Doc. no. P0695r0
Date: 2017-02-19
Programming Language C++
Audience: Evolution
Reply to: Bjarne Stroustrup (bs@ms.com)

# Alternative concepts

Bjarne Stroustrup

www.stroustrup.com

This note documents some of my observations and opinions on *Concepts TS revisited* (P0587R0) by Richard Smith and James Dennett. To distinguish, I will refer to the current concept design as documented in the Concepts TS (*Working Draft, C++ extensions for Concepts*, N4641) and papers such as Concepts: *The Future of Generic Programming* (P0557R0) as "Concepts" or "Current concepts" and the novel design sketched out by "Concepts TS revisited" (P0587R0) as "Alt-concepts" (alternative concepts).

In the form of comments to the Concepts TS, alt-concepts sketches out a design that

- Is syntactically different
- Is semantically different
- Has a different programming model

Obviously, it breaks all existing code using concepts and invalidates all that has been written about concepts over the last few years. Furthermore, I consider most suggested changes not to be improvements.

## Suggested changes

Let me first list the suggested changes/improvements to the current concept design as summarized throughout the alt-concepts document:

- => Add first-class concept definition syntax, remove `concept bool`
- => (Optional) do not allow concept overloading
- => Remove full decomposition and ordering based on textual duplication; only order concepts
- => Concepts explicitly specify which concepts they refine and which they are refinements of
- => Requires clause specifies list of required concepts, not arbitrary boolean expression.
- => Requirements of a template cannot contain arbitrary expressions, just named concepts.
- => Replace syntax with `'requires` parameter-declaration-clause compound-statement'

    - o - can use any syntax permitted in a function or lambda body
    - o - result is `true` if substitution into body succeeds, `false` if not
    - o - extend and reuse existing language syntax, do not invent a new mini language
- => For syntactic convenience, add nested-requirement form as regular declaration
    - o - `requires x;` is equivalent to `static_assert(x);` but more natural
- => Replace template-introduction syntax with a syntax based on decomposition declarations that permits additional template parameters to be declared
- => Require an explicit sigil to declare a function to be a template
- => Remove or revise terse template notation

I will not attempt a point-by-point refutations of the suggestions made. Hopefully, the list will make it obvious that we are not talking about a minor tweak, but about a very different proposal. Should we decide to go this way, alt-concepts will not be stable and useful in production code for several years. We'd need design, documentation, formal wording, implementation, usage experience, teaching experience, rationale, popularization, and an outline of an alt-concept standard library. All of this, and more, has been done for the current concepts design. So far, none has been done for alt-concepts. We should not lower the bar for alt-concepts

- out of hope that it might be better (non-specific proposals always seem better because we imagine the final detailed design will be what we dream about),
- feelings that it is "more natural" ("natural" is a very subjective notion), or
- out of hope that it will turn out to be easier and/or simpler to implement, use, teach, etc. (we always underestimate the time and effort needed).

Whatever criteria (fair or foul) were applied to concepts should be applied to alt-concepts. Currently, it is not really a design, but just a collection of suggestions.

A few of the suggestions in the alt-concepts documents could possibly be used to improve concepts. Most have been considered in the past, but I will re-consider some here even though they would break existing code: concepts have been available for experimentation for 4 years (GCC branch) and widely available for almost a year (GCC 6). Concepts is a foundational feature and we should be willing to break a little code now (in ways that are easily repaired) if the benefits to the C++ community are major, but only if the benefits are major – every change is likely to break some existing code.

Before reading further, I strongly suggest that you – if you haven't already – read my paper *The Future of Generic Programming* (P0557R0). That paper outlines some design rationale for concepts, give some examples of their design and use, and documents some historical background. This paper has been widely read in the community and was – as far as I can tell from Reddit, Slashdot, comments to me, and reactions to talks and discussions on the same theme – very well received.

Fundamentally, my opinion is that the C++ community badly needs concepts ***now***. They were considered central in the development of generic programming (Stepanov) but we couldn't provide direct support. Some of us have worked on this for 20 years and are in no mood to wait another 5 while the committee consider "novel" approaches and designs. With the growth of generic programming and metaprogramming concepts are needed to improve our designs, interfaces, and implementations. The alternative is increasing complexity and chaos. Even experienced committee members need concepts to

express their designs concisely, precisely, and correctly (for example see Ville Voutilainen's recent email about optional<T> assignment and Beeman Dawes paper on iterator facades [P0186R0]).

