Document number:	D
Date:	2021-06-16
Project:	Compile-Time Static Failure Language Feature
Reply-to:	David Aaron Braun da.braun1@hotmail.com

Table of Contents:

Introduction:

I proposal to add a new type of static assertion to the C++ Language standard which will always fail at compile time if the code reaches that declaration.

Motivation:

The motivation here is create a better solution to a new problem created by the introduction of *if constexpr.*

I think this is best explained through an example. Let's say you have a function; which for your use case needs to take a templated parameter of any standard integer type except one. Let's say *int32_t*. Thus, you'd like to use templates instead of overloads. But you need to test which integer type was passed in order to get the desired code to compile.

Current Solution	My Proposal
*template<typename T> requires std::is_integral<T> void my_func(T param) { if constexpr (std::is_same<T, uint8_t>()) { /*uint8_t specific Code here*/ } /*…*/ else if (std::is_same<T,int64_t>()) { /*int64_t specific Code here*/ } else { /*But what do I do if they passed an int32_t when we don't want them to? Throw an exception? Nope, those only get evaluated at run-time and we want a compile error here! We have to do this: */* ***[] <bool flag = false>() { static_assert(flag,"Compile Error! You passed an int32_t into my_func !"); }();*** *} //End else } //End my_func*	*template<typename T> requires std::is_integral<T> void my_func(T param) { if constexpr (std::is_same<T, uint8_t>()) { /*uint8_t specific Code here*/ } /*…*/ else if (std::is_same<T,int64_t>()) { /*int64_t specific Code here*/ } else { /*'static_failure' always fails at compile time for the specified reason*/* ***static_failure("You passed an int32_t into my_func !");*** *} //End else } //End my_func*

The current solution is unintuitive, verbose and honestly kinda hacky. It is especially annoying and confusing that the *static_assert* must be wrapped in a lambda in order to work. (I have no idea why.)

My solution, I feel is clearly written and will do what it says in a more intuitive way.

Impact On the Standard:

As far as I know my proposal will not present any negative impacts to the current C++ standard. As it is an entirely new language keyword.

Care will need to be taken to educate C++ developers on the purpose of *static_failure* so that they do not try to use it in place of *static_assert.*

Design Decisions:

I chose this design instead of trying to modify *static_assert* for a 2 simple reasons:

1. ABI Breakage
2. Clarity of the code

Technical Specifications:

static_failure as described in this paper is a new declarer designed to induce a customized compile-time error which is triggered if/when the new *static_failure* keyword is reached in the code. The *static_failure* declaration is always evaluated at compile time and will produce output in the "errors / build output" section of which ever compiler is used.

I also propose that blank (length of zero) messages should not be allowed as this would trigger a compile failure with no specified reason.

Syntax:

static_failure (const char (literal) message)

Example (also see above):

static_failure("I am error!"); //Code does not compile, build output shows this message

✓**program is well-formed.**

static_failure("Press 'f' to pay respects."); //Code does not compile, build output shows this message

✓**program is well-formed.**

static_failure(""); //Code does not compile, build output shows "You must specify a reason for static failure in …" **X program is malformed.**

static_failure(); //Code does not compile, build output shows "You must specify a reason for static failure in …"
X program is malformed.

static_failure; //Code does not compile, build output shows "You must specify a reason for static failure in …"

X program is malformed.

static_failure //Code does not compile, build output shows "missing semi-colon on line #…"

X program is malformed.

Acknowledgements:

Thank You to the creators of the current C++ standard.

References: none.

goto Top;