Source-Code Information Capture

Robert Douglas

2014-05-17

Document Number:	N3972
Date:	2014-05-17
Project:	Programming Language C++

1 Introduction

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.

2 Background

In Issaquah, I presented the idea of "Call-Site Informed Default Variables" to the reflection study group, to get a sense of what direction to pursue. The goal was to craft a feature to allow a function to capture information about the source code calling it, moving toward obsoleting function-macros for these use cases. The response from the room was largely positive, and requests were focused on the desire to present a means to convey source code information to be passed to a function. This proposal seeks to do just that.

3 Design Examples

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

3.1 Test-Assertions/Invariant-Checks

```
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);
        }
}
```

3.2 Logging

```
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);
}
```

3.3 operators

The reflector brought up an issue of how to handle passing `std::source_context`s to functions with a set number of parameters, such as operators. To address the issue, this proposal includes language to make sure that a wrapper object can be crafted for a type, to gather `source_context` information on its construction. It should be noted, however, that such an object is not part of this proposal. The following example is intended for purely illustrative purposes.

```
template<typename OperableT>
struct wrapper_thing {
        wrapper_thing(OperableT that) : m_that(that) {}

        OperableT getWrapped() const { return m_that; }

        std::source_context m_source_context;
        OperableT m_that;
};

struct SpecialInt {
        explicit SpecialInt(int val) : m_value(val) {}

        // std::source_context of wrapper_thing is constructed in terms of the call-site
        SpecialInt operator+(wrapper_thing<SpecialInt const&> wrappedObj){
                if (m_value == wrappedObj.getWrapped().m_value){
                        std::ostringstream os;
                        auto const& sc = wrappedObj.m_source_context;
                        os << sc.file_name() << ":" << sc.line_number()
                                << ":" << sc.function_name()
                                << " Error: values matched here, but "
                                        "for some strange reason, should not";
                        throw std::runtime_error(os.str());
                }
                return SpecialInt(m_value + wrappedObj.getWrapped().m_value);
        }

        int m_value;
```

```
};
```

4 Proposal

4.1 Library Additions

4.1.1 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 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. When used as a default-variable expression, (dcl.fct.default) the resulting values shall be as-if it were instantiated at the call-site. When used as a member of a class, if no initialization is given in the constructor initialization-list, then the value shall be as-if it were constructed at the point where the constructor is called. [*Examples:*

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

struct Container {
        Container() {}
        Container(std::source_context given) : ctx(give) {}
        std::source_context ctx;
};

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

        Container container; // container.ctx is constructed as of this line, file, and function_name
        Container container2(c); // container2.ctx is constructed with the values of "c", above
}
```

– *end example*]

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

2 *Returns:* Integer representing the line number of the declaration. It is implementation-defined, how this value is generated, except that the value captured shall be modified by #line in the same manner as for __LINE__

```
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 implmentation-defined, how this value is generated.

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

4 *Returns:* NTBS representing the name of the file in which it was instantiated. It is implementation-defined, how this value is generated, except that the value captured shall be modified by #line in the same manner as for __FILE__

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

5 *Returns:* NTBS representing the name of the function in which it was instantiated, if applicable. It is implementation-defined, how this value is generated, except that it shall have the same representation as __func__ for the same site.

4.2 Language Changes

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

5 Design Notes

5.1 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.

5.2 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.

There was a request to add a columnar-offset field, representing characters since the start of the file. Pending arbitration with `#line`, I have chosen to leave this feature off of this proposal, for now.

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

5.3 Notes Regarding Operators

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

5.4 Open Questions

- Is it necessary to specify behavior for usage in unions?
- Can we mandate parity between `source_context`'s functions and the corresponding macros/__func__? Do we need to, or is QoI enough?
- What would the implementability be of columnar-offset-in-file? How might it work with #line?
- What would a feature-testing name look like?

6 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.