Top Level Is Constant Evaluated

Document No P1955r0
Date November 3, 2019
Project The C++ Programming Language
Audience EWG
Authors Frank Birbacher (Bloomberg LP), frank.birbacher@gmail.com

With the discussion and outcomes of the `std::is_constant_evaluated()` functions we propose a different way to solve the need. The alternative is easier to understand, allows extension to more use cases, and should be less liable to wording oversights.

Contents

1 Revisions

Initial revision.

2 Introduction

The introducion of the `std::is_constant_evaluated()` function has created a lot of discussion about its correct use. Also it has fallen afoul of the maintenance trap resulting from concurrent modifications to the Standard resulting in a NB comment about missing support for calling `consteval` qualified functions. The current wording also comes with an example to convince the commttee members themselves of its meaning. While essential to the implementation of compile time containers, e. g. strings, we'll present alternatives that are easier to understand, more general in nature, and should result in less maintenance issues for the wording.

3 Too Long, Didn't Read

We propose to answer the question whether something is evaluated at compile time or not for a whole function body, that is, allow two different function bodies for the two cases respectively. This makes the facility easier to understand and harder to use wrong. Furthermore it also allows extension for more use cases, such as moving the runtime body out-of-line.

Listing 1: Tony Table: Example from *meta.const.eval* contrasted with idea for top-level distinction.

```
// BEFORE: Using is_constant_evaluated().
constexpr void f(unsigned char *p, int n) {
        if (std::is_constant_evaluated()) {
                for (int k = 0; k<n; ++k) p[k] = 0;
        } else {
                memset(p, 0, n);
        }
}

// AFTER: Proposed idea.
constexpr void f(unsigned char *p, int n)
consteval = {
        for (int k = 0; k<n; ++k) p[k] = 0;
}, {
        memset(p, 0, n);
}
```

4 We Propose

- Introduce a function body syntax to distinguish code for constant evaluation.
- Use language syntax to provide a language feature instead of a magic function.
- Reuse wording about `constexpr` functions and lose the need to speak of the semantics in `if`-branches.

Allow for the body of a function definition to give a `constexpr =` {...} body between the signature and the regular body. The meaning is to use the designated body for constant evaluation. All rules for what is possible inside a `constexpr` function would apply to that body. The regular body will be used for the runtime case as usual. Refer to the listing 1 for the proposed syntax. The items serve use cases that are detailed in the following sections.

Listing 2: Issues of using it wrong

```
// in Header:
constexpr void f(unsigned char *p, int n)
{
        // mistake here:
        if constexpr(is_constant_evaluated()) {
                for (int k = 0; k<n; ++k) p[k] = 0;
        } else {
                memset(p, 0, n);
        }
}
```

5 Use Case: Easy to Understand and Use

> As a C++ user I want to use the facilities offered with ease. As a Standards Committee Member I want to understand the dependencies between different features.

The definition and effects of `is_standard_evaluated()` confuses a number of people thereby raising the need to put an explaining example into the Standard. Also the particularities of it being a magic function incurs wording overhead regarding other features, such as `consteval` functions, see listing 2. These issues are also outlined in P1938R0.

By referring to the wording of definitions of `constexpr` functions the meaning of calling `consteval` functions becomes clear.

6 Future Use Case: Supply Out-Of-Line Definition

> As a C++ user I want to have functions with implementation details hidden in a translation unit and still let them have a `constexpr` part in the header.

Although this can be worked around by defining two functions, one of which is `constexpr` and calls the other for the not constant evaluated case, it's much more straight forward to only declare one function. Moving the definition out of the header allows to pull in additional includes without exposing them.

7 Future Use Case: Separate inline Implementation

> As a C++ user I want to optimize the code for an `inline` function in the non-inline case.

Listing 3: How to move runtime case into .cpp

```
// in Header:
constexpr void f(unsigned char *p, int n)
consteval = {
        for (int k = 0; k<n; ++k) p[k] = 0;
};  // Semicolon concludes declaration with missing definition

// in .cpp:
void f(unsigned char *p, int n)
{
        memset(p, 0, n);
}
```

Listing 4: Hide detail in .cpp

```
// in Header:
void f(unsigned char *p, int n)
inline = {
        for (int k = 0; k<n; ++k) p[k] = 0;
};  // Semicolon concludes declaration with missing definition

// in .cpp:
#include <cstdlib>
void f(unsigned char *p, int n)
{
        std::memset(p, 0, n);
}
```

This use case was suggested in P1220R0 for San Diego. The feature has been deemed too specific at that time, but when combined with the `constexpr` case discussed here it makes for a stronger argument. The proposed split between constant evaluated and runtime case could also allow to split inline and non-inline case in the same way. This makes this proposal more general than either of the individual feature proposals.

8 Summary

The proposed new syntax avoids a number of misunderstandings by limiting the facility to function body selection. It allows to support future extensions that serve additional use cases by omitting certain bodies and defining them later. Overall we think this to be a better approach to distinguishing constant evaluation.