistent vs. independent binding in function bodies (and classes) is and do not propose any change until we have some better analysis and hopefully some usage numbers. What little I have seen supports consistent binding, but there are examples where independent resolution is more appropriate (exactly as for declarations). It is not possible to have a simple general language with ideal defaults for every use case.

3.7 auto and concepts

I often explain that **auto** can be seen as the least constrained concept. For example

```
void f(auto);     // first proposed in 2002!
```

is a function that can take a value of any type as an argument. This declaration is equivalent to

```
template<typename T> void f(T);
```

so **typename** is the name of the least constrained concept.

Some have interpreted that to mean that we could define **auto** as

```
template<typename T> bool auto = true;
```

They then tried to deduce the meaning of all uses of **auto** from that. That is being too literally minded. We did not define **auto** that way and the keyword **auto** has been reused with slight variations in meaning since its introduction. Consider

```
auto fct(int,double) -> double;  // auto is a syntactic place holder(only)

auto fct2(int i) { return g(i); }     // auto indicates that the return type is deduced
```

```
template <auto x> constexpr auto constant = x;          // from P0127R2

auto v1 = constant<5>;            // v1 == 5, decltype(v1) is int
auto v2 = constant<true>;         // v2 == true, decltype(v2) is bool
auto v3 = constant<'a'>;          // v3 == 'a', decltype(v3) is char
```

What I mean by "**auto** can be seen as the least constrained concept" is that the constraints imposed by **auto** are those of the least constrained type concept, that is "the argument must be a type." In my experience, only language lawyers are confused by that.

Naming a concept and using it repeatedly is different from just considering the minimal constraint of type-binding imposed by the keyword **auto**. Consider:

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

C f(C c1, C c2, auto a1, auto a2);
```

Here, the four arguments are not constrained, we just have to decide whether the name **C** has to be consistently used or the bindings are independent. The issue of how to use names is separate from the (non-controversial) issue of how to constrain types. Both choices are logically possible and both choices have their problems that would need to be addressed. I consider the (current) requirement of consistency a useful tool (e.g., see §4 and §5.3).

I suspect that some the attraction of independent resolution of concept names comes from the view that **auto** (unconstrained genericity) is somehow "normal" or the ideal. I think that's completely against the notion of generic programming as defined by Alex Stepanov and embodied in the current concept design: Algorithms should be expressed in terms of concepts and unconstrained arguments should be rare and viewed with suspicion. That is, **auto** and **typename** are relicts of an imperfect past where we could not express concepts directly and should as far as possibly be eliminated: **auto**/**typename** is generic programming's equivalent to **void***, the lowest level and least specific way of expressing something. I know that there are uses of templates beyond Alex's vision of generic programming, but I don't think those should drive the design of concepts. It would be the tail wagging the dog: unconstrained genericity is already well supported.

For completeness, we must also consider the use of **auto** and concepts in function bodies:

```
void use(vector<int>& vi, list<int>& lsti)
{
        auto b = vi.begin();
        auto e = lsti.end();

        Forward_iterator b2 = vi.begin();
        Forward_iterator e2 = lsti.end();

        // …
}
```

The two uses of **auto** are (of course) independent. Currently, the two uses of **Forward_iterator** are also independent and that is the only place in C++ where the same unqualified type name can have different meanings in the same scope (which wasn't part of the original concept design [Sut13]). I think the change was a mistake. For example, the examples above are likely bugs. If we want to allow different bindings to the same name in the same scope, we can use different names. Requiring consistent resolution within each scope (including function parameter scopes) is the most useful and most consistent (pun intended).

Also, ***not*** having consistent resolution here breaks the fundamental equivalence of the notations (§2.2). See also §6.

3.8 Lambdas

One of the reasons for the natural syntax is to simply constrain lambdas. Consider:

```
sort(v.begin(), v.end(), [](auto b, auto b) { return *a>*b; });
```

This postpones checking of ***a** and ***b** until instantiation time. We are back to unconstrained calls and consistency vested in implementation code. With concepts that becomes

