Document Number:	N4186
Date:	2014-10-10
Project:	Programming Language C++, Evolution Working Group
Reply-to:	Matthias Kretz <kretz@compeng.uni-frankfurt.de>, Jens Maurer <Jens.Maurer@gmx.net>

Supporting Custom Diagnostics and SFINAE

ABSTRACT

Use of `static_assert` in the body of a function will not lead to a substitution failure in template instantiation, thus making it impossible to create a trait that can distinguish between the intended and unintended use of the function. This paper discusses the current situation and possible solutions to allow custom diagnostics while at the same time enabling traits to test for usability of a function. The suggestion of this paper is to extend the syntax of deleted functions to allow custom diagnostics.

CONTENTS

```
class simd_float;
template <typename T> simd_float operator+(simd_float, T) {
  static_assert(has_compatible_vector_size<simd_float, T>::value,
                "Incompatible operands: the SIMD register sizes for "
                "both operands must be equal on all possible target "
                "platforms to ensure portable code. Use an explicit "
                "type conversion to make the code portable.");
  return ...;
}
```

Listing 1: Example usage of `static_assert` for a more informative error message.

```
template <typename T, typename U,
          typename = decltype(std::declval<T>() + std::declval<U>())>
std::true_type test(int);
template <typename T, typename U> std::false_type test(...);
template <typename T, typename U = T>
struct has_addition_operator : public decltype(test<T, U>(1)) {};
```

Listing 2: A type trait that checks for the existence of `operator+(T, U)`.

1 PROBLEM

Static assertions are a very useful tool to improve error messages if a library interface is used incorrectly. Consider the addition operator in Listing 1.

This has the following effects:

1. `operator+` is a *viable* function for portable and unportable uses of the addition operator.
2. The program is *ill-formed* if an unportable type combination is used.
3. The compiler will output the second argument to the `static_assert` as *custom diagnostic* output if an unportable type combination is used.
4. A *SFINAE* (or concept) check for the *usability* of the addition operator for an unportable type combination is impossible to implement.

The `has_addition_operator` trait in Listing 2 will not tell whether a call to `operator+(T, U)` leads to a failed static assertion. This depends on the rules of substitution failure: The expression `decltype(std::declval<T>() + std::declval<U>())` yields a valid type even if `has_compatible_vector_size<simd_float,`

```
template <typename T>
enable_if_t<has_compatible_vector_size<simd_float, T>::value,
            simd_float>
    operator+(simd_float, T);
template <typename T>
enable_if_t<!has_compatible_vector_size<simd_float, T>::value,
            simd_float>
    operator+(simd_float, T) = delete;
```

Listing 3: Using a deleted function as an alternative implementation to Listing 1.

`T>::value` is `false`.[1] The substitution rules do not depend on whether a static assertion fails on instantiation of a template function. They do depend on whether the (viable) function is accessible (`public` vs. `private`) or *deleted*, though. Thus, the `has_addition_operator` trait will tell whether an addition operator is inaccessible or deleted.

Listing 3 shows an implementation of `operator+` that solves the SFINAE issue of Listing 1 but at the cost of losing custom diagnostics output. The compiler has no idea why the library developer decided to declare the function as deleted. Thus, all it can do is tell that a deleted function was used. This tells a user of the library that either the library developer made a mistake or it was really intended that this overload is forbidden.

There is no way in current C++ to declare a function in such a way that all four items are satisfied:

1. *viable* for incorrect use
2. *ill-formed* for incorrect use
3. *custom diagnostics* output for incorrect use
4. *SFINAE* or Concepts can check for *usability*, not only viability

The *Custom diagnostics* and *SFINAE* features are mutually exclusive.

2 POSSIBLE SOLUTIONS

Approaches:

1. Introduce a new type trait (which requires compiler support) that can detect whether a given expression fails a static assertion.

1 It would be possible to modify Listing 1 such that the return type is invalid, but then the function would not be viable for unportable type combinations and the `static_assert` would never fail...

