Document Number: P0791R0
Date: 2017-10-10
Reply To: Jakob Riedle (jakob.riedle@tum.de)
Audience: EWG

Concepts are Adjectives, not Nouns

Introduction

This Paper aims to partially refine the big picture of Concepts in C++ – not to give concise articulation of surely necessary specification details. The author hopes to give a more intuitive understanding of Concepts and hence more powerful support of Constraint-based Programming in C++.

Starting point for the following syntactical deliberations is [1] ("N4687: Working Draft, Standard for Programming Language C++").

Status Quo

Given the following Code Snippet that facilitates Concepts as described by [1],

```
template<Red X>
void foo();
```

how do we know, `Red` is a concept? Or does `Red` refer to a plain type and X is a value? This gets even worse considering the idea of (previously proposed) Non-Type Concepts – constraints on values.
`Red` may then eventually be defined like this:

```
template<int N>
concept Red = requires{ /*...*/ };
```

… which would leave us with even more ambiguity in the previous example, namely that `X` might be a value of type `int`. This visual indifference does not sufficiently express the very difference between what needs to be passed to those templates. Consider the following declaration:

```
template<Dimension D>
class 2D_Field;
```

We can deduce that `2D_Field` is a class representing a two-dimensional Field with Values of unknown type, presumably `double` or `int`. Since Concepts as currently proposed allow D to resolve to both a type and a value, we don't know how to use it. Are we supposed to pass a type like `int[48][48]` or just the length of one side (`48`) and assume a square field? One might argue that we already have similar visual indifferences (forwarding reference vs. rvalue reference, variable definition vs. function declaration (Most Vexing Parse)) and more or less have happily grown accustomed to them. However, the abovementioned visual indifference this time impacts the core of the language.

The possibility to declare a Type name in syntactically identical way as a normal variable without any hint suggesting a type declaration *besides hopefully descriptive names (!)* represents an unprecedented inconsistency within C++. The ability to do so weakens an essential pillar upon which we read and understand data flow and data structures. This leads for example to the following situation: Given a Template, we cannot up front determine whether identifiers used within refer to type names or values, that are *declared by the template they are used in!*

Furthermore, if the name of a Concept used in the declaration of a template is accidentally shadowed by a concrete type, the compiler will try to at least syntactically understand the templates contents. Since the former Type Template Parameters using the respective concept are now of concrete type and thus values, the error messages generated within the template will be very cryptic. We have subsequently achieved

the exact opposite of what Concepts try to solve: We have silently weakened fundamental preconditions of a template: Namely that specific identifiers it uses shall resolve to types and specific identifiers shall resolve to values. Given that this can happen vice versa (A concept declaration shadows a concrete type used by the template declaration), we have even *weakened the preconditions of templates that do not facilitate Concepts*. Even though the kind of preconditions being weakened is different than what is aimed for by Concepts (preconditions on the templates implementation context vs. the templates use), the argument is still applicable.

Solution

The problem in the opinion of the Author lies in the fact, that Concepts are (ontologically speaking) wrongly perceived as a constrained type instead of a constraint applicable to a types domain.

Note: `typename` is the context of abstract ideas understood as an datatype whose domain is the set of all hypothetical datatypes.

To examine this more closely, let's consider the following example:

```
template<RandomAccessIterator T>
void sort( T begin , T end );

template<typename T>
   requires LessThanComparable<T> && CopyConstructible<T>
T min( T&& a , T&& b );
```

Why is it not possible in the second case to frankly state, we want "a less-than-comparable and copy-constructible Type T?" If you look at the phrasing of this question, you might notice that "less-than-comparable" and "copy-constructible" are *Adjectives*, not Nouns. The Noun however in this question is "Type". It becomes very clear now, why the Syntax currently proposed by N4687 [1] is so ambiguous: Named concepts are falsely perceived as Nouns instead of Adjectives. Let's see how this can give us a clue on how we can help Concepts to be more descriptive and less ambiguous.

Type Concepts

There is no such thing as a *CopyConstructible* or *Regular* but there surely are *CopyConstructible* and *Regular Types*. Leaving out the Noun "Type" in the statement `CopyConstructible T` translates to English as

> *Naming it `T`, let there be a copy-constructible!*

And everyone would strongly feel the need to ask, "A copy-constructible *what*?"
Just because we gave it a name doesn't mean we have said, *what it is*.