```
sort(v.begin(), v.end(), [](Iterator b, Iterator e) { return *a>*b; });
```

Until C++17, that was the only way, but soon we can write:

```
sort(v.begin(),v.end(),[]<Iterator Iter>(Iter b, Iter e) { return *a>*b; });
```

This implies some the tradeoff between among notations become available when writing lambdas. The usual questions about aesthetics and the frequency of use apply here, but since lambdas are on average shorter and simple than library functions, I think that the natural syntax has a distinct advantage here.

4. Usage numbers

One way to look at the choice between independent and uniform resolution is to estimate how many function declarations will be simplified by each. That's easier said than done, though, because the relatively low amount of code using the feature (non-student users of concepts are holding back because the ongoing complaints about the natural notation gives an impression of instability).

4.1 The current STL

When we first designed concepts, we looked at the STL (see the Palo Alto TR [Str12]). Looking at the STL algorithms in C++17 (N4659, section 28), we find:

- 29 with no repeated concepts
- 142 where all repeated concepts must bind to the same type
- 1 where all repeated concepts can bind to different types (**iter_swap**)
- 79 where some repeated concepts can bind to different types and some replicated concepts must be the same

I counted by hand, so the numbers may be slightly imprecise.

These numbers give an idea why we chose consistent resolution: 171 algorithms can be declared with the natural notation without trickery, 80 cannot. If we defaulted to independent resolution, 215 algorithms would need to somehow have a same-type constraint expressed.

The 80 algorithms for which the resolution of replicated concepts can differ all require some relation between those concepts to be expressed. This implies that a type name introducer technique is likely to be superior. The **template** type-name introduction plus **requires** clauses tends to get verbose. Compare:

```
template< Input_iterator In1, Input_iterator In2,  Output_iterator Out>
        requires Equality_comparable<Value_type<In1>, Value_type<In2>> &&
                Assignable<Value_type<Out>,Value_type<In1>> &&
                Assignable<Value_type<Out>,Value_type<In2>>
Out merge(In1, In1, In2, In2, Out);
```

That's larger than the body of **merge()**.

The concept type-introducer syntax reduces that significantly and reduces opportunities for mistakes:

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

The difference between the two styles is exactly equivalent to the difference between expressing an action as inline code and as a function call. Replicating code in several places is usually considered bad programming, a source of errors, and a maintenance hazard. We should not encourage that.

Of the 171 algorithms for which repeated concepts must bind to the same type

- 113 Need to express one or more constraints among argument types
- 42 Do not need to express constraints among argument types

Typically, an algorithm that does not need added requirements is the simpler and more common of a pair where the other has a value or predicate extra argument. For example (using conventional **template** type-name introducer):

```
template<Random_access_iterator Ran> void sort (Ran, Ran);

template<Random_access_iterator Ran, Relation Comp>
        requires Callable_with<Comp,Value_type<Ran>>
void sort(Ran, Ran, Comp);
```

Given a suitable definition of **Relation** this is easily expressible with the natural syntax**:**

```
void sort (Random_access_iterator, Random_access_iterator);

void sort(Random_access_iterator, Random_access_iterator,
                Relation<Value_type<Random_access_iterator>>);
```

Since **Random_access_iterator** is such a long name and used three times, we might prefer to use it as a type-introducer

```
Random_access_iterator {Ran}
void sort (Ran, Ran);
```

```
Sortable_with{Ran,Pred}          // Pred must be a Relation<Value_type<Ran>>
void sort(Ran, Ran, Pred);
```

How many STL algorithms could reasonably be expressed using the natural notation? Fundamentally, the natural notation has an advantage when the arguments can be expressed so that dependencies among argument can be expressed strictly linearly left to right. For example

```
void sort(Random_access_iterator, Random_access_iterator,
                Relation<Value_type<Random_access_iterator>>);
```

When that is not the case, the natural notation easily gets clumsy. Consider:

```
template<Input_iterator In, Output_iterator Out>
        requires Assignable<Value_type<Out>,Value_type<In>>
Out copy(In b, In e, Out);
```

This sequence-in-sequence-out idiom is very common in the STL. This implies that we should look for a concept to express this pattern (§3.4). Consider

