# N4419 - Source-Information Capture Extensions

Authors: Robert Douglas
April 8th, 2015

## Introduction

In Urbana, N4129 was discussed, generating a variety of requests for new features. This paper details some of the design and implementation implications for the various requests. The goal of this paper is solicit feedback as to how desirable various features are.

## Offset from Start of File

### Motivation

Tooling on output of logs and unit tests suffer from file/line reporting, where the tool is required to scan the entire file to determine the caret by counting line endings. Tool authors would prefer to have access to the offset from the beginning of file.

### Previous Discussions

This feature was considered harmless by all of LEWG and desirable by some, but was omitted from the initial form of std::source_context out of concern for keeping the initial feature to a minimal feature-set.

### Proposal

Add an accessor to `std::source_context`:
`constexpr int offset_from_start_of_file() const noexcept;`
*Returns:* Integer

Add a section for `current_source_context()`:
Created `source_context::offset_from_start_of_file()` shall return implementation-defined value representing the byte-offset from the start of the file.

## Implementation-Defined Source Location Identifier

Proposal is to add an accessor to the `source_location` class, to get an implementation-defined identifier for the line of the source code. This would be an alternative to requiring pieces of information that some vendors may not find useful.

## Finer-Grained Source-Context Intrinsics

Some code bases may generate very large binaries, resulting from many named functions, each using doing invariant checking which would want to use `source_location`. As such, it was noted that having the result of `__func__` generated in the binary for every single function using `source_location`, may itself be prohibitive to adoption of the feature, as the amount of data would be too large, when `function_name` is not needed.

N4129, intentionally omitted the discussion of intrinsics for later design work, as it was a larger issue than was needed to make progress, and N4129 was believed to in no way prohibit the later specification of such intrinsics.

The following list gives possible examples of finer-grained intrinsics that could be supported. That is, each grouping represents a different set of intrinsics, comma delimited. Intrinsics returning a tuple of items are denoted by {}'s.

1. {File, Line}
2. {File, Line}, Function, Column
3. {File, Line}, File, Line, Function, Column
4. {File, Line, Column}, Function
5. File, Line, Function, Column

If desired, and with guidance, a follow-up proposal will specify the intrinsics by which `source_location` could be theoretically built, or an program could piecemeal consume a subset of data.

## Additional Magic Function(s) Specifying Capture Set

```
source_location::current(File | Line | Column);
```

This proposal would define an overload of `current`, which uses tags to determine what is (and thus what is not) captured. For simplification, we may consider having one overload with a default argument.

## User-Defined Data

A request was made to allow the user to decorate, in some fashion, the information stored in `source_location`, with additional user data. Since the data is stored in the binary, itself, the user could presumably add in their own comments or other such insights.

## Pretty Function

The result of `function_name()` is presumed to be the same as `__func__` for the line captured. Many compilers also provide a utility to get a demangled version of the function name, or otherwise friendlier string for human consumption. This proposal is to define such an extension to `source_location`, that such an accessor would exist.

## Acknowledgements

Thanks to Peter Sommerlad, John Lakos, Chandler Carruth, and many other Urbana attendees for their suggestions and feedback.