Doc. no.: P0956r0
Date: 2018-02-11
Programming Language C++
Audience: EWG
Reply to: Bjarne Stroustrup (bs@ms.com)

Answers to concept syntax suggestions

Bjarne Stroustrup

Abstract

There have been a few suggested improvements to the syntax of concepts. My questions are:

- What problem are you trying to solve?
- How do you know that it's a problem?
- For whom is it supposed to be a problem?

My conjecture, based on years of experience with use and teaching, is that there are no serious problems with the current "natural/tense/conventional/whatever" syntax and that users like it. It is the simplest and is taken as "natural."

When suggesting an improvement, we should avoid adding verbosity, duplicate expressiveness already supplied in the TS, and avoid untried schemes.

There are so many papers, ideas, and opinions that to be brief, I am addressing only comments that closely relates to the current design.

Summary

Much of what can be said in favor the natural syntax has been said in Concepts: The Future of Generic Programming (P00557r1) and Function declarations using concepts (P0494r0) together with examples and discussions of implications and potential problems. If you haven't read them (recently), you might want to. Points in favor of the natural syntax

- Programmers (experienced and students) like it.
- Programmers hate verbosity.
- It does not repeat the mistake of C by requiring a logically redundant keyword the indicate a semantic class (i.e., it is not like having to say **struct S** for uses of **S**).
- It applies equally to functions and lambdas.
- It makes **auto** a natural part of the system.

I have been explaining and teaching concepts for years – often emphasizing the natural syntax. I have never had problems with students or industrial programming having problems with understanding or use. This experience covers many hundreds of people of widely varying backgrounds. Others have had similar experience.

I have heard conjectures about problems and I have seen many clever examples that would puzzle everybody, but so I have for just about every language feature and syntactic construct in C++. We are very competent language technicians and implementers, let's give the users what they want and like. We can handle the technical details.

Whatever you do, don't repeat C's mistake of requiring **struct** to mention a type by requiring something like **concept** to mention a concept. We also have a few cases where we have to say **typename** to mention a type and **template** to mention a template. Users hate those (and are not open to logical reasons for the need). Most programmers are not, should not be, and don't want to be language lawyers.

What are the problems with the current notation?

I have discussed most concerns raised by others in my recent papers, so here I will discuss only what I consider remaining problems.

Consistent use of concept names

As I (e.g., P0694r0) and others have observed, the demand for consistent use of concept names in a function declaration is not consistent with the rule that concept names used for local variables. This should be fixed. We can either go with consistent binding (a concept name is bound to a single type in a scope) or independent binding (each mention of a concept name can be bound to any type that matches it). I don't think the current (concepts TS) mixture of the two is viable.

Independent binding

If we choose independent binding (see P0694r0), the fundamental equivalences between notations need to change for

C f(C,C); means **template<C T> T f(T,T);**

To

C f(C,C); means **template<C T1, C T2, C T3> T1 f(T1,T2);**

In particular, this will imply changes to the meaning of returning a value of a type of the same concepts as an argument (as in the example above) from converting to a specific type to returning a value of a potentially different type.

We can no longer use a concept name to stand in for its type, so we need another way (using **decltype** is possible, but ugly).

We need a way of expressing "same type" for two uses of a concept.

Consistent binding

If we choose consistent binding (see P0694r0), we need to change the meaning of use of concepts for local variables to be consistent.

We need a way of saying "different type" for two uses of a concept.

Constraining the type of variable

I find this increasingly useful. In particular, programmers find that the use of **auto** for values returned from function calls can make code hard to read and resorts to specific types (damaging flexibility) or comments (unreliable and unchecked). We badly need

```
C x = f(y);
```

This is not a frill. It is essential for large-scale usability of concepts. It is user experience feedback.

For the same reason, programmers ask for the natural syntax for lambda argument

```
[](C x) { … }
```

This was after all one of the motivations for introducing the natural syntax.

The natural syntax is not the only notation

Sometimes the arguments seem to assume that the natural syntax is supposed to express every aspect of concepts. That was never the intent. I am perfectly happy to leave complex solutions to the more verbose notations.

The major advantage of the natural notation is exactly that it is simple and similar to the familiar use of types. Don't try to improve it by making it verbose to handle complicated examples already handled well by the other notations.

Note the suggestions for improving the combined use of the natural syntax and other notations in P0694r0. In particular, I think we need the ability to add constraints:

```
void foo(Concept, Concept, Sentinel<Concept>)
        requires OtherConcept<Concept> // added requirement
{
        // …
}
```

In this example from P0694r0, I assume consistent binding to be able to use the concept name to refer to its type. However, the idea applies equally to independent binding.

The concept type-name introducer mechanism is essential

In P0694r0**,** I demonstrated that the concept type-name introducer syntax is the only fully general and manageable notation. If someone insists on using a single notation (as an organization or style guide may), the concept introducer mechanism is the only one that fits the bill. For example:

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

The exact notation is less important, but the current (concepts TS) notation works and I don't know of a better one.

Note that this works even better with ability to add constraints in a **requires**-clause (seen next section).

Comments on P0745R0

Herb Sutter has written an analysis and proposal P0745R0, which is partially based on a paper by Botond Ballo and Andrew Sutton (N3878). This design in turn partially goes back to the earliest discussions of designs of syntax for concepts by Andrew with Gaby and me on my whiteboard at TAMU – there is little fundamentally new in these discussions, just many important design details.

The basic idea is to consistently use a concept name as a typename introducer. For example:

```
OutputIterator{Out} copy(InputIterator{In},In,Out);
```

Or more completely specified

```
OutputIterator{Out} copy(InputIterator{In},In,Out)
        requires Assignable<ValueType{Out},ValueType{In};
```

This example, like many other complex examples, is better expressed using the concepts TS typename introducers

```
Copyable{In,Out} copy(In,In,Out);
```

We can (I assume) specify return types that are not also argument types like this:

```
auto f(C{In}) -> C{Out};

C{Out} g(C{In});
```

As far as I can see, this is consistent with concepts TS typename introducers.

There is much to be said for this way of introducing names and when N3878 was first presented, EWG didn't reject it. Rather, we postponed the discussion of this added feature until after the approval of the basic concept proposal and the assumed greater experience with that (that was 4 years ago).

In P0745R0, Herb proposes that if we don't need to refer directly to a constrained type, we don't have the mention it. For example:

```
Iterator{} f(Range{});

Integral{} x = f(y);
```

That's reasonably logical and offers some verbosity to people who like syntax. I strongly prefer the (also mentioned) obvious simplification:

```
Iterator f(Range);

Integral x = f(y);
```

This is what we are using today and has used in code and presentations for many years. It works. I don't want to have to explain why "nothing" is spelled **{}** in contexts where that's obvious.

References

- Botond Ballo and Andrew Sutton: Extensions to the Concept Introduction Syntax in Concepts Lite. N3878. 2014-01-13.
- Herb Sutter: Concepts in-place syntax: P0745R0. 2018-02-06.
- B. Stroustrup: Concepts: The Future of Generic Programming. P00557r1. 2017-01-31.
- B. Stroustrup: Function declarations using concepts. P0694r0. 2017-06-18.