static_printf: Standardising build-time output

Document #:	PXXXXR0
Date:	2021-01-19
Project:	Programming Language C++
Audience:	XXX

Reply-to: Jake Arkinstall <jake.arkinstall@gmail.com>

Abstract

There is currently no standardised way to print the values of constant expressions at compilation time. Whereas `static_assert` provides the ability to compare constant expressions, the optional failure message must be a string literal. A standardised mechanism to provide contextual information to the at build time could assist developers and users of C++ libraries to understand and resolve compilation errors. `static_printf` and `static_assertf` are proposed for such a purpose. Such functionality could also provide a method of providing simple compile-time analytics using standard C++ code, without the need for compiler-specific extensions.

1 Introduction

When an error occurs in a program at runtime, it is often useful to provide the user with sufficient contextual information to identify the source of the problem. For example, if a function expects that two integral values are expected be equal after a calculation, but they fail to reach equality, it would be pertinent to provide those values, perhaps alongside any intermediate calculations, as well as a message explaining that they are expected to be equal.

The same argument might also apply to an error that occurs at compile time, particularly if the cause of the error is a failing static assertion. However, C++ currently lacks the ability to output values at compile time. As a result, static assertion messages can be vague, and debugging code can be a difficult task.

This exact issue is highlighted by the examples of `static_assert` provided by the C++ standard itself, as explained in . This is by no means a fault of the authors, nor of the documentation, but of the lack of ability to communicate contextual information to the user during compilation. This is explored futher in section 2.

A possible solution to this problem is then provided in section 3, through new `static_printf` and `static_assertf` declarations.

The ability to output information at build time may also serve purposes outside of debugging. An example of this could be the output of metadata during a build process, which might later be extracted and digested for future monitoring of build analytics without the need for a compiler-specific extension. Such considerations are explored in subsection 4.2.

Finally, possible future work is provided in section 5. This covers such ideas as the encorporation of `std::format` in a compilation-time setting to allow easier output of custom objects.

2 When `static_assert` is insufficient

There are three classes of error message that are possible through `static_assert`. Using examples taken from the C++20 final working draft[Smith2020] as illustration, these three classes are

1. A specific, contextual but hardcoded message:

```
static_assert(ratio_multiply<ratio<1, 3>, ratio<3, 2>>::num == 1,
              "1/3*3/2 == 1/2");
```

2. A message that conveys the general problem, without specific information

```
static_assert(sizeof(int) == sizeof(void*), "wrong pointer size");
```

3. No message at all, as of C++17:

```
static_assert(same_as<decltype(result2), vector<int>::iterator>);
```

In all cases, the user needs to extrapolate contextual information from additional messages from the compiler. Such additional messages are not standardised, which leads to varying user experiences across different compilers.

Using the "wrong pointer size" example from item 2 with three popular C++ compilers highlights these differences.

```
# Clang 11.0.0
<source>:1:1: error: static_assert failed due to requirement
    `sizeof(int) == sizeof(void *)' "wrong pointer size"
static_assert(sizeof(int) == sizeof(void*), "wrong pointer size");
^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# Microsoft Visual C++ v19.28
<source>(1): error C2338: wrong pointer size

# GCC v10.2
<source>:1:27: error: static assertion failed: wrong pointer size
1 | static_assert(sizeof(int) == sizeof(void*), "wrong pointer size");
  |               ~~~~~~~~~~~~^~~~~~~~~~~~~~~~
```

Each implementation provides breadcrumbs of information for a developer to start their debugging journey: the line number and message. Two implementations even highlight the failing assertion itself. But no implementation provides the key piece of information that a user may need for further investigation: `sizeof(int)` and `sizeof(void*)`.

This is a trivial example, and `sizeof(int)` can be found quite easily. As complexity grows, the ease of interpreting the assertion and the accompanying message diminishes. It is neither unreasonable nor uncommon for static assertions to be performed on constant expressions derived from type information provided from an outside source, such as the call site within user code, which restricts the ability of template-metaprogramming code to report back to the user in the case of an error with standard C++ code.

We turn our attention to a slightly less trivial example to explain the kind of information that a library author might wish to convey to the user upon error.

```
#include <type_traits>
#include <tuple>
```

```cpp
#include <array>

template<std::size_t N>
constexpr bool foo(auto left, auto right){
    using L = decltype(left);
    using R = decltype(right);
    static_assert(std::is_same_v<std::tuple_element_t<N, L>,
                                 std::tuple_element_t<N, R>>,
                  "left and right inputs have different types at index N");
    return std::get<N>(left) < std::get<N>(right);
}
int main(){
    std::tuple left{true, 42, "Bar"};
    std::array right{16, 2, 23};
    return foo<1>(left, right) ? 1 : 0;
}
```

In this example, `foo` compares the N^{th} elements of two different inputs, which may independently be a tuple, pair or array, with the requirement that the N^{th} elements have the same type.

