Paper Number: P1936R0
Title: Dependent Static Assertion
Authors: Ruslan Arutyunyan <Ruslan.Arutyunyan@intel.com>
Audience: EWG-I (Evolution Working Group-Incubator)
Date: 2019-10-07

I. Introduction

We need to introduce a generic solution to create the dependent scope for `static_assert(false)` expression where it is semantically necessary.

II. Motivation and Scope

In several scenarios `static_assert(false)` expression is a useful construction. That happens when better diagnostics should be provided to the user. However, if implementer just writes `false` as the first argument of the `static_assert` the program has never been compiled successfully.

Consider the following examples where such semantics can be useful:

Suppose the user have to implement the function with the signature:

```
template <typename T>
int my_func(const T&)
```

and it is necessary to implement it in accordance with the following requirements:

- If T is integral type, returns 1
- Otherwise if T is convertible to `std::string`, returns 2
- Otherwise the program is ill-formed.

Possible implementation might be:

```
template <typename T>
int my_func(const T&)
{
    if constexpr(std::is_integral_v<T>)
    {
        return 1;
    }
    else if constexpr (std::is_convertible_v<std::string, T>)
    {
        return 2;
    }
    else
    {
        // Always Compile-time error
        static_assert(false, "T is not integral and is not
                              convertible to std::string");

    }
}
```

But as mentioned above this code cannot be compiled successfully due to `static_assert(false)` expression.

Another example where `static_assert(false)` might be useful is the class template for which primary template is not defined. Instead, user should always pass correct template arguments that one of specializations has been chosen.

Consider the following code snippet:

```
// Primary template
template <typename T, typename U>
struct my_struct;

// Partial specialization
template <typename T, typename Alloc>
struct my_struct<int, std::vector<T, Alloc>>
{
};

// User code
int main()
{
    my_struct<int, int> s;
}
```

Examples of compiler messages are:

- Clang: `error: implicit instantiation of undefined template 'my_struct<int, int>'`
- GCC: `error: aggregate 'my_struct<int, int> s' has incomplete type and cannot be defined`
- Intel Compiler: `error: incomplete type is not allowed my_struct<int, int> s;`

Implementer might want to provide better diagnostics to the user. The possible approach might be:

```
template <typename T, typename U>
struct my_struct
{
    // Always Compile-time error
    static_assert(false, "Type T and Type U cannot be used in such
                          combination. See the documentation");
};
```

Unfortunately, the static assertion in the code above is always failed despite if primary template has been chosen or not.

III. Problem statement

To overcome the mentioned issue the implementer should write some implementation to create dependent scope for the `static_assert(false)` expression.

Possible implementation:

```
template <typename T>
```

```
constexpr bool always_false()
{
    return false;
}

template <typename T, typename U>
struct my_struct
{
    // static_assert fails only if primary template is chosen
    static_assert(always_false<T>());
};
```

Many template libraries implement the approach above in their manner. It's better to have one standard solution instead of having a lot of workarounds everywhere implemented differently.

IV. Proposal

The issue may be addressed by introducing the generic solution for such problem.

Introduce new syntax for static_assert with optional angle brackets:

static_assert<T>**(** *bool_constexpr* , *message* **)**

static_assert<T>**(** *bool_constexpr* **)**

that would not be calculated unless entered the dependent scope.

In that case the `static_assert` would look like either:

```
template <typename T, typename U>
struct my_struct
{
    static_assert<T>(false);
};
```

A dependent static assertion is created with help of optional template parameter. It would be evaluated only if primary template is chosen.

The proposed API should work with all:

- Type template parameters
- Non-type template parameters
- Variadic template parameters

For convenience it also may work with variadic templates pack but it is optional

V. Additional notes

Another parallel paper (P1830R1) that tries to solve this problem on the library level is submitted.

Unfortunately, it cannot fulfill all use-case since it is hard to impossible to support all combinations of template template parameters in the dependent scope.