```
IO_iterators{In,Out} Out copy(In b, In e, Out);
```

What we should ***not*** do is to repeat the pattern 100 times through cut and paste.

We can linearize copy and use the natural notation, but it is repetitive and the conventional names are long:

```
auto copy(Input_iterator b, Input_iterator e,
            Output_iterator<Value_type<Input_iterator>> Out) ->
Output_iterator<Value_type<Input_iterator>> Out)
```

I cannot recommend that.

Excluding algorithms using the sequence-in-sequence-out pattern, I count 151 algorithms that arguably are best expressed using that natural syntax. Some look verbose and repetitive, exactly like the current template notation with long names that can get in the way of readability.

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

In both the pair-of-iterator style and the sequence-abstraction style of algorithms, about two thirds of the algorithms are easily expressed using the natural declaration notation:

```
void sort (Sortable&);
void sort(Sortable&, Relation<Value_type<Sortable >>);
```

The concept type introducer notation is the obvious fully general alternative:

```
Sortable{Seq}  sort (Seq&);
Sortable_with{Seq,Comp} void sort(Seq& ,Comp);
```

Explicit use of **requires** clauses is rarely the shortest or cleanest notation.

If we have a sequence abstraction, **Seq**, we need to decide whether to return a sequence or an iterator for most STL algorithms:

```
Seq find(Seq,Equality_comparable<Value_type<Seq>>);                    // one way
auto find(Seq,Equality_comparable<Value_type<Seq>>) -> Iterator<Seq>;   // another way
```

We need to compare the value types of iterators so often that we should probably have a concept for that (as some libraries do, e.g., Ranges' **IndirectPredicate**):

```
Seq find(Seq, Indiret_comparable<Seq>);                     // one way
auto find(Seq, Indirect_comparable<Seq>) -> Iterator<Seq>;   // another way
```

Using a sequence abstraction makes the STL clearer and the notation shorter, but it doesn't change the number of algorithms for which the natural syntax is reasonable: 151.

4.3 Ranges

Ranges (N4651) is a prime example of a modern library based on concepts. Compared to the original STL, the relationships among template arguments are plentiful, varied, and subtle. It seems a candidate for simplifying and regularizing its definition using concept type-name introducers. Consider a relatively example:

```
// 10.4.1, copy:
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
        requires IndirectlyCopyable<I, O>()
tagged_pair<tag::in(I), tag::out(O)>
copy(I first, S last, O result);

// 10.4.2, move:
template <InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
        requires IndirectlyMovable<I, O>()
tagged_pair<tag::in(I), tag::out(O)>
move(I first, S last, O result);
```

With a small change to **IndirectlyCopyable** and **IndirectlyMovable**, this could be simplified to something like:

```
// 10.4.1, copy:
IndirectlyCopyable{I, O}
```

```
tagged_pair<tag::in(I), tag::out(O)> copy(I first, Sentinel<I> last, O result);

// 10.4.2, move:
IndirectlyMovable{I, O}
tagged_pair<tag::in(I), tag::out(O)> move(I first, Sentinel<I> last, O result);
```

Experiments with simplifying notation are useful because they can indicate potential generalizations and/or simplifications.

My favorite example, sort, has mutated into:

```
// 10.5.1.1 sort [sort]
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{});

template <RandomAccessRange Rng, class Comp = less<>, class Proj = identity>
        requires Sortable<iterator_t<Rng>, Comp, Proj>()
safe_iterator_t<Rng>
sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

Due to its complexity/flexibility and heavy use of default template arguments, this does not appear to lend itself to simplification using the natural syntax. I doubt that Ranges is the kind of library that we can make easy for non-experts to read and write.

Using default arguments, we could possibly reduce syntactic noise further:

```
template<typename I, typename C = less<>, typename P = identity>
concept bool Sortable = /* … */;

// 10.5.1.1 sort [sort]
Sortable{I, Comp, Proj}
I sort(I first, Sentinel<I> last, Comp comp = Comp{}, Proj proj = Proj{});

Sortable_range{Rng, Comp, Proj}
safe_iterator_t<Rng>
sort(Rng&& rng, Comp comp = Comp{}, Proj proj = Proj{});
```

Note that if we wanted to have just one **Sortable** concept, it would have to be overloaded on **Iterator** and **Range**.

