Document: P3802R0
Revises: (none)
Date: 15-Jul-2025
Audience: EWG, LEWG
Authors: Daveed Vandevoorde (daveed@edg.com)

Poor Functions

Intro

C++ functions have never been limited to functions in a "mathematical sense". However, recently, we have introduced a few functions that are less "function-like" than any C++ functions before that. I'd like us to step back from that decision, and move the non-function-like behavior to a construct of its own.

To illustrate the issue, consider the original culprit: `std::source_location::current()`. It is declared as follows:

```
namespace std {
  struct source_location {
    static consteval source_location current() noexcept;
    ...
  };
}
```

This looks like a normal C++ function, which would imply that I can wrap it with my own convenience function:

```
consteval auto curr_loc() {
  return std::source_location::current();
}
```

but that doesn't work: `std::source_location::current()` is dependent not on parameters (there are none) or global state (e.g., the AST-like structure maintained by the implementation), but also on the local context in which its invocation appears. For example, if I write:

```
using srcloc = std::source_location;
consteval auto number(srcloc loc = srcloc::current()) {
  return loc.line();
}
auto r = number();
```

I cannot replace `srcloc::current()` by `curr_loc()` because `srcloc::current()` (i.e., `std::source_location::current()`) is "doubly magical":

1) ordinarily, it picks up the source-location context in which it appears, but
2) when it appears as a default argument it instead picks up the source-location context where that default argument is used.

My function `srcloc` above has no way to model either of those behaviors.

`std::source_location::current()` was the first such "function" introduced in the language, but P2996 added another one: `std::meta::access_context::current()`. It is also a function that cannot be wrapped because it sneakily passes in local context.

Furthermore, there is already talk about introducing additional "functions" of this sort. So far I've been in discussions that suggest `std::meta::current_function()` and `std::meta::current_class()`, but no doubt there are other "local context queries" that we will want to add in the future.

Proposal

Instead of hiding the dependence on local context in the *identity* of a growing set of functions, I think we should make that local context explicit and pass it into the functions under discussion as a (default) argument written as an independent construct (i.e., not a function call, or a magic variable, etc.). For example:

```
namespace std {
  struct source_location {
    static consteval
      source_location current(std::meta::info = __local_ctx) noexcept;
    ...
  };
}
```

With that declaration, I can write my `curr_loc()` function as follows:

```
consteval auto curr_loc(std::meta::info c = __local_ctx) {
  return std::source_location::current(c);
}
```

and it will have all the expected/required behaviors: The magic is no longer in the function, but in a separate construct (`__local_ctx`) that need not be bound by existing expectations.Here, `__local_ctx` is a keyword, but it could also be formulated some other way, e.g. involving punctuation: Its *raison d'etre* is to capture a unique language mechanism so that other existing mechanisms do not

have to be corrupted to shoe-horn desired semantics. In particular, __local_ctx becomes subject to the special rules for default arguments and default member initializers that were previously applicable to `source_location::current(...)` and `meta::access_context::current(...)`, and those functions no longer need those special rules themselves.

For C++26, we can let `__local_ctx` be an implementation-dependent construct. It doesn't have to produce a `std::meta::info` object *per se*, but since `std::meta::info` is already the type to represent implementation internals, it seems entirely appropriate for this use.

Eventually (after C++26), we could raise something like `__local_ctx` to a standard-syntax construct, especially if we encounter a need for many more context-sensitive queries. We can use one `__local_ctx` construct (with an alternative spelling) for each of the functions that need this behavior (in C++26 that would be `std::source_location::current(...)` and `std::meta::access_context::current(...)`), or we can use different constructs for each (e.g., `__local_ctx<"location">`, `__local_ctx<"access">`, etc.).