Since the keyword `typename` is the only indication that we declare a **type**, we should re-add this keyword in a fashion we would naturally do in spoken language – T is still a `typename` after all.
This would result in the following expression:

```
CopyConstructible typename T
```

Since we naturally wouldn't hesitate to prepend any number of adjectives to nouns in spoken language, we find no problem anymore in using any number of Constraints exactly where we would naturally expect them to be:

```
template<CopyConstructible LessThanComparable typename T>
T min( const T& a , const T& b );
```

This really does turn out surprisingly intuitive. The conjunction of Concepts suddenly feels natural, because we don't try to "*interleave two constrained types*" but rather apply two criteria to a domain of types.

But wait, one might reply: "How about the `Iterator` or `Container`-*Concepts*, they are nouns." And while that is true, we at the same time don't expect a *Container* to be passed to our templates (as is suggested

by current Syntax `template<Container C>)` but rather a *Container-Type*. So even in this case it makes perfect sense to insert the keyword `typename` after a Concepts name. How about Non-Type Concepts?

Note: Suggesting a syntax for the conjunction of concepts, at their use does not imply removing the proposed `requires`-clause.

Application to Non-Type Concepts a.k.a "Value Concepts"

Note: It is assumed here, that there is need for named Non-Type Constraints, which is the reason this area is being explored here.

Note: In the following deliberations, we speak about the constraining of variables. Types are nonetheless thought to be constrainable outside the context of a variable declaration. However, for the sake of explainability, this paper will discuss these constraints w.l.o.g. at the declaration of a variable.

As Non-Type Concepts have been dropped in [1], we will approach them from the perspective of Concepts TS [4]. Given may be the following Value Concept definition:

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

We will not discuss the above syntax at this point but rather assume it as temporary starting point for the following considerations.

Along the design rules explored in the last paragraphs, one might try to write a variable of *constrained type* (e.g. copy-constructible type) as

```
CopyConstructible auto foo
```

. With this idea in mind, let's get back to the example of `Even`.

We can ask ourselves, does the datatype `int` have anything to do with the property of being even? Yes and No. While a *Number* can be even in a mathematical sense, an *Iterator* cannot and in fact an Iterator is unlikely to have a modulo operator that we can facilitate in the above fashion. We ultimately need to constrain our Value Concept itself with a Type Concept. In the case of `Even`, we will call this Type Concept *Numerical*. Adapting the above syntax for the declaration of `N`, we can now write

```
template<Numerical auto N>
concept bool Even = N%2 == 0;
```

When using this Value Concept along the Idea of treating Concepts as adjectives, there are four different aspects for which we need to clarify their semantics:

1. We constrain the domain of a concrete datatype, e.g. `int`: This will most intuitively be written as

   ```
   Even int foo
   ```

2. We want to constrain the values of a variable regardless of its type. Since a declaration of a variable of unconstrained type is done using `auto`, this would probably be done as

   ```
   Even auto foo
   ```

   However, `Even` is at this point only defined for a *Numerical* value. As one would automatically expect, this already implicitly constrains the datatype.
3. We want to constrain the values of a constrained set of datatypes. If we just continue applying the pattern from the previous examples, this would look like the following (we will exemplarily use the Type Concept `CopyConstructible`):

   ```
   Even CopyConstructible auto foo
   ```

   The type of `foo` now has to fulfill the Type Concepts `Numerical` (implicitly because of `Even`) and `CopyConstructible`, its value must fulfill the Value Concept `Even`.
4. We want to constrain the type of a variable of already concrete type:

```
CopyConstructible int bar
```

Since we already know, `int` fulfills the Type Concept `CopyConstructible`, the Concept is essentially useless. This situation however is likely to happen, especially in templated code and should therefore be allowed – of course, only as long as the concrete type **does fulfill** the prepended Concept.

Note: In the above item "3", both Type Concepts and Value Concepts are being prepended to the keyword `auto`. This ambiguates, which Adjectives constrain the Values and which constrain the Type of the variable. Whether this is an actual Problem at all and if yes, which syntax adjustments should be made to resolve this visual indifference is to be discussed by subsequent papers.

Mixing Type Qualifiers and Concepts

Let's consider the following example:

```
CopyConstructible const auto bar
```

Should `CopyConstructible` be applied to the whole (even though `const`-qualified) datatype of `bar` (as is presumed in the previous paragraphs) or to the type that '`auto`' would resolve to (the non-`const`-qualified type)? Well, if we wanted to apply a Concept to only '`auto`' and would follow the rules of natural speech, we'd rather write something like the following:

```
const CopyConstructible auto bar
```

This starts to make clear, that the idea of Concepts as predicates of a variable's datatype/values is unnecessarily constrained. The author feels at this point that Concepts should rather follow the precedence rules already present at declaring datatypes (as the last scenario suggests).

In order to get a bit into detail, let's consider the case of reference qualification:

```
CopyConstructible auto&& foo
```

This would presumably translate to an rvalue reference to a value of arbitrary but `CopyConstructible` type.

