Today's *return-type-requirement*s Are Insufficient

Document #: WG21 P1084R2
Date: 2018–11–06
Audience: CWG, LWG
Reply to: Walter E. Brown <webrown.cpp@gmail.com>
Casey Carter <casey@carter.net>

Contents

Abstract

This paper proposes a more concise way to specify the same set of constraints that result from requirements of the following form: `E; requires Concept<decltype((E)), Args...>;`. The current *return-type-requirement* ([expr.prim.req.compound]) is demonstrated to be insufficient to the task.

Nobody puts constraints on God. She doesn't like it.

— Andrew Greeley

Деспотизм ограничения только помогает достичь точности исполнения.
(The despotism of a constraint helps only to achieve accuracy of execution.)

— Igor Fyodorovich Stravinsky

1 Introduction

Issue 12[1] ("Add same-type constraints for expressions") of the now-retired C++ Concepts Issues List was opened in 2015 in response to a national body comment on the then-nascent Concepts TS.

- The issue first observes that we have concepts notation "to specify (via the trailing-return-type notation `->`) that a constraint is satisfied iff an expression `E` is convertible to a type `T`."[2]
- Then, the issue proposes that "It would be very useful to have ... constraints that are satisfied iff `decltype(E)` is exactly the type `T`."
- Finally, to address the above perceived need, the issue suggests to "Introduce new notation (e.g., `E => T`) to denote a constraint that is satisfied iff the expression `E` has precisely the type `T`."

[1] See http://cplusplus.github.io/concepts-ts/ts-closed.html#12.

[2] At the time, such notation was specified in subclause 14.10.1.1 [temp.constr.conv].

According to that issues list, the NB comment-*cum*-issue was at the time treated as an extension request and sent to EWG for triage. In turn, EWG declared that it was "not in favor of adding this feature."

Today, with more than three years of additional experience in the development and use of concepts, we believe it is time to revisit this issue and propose a generalization of the above issue's same-type constraint. In particular, experience with the Ranges TS draft [N4382] and follow-on papers (e.g., [P0898R0]) strongly suggests that we need a more concise way to specify the same set of constraints that result from the following requirements:

```
E; requires Concept<decltype((E)), Args...>;.
```

To achieve this desired concision, this paper proposes to adopt the following syntax:

```
{ E } => Concept<Args...>;.
```

As an alternative, this paper proposes:

```
{ E } -> Concept<Args...>;,
```

in which the right arrow from a *return-type-requirement* ([expr.prim.req.compound]) is respecified to have the interpretation shown above.

2 Background

[N4382] contains many nested requirements of the form `requires Same<T, decltype(E)>;`, effectively requiring that the type and value category of expression `E` is exactly `T`. During review in Lenexa, Casey Carter suggested to Eric Niebler that these could be reformulated as `{ E } -> Same<T>;`. Splitting the primary components of the nested requirement into these distinct pieces of syntax has two benefits:

- it's more easily parsed by humans, since the syntax separates the required expression from the constraint placed upon the required expression, and
- it results in clearer error messages for uses of the concept in which `E` is not a valid expression, since the syntax creates separate constraints for the distinct bits of syntax, namely (a) the expression constraint "`E` must be a well-formed expression" and (b) the deduction constraint "the type of `E` must satisfy `Same<T>`").

Eric agreed with Casey that this was an improvement to the specification of the Ranges concepts. They then applied that transformation throughout the working paper.

Alas, when Casey began implementing the working paper, he quickly discovered bugs in GCC's implementation of deduction constraints that rendered them effectively useless. To work around this limitation, Casey devised the following macro:

```
#define STL2_EXACT_CONSTRAINT(E, T) E; requires Same<T, decltype((E))>
```

which he then applied in lieu of `{ E } -> Same<T>;` in his implementation. This form was well-supported by GCC, and proved sufficient to implement all of [N4382].