Please note that my purpose here is ***not*** to redesign Ranges, but just to point to possibilities for cleaner expression of ideas.

4.4 Conclusion

We cannot make conclusions from numbers alone, but I think that the natural syntax appears to serve its intended use reasonably well. Simple uses can be expressed simply and as the complexity of the interfaces increase, so does the need for more elaborate specifications. In particular, the concept type-name introducers seem most useful for dealing with complexity.

I suspect that the STL (in its various disguises) is about as hard a test as we can apply. The demands on foundation libraries are greater than on most libraries. That is, the STL and Ranges provide the kind of interfaces for which I expect the natural syntax to be least useful. To contrast, I have recently seen networking and graph libraries that could be expressed exclusively by the natural syntax.

The STL pair-of-iterators style isn't going away. For the foreseeable future, the STL and its style will be common, and will be used with concepts. Moving the standard to Ranges will not change that, though obviously it will eventually lead to a reduction in frequency of use. Also, many lambdas will be using the pair-of-iterators style in application code such uses will not always easy to find and update to "more modern styles."

5. Consistent style

There is an argument for using a single style for all declarations (incl. definitions) in a library.

5.1 Explicit **requires**-clauses

Perfect consistent style can be achieved using the most verbose and expressive form

```
template<typename In1, typename In2,  typename Out>
        requires  Input_iterator<In1> &&
                  Input_iterator<In2> &&
                  Output_iterator<Out> &&
                 Equality_comparable<Value_type<In1>, Value_type<In2>> &&
                 Assignable<Value_type<Out>,Value_type<In1>> &&
                 Assignable<Value_type<Out>,Value_type<In2>>
Out merge(In1, In1,  In2,  In2, Out) ;
```

A good argument can be made for the shorthand notation being consistent with that

```
template<Input_iterator In1, Input_iterator In2,  Output_iterator Out>
        requires Equality_comparable<Value_type<In1>, Value_type<In2>> &&
                 Assignable<Value_type<Out>,Value_type<In1>> &&
                 Assignable<Value_type<Out>,Value_type<In2>>
Out merge(In1, In1,  In2,  In2, Out) ;
```

Even this simplification can lead to vigorous arguments about whether, where, and when to use the shorthand notation in preference to explicit **requires**-clauses (I have heard them).

However, it is usually considered bad programming style (and error-prone) not to factor out common code patterns. Stronger: the most fundamental reasons for introducing language support for concepts include

- express the idea of fundamental concepts of an application area directly
- discourage fragmentation of interfaces with minute differences that add complexity, complicate understanding, and sometimes hinder interoperability.

Expressing constraints on an interface as a long list of individual properties seems to be a consequence of explicit **requires**-clauses. There are people who like explicit **requires**-clauses exactly because they can provide different constraints for each algorithm, rather than applying the foresight needed for systematic generalization. One reason some of us reacted strongly against the C++0x concepts was the proliferation of tiny "concepts" that were meaningless on their own (had no semantics). We used 104 concepts to express the standard-library algorithms! I fear that overuse of explicit **requires**-clauses is pushing us in this direction again. Thus, I cannot recommend explicit **requires**-clauses as the basis for a consistent generic function declaration style.

5.2 Concept type introducers

To properly factor out requirements into semantically meaningful concepts, we can use concept type-name introducers. For example:

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

This saves us from code replication, and – if systematically used to express general notions – from unmanageable designs.