If we change the template argument of `foo` to 0, then the static assertion fails, and the user is faced with a vague error message. They might be able to piece together the necessary information from the function signature output, but the output of this is neither mandated by the standard nor output in a uniform manner.

```
# Clang 11.0.0
<source>:8:1: error: static_assert failed due to requirement
'std::is_same_v<bool, int>' "left and right inputs have different types at index N"
static_assert(std::is_same_v<std::tuple_element_t<N, L>,
^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<source>:16:8: note: in instantiation of function template specialization
'foo<0, std::tuple<bool, int, const char *>, std::array<int, 3>>' requested here
return foo<0>(left, right) ? 1 : 0;
       ^


# Microsoft Visual C++ v19.28
<source>(8): error C2338: left and right inputs have different types at index N
<source>(16): note: see reference to function template instantiation
'bool foo<0,std::tuple<bool,int,const char *>,std::array<_First,3>>(_T0,_T1)'
being compiled
        with
        [
            _First=unsigned int,
            _T0=std::tuple<bool,int,const char *>,
            _T1=std::array<unsigned int,3>
        ]

# GCC v10.2
<source>: In instantiation of 'constexpr bool foo(auto:11, auto:12)
[with long unsigned int N = 0; auto:11 = std::tuple<bool, int, const char*>;
auto:12 = std::array<int, 3>]':
<source>:16:26:   required from here
<source>:8:20: error: static assertion failed: left and right inputs have different
types at index N
    7 | static_assert(std::is_same_v<std::tuple_element_t<N, L>,
      |               ~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    8 | std::tuple_element_t<N, R>>,
      | ~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

From each of these messages, it is possible to piece together the problem from the error output, but it requires some legwork on behalf of the user. As the inputs grow in complexity, or if `foo` is used within a template metaprogramming library several calls from user-facing code, understanding the problem and understanding how to fix it can become more difficult.

3 Proposed Solution: `static_printf` and `static_assertf`

When static_assert is evaluated, the values of the constant expressions that form its assertion clause are already available to the compiler. The reader might be convinced by the examples in section 2 that providing a standardised mechanism to output values can aid in debugging static assertion failures.

For this purpose we propose a `static_printf` declaration, named for consistency with `static_assert`. We also propose a `static_assertf` declaration, which provides a formatted version of `static_assert`. The two differ in that `static_assertf` requires an contextually converted constant expression of type `bool` as the first argument, and like `static_assert` the program is ill-formed if this argument is `false`.

3.1 `static_printf`

The `static_printf` declaration behaves in the same way as `std::printf` behaves at runtime, but is only evaluated at compilation time, arguments must be constant expressions, and `stdout` adopts the compiler's `stdout`. The formatting rules follow the same formatting rules as `std::printf`.

The syntax is thus already familiar:

```
template<unsigned N>
consteval void print_N(){
    static_printf("The value of N is %u\n", N);
}
```

For example, when `print_N` is specialised with `N = 10` at compilation time, the "The value of N is 10\n" is output to the compiler's stdout. The compiler may opt to provide a preamble to the message, either in a previous line or at the start of the output line, but "The value of N is 10\n" must be visible and uninterrupted in the output. This should be done in an ordered manner, such that successive `static_printf` calls are necessarily output in the order that they are evaluated within a single compilation unit.

`static_printf` is to be used when the compiler should output information and continue. Except in the case of an invalid format string or incompatible parameters, `static_printf` should not have any effect on the output binary. When a `static_printf` format string or input parameter is erroneous, however, the program is ill-formed.

In the absense of `static_assertf` (introduced in subsection 3.2), a static assertion with formatted output could be emulated by handling the condition within an 'if constexpr' block, using `static_printf` to output contextual information, and finally using `static_assert` to make the program ill-formed and halt compilation. For example,

```
#include <type_traits>
#include <tuple>
#include <array>
template<std::size_t N>
constexpr bool foo(auto left, auto right){
    using L = decltype(left);
    using R = decltype(right);
    using L_N = std::tuple_element_t<N, L>;
    using R_N = std::tuple_element_t<N, R>;
```

```
    if constexpr(!std::is_same_v<L_N, R_N>){
        static_printf("The left argument's type at index %lu is '%s'.\n",
                      N, std::meta::name_of(reflexpr(L_N)));
        static_printf("The right argument's type at index %lu is '%s'.\n",
                      N, std::meta::name_of(reflexpr(R_N)));
        static_assert(false, "Foo requires the left and right arguments "
                             "to have the same type at the specified index.");
    }
    return std::get<N>(left) < std::get<N>(right);
}
int main(){
    std::tuple left{true, 42, "Bar"};
    std::array right{16u, 2u, 23u};
    return foo<0>(left, right) ? 1 : 0;
}
```

should output, taking into account the choice of the implementation to provide additional output,

