

Paper Number: P1830R0
Title: `std::dependent_false`
Authors: Ruslan Arutyunyan <Ruslan.Arutyunyan@intel.com>
With Input from: Billy O'Neal <bion@microsoft.com>, CJ Johnson <johnsoncj@google.com>
Audience: LEWG-I (Library Evolution Working Group-Incubator)
Date: 2019-07-18

I. Introduction

We need to introduce a generic solution to create the dependent scope for `static_assert(false)` expression where it is semantically necessary and understand if there are use cases for the abstract solution.

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 when it 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 has never been compiled successfully.

Another example where in `static_assert(false)` might be useful is the class template for which primary template is not defined. Instead, user should always pass correct template parameters 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 whatever stuff to create dependent scope for the `static_assert(false)` expression.

Example:

```

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>());
};

```

IV. Proposal

The issue may be addressed by introducing the generic solution for such problem. There may be several approaches to achieve that:

a) dependent_false variable template

```

template <typename... Args>
constexpr bool dependent_false = false;

```

In that case the `static_assert` would look like:

```

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

```

We create a dependent expression with help of `dependent_false` variable template that it would be evaluated if primary template is chosen.

In the proposed API `Args...` may be missed by mistake but such kind of error would be easily caught at compile-time.

Here and in other possible solutions we propose to use variadic templates for the interface. The example above shows why it is convenient.

Suppose we have `my_struct` declaration as follows:

```

template <typename... Args>
struct my_struct;

```

and the API for dependent scope is

```

template <typename T>
constexpr bool dependent_false = false;

```

In that case `my_struct` definition is

```

template <typename... Args>
struct my_struct
{
    static_assert(dependent_false<std::tuple_element_t<0,
                                                std::tuple<Args...>>>);
};

```

With the proposed API it looks like

```

template <typename... Args>
struct my_struct
{
    static_assert(dependent_false<Args...>);
};

```

b) dependent_bool_value variable template

```

template <bool value, typename... Args>
constexpr bool dependent_bool_value = value;

```

In that case the static_assert would look like:

```

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

```

c) dependent_value variable template

```

template <typename T, T t, typename... Args>
constexpr T dependent_value = std::integral_constant<T, t>{};

```

In that case the static_assert would look like:

```

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

```

d) dependent_constant class template

```

template <typename T, T t, typename... Args>
struct dependent_constant : std::integral_constant<T, t> {};

```

And dependent_bool_constant alias template for convenience and for consistency with integral_constant:

```

template <bool b, typename... Args>
using dependent_bool_constant = dependent_constant<bool, b, Args...>;

```

In that case the static_assert would look like:

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

V. Further possible improvements

- a) Is additional API required to support non-type template parameters?
- b) Is additional API required to support template template parameters?

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2019, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804