Constraining Pointers

Now something a bit more complex:

```
CopyConstructible auto* Even bar
```

Using the precedence rules we already internalized in our minds, the above should translate to an `Even` (2-aligned) pointer, that points to a `CopyConstructible` variable. Similarly,

```
CopyConstructible auto* const Even& bar
```

Should translate to a `const` lvalue reference to an `Even` (2-aligned) pointer, which points to an object of arbitrary but `CopyConstructible` Type.

Note: Whether a pointer is applicable to the concept `Even` solely depends on the definition of `Even` and eventually, whether a pointer fulfills the hypothetical Concept `Numerical`.

The reader should at this point keep in mind, that this paper doesn't aim at making precedence rules explicit but rather to suggest making Concept more versatile by perceiving them as adjectives (similar to e.g. `const`) while following established 'intuition'.

Outcome

In all aforementioned scenarios, the outcome has intuitive semantic implications, follows the design principles already present in C++ and is therefore easily understandable even by C++ beginners.

The plausibility of this approach stems from the fact, that we *are* adjectivizing types and values in a syntactically identical fashion for a very long time now. The most clear-cut examples include the following:

- The keyword `unsigned` is in *some sense* restricting a datatype to non-negative values. The fact, that `unsigned` and `signed` can be used without `int`, `short`, `char` shows how even here Adjectives and Nouns are ambiguated.
- The keyword `double` designates a double-precision floating point type. Even here, the name does not suggest *what is being doubled* – once more the noun is being left out.
- The keywords `short` and `long` designate a smaller resp. longer version of an `int`. Like `unsigned`, `short` and `long` can be used without the noun `int`.

What about Terse Syntax?

We are unlikely to apply the syntax form proposed by N4687 [1] to function or lambda parameters (as outlined by P0587R0 [2][1]). Hence, the syntax for applying a concept is most likely to differ in the context of function and lambda parameters, in order to sufficiently provide for both brevity *and* descriptiveness, which should be present in the first place.

The here proposed syntax however can be carried without modification not only into the declaration of lambda a well as (non-lambda) function parameters. This is, because the proposed approach resolves the visual indifference between a non-templated function and an implicitly templated function. The term "Terse Syntax" would become obsolete in this case: We would not be forced to make another visual differentiation between the use of concepts in template vs. function/lambda parameter declarations.

Example

Assuming the identifier `NonEmpty` is unknown to the programmer and may stand for either a concrete Type or a Type Concept,

a) a **non-templated** function would be declared as
```
void foo( NonEmpty obj );
```
resp.
```
auto foo = []( NonEmpty obj ){ /*...*/ }
```
b) while an **implicitly templated** function/lambda (because Concept-constrained) would be declared
```
void foo( NonEmpty auto obj );
```
resp.
```
auto foo = []( NonEmpty auto obj ){ /*...*/ }
```

As should be clear by now, the proposed approach suggests a very intuitive view on concepts and is able to pave the way for a consistent, descriptive and intuitive usage of them across all their application contexts, which is not possible with the wording of Concepts in N4687 [1].

Q&A or "Variants"

Is this gain in clarity and descriptiveness worth the additional writing work?

We have examined in the first chapter, that the current Concepts proposal breaks with fundamental habits and additionally implies a visual indifference between constrained Type and Non-Type Parameters. Furthermore, the proposed syntax is hardly usable for terse syntax, as one cannot differentiate between an implicitly-templated function and a regular one.

In case we constrain identifiers by exactly one Concept, the proposed syntax form requires 5 resp. 9 additional letters for resolving all of the above issues. When applying more than one concepts to an identifier, the proposed syntax can spare the trailing `requires`-clause and is thus *shorter* (in case of a logical conjunction of them, which is the expected use case as discussed in the next paragraph).

Apart from that, the keyword `typename` after a Concept's name could be made optional inside a `template<>` statement – of course with the profound consequences outlined before.

[1] This is mostly because the use of a Concept in a functions parameter list silently makes this function an implicit template which has profound impact on how to reference it.

One may logically conjunct multiple Concepts by chaining them. Do you also propose Concept disjunctions?

