P3068R1: Allowing exception throwing in constant-evaluation

Date: 2023-03-18
Audience: EWG
Authors: Hana Dusíková (cpp@hanicka.net)

Motivation

Since adding the constexpr keyword in C++11, WG21 has gradually expanded the scope of language features for use in constant-evaluated code. At first users couldn't even use if, else, or loops. C++14 added them. C++17 added constexpr lambdas. C++20 finally added the ability to use allocation, std::vector and std::string. These improvements have been widely appreciated by many users, and they lead to simpler code that doesn't need to workaround differences between normal and constexpr C++.

The last major language feature from C++ still not present in constexpr code is the ability to throw an exception. This absence forces library authors to use more intrusive error reporting mechanisms. One example would be the use of std::expected, std::optional. Another one is complete omission of error handling. This leaves users with long and confusing errors generated by the compiler.

Throwing exceptions in constant evaluated code is the preferred way of error reporting in the proposal adding Static Reflection for C++26. Some meta-functions can fail and allowing them to throw will significantly simplify reflection code.

Changes

- R0 → R1: Changed wording after consultation with Jens, added example to show exception can't leak via std::exception_ptr

Proposed changes in wording

7.7 Constant expressions [expr.const]

(5.25) a throw-expression ([expr.throw]), unless the lifetimes of the exception object and any implicit copies of exception object created by calls to std::current_exception() or std::rethrow_exception() end within the evaluation of E

Implementation experience

None in a C++ compiler. The author implemented this in simple AST walking scripting language which is how constexpr code is evaluated in most of C++ compilers.

The implementation strategy is usually registering a handler for specific exception types when try and catch blocks are found and unregistering when the try block is left.

Impact on existing code

This change shouldn't break any existing code as throwing exceptions without catching them is already an error and is used by various libraries to improve compile-time errors.

The intent is to keep this useful mechanism intact. The proposed wording change will only modify behavior in cases where there is try/catch block present.

Only language changes in this paper

Intention of this proposal is to allow throwing any type of exception as long as it can be constructed in constant evaluated context. We don't propose making a magic type of a constexpr exception type to be inherited from.

This paper doesn't propose any library changes, but we think another paper should mark helper functions to be constexpr, namely: std::current_exception, std::uncaught_exception, and std::rethrow_exception.

1 https://wg21.link/P2996R1#error-handling-in-reflection
2 libfmt (https://github.com/fmtlib/fmt/blob/master/include/fmt/format.h#L2245)
CTHASH (https://github.com/hanickadot/cthash/blob/main/include/cthash/sha2/sha512/t.hpp#L18)
What is allowed and what's not?

Defining a new constexpr variable always creates a new constant evaluation context. try/catch blocks around such definition won't catch the exception thrown from inside of it and it will lead to a compile-time error. This behavior is similar to constexpr memory allocations, which can't leave the constexpr context.

```cpp
constexpr auto just_error() {
    throw my_exception("this is always an error");
}
```

```cpp
constexpr void foo() {
    try {
        auto v = just_error(); // OK
    } catch (my_exception) { }
    try {
        constexpr auto v = just_error(); // ERROR: constexpr variable creates new constant evaluation context
    } catch (my_exception) { }
}
```

Exceptions must be caught

```cpp
constexpr unsigned divide(unsigned n, unsigned d) {
    if (d == 0u) {
        throw std::invalid_argument("division by zero"); // if std::invalid_argument has constexpr constructor
    }
    return n / d;
}
```

```cpp
constexpr auto b = divide(5, 0); // UNCHANGED: still a compilation failure
```

```cpp
constexpr std::optional<unsigned> checked_divide(unsigned n, unsigned d) {
    try {
        return divide(n, d);
    } catch (const std::invalid_argument &)
    {
        return std::nullopt;
    }
}
```

```cpp
constexpr auto a = checked_divide(5, 0); // BEFORE: compilation failure
// AFTER: std::nullopt value
```

Constant evaluation violation behavior won't be changed

```cpp
constexpr int throw_if_odd(const unsigned* p) {
    if (*p % 2 == 1) {
        throw 0;
    } else {
        return 1;
    }
}
```

```cpp
constexpr int g() {
    try {
        return throw_if_odd(nullptr);
    } catch (...) {
        return 2;
    }
}
```

```cpp
static_assert(g() == 2); // UNCHANGED: still an error, not magically okay because dereferencing an int throws
```
Lifetime of exception object must stay inside constant evaluation

```cpp
constexpr auto fail() -> std::exception_ptr {
    try {
        throw failure{};
    } catch (...) {
        return std::current_exception();
    }
}

constexpr auto stored_exception = fail(); // NOT ALLOWED: no exception object must leak from constant evaluation
```

Special thanks