If we do that, we must base our perfectly consistent style on the concept type-name introducers. In that case, we get

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

Sortable{Seq,Comp}
void sort(Seq&, Comp);
```

Assuming that the second argument to **Sortable** has a default.

5.3 The natural notation

The natural declaration syntax is (by design) not fully general, so we cannot use it as the only style for an STL-like library. In particular, two mutually dependent types cannot be expressed:

```
template<Forward_iterator For1, Forward_iterator For2>
        requires Equality_comparable<Value_type<For1>,Value_type<For2>>
bool same(For1, For1, For2, For2);
```

However, I personally don't feel the need for perfect stylistic consistency (the onion principle applies [Str17]) and prefer to use the natural syntax for the simplest cases (for which it was devised):

```
void sort (Sortable&);

void sort(Sortable&, Relation<Value_type<Sortable>>);
```

The language should allow both alternatives and leave the choice to style guides.

6. The meaning of declarations

Consider

```
void f(X a);
```

What does this mean? **X** could be a type or a concept. In the latter case, **f** is a template.

It has been suggested that this could be a source of confusion and teaching problems. This doesn't bother me at all and I have seen no real-world problems with that. Over the last few years, I have taught people of a wide variety of backgrounds with widely varying degrees of experience with C++. This notation has never emerged as a significant concern. On the contrary, the natural syntax has been popular and helped people progress from simpler forms of programming (non-generic and unconstrained generic programming) to concept-based generic programming.

Of course, someone at some point is going to be surprised and confused by that syntax, making a decision based on the assumption that **X** is a type and **f** is an ordinary function rather than a concept and a template function (or vice versa). What kind of surprises? And what would be the consequences? I can't prove a negative ("nothing bad can happen"), but I can argue that it will be less of a problem than many other problems in the language and therefore minor compared to the benefits. I addressed aspects of this in §8.6 of [Str17], as did the original concepts proposal [Sut13].

Concepts are not as different from types as some people think. Concepts can be seen as a generalization of types. That is, a type imposes a set of constraints defined by a built-in type or expressed by a programmer as a user-defined type. A concept imposes a set of constraints defined as a predicate. Consider:

```
template<class T> concept bool Int = Same_type<T,int>;

Int ff(Int i) { return ++i; };

void user()
{
        Int x = ff(2+1);
        Int y = ff(x);
        Int z = ff(x+y);
        Int zz = 'c';       // error: a char is not an Int
        Int xx = 2.0;       // error: a double is not an Int
        ff(2.0);            // error: a double is not an Int
}
```

Here, **Int** is an **int** stripped of its implicit conversions. I could easily add the value-preserving conversions to **Int**. Both versions work today. Why should I have to declare functions taking **Int**s different from functions taking **int**s? In which sense is **Int** not a type?

As ever, note that the natural notation is not and never was meant to be the only syntax. When something isn't easily and naturally expressed using the natural syntax, use concept type-name introducers. The equivalence of notations ensures that we can smoothly change from one notation to another.

6.1 Potential definition problems

Consider a few examples of suggested potential problems with function definitions.

Consider

```
void f(X x)
{
        // …
}
```

Is **f** a template or an ordinary function? The lookup rules for the body of **f** differs depending on that (two-phase lookup or not). However, the compiler will catch most results of naïve confusion and experts will know what they are doing.

Consider:

```
void f(X x)
{
        static int sss;
        // …
}
```

If **X** is a type, there will be one **sss** in the program. If **X** is a concept, there will be one **sss** per instantiation. What a programmer would expect is not obvious. Personally, I would expect one **sss** per function just as the TS requires. I would be very surprised if **sss** suddenly became a shared object, potentially requiring synchronization for access.

Consider:

```
void f(X x)
{
  // …
}
```

If **X** is a concept, **f** should be in a header. If **X** is a type, **f** should be in a **.cpp** file with a declaration in a **.h** file somewhere. As pointed out in [Str17], this is an old problem that is independent of the type system and that will be addressed by modules.

Consider:

```
void f(X x)
{
  X::type t;
  // …
 }
```

If **X** is a concept, we need to add **typename**. If **X** is a type, **typename** is outlawed in C++11, but allowed in C++17. That’s another example of a minor problem caused by unintended and unanticipated language irregularities unrelated to concepts. One reason this problem is minor is that the compiler will consistently catch such mistakes.

My fundamental answer to problems related to function definitions and concepts as arguments is: a programmer modifying a function must understand the basic semantics of its arguments and the basic concepts of the application. Yes, it can be less than trivial for functions that are huge, programmers with limited experience, and if support tools (such as code coloring) is unavailable. However, I don't see this being major compared to other real-world problems. Compilers, testing, and simple code reviews will catch such problems.

If you feel tempted to address such possible confusions with heavier syntax, please re-read [Str17]. Once a "heavy notation" is widespread, it becomes widely disliked as verbose.

6.3 Potential usage problem

Consider