During further development of the Ranges TS, Casey and Eric ran across similar instances of requirements that an expression be valid and that its type and value category satisfy some particular concept. For example, the `Boolean` concept requires (in part) that the expression `!b` (a) be valid and (b) be both implicitly and explicitly convertible to `bool`, with both kinds of conversion producing the same value. They wrote this requirement as `{ !b } -> ConvertibleTo<bool>;`, generalizing the style of the "expression with exact type and value category" requirement to a "expression whose type and value category satisfy *partial-concept-id*" requirement. They implemented this with a similar macro `STL2_CONVERSION_CONSTRAINT(E, T)` and all was well with the world.

Until one day it wasn't.

Late in the TS process, Eric decided to test whether GCC's implementation of deduction constraints had improved such that the preferred syntax could be used directly in the implementation. He therefore redefined the macros so as to emit that syntax in lieu of the workaround. Unfor-

tunately, "everything halted and burst into flames."[3] After considerable study of the Concepts WP wording, they realized that they had misunderstood the lowering of deduction constraints: `{ E } -> Same<T>` produces a deduction constraint that requires `f(E)` to be a valid expression for an invented abbreviated function template `void f(Same<T>);`. That function template lowers to the syntax:

```
template<class U> requires Same<U, T> void f(U);
```

This results not in the intended requirement that `E`'s exact type and value category be `T`, but requires instead that `E` be convertible to `remove_cvref_t<decltype((E))>` and that `remove_cvref_t<decltype((E))>` and `T` be the same type. This is a totally different requirement than the intent: for example, it's not satisfiable at all when `T` is a reference type.

The solution they devised was to reformulate their requirements in a more obscure style; they used either `{ E } -> Same<T>&&;` or `{ E } -> Same<T>&;`. The first reformulation requires `f(E)` to be valid for the invented function template:

```
template<class U> requires Same<U, T> void f(U&&);
```

which also is not an "exact type and value category" requirement but an "exact type and same rvalue-ness" requirement. The second reformulation requires `f(E)` to be valid for:

```
template<class U> requires Same<U, T> void f(U&);
```

which is intended to be a "exact type and lvalue" requirement.

Since then, they discovered that both reformulations are still broken. The first is unsatisfiable when `T` is an rvalue reference type, since `U` always deduces to either an lvalue reference or a non-reference type. The second breaks when `T` is a reference type.

3 Proposal summary

In brief, every attempt to express needed constraints via a *return-type-requirement* has failed. To meet the needs demonstrated above, we must either (a) provide a different operator (e.g., `=>`) or (b) respecify today's *return-type-requirement* ([expr.prim.req.compound]).

4 Addendum

Upon review (in Rappersville, 2018-06) of the initial proposal, EWG voted (5-20-6-0-0) to "Change the specification of the *return-type-requirement* so it is defined in terms of concept check instead of template argument deduction." The following proposed core wording reflects that endorsement, as well as the group's stated preference for option (b) above, namely "respecify today's *return-type-requirement* ([expr.prim.req.compound])." We also provide editorial direction to take advantage of those core language adjustments in specifying the recently-integrated concepts library (clause 17).

5 Proposed wording[4]

5.1 As shown below, edit [expr.prim.req.compound] so that a *return-type-requirement* of the form `{ E } -> Concept<Args...>;`
will behave as if written `E; requires Concept<decltype((E)), Args...>;`.

compound-requirement:
`{` *expression* `}` **noexcept**$_{opt}$ *return-type-requirement*$_{opt}$

[3] See https://github.com/ericniebler/stl2/issues/330.

[4] All proposed additions and ~~deletions~~ are relative to [N4762]. Editorial notes are displayed against a gray background.

return-type-requirement:
trailing-return-type
~~-> *cv-qualifier-seq*$_{opt}$ *constrained-parameter cv-qualifier-seq*$_{opt}$ *abstract-declarator*$_{opt}$~~
-> *qualified-concept-name*

1 A *compound-requirement* asserts properties of the *expression* **E**. Substitution of template arguments (if any) and verification of semantic properties proceed in the following order:

(1.1) — Substitution of template arguments (if any) into the *expression* is performed.

(1.2) — If the **noexcept** specifier is present, **E** shall not be a potentially-throwing expression (13.4).

(1.3) — If the *return-type-requirement* is present, then:

(1.3.1) — Substitution of template arguments (if any) into the *return-type-requirement* is performed.

