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.
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, T>::value`.

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

```cpp
class simd_float;

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

```cpp
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)) {}
```

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

## 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 \texttt{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 \texttt{has\_addition\_operator} to check for usability of the addition operator.

```cpp
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 \cite[§13.3 over.match]{cpp17}, in the same spirit as the check for accessibility:

\begin{quote}
\textsc{§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 \texttt{static\_assert}, the program is ill-formed.

The intention is to trigger a substitution failure when a \texttt{static\_assert} would fail and thus enable SFINAE.
\end{quote}

4. Extend the \texttt{delete} expression for deleted functions \cite[§8.4.3 dcl.fct.def.delete]{cpp17} to accept an optional string argument that will be used for diagnostics output.

\begin{quote}
\textsc{§8.4.1 [dcl.fct.def.general]}
Function definitions have the form

\texttt{function-definition:}
\begin{verbatim}
attribute-specifier-seq\_opt decl-specifier-seq\_opt declarator virt-specifier-seq\_opt function-body
\end{verbatim}

\texttt{function-body:}
\begin{verbatim}
ctor-initializer\_opt compound-statement
function-try-block
\end{verbatim}

\texttt{deleted-definition:}
\begin{verbatim}
= delete ;
\end{verbatim}
\end{verbatim}
\end{quote}
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.
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).

REFERENCES