```
void f(X a);
```

Is **f** a template or an ordinary function? The concern about this is similar to the worries expressed about the possibility that **f** might be overloaded. Yes, you could get a surprise from overloading, but we have lived reasonably happily with that for 34 years and I think that a (constrained) template is less surprising because it is better behaved than a random overload (overloading is an ad-hoc form of genericity).

Why would you want to know? In generic programming, a generic function is the norm. We tend to know the names of the fundamental types in an application, and when we spot one we do not know and we need to understand something that's not obvious from its name, we look it up. If you like code coloring (syntax highlighting), just look at the color of **X**. As said in [Str17], I heard such concerns when I removed the need to prefix every mention of a user-defined type by **struct**. To this day, there are C programmers who prefer to write **struct S** instead of plain **S** when writing C++, but I think that there is a net increase in readability from using plain **S**.

Consider:

```
C f(C,X);       // #1
C f(C,C);       // #2
X f(C,X);       // #3
auto f(C,X);    // #4
```

Assuming **C** is a concept and **X** is a type, the return type in #1 and #2 would be the type bound to the argument **C**. The return type in #3 would be **X**, and the return type of #4 would be deduced from a **return**-statement. It has been suggested that this is potentially confusing. I don't see that: If the name of a return "type" is the same as an argument name, it is the same type as the argument.

Note that

```
auto f(C,X) -> C;       // #5
C f(C,X);               // #6
```

Are equivalent. In each case the return value must be of the type bound to **C**. If anything is confusing here, it's the use of **auto** as a placeholder for a return type (which is unconstrained by default).

This currently works:

```
auto f(C,X) -> auto;      // #7
auto f(C,X);              // #8
```

This is a use of **auto** as the least constrained concept.

Consider:

```
C f();    // C is a concept
```

Is **f** a template or an ordinary function? Why should I care? Obviously, **f** has no dependent names, so there is no difference between one-phase and two-phase lookup. If it is a template, we have to stick it in a header until we have modules, but I doubt it is a big deal either way.

By notational equivalence, **C f();** means

```
template<class T> requires C<T> T f();
```

So **f** is a template.

Consider:

```
C f(C x) { return f(x); } // C is a concept
```

There is a potentially significant difference between independent and consistent resolution lurking here: Is the type of **f(x)** just constrained by an (independent) **C** or is it converted to the type bound to the **C** argument? Status quo is the latter, and I think that's good. Independent resolution would imply that declarations that were not also definitions would not be possible, and that every function that needed a return type constrained would need to rely on deduction (or in-body workarounds).

Consider:

```
void f(X&& a);
```

If **X** is a type, an argument must be an rvalue, but if **X** is a concept, we are dealing with a template and **X&&** is a forwarding reference that can bind to an lvalue. That's obviously a significant sematic difference that could affect a caller.

However, it is not a problem that stems from concepts of the declaration syntax. It is a result of us deciding to overload the meaning **&&** for function parameters.

```
#include “operations.h”
// …
X x = init;
f(x);
g(x);    // safe?
```

Just by looking at this, we cannot (without looking in **operations.h**) tell if **f(x)** will compile and if so if it modifies the value of **x**. If **f()** is a template, the programmer must look at its definition to see if it really forwards. If **f()** just reads its **X&&** argument, there is no problem related to templates – no difference between template and non-template implementations. It has nothing specific to do with concepts.

I don't think that the natural syntax using concepts makes this problem any worse.

If we want to solve the problems related to the overloading of **&&** in declarations, we could introduce a specific syntax for forwarding reference or – my preferred solution – consistently warn against later use of an lvalue that has been passed to a forwarding reference. The latter can be done by compiler and tool writers today.

See §2, §7, and §8.6 of [Str17] for a rationale for the use of the natural syntax and a discussion of potential problems.

7. Suggested language extensions

I am not trying to address all suggestions for improvements of concepts (e.g., elimination of redundant bool, the functional form of concept definitions, and the use of concepts as concept arguments). This section simply addresses a few suggestions for extensions to make the natural syntax to be make the natural syntax more expressive.

7.1 type-name introduction

Here are a few suggestions, assuming independent resolution, to accommodate consistent resolution:

```
void sort(!Random_access_iterator, Random_access_iterator);   // all RAIs must be the same

void sort(Random_access_iterator!, Random_access_iterator!);  // all RAI!s must be the same

void sort(Random_access_iterator.1, Random_access_iterator.1);

void sort(Random_access_iterator[1], Random_access_iterator[1]);
```

Here are a few suggestions, assuming consistent resolution, to accommodate independent resolution:

```
auto merge(Input_iterator{In1},  Input_iterator{In1},    // name [Bal14]
           Input_iterator{In2},  Input_iterator{In2},
           Output_iterator) -> Output_iterator;

auto merge(Input_iterator.1, Input_iterator.1,           // dot
           Input_iterator.2, Input_iterator.2,
           Output_iterator) -> Output_iterator;

auto merge(Input_iterator[1], Input_iterator[1],         // []
           Input_iterator[2], Input_iterator[2],
           Output_iterator) -> Output_iterator;
```

You might even insist on naming a type even if there is no potential ambiguity

```
auto merge(Input_iterator{In1}, Input iterator{In1},     // always name
                    Input_iterator{In2}, Input iterator{In2},
                    Output_iterator{Out}) -> Output_iterator{Out};
```

Or even

```
auto merge(Input_iterator{In1}, In1,      // always name
           Input_iterator{In2}, In2,
           Output_iterator{Out}) -> Out;
```

I'm not sure that these are problems worth solving. That is, the natural syntax need not be extended to handle such cases. The natural syntax was introduced to make simple cases simple and conventional. Concepts as type-name introducers, the explicit use of template type-name introducers, and/or requires-clauses are always there for the more complicated cases. Added syntax, like the suggestions above, could easily become a barrier to understanding (incl., teaching and learning).

7.2 Local concept names

For the (current and my preferred) consistent resolution, we can always simply give a concept a second name and use that (§3.6). This can currently be done only in namespace scope. Doing so in local scope would require an extension. For example:

```
Number compute(Number a, Number b)
{
        concept N2 = Number;  // same constraints as Number
        Number x = a+b;       // a+b must be the argument type
        N2 y = a+b;           // a+b can be a different Number type
        // …
}
```

As with the other syntactic extension suggestions, I would prefer to wait to see if there is a real need.

7.3 Natural syntax with **requires**-clauses

Consider

