Document: **P3590R0**
Revises: **(none)**
Date: **13-Jan-2025**
Audience: **EWG, CWG**
Authors: **Daveed Vandevoorde (daveed@edg.com)**

Constexpr Coroutines Burdens

Intro

I was asked to write a brief overview of what makes "constexpr coroutines" somewhat difficult to support in current implementations (an issue that arose during the discussion of P3367 "constexpr coroutines" in CWG). This is that overview.

What's Needed

In order to implement "constexpr coroutines", a C++ front end's consteval implementation needs a way to keep track of multiple evaluations simultaneously and suspend/resume between those evaluations. That also means that local variables will have different mappings between those evaluations, and lifetime checking may have to adjust as well. In principle, there are multiple ways to achieve that (P3367R3 includes an overview), but in practice front ends currently in production are unlikely to implement any of those techniques in the near future.

The Current State Of Consteval Implementations

As far as I know, all the current C++ consteval implementations that are in production (as opposed to "experimental") recursively evaluate a graph constructed from parsing expressions and constexpr functions (that graph is often called AST — abstract syntax tree — or parse tree). My understanding is also that all current "production" implementations perform this evaluation using recursive function calls; e.g., to evaluate x + (y - z), a function like evaluate_expr is called for the top-level node (representing the "+"), and that function will recursive call evaluate_expr for the "-" node.

This approach has some significant disadvantages, including:

- It tends to be low-performance because the "nodes" are typically fairly abstract, and so the evaluator needs to do a fair bit of "decoding work" (compared to, e.g., byte code interpreters).
- For complex evaluations, it's easy to run out of space on the call stack.

Additional Constraints

Some of the C++ front ends (which include the consteval implementation) are available as "components" that are used for multiple tools. This introduces some constraints:

- The representation of the parse tree must be close to that of the source so that source analysis tools can use that representation. This, e.g., makes many transformations (e.g., of a coroutine to a set of blocks with exits/entries for suspend/resume) impractical.
- The components do not want to impose undue environmental burdens on their client code. E.g., the initial Clang-based consteval coroutine evaluator used a "fiber" framework to allow suspending/resuming the state of the evaluator. However, adding a dependency on such a framework is not actually practical for mainline Clang (according to at least two principal Clang maintainers). Similarly, one could imagine using C++20 coroutines in the implementation of the constant evaluation of coroutines, but imposing C++20 on the client code is not always an option for production tools.

An Anticipated Future

My expectation is that over the next decade or so, C++ consteval implementations will switch to something like virtual-machine-based (VM) evaluators (i.e., "byte code interpreters"), driven by the increased reliance on constant evaluation in modern code (including due to anticipated reflection facilities in C++26). Clang has already started that transition (https://clang.llvm.org/docs/ConstantInterpreter.html), but it's a project that started several years ago and hasn't completed yet.

In light of all of the above, it seems to me that "constexpr coroutines" are being proposed a little too early. If they become part of C++26, I suspect most of the principal C++ implementations will opt not to be standard compliant for the better part of a decade, while they (a) keep working on outstanding pre-C++26 language features ("modules" being most notable) and (b) prioritize newer C++26 and C++29 features that provide "more bang for the buck".

I therefore recommend we wait with standardizing "constexpr coroutines" until implementations are well underway in their transition to VM-like implementations.