

Source-Code Information Capture

Robert Douglas

2014-10-10

| | |
|------------------|--------------------------|
| Document Number: | N4129 followup to N3972 |
| Date: | 2014-10-10 |
| Project: | Programming Language C++ |

1 Introduction

Unchanged from N3972 Logging, testing, and invariant checking each produce messages containing information such as file names, line numbers, and function names. Currently, the only way to get at this information, while avoiding code duplication at each invocation, is through the use of function macros. Function macros expand to the location of use, thus allowing `__LINE__`, `__FILE__`, and `__func__` to be interpreted in the context of the callers site. If this is where the issue ended, it may not be such a big deal. Unfortunately, for each of these domains, we end up with hundreds of macros, used widely across most code bases.

1.1 Changes from N3972

Removed Section on wording to support operators and specifying behavior for `std::source_context` as a member of class. Included `column_offset_from_start_of_file()`. Included a variety of other changes, given the feedback from Rapperswil.

1.2 Straw Poll Results

For reference, straw poll results can be found here: <http://wiki.edg.com/twiki/bin/view/Wg21rapperswil2014/N>

- Straw poll: do we want this feature? 17/4/1/2/0
- Straw poll: Remove special rules for non-static data member of type `source_context`? Yes (unanimous)
- Straw poll: adding "offset from beginning of file" to v1? 2/8/12/3/0 "meh, yes"

- Straw poll: adding a pretty function name member function to v1? 1/8/8/3/2
"meh, no"
- Straw poll: add comparison operators (or equivalent) to v1? 2/4/6/8/3
- Straw poll: do we require a reference implementation of this proposal before it goes to LEWG? 0/4/6/9/1
- Straw poll: if so, do we want to see a revision of v1 before sending to LEWG? No (unanimous)
- Straw poll: Should we do the type's name bike shed here or leave it to LEWG? No (unanimous)
- Straw poll: Should we leave the delivery vehicle to LEWG? Yes (unanimous)

2 Design Examples

The following examples illustrate some of the expected usage of source information being passed to library functions.

2.1 Test-Assertions/Invariant-Checks

Unchanged from N3972

```
template<typename T>
void assert_equal(T const& l, T const& r, source_context sc = source_context()) {
    if (!(l == r)) {
        std::ostringstream os;
        os << sc.file_name() << ":" << sc.line_number()
           << ":" << sc.column()
           << ":" << sc.function_name()
           << " Error: " << l << " != " << r;
        throw std::runtime_error(os.str());
    }
}

template<template<class, class> class C, typename T, typename Allocator>
void assert_equal(
    C<T, Allocator> const& l,
    C<T, Allocator> const& r,
    source_context sc = source_context()) {
    if (l.size() != r.size()) {
        std::ostringstream os;
        os << sc.file_name() << ":" << sc.line_number()
           << ":" << sc.column()
           << ":" << sc.function_name()
           << " Error: container sizes mismatch: ("
```

```

        << l.size() << ", " << r.size() << " ";
        throw std::runtime_error(os.str());
    }

    for (typename C<T, Allocator>::const_iterator
         lit = begin(l), lEndIt = end(l), rit = begin(r);
         lit != lEndIt; ++lit, ++rit) {
        // Explicit about source information, so that
        // the information of the caller of this site is used
        assert_equal(*lit, *rit, sc);
    }
}

```

2.2 Logging

Unchanged from N3972

```

template<typename LoggerT, typename MessageT>
void log(
    Logger & l,
    LogLevel level,
    MessageT const& message,
    source_context sc = source_context()) {
    if (logger.level() >= level) {
        l << sc.file_name() << ":" << sc.line_number()
          << ":" << sc.column()
          << ":" << sc.function_name()
          << ":" << message << std::endl;
    }
}

template<typename LoggerT, typename MessageT>
void log_debug(
    Logger & l,
    MessageT const& message,
    source_context sc = source_context()) {
    log(l, LogLevel::Debug, message, sc);
}

```

2.3 Column Offset from Beginning of File

New in N4129

```

#include <iostream>
#define PI 3.1415926535898
void foo(source_context sc = source_context())
{

```

```

        cout << sc.column_offset_from_start_of_file() << std::endl;
    }
    void bar()
    {
        foo(); // prints 178, I believe. directives need to be included, so do new lines.
        May thus be different result on Windows where build tools convert line endings automtically.
        counting tabs as 1 character
    }

```

3 Proposal

3.1 Library Additions

3.1.1 Create an object `std::source_context`

```

namespace std {
    struct source_context {
        constexpr source_context() noexcept;
        constexpr size_t line_number() const noexcept;
        constexpr size_t column() const noexcept;
        constexpr size_t column_from_start_of_file() const noexcept;
        constexpr char const* file_name() const noexcept;
        constexpr char const* function_name() const noexcept;
    };
}

```

```
constexpr source_context::source_context() noexcept;
```

- ¹ *Effects:* Constructed with values corresponding to its source location. Values shall show parity with corresponding macros `__FILE__`, `__LINE__`, `__func__`. When used as a default-variable expression, (`dcl.fct.default`) the resulting values shall be as-if it were instantiated at the call-site.

[Examples:

```

void f(std::source_context a = std::source_context()) {
    std::source_context b; // values for "b" represent this line of code
}

void g() {
    f(); // f's "a" represents this line of code

    std::source_context c;
    f(c); // f's "a" gets the same values as "c", above
}

```

– end example]

```
constexpr int source_context::line_number() const noexcept;
```

2 *Returns:* Integer representing the line number of the declaration.

```
constexpr int source_context::column() const noexcept;
```

3 *Returns:* Integer representing the character offset of the declaration, from the start of the line. It is implementation-defined, how this value is generated.

```
constexpr int source_context::column_offset_from_start_of_file() const noexcept;
```

4 *Returns:* Integer representing the character offset of the declaration.

```
constexpr char const* source_context::file_name() const noexcept;
```

5 *Returns:* NTBS representing the name of the file in which it was instantiated.

```
constexpr char const* source_context::function_name() const noexcept;
```

6 *Returns:* NTBS representing the name of the function in which it was instantiated, if applicable.