```
void foo(Concept, Concept, Sentinel<Concept>)
        requires OtherConcept<Concept>         // added requirements?
{
        difference_type_t<Concept> x = 0;      // In body?
        // …
}
```

The declaration line works as intended. The use of the explicit **requires**-clause with the concept would be an extension that would render the natural syntax more widely applicable. So would the use of the concept in the function body.

This potential extension would not interfere with the fundamental equivalence among the notations (§2.2).

8. Naming: Why "natural syntax"?

Why do I call **T f(U)** the "natural syntax"? After all, there is nothing "natural" about programming language constructs. Further, we should always suspicious about things labeled "natural", "obvious", "normal", or "intuitive" because these terms tend to distract from consideration of reasons. However, after about 400 years of usage of the **f(x)** notation in Mathematics (since Descartes) and about 70 years of use in programming (since Fortran), it feels natural to many. I have considered many alternatives:

- Terse – but not all examples are short
- Conventional – correct but boring, and seems to have negative connotations
- Functional – possibly, but this is the declaration, rather than the use
- Abbreviated – not an abbreviation of any other notation, and sometime longish
- Agile – no, not really
- Traditional – for functions, though unfortunately not for templates
- Common – common in which context?
- Simple – tempting, but nondescriptive
- Normal – normal to whom and why?
- Original – the first template design didn't use the **template** prefix

The main point is that this notation matches the way we traditionally define and use functions, not that it requires the fewest number of characters. I suspect that no name will be liked by all people and that any name that is "sort of OK" (e.g., "natural syntax") will after a while feel natural.

9. Conclusions

I started this examination of the natural syntax and the STL algorithms with the assumption that some improvement/simplification of concepts would be warranted based on arguments for consistency of style and/or low numbers of use-cases of features. Having listened to suggestions that independent resolution of binding to types would lead to simplifications comparer to consistent binding, I specifically looked at that possibility. However, I did not find those simplifications. In particular, both resolutions are consistent with the independent resolution of types bound to **auto**: with **auto**, there is no name to bind to for repeated use.

Further, independent resolution breaks the fundamental equivalence among the notations (§2.2). In addition, it raises many unexplored issues to do with return values and general usage.

We should stick to the consistent resolution of types to repeated uses of a concept in a declaration. Without that, the use of the natural syntax is reduced to cases without repeated use of a concept. For consistency, we should explore extending that requirement to the whole definition. If we decide on this extension (really a reversion to the original concepts design), we should consider adding a mechanism for defining local concepts.

My appreciation of the concept type-introducer increased. In fact, it is the only notation that could be exclusively used to achieve a "completely consistent good style". Compared to the concept type-name introducer notation, the conventional **template** type-introducer notation and explicit **requires**-clauses are verbose, ad hoc, repetitive, and (therefore) error-prone in many cases. In such cases, they should be avoided (and not just for aesthetic reasons). It is worth remembering that the natural syntax was never meant to be the only notation.

My appreciation of concepts as the foundation for generic programming increased. Focusing on unconstrained template arguments (using plain **auto** and **typename**) distracts from the need to design the proper concepts (with semantic meaning) that are the foundation of solid, widely useful generic code (See Alex Stepanov's writings). Concepts are not as different from types as some people assume, so disallowing the natural syntax would unnecessarily separate generic programming from "ordinary programming."

I found no genuine language problems for which the natural syntax was the root cause of a comprehension problem and only one (binding to a forwarding reference vs. binding to rvalue reference) where the natural syntax could possibly make current mistakes/confusion a bit more common.

The natural syntax is still the shortest and simplest in many cases; especially in the simplest and most common cases. In particular, it simplifies about 2/3rd of the STL algorithms. It is a valuable addition to the language as well as the ideal [Str17]. If someone doesn't like it for a perceived need for some form of syntactic consistency or fear of confusion, they can rely on concept type-name introducers and style guides rather than language restrictions.

10. References

- [Bal14] B. Ballo and Andrew Sutton: *Extensions to the Concept Introduction Syntax in Concepts Lite*. WG21-N3878. January 2014.
- [Nie17] Eric Niebler and Casey Carter: *Working Draft, C++ Extensions for Ranges*. N4651. March 2017.
- [Smi17] Richard Smith: *Working Draft, Standard for Programming language C++*. N4659. March 2017.
- [Str12] B. Stroustrup and A. Sutton (Editors): *A Concept Design for the STL*. WG21-3351. January 2012. "The Palo Alto TR."
- [Str17] B. Stroustrup: *Concepts: The future of generic programming*. WG21-P0557R0. January 2017.
- [Sut13] A. Sutton, B. Stroustrup, and G Dos Reis: *Concepts Lite*. WG21-N3701. June 2013.
- [Sut17] A. Sutton: Working Draft, C++ extensions for Concepts. WG21-N4641. February 2017.
- [Tou16] James Touton, Mike Spertus: Declaring non-type template arguments with auto. P0127R2.

Acknowledgements

Thanks to Inessa Bliznetsova, Casey Carter, J-D. Garcia, Eric Niebler, Gabriel Dos Reis, Richard Smith, Andrew Sutton, Herb Sutter, Ville Voutilainen, J.C. van Winkel, and Sergey Zubkov, for helpful comments on drafts of this paper. Casey Carter pointed to the possibility of extending the natural syntax with an explicit **requires**-clause.