```
  # optional preamble A #
  # optional preamble B # The left argument's type at index 0 is 'bool'.
  # optional preamble C #
  # optional preamble D # The right argument's type at index 0 is 'unsigned int'.
  # optional preamble E #
  # optional preamble F # Foo requires the left and right arguments to have the
  same type at the specified index.
```

3.2 static_assertf

It is conceivable that many uses of `static_printf` will be for diagnostic purposes; that is, after all, the motivation of this proposal. However, it would be unnecessarily restrictive to have `static_printf` render the program ill-formed and cause a termination of compilation. As seen in subsection 3.1, this presents a potential problem: the printing of diagnostic information would need to be encapsulated within an `if constexpr` block which leads to a failing `static_assert`.

To remedy this, `static_assertf` is also proposed. This declaration combines `static_assert` and `static_printf` to enable a `printf`-like interface to static assertions. In our example, we can then change the argument to

```
#include <type_traits>
#include <tuple>
#include <array>
template<std::size_t N>
constexpr bool foo(auto left, auto right){
    using L = decltype(left);
    using R = decltype(right);
    using L_N = std::tuple_element_t<N, L>;
    using R_N = std::tuple_element_t<N, R>;
    static_assertf(std::is_same_v<L_N, R_N>,
                   "The left argument's type at index %lu is '%s'.\n"
                   "The right argument's type at index %lu is '%s'.\n"
                   "Foo requires the left and right arguments to have "
                   "the same type at the specified index.",
                   N, std::meta::name_of(reflexpr(L_N)),
                   N, std::meta::name_of(reflexpr(R_N)));
    return std::get<N>(left) < std::get<N>(right);
}
```

`static_assertf`, just like `static_assert`, renders a program ill-formed if the first argument evaluates as false. The key difference is that the `message` parameter becomes a format string, and subsequent arguments provide the corresponding values at compilation time. The `static_printf` rule of sequential output should also be obeyed: any prior `static_printf` will be evaluated, and output to the compiler's `stdout`, before the formatted message from `static_assertf`.

4 Uses Outside of Debugging

Although debugging is the motivation for this proposal, allowing users and libraries with the ability to provide information at compile time could be beneficial for additional purposes. The general argument this proposal puts forward is that compile-time functionality is, in itself, a form of programming, and untold benefits could emerge from providing compile-time code with more opportunity to inform the user, through the compiler, of what the code is doing.

4.1 Improving the Compile-Time Learning Experience

`static_printf` might offer beginners with a more comfortable journey into compile-time code. The first thing that most beginners will write in a new programming language is "hello world". The second, more exciting step will typically involve creating variables, perhaps their name and age, and outputting them in a friendly message. These steps are far from trivial: they provide inspiration and confidence to every learner.

Compile-time programming in C++ is, in many ways, a language of its own; it has its own syntax, it operates within a different domain to the runtime code, and it even has its own books. But we currently have no direct "hello world" equivalent in compile-time code. We have indirect approaches, of course, where users might create a number at compile time and output it at runtime noting that the assembly output had the pre-calculated value hardcoded.

Being able to output information directly at compilation time, though, without resorting to non-standard tricks or relying on runtime code to serve information, could instil the same inspiration and confidence that learners get from any other language. Moreover, as they start creatively toying with compile-time logic to perform more complex tasks, they can perform actions step by step, observe the changes that they are making. They can see other peoples' code, output intermediate values and types to see how it works. This might make compile-time programming less intimidating and more intuitive to beginners and experts alike.

4.2 Compilation Analytics

Another potential use is that of analytics. The general idea is simple: by outputting information in several places, with a common format, build logs can be digested and analysed for the purpose of providing details about the processes that occur during the build.

An example of this could be a 3rd party library which a user or developer could utilised for the purposes of generating build-time metrics. This is a practice that is already deemed useful enough for the emergence of compiler-specific tools[msvcinsights], but there is no feasible approach for this in standard C++.

The types of output that such a library might wish to output could include lists or hierarchies of template instantiations, perhaps including the time of instantiation for benchmarking purposes [1]. Outside of benchmarking, such output could provide insights that can be useful for dead code analysis, as well as scalability and complexity analysis to compare multiple solutions to the same problem.

[1] Benchmarking would require a method to get the current time during template instantiation. Such a method does not yet exist, because it would serve little purpose before `static_printf`.

5 Future considerations

5.1 `std::format`

In C++20, `std::format` was adopted, allowing users much greater flexibility in how their types should be output. At present, `std::format` lacks the `constexpr` functionality to be used for the purposes of compile-time string formatting. However, it is conceivable that this could change in future revisions of the standard. If this were to happen, a `static_printf` variant that could utilise `std::format`'s advantages over `printf`, both in terms of syntax and flexibility, might be considered.

5.2 `Compile-Time Time`

As noted in subsection 4.2, benchmarking template instantiations would require a 'consteval' method for getting the current time, serving the purpose of `__TIME__` but at the time of evaluation instead of parsing.

References

[Smith2020] Richard Smith. Final Working Draft, Standard for Programming Language C++. https://isocpp.org/files/papers/N4860.pdf, 2020 (accessed 2021-01-18).

[msvcinsights] Kevin Cadieux. Profiling template metaprograms with c++ build insights. https://devblogs.microsoft.com/cppblog/profiling-template-metaprograms-with-cpp-build-insights/, 2020 (accessed 2021-01-18).