People are finding concepts useful in experimentation, design, and production code ***now***. See also [P0225R0], text_view: A C++ concepts and range based character encoding and code point enumeration library and Seastar C++ framework looks to simplify coding .

## Syntax issues

The first two suggestions

- => Add first-class concept definition syntax, remove `concept bool`
- => (Optional) do not allow concept overloading

are quite reasonable, not radical, and have been considered repeatedly. Allowing people to say

```
template<typename T>
concept Equality_comparable =
        requires (T a, T b) {
                { a == b } -> bool;        // compare Ts with ==
                { a != b } -> bool;        // compare Ts with !=
        };
```

Rather than

```
template<typename T>
concept bool Equality_comparable =
        requires (T a, T b) {
                { a == b } -> bool;        // compare Ts with ==
                { a != b } -> bool;        // compare Ts with !=
        };
```

Is a truly minor change, but having to say "bool" seems to bother some people to an amazing degree. We chose the current, slightly redundant syntax to avoid messing with the already baroque C++ grammar; see also *The Future of Generic Programming*. If only the former syntax was allowed, existing code would break, but loudly and easily repaired. The current syntax could be supported by current compilers for a transition period.

Allowing concept overloading is a different matter. If we don't need it, we can (of course and as we have known for years) simplify the concept definition syntax (by eliminating concept functions) and more importantly the concept use syntax (by never requiring **()** for concept function call).

Looking for concrete examples of concept overloading, discussing them, and eliminating concept functions if no significant uses are found is a viable plan, and I'd be very happy if we found we could simplify by leaving them out. This should not delay the introduction of concepts into the standard. I encourage people to look for concrete examples of concept overloading.

## Hierarchies of concepts

Alt-concepts proposes to eliminate the deduction of template relations and replace it with an "object-oriented" declarative mechanism.

- => Remove full decomposition and ordering based on textual duplication; only order concepts
- => Concepts explicitly specify which concepts they refine and which they are refinements of

This, I think is a major mistake. It makes code more verbose and less flexible. In particular, it introduces OO-style need for designer foresight and complicates joint use of multiple code bases. It is also a dramatic change to a fundamental aspect of concepts and a reversion to a troublesome part of C++0x concepts. The absence of explicit hierarchies is one of the strengths of generic programming compared to traditional object-oriented programming and we should not lose it by requiring explicit hierarchies in template interface specifications.

If you want to name concepts and use composition to build concepts, you get the notational benefits from traditional derived class notation without the problems. Consider:

```
template<typename T>
concept bool Sortable =
        Sequence<T> &&
        Random_access_iterator<Iterator_of<T>> &&
        Less_than_comparable<Value_type<T>>;
```

In a full-scale library, **Sequence<T>**, **Random_access_iterator<Iterator_of<T>>**, and **Less_than_comparable<Value_type<T>>** will be used as parts of many concepts. Explicitly building a sequence hierarchy, an iterator hierarchy, and a comparison hierarchy of concepts and then fitting those together into a lattice would be inflexible and rigid.

Incidentally, I consider occurrences of **requires requires** and especially frequent use of **requires requires** a sign of sloppy design or inexperience. Concepts should be designed to represent domain concepts in code, rather than to create a mess of ad hoc constraints. Most constraints on template arguments should be named concepts.