2. Extend concepts to do “negative matching” to enable customized diagnostics. Thus, a call to `simd_float + double` would match the second overload in Listing 4 as best viable function and make such a program ill-formed with the string after `error` used for diagnostics. In a template parameter substitution this would lead to a failure and thus enable `has_addition_operator` to check for usability of the addition operator.

```
template <typename T>
  requires has_compatible_vector_size<simd_float, T>::value
simd_float operator+(simd_float, T);
template <typename T>
  requires !has_compatible_vector_size<simd_float, T>::value
  error "<how to use + correctly>"
simd_float operator+(simd_float, T);
```

Listing 4: Notion of “negative matching” as an extension to concepts.

3. Introduce an additional check at the end of overload resolution [1, §13.3 over.match], in the same spirit as the check for accessibility:

§13.3 [over.match]

If a best viable function exists and is unique, overload resolution succeeds and produces it as the result. Otherwise overload resolution fails and the invocation is ill-formed. When overload resolution succeeds, and the best viable function is not accessible (Clause 11) in the context in which it is used or template instantiation would lead to a failed `static_assert`, the program is ill-formed.

The intention is to trigger a substitution failure when a `static_assert` would fail and thus enable SFINAE.

4. Extend the `delete` expression for deleted functions [1, §8.4.3 dcl.fct.def.delete] to accept an optional string argument that will be used for diagnostics output.

§8.4.1 [dcl.fct.def.general]

Function definitions have the form

function-definition:

  attribute-specifier-seq$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator virt-specifier-seq*$_{opt}$ *function-body*

function-body:

  ctor-initializer$_{opt}$ *compound-statement*

  function-try-block

  deleted-definition

  `= default ;`

  ~~`= delete ;`~~

deleted-definition:

  `= delete (` *string-literal* `) ;`

  `= delete ;`

```
template <typename T>
enable_if_t<!has_compatible_vector_size<simd_float, T>::value,
            simd_float>
    operator+(simd_float, T) =
        delete ("Incompatible operands: the SIMD register sizes for "
                "both operands must be equal on all possible target "
                "platforms to ensure portable code. Use an explicit "
                "type conversion to make the code portable.");
```

Listing 5: Providing custom diagnostics to a deleted function.

§8.4.3 [dcl.fct.def.delete]

A function definition of the form:

attribute-specifier-seq$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator* *virt-specifier-seq*$_{opt}$ ~~= delete ;~~ *deleted-definition*

is called a *deleted definition*. A function with a deleted definition is also called a *deleted function*.

A program that refers to a deleted function implicitly or explicitly, other than to declare it, is ill-formed, and the resulting diagnostic message (1.4) shall include the text of the string-literal, if one is given, except that characters not in the basic source character set (2.3) are not required to appear in the diagnostic message. [*Note:* This includes calling the function implicitly or explicitly and forming a pointer or pointer-to-member to the function. It applies even for references in expressions that are not potentially-evaluated. If a function is overloaded, it is referenced only if the function is selected by overload resolution. — *end note*]

With this solution the deleted function in Listing 3 can be extended as shown in Listing 5.

3 EVALUATION

Approaches 1 and 3 require instantiation of the constant part of the function body to evaluate the `static_assert` during overload resolution, before actually selecting the function. This seems novel territory for such a small feature.

Similarly, approach 2 requires extensions to the concepts design, which is currently not even a working draft for a Technical Specification.

Approach 4 can be implemented as a fairly small extension to the current check whether a function is deleted at the end of overload resolution. The issue of integrating string-literals from program source code to diagnostic compiler output was already solved for `static_assert`.

The recommendation is to proceed with approach 4.

A ACKNOWLEDGEMENTS

- This work was supported by GSI Helmholtzzentrum für Schwerionenforschung and the Hessian LOEWE initiative through the Helmholtz International Center for FAIR (HIC for FAIR).

B REFERENCES

[1] Stefanus Du Toit, ed. N3936: Working Draft, Standard for Programming Language C++. ISO/IEC C++ Standards Committee Paper, 2014. URL `http://www.open-std.org/jtc1/sc22/wg21/prot/14882fdis/n3936.pdf`.