First of all, this paper does not propose to remove the trailing requires clause after a templates parameter list or a functions parameter list. Secondly, the disjunction of concepts can be expected to be used far less often than the conjunction of them. This is because Concepts represent preconditions in an abstract sense, which are to be met before a certain piece of software can be used. This abstract list of preconditions is naturally built of conjunctions for the most part.
Other than that, there are three alternatives to an additional `requires`-clause with an explicit disjunction:

1. We could allow Concepts to be passed as template parameters to other templates, which enables us to come up with a disjunction of Concepts:

   ```
   template<typename T, template<typename> concept... Cs>
   concept Or = ( Cs<T> && ... );

   // Usage
   template<Or<Even,Odd> int N>
   void foo();
   ```

   This implementation option is shorter and more descriptive in its purpose than a corresponding `requires`-clause. Possible meta concepts would include e.g. `Not<>`, `Neither<>`, `XOr<>` or `IfThenElse<>`.
2. We could provide for a Concept-capable disjunction operator such as "`|`":

   ```
   template<Even|Odd int N>
   void foo();
   ```

   The precedence rules and semantics of this operator would still need to be made explicit though.
3. Do nothing. Disjunctions of concepts in some cases prevent us from preserving a total ordering concepts. Therefore we might discourage writing them by not providing an alternative.

Are there any other benefits of this approach?

Concepts as (partial) manifestation of Contract-based Programming in C++ currently limit themselves to the compile time. Anticipating that there will be people raising their voice for the wish to write Value-Concepts and furthermore apply already-written Value-Concepts to function parameters:

The concept of Adjectives represents a powerful and intuitive tool at fully implementing Contract-based Programming in C++ consistently across compile-time and runtime. It can easily be extended to come up for e.g. runtime constraints (in the form of concept functions), which can avoid the multiple evaluation of a constraint by including the constraints into the type system (just like other adjectives such as `volatile`, `const` or `constexpr` do). Proposing this however is not goal of this paper.

Note: [3] (P0380R1: A Contract Design) was kept in mind while writing the above comment. The author however considers the Adjective Syntax (as conceptual idea) to be vastly superior to the proposed attribute syntax.

Formalization

The syntactical resp. semantical revisions of Concepts can now be summarized as follows:

1. Concepts can emerge in **three kinds**:
   a. *Type-constraining*. A type constraining Concept represents a property enforceable on the domain of variables of type "`typename`".
   b. *Value-constraining*. A value constraining Concept represents a property enforceable on the domain of an arbitrary (but excluding "`typename`") Datatype.
   c. *Type- and Value-constraining*. A Value- and Type-constraining Concept represents a property that is only defined (decidable) for Values of a certain class of Datatypes (which is itself constrained using a Type Concept). Therefore, a *Type- and Value-constraining Concept* – once applied to a Value Variable – constrains its set of values as well as its datatype.

Note: The domain of a variable is for practical purposes understood as the set of values that may be assigned to this variable.
Note: Pure value-constraining Concepts are expected to be rare, since only few and only very general properties can be associated with values of arbitrary datatype.

2. **Enforcing a Concept** on the domain of a variable (type or value variable) happens by inserting the Concepts name into the variable's datatype. More specifically:
    a. For Type Variables, this means to prepend its name to the `typename` keyword.
    b. Once inserted into a variable's datatype, it should constrain the contextually closest part of the datatype (whatever that shall be, but with present precedence rules in mind).
    c. Type-constraining Concepts may be applied to Type Variables, as well as Value Variables.
        – In case of Type Variables, the applied Concept constraints the set of types that may be assigned to this variable.
        – In case of `auto`-typed Value Variables, the applied Concept constraints the datatype of values that may be assigned to them. However, the datatype of the initializing value still determines its datatype (for its whole lifespan).
        – In case of non-`auto`-typed Value Variables: Iff the Concept is fulfilled, the Variable declaration behaves as if written without the Concept applied. Otherwise, the declaration is ill-formed.
    d. Value-constraining Concepts may be applied to Value Variables.
        – Once applied to a Variable, the Concept constraints the set of values that may be assigned to it. The precedence rules shall be identical to the application of Type-Constraining Concepts.
    e. Value- and Type-constraining Concepts may be applied to Value Variables.
        – Once applied to a Variable, the Concept constraints the set of possible declared datatypes of this variable as well as the set of values that may be assigned to it. The precedence rules shall be identical to the application of Type-Constraining Concepts.
        – Again, in case of `auto`-typed value variables, the datatype of the initializing value still determines its final datatype – given it fulfills the Concept.

Acknowledgements

Thanks to Arthur O'Dwyer, who gave me very valuable feedback on the topic as well as this paper and who offered to shepherd this paper to Albuquerque.
Thanks to Richard Smith, who coined the phrase "Concepts are Adjectives, not Nouns".

References

[1] N4687: Working Draft, Standard for Programming Language C++

[2] P0587R0: Richard Smith, James Dennett: Concepts TS revisited

[3] P0380R1: G. Dos Reis, a.o.: A Contract Design

[4] N4549: Programming Languages — C++ Extensions for Concepts