The need for a new syntax "provided to retroactively annotate that an existing concept is a refinement of a newly-introduced concept" with associated semantics is a band aid needed only because atl-concept eliminates general Boolean composition of concepts. I find

```
template<typename T> concept bool Primitive = Integral<T> || Floating<T>;
```

a far clearer statement of the intent than

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

template<typename T>
```

```
extern concept Integral<T>
        : Primitive<T>;

template<typename T>
extern concept FloatingI<T>
        : Primitive<T>;
```

Concepts simply are not inherently hierarchical.

Another realistic example of disjunction is

```
template<typename T>
        requires Same<T, void> || TotallyOrdered<T>
struct less;
```

I find it hard to imagine a simple OO-style solution.

## Requires expressions

The opening sentences of the alt-concepts paper on requires expressions expresses concerns about the possibility of later addition of definition checking to the concepts design. Not everyone likes definition checking and based on experiments we know how to do it for the current design (as far as you can be certain without having done it in complete detail); see *The Future of Generic Programming* (P0557R0). Doing definition checking for a more general set of expressions as alt-concepts suggests should be no easier than for a subset. Alt-concepts claims "trivially" without supporting arguments. In particular, I wonder how **constexpr if** would be handled without a C++0x-style **late_check** mechanism. This is just one example of the technical problems we have to address when considering an idea, as opposed to a fleshed-out proposal (and I consider the usage, programming techniques, problems even more difficult to address in advance).

The suggested improvement

- => Replace syntax with '`requires` parameter-declaration-clause compound-statement'
  - - can use any syntax permitted in a function or lambda body
  - - result is `true` if substitution into body succeeds, `false` if not
  - - extend and reuse existing language syntax, do not invent a new mini language
- => For syntactic convenience, add nested-requirement form as regular declaration
  - - `requires x;` is equivalent to `static_assert(x);` but more natural

This seems to me to be nothing but a wild generalization with little concern about use. One thing that the designers of concepts are quite certain about is that the constraints on an algorithm is not the simple needs of its current algorithm. That leads to interface instability. We do not want simply require that the implementation code works by restating it as a **requires** expression (to reduce duplication, some people would use macros, thus eliminating much of the benefits from concepts). The aim is for requires clauses to specify a comprehensible set of constraints, not to by default to exercise every

obscure language use. What we have in concepts has proven sufficient to handle the STL (see Ranges). If in the future, we find problematic gaps in our coverage, we can extend the rules to handle genuinely useful cases.

In general, I find **requires** expressions seriously overused in the alt-concepts discussion. It's a bit like looking at unstructured code written by someone who doesn't appreciate functions and classes as structuring mechanisms. The requires expressions started out as an implementation mechanism for concepts and is best understood as such, rather as a primary mechanism for the user to directly express constraints in a template declaration.

## Terse notations

I consider the terse use of concepts in function declarations fundamental. For example:

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

It is needed to minimize the distinction between generic and "ordinary" programming. It is not some random notational quirk. See *The Future of Generic Programming*. This notation is wildly popular among some users. We should not mess with it.

The notion of template introducers was discovered in real use. Repeated use of complicated patterns of template argument introduction and requires clauses were common enough to be a nuisance and a bug source. The standard-library family of merge functions is the canonical example.

On the other hand, the template introducer syntax is not fundamental. For example:

```
concept{A,B,C} void f(A,B,C) {}
```

Could be expressed in some other way. Alt-concepts suggests the slightly more verbose

```
template<concept [A,B,C]> void f(A,B,C) {}
```

"or similar". There seems to be a dislike of brackets **{}** in some parts of the C++ standards community and a curious desire to see the keyword **template** in as many places as possible. Many in the user community disagrees and find the current syntax-heavy syntax painful. I am not averse to improving the introducer syntax, but I don't see the suggested syntax as an improvement. For starters, it's more verbose. I consider this as an example of the well-known phenomenon of people being nervous about novel features and demanding LOUD syntax. Invariably, as a feature becomes common, people complain bitterly about verbosity.

If the syntax was changed, the resulting breakage would be manageable specifically because template argument introducers limit repetitive use.

## A closing point

People have waited for and talked about concepts for years, a dramatically different proposal, such as alt-concepts, would undermine the credibility of the committee and of many of its supporters who have presented concepts in public over the last few years.

## Summary

No major changes to the current concepts design is needed, but minor tweaks may be feasible. The alt-concepts design is a radical departure from the current design, is at best quarter-baked, and in places reverts to previously rejected approaches. It does not address some of the fundamental needs of generic programming and leads to more syntax laden and less flexible code. It would take years to bring it up to the quality of the current TS and concepts is badly needed now; in fact, they are in increasing current use. If we want better concepts, we should build on the current concepts, rather than – yet again – try to go back and start over.