Document number: N3878
Date: 2014-01-13
Study group: Concepts
Reply to: Botond Ballo <botond.ballo@utoronto.ca>
Andrew Sutton <asutton@cs.tamu.edu>

Extensions to the Concept Introduction Syntax in Concepts Lite

Botond Ballo, Andrew Sutton

1 Introduction

Concepts Lite [1] introduced the notion of "concept introduction", a shorthand for declaring a template whose template parameters are constrained by a multi-argument constraint:

```
Mergeable{For1, For2, Out}
void merge(For1 p, For1 q, For2 p2, For2 q2, Out o);
```

Here `Mergeable` is a three-argument constraint, and the above code is a shorthand for:

```
template<typename For1, typename For2, typename Out>
  requires Mergeable<For1, For2, Out>()
void merge(For1 p1, For1 q1, For2 p2, For2 q2, Out o);
```

In this paper, we propose a few small extensions to concept introductions. We believe these extensions will make the language more expressive and fill some holes in the Concept Lite proposal.

2 Proposal

2.1 Concept introduction in a template declaration

2.1.1 Overview

In the above example, the concept introduction took the place of a template parameter list in a template declaration. We propose that, additionally, a concept introduction be allowed as an element of a template parameter list:

```
template<Mergeable{For1, For2, Out}>
void merge(For1 p, For1  q, For2 p2, For2 q2, Out o);
```

This would be equivalent to the examples in 1.

2.1.2 Motivation

There are three motivations for this extension.

First, it bridges the syntax between regular template parameters and concept introductions. In the degenerate case of a constraint with a single argument:

```
template <UnaryConcept{C}>
void function(C);
```

the concept introduction syntax looks almost exactly the same as a regular constrained parameter declaration (the only difference is the braces):

```
template <UnaryConcept C>
void function(C);
```

One can even think of our proposed syntax as an extension to regular template parameter declarations, with the braces added where necessary to group multiple arguments to single constraint.

Second, we gain greater expressivity by mixing concept introductions and regular template parameters in a template parameter list, something that could not be done before without using the long-hand **requires** form:

```
template <UnaryConcept A, BinaryConcept{B, C}>
void function(A, B, C);
```

Third, we can now have multiple concept introductions per template declaration, also something that could not be done before without the **requires** form:

```
template <BinaryConcept{A, B}, BinaryConcept{C, D}>
void function(A, B, C, D);
```

2.1.3 Details

Our proposed semantics for concept introduction is that each occurrence of a name that appears in an introduction constitutes a declaration of that name as a template parameter. This implies that if an introduction uses a name that already appears in a previous element of the template parameter list, the program is ill-formed. For example,

```
template <typename A, BinaryConcept{A, B}>
void function(A, B);
```

is ill-formed because it is a shorthand for

```
template <typename A, typename A, typename B>
  requires BinaryConcept<A, B>()
void function(A, B);
```

which illegally tries to redeclare the template parameter `A`.

For a rationale of this choice, see the Discussion 3 section.

2.2 Concept introduction in a parameter declaration

Concepts Lite introduces the following "terse notation" for declaring a constrained function template:

```
void sort(Container& c);
```

Recall that this is a shorthand for:

```
template <Container __Container>
void sort(__Container& c);
```

In case where the same concept is used in two different parameters:

```
void sort(RandomAccessIterator p, RandomAccessIterator q);
```

the rule is that the two parameters are of the same type. That is, the above is a shorthand for

```
template <RandomAccessIterator __Ran>
void sort(__Ran p, __Ran q);
```

.

Sometimes, however, we want two arguments that each satisfy the same concept without them being the same type. For example:

```
template <Range R1, Range R2>
auto join(R1 r1, R2 r2) { ... }
```

Here, we want both arguments to be ranges, but not of the same type. There is currently no way of expressing this in "terse notation".

We propose that concept introduction be allowed in a function parameter declaration, as in the following example:

```
auto join(Range{R1} r1, Range{R2} r2) {...}
```

Here, for any given instantiation, `R1` and `R2` are the concrete types of `r1` and `r2`, respectively. Since `R1` and `R2` are different names, `r1` and `r2` are allowed to be of different types.

Introductions in parameter declarations also give rise to another way of expressing the `sort` example above:

```
void sort(RandomAccessIterator{R} p, R q);
```

Note that since the introduction in the first function parameter declares the name `R` (which is the name of a template parameter being implicitly declared), we can use the type `R` as the type of the second parameter directly (there is no need for another introduction), and this forces the types of the two parameters to be the same, as we desire.

This form has two advantages over

```
void sort(RandomAccessIterator p, RandomAccessIterator r);
```

First, it makes it more clear that the two parameters must be of the same type. Second, it gives the function implementer a type name to work with (`R`) that is more convenient than `decltype(p)` or `decltype(r)`.

Note that the following would be ill-formed:

```
void sort(RandomAccessIterator{R} p, RandomAccessIterator{R} q);
```

because it would be a shorthand for:

```
template <RandomAccessIterator R, RandomAccessIterator R>
void sort(R p, R q);
```

which illegally tries to redeclare the template parameter `R`.

This is consistent with the treatment of concept introductions in template parameter lists, where each name in the introduction declares a new template parameter.

2.3 Concept introduction in a variable declaration

Previous Concepts Lite proposals indicated that a concept name could be used as a type specifier in a variable declaration:

```
RandomAccessIterator r = find(p, q, v);
```

Here, the type of `r` is deduced from the type of `find(p, q, v)`, but the compiler additionally checks that the deduced type satisfies the `RandomAccessIterator` constraint.

If the above is accepted, then for consistency we propose that the concept introduction syntax be allowed in this context as well:

```
RandomAccessIterator{R} r = find(p, q, v);
// R is the type of r
```

This is equivalent to the above, but it introduces a name for the concrete type of `r` that is more convenient than `decltype(r)`.

It would then be natural to allow using this with `auto` as well:

```
auto{T} t = foo();
// T is the type of t
```

3 Discussion

As stated in 2.1.3, we propose that each use of a name in a concept introduction constitutes a declaration of that name as a template parameter, so that

```
template <typename A, BinaryConcept{A, B}>
void function(A, B);
```

is ill-formed because it is a shorthand for

```
template <typename A, typename A, typename B>
  requires BinaryConcept<A, B>()
void function(A, B);
```

An alternative would be to say that a use of a name in a concept introduction constitutes a declaration of that name as a template parameter only if that name has not already been declared. Under this interpretation, the introduction above would be a shorthand for:

```
template <typename A, typename B>
  requires BinaryConcept<A, B>()
void function(A, B);
```

and would thus be valid.

We do not propose this alternative because it would mean that the introduction of a name might be a declaration of that name, or just a use of the name depending on the context. This would be akin to C++ allowing variables to be used without being declared, with the first use acting as a declaration. It would be very easy to get into a situation like the following:

```
// Earlier on in the translation unit
class B { ... };

...
template <UnaryConcept A, BinaryConcept{A, B}> // oops, B refers to the class
declared above
void f(...);
```

Acknowledgements

We are grateful for the feedback provided by Danny Ferreira for this proposal.

References

[1] Andrew Sutton, Bjarne Stroustrup, *et al.*, *Concepts Lite*, Technical Report N3701, ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++, Jun 2013.