3.2 Language Changes

This proposal should be implementable without any changes to the core language.

4 Design Notes

4.1 General Notes

Previous discussions considered a language feature to indicate whether the default-variable expression should be evaluated at the call-site. Upon further reflection, I believe that we can specify this feature without need to extend or amend the grammar. Rather, this proposal simply defines how `source_context` (a new feature) should be handled in terms of default-variables. It may be of worth to note that `__func__` is undefined when used as a default variable.

It is expected that `source_context` can be implemented as a pointer to an entry in a table in memory. Thus, copying should be a constant cost, regardless of the data referenced by the `source_context` object.

For both performance and correctness concerns, the data referenced by `source_context` is immutable.

Constructor overloads were omitted due to the lack of any convincing use-case for them, and the potential complexity they might otherwise impose on the class. Nothing in this proposal should forbid such extensions in the future. For instance, one might want to create a `source_context` object from a reference to a callstack and an index, should such objects be defined at some point.

All function and class names should be considered purely placeholders until such time as bike-shedding is done within the committee. Some additional names suggested through the reflector:

- `std::source_info` (original draft proposal name, changed, given feedback from reflector)
- `std::source_location`
- `std::source`
- `std::source_loc`
- `std::src_loc`
- `std::caller`
- `std::caller_location`
- `std::caller_loc`
- `std::call_location`
- `std::call_loc`
- `std::called_from`
- `std::called_by`

Author's Note: This class should not be interpreted as purely location-centric information for now and forever. Nor is it solely a means to gather information on who called you. These are both problems this class means to address, but neither are the totality of it. It is intended to be a means to capture information, of whatever form, of the actual source code at the point of instantiation of the object, and to aggregate that information into a single object which can be passed through memory inexpensively.

This class must be copyable and assignable, so that it may be moved between threads, or stored for later consumption, with minimal cost. Discussion has centered around what functions can be made constexpr. Being admittedly ignorant of the nuances of literal-types, I have taken a stab at adding these identifiers. It is my opinion that optimizing for compile-time operations is far less critical than run-time copying. Logging, in particular for real-time systems, is typically done in parallel tasks. As such, copying the data to

log is an extremely time-sensitive operation, in contrast to the subsequent formatting. Thus, if contention between the two appears evident, this proposal should be interpreted to favor the run-time performance of copying and assignment.

4.2 Known Limitations and Future Directions

Use of default-parameters alongside a parameter-pack cannot be represented by the features in this proposal. This proposal's focus is to introduce the `source_context` object, and define its behavior when initializing a default-variable, and as a class-member. This proposal does not preclude a future feature to allow some form of call-stack introspection, or other means of gathering call-site information off the runtime-stack. Thus, the language should not be hindered, either now or in the future, to getting call-site information on a function taking a parameter-pack.

A request was made for multiple versions of `function_name()`. One for the compiler preferred, one for the developer-preferred*. My lack of inclusion of this feature is purely based on my need to appeal to the more experienced in standardese, for input on how to specify such a thing, considering that this proposal seeks to avoid mandating the contents of any strings. I would love to see such a feature, but the means to specify it, elude me. *paraphrased

4.3 Notes Regarding Operators

Unchanged from N3972 A discussion on the reflector ensued regarding adding logical-comparisons, equality-comparison, and a hash-function to `source_context`. The motivation for the feature request was to allow for the object's usage within a sorted container, such as `std::map` or in an unsorted-container such as `std::unordered_map`. This request was driven by a desire to develop analytics, such as tracking the number of times a function is called from one site or another.

It is my strong opinion that there is no natural ordering for `std::source_context`. It was observed that within related discussions, opinions varied as to what it would mean to compare two lines of code as "equal." Also, to give a definition for sorting behavior, would necessitate defining how the component strings should be formatted. With no such guarantees, there can be no guarantee that two implementations will agree on how many times a single call-site actually calls a particular function. Considering also, that there is no known implementation of sorting aggregated source-code information, the feature has been intentionally omitted from this proposal.

It should be noted that this proposal does not prohibit such a feature from being added later. If such a proposal would be offered, I strongly suggest that it be driven by some implementation experience.

It is the author's opinion that, should these operators be considered, then future designs should prefer solutions which provide standalone comparators and hash-functors, supporting at least:

- Comparison based on just `file_name`, to facilitate grouping by calls from anywhere in a file
- Comparison based on `file_name` first and `line_number` second to facilitate what some may consider a "natural ordering"
- Comparison based on `file_name` first, `line number` second, and `column third`, a clause stating new standards may extend it with information like "compile time/date", for the finest-grained groupings
- Comparison based on just `function_name` to facilitate grouping by calls from a particular function
- Comparison based on `function_name` first, and `line_number` second, as a means to accomplish the same as `file_name-then-line_number`, above, except where file paths or changed header names create problems with identification of equivalent call-sites

4.4 Open Questions

- What would a feature-testing name look like?

5 Acknowledgements

I want to thank all the members of SG7 and the reflector community, who have provided feedback on the direction and wording of this proposal.

A special thanks to Nevin Liber, Faisal Vali, Jay Miller, Jason Smith, Richard Berlint, the rest of my coworkers at KCG, and the Informal C++ Chicago Group, for their support and input on this topic.

And especially, a I want to acknowledge and thank KCG for their support in my involvement with PL22.16.