(1.3.2) — If the *return-type-requirement* is a *trailing-return-type* ([dcl.decl]), **E** is implicitly convertible to the type named by the *trailing-return-type*. If conversion fails, the enclosing *requires-expression* is **false**.

(1.3.3) — If the *return-type-requirement* ~~starts with a *constrained-parameter* (12.1), the *expression* is deduced against an invented function template `F` using the rules in 12.9.2.1. `F` is a `void` function template with a single type template parameter `T` declared with the *constrained-parameter*. A *cv-qualifier-seq cv* is formed as the union of `const` and `volatile` specifiers around the *constrained-parameter*. `F` has a single parameter whose *type-specifier* is *cv* `T` followed by the *abstract-declarator*. If deduction fails, the enclosing *requires-expression* is `false`.~~ is a *qualified-concept-name* ([temp.param]) of the form *nested-name-specifier*$_{opt}$ *concept-name*, the concept it denotes shall be satisfied with **decltype((E))** as its sole argument. If the *return-type-requirement* is a *qualified-concept-name* of the form *nested-name-specifier*$_{opt}$ *concept-name* **<** *template-argument-list*$_{opt}$ **>**, the concept it denotes shall be satisfied with **decltype((E))** as its first argument and with the elements, in the order listed, of the *template-argument-list* comprising the concept's subsequent arguments. [*Note:* Thus, constraints of the form **{ E } -> Concept;** or of the form **{ E } -> Concept<>;** are equivalent to **E; requires Concept<decltype((E))>;** while a constraint of the form **{ E } -> Concept<A**$_1$**, A**$_2$**, ..., A**$_n$**>;** is equivalent to **E; requires Concept<decltype((E)), A**$_1$**, A**$_2$**, ..., A**$_n$**>;**. — *end note*]

[*Example*:
. . .
~~`template<typename T, typename U> concept C3 = requires (T t, U u) {`~~
~~`  t == u;`~~
~~`};`~~
~~`template<typename T> concept C4 = requires(T x) {`~~
~~`  {*x} -> C3<int> const&;`~~
~~`};`~~
~~The *compound-requirement* requires that `*x` be deduced as an argument for the invented function:~~
~~`  template<C3<int> X> void f(X const&);`~~
~~In this case, deduction only succeeds if an expression of the type deduced for `X` can be compared to an `int` with the `==` operator.~~
. . .
— *end example*]

5.2 Throughout Clause [concepts], apply the revised interpretation of a *return-type-requirement*.

- Replace each constraint of the form `E; requires Concept<decltype((E))>;` by the now-equivalent form `{ E } -> Concept;`.
- Replace each constraint of the form `E; requires Concept<decltype((E)), Args...>;` by the now-equivalent form `{ E } -> Concept<Args...>;`.

6 Acknowledgments

We gratefully acknowledge Barry Rezvin, Tim Song, and Zhihao Yuan for their respective reviews of early drafts of the proposed wording. Thanks also to Lev Minkovsky for suggesting improvements in translating the Stravinsky epigraph.

7 Bibliography

[N4382] Eric Niebler: "Working Draft, C++ Extensions for Ranges." ISO/IEC JTC1/SC22/WG21 document N4382 (pre-Lenexa mailing), 2015–04–12. http://wg21.link/n4382.

[N4762] Richard Smith: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N4762 (corrected post-Rappersville mailing), 2018–07–07. http://wg21.link/n4762.

[P0898R0] Casey Carter: "Standard Library Concepts." ISO/IEC JTC1/SC22/WG21 document P0898R0 (pre-Jacksonville mailing), 2018–02–12. http://wg21.link/p0898r0.

8 Document history

Rev.	Date	Changes
0	2018–05–06	• Published as P1084R0, pre-Rappersville.
1	2018–10–07	• Adopted EWG guidance received in Rappersville, and provided corresponding proposed core wording. • Provided proposed library concepts guidance for taking advantage of repurposed notation. • Improved the manuscript via minor editorial tweaks. • Published as P1084R1, pre-San Diego.
2	2018–11–06	• Adopted CWG guidance received in San Diego. • Published as P1084R2, post-San Diego.