# N4418 - Parameter Stringization

Authors: Robert Douglas
April 8th, 2015

## Background

Macros provide 3 features which normal functions do not, that are useful for logging/contracts/unit-tests:

1. Access to filename, line number, and function name of the call site
2. Concatenation of compile time strings
3. Converting the expressions used in the parameter list to strings

With N4129/N4417 (Source-Code Information Capture) in-flight on addressing the first feature for regular functions, this paper seeks to explore the design space for addressing the third. In doing so, this paper will also discuss aspects of the various designs which could allow these 2 features to be used in conjunction with each other. The second feature is out of scope, but will hopefully be addressed in separate papers. (N4121) (N4236)

## Design Considerations

All designs here presume that the usage of reflection must be visible in the function signature. This constraint allows the caller of a function to know that aspects of their code may leave their control. Since they may be unable to see the implementation of the code they are calling, they must ultimately have the ability to in some way obfuscate the reflection information, or opt to not call that code.

Note: As this is intended to open discussion, little-to-no consideration has been done for bikeshed purposes. As such, language elements like operators/labels/etc are placeheld with '@' or something similarly controversial.

There are 2 use cases considered for each:

1. Caller is directly calling the function which will ultimately consume the strings.
2. Caller is calling a function which will forward the strings onward to another forwarder, or to the function that consumes the strings.

### In-Signature, Piecewise Capture

```
template<typename T, typename S>
void assert_equal(T const& left, S const& right, char const* leftAsString = @left, char
const* rightAsString = @right);
```

Here we allow the user the flexibility to optionally pass in their own strings. The default parameters grab the specified argument as text. This makes the forwarding-call version trivial, providing an insertion point for forwarded strings.

Obviously, the verbosity has the + and -'s of verbosity: More typing, more (coerced) control.

An alternative version of In-Signature Piecewise Capture:

```
template<typename T, typename S>
void assert_equal(T @ const& left, S @ const& right)
{
    std::cout << @left; // prints the stringized version of left
    std::cout << left; // prints the value of left
}
```

Here, the markup is part of the qualifiers of the type. This involves less typing, but does not allow the user to specify their own replacements for the parameter strings. As such, code bases which want to allow for obfuscation, would need a second overload without the reflection. This also would be compatible with variadic templates, which is a known shortcoming of using default variables for capture. It requires more magic in the body of the function, though.

## In-Signature Complete-Capture

If we are not concerned with picking and choosing which parameters we capture as strings, rather just whether or not to capture, then we can get a tuple of strings. This could look like:

### As parameter:

Direct Call:

```
template<typename T, typename S>
void assert_equal(T const& left, S const& right, std::reflection_stuff reflect =
std::reflection_stuff())
{
    auto leftAsString = reflect.param(0).as_string();
}
```

Forwarding call:

```
template<typename T, typename S>
void assert_vecs_equal(std::vector<T> const& left, std::vector<S> const& right,
std::reflection_stuff reflect = std::reflection_stuff())
{
   assert_equal(left.size(), right.size(), reflect);
}
```

This introduces a new type, similar to source_context from N4129. Note that parameters are selected at runtime. The type reflection_stuff is not templated on the number of parameters.

### As part of function:

Direct Call:

```
template<typename T, typename S>
void assert_equal(T const& left, S const& right) reflect
{
```

```
    auto leftAsString = assert_equal.param<0>().as_string();
}
```

Forwarding Call:

```
template<typename T, typename S>
void assert_vecs_equal(std::vector<T> const& left, std::vector<S> const& right) reflect
{
    assert_vecs_equal.forward_to(assert_equal, left.size(), right.size());
}
```

Forwarding is not well thought out, here. It does not address how to choose what information is or is not forwarded. Further work would be needed to generate more ideas on how to handle forwarding.

## Integration as part of Complete-Capture

If willing to allow for complete-capture, we can use this to capture both parameter information and std::source_context as defined in N4129. This also leaves us an expansion point in the future, to capture additional information consistently.

```
template<typename T, typename S>
void assert_equal(T const& left, S const& right) reflect
{
    if (!(left == right))
    {
        std::cout
            << current_time().to_string()
            << assert_equal.source_context().file_name() << ":"
            << assert_equal.source_context().line_number() << ":"
            << assert_equal.source_context().function_name() << " Failed: "
            << assert_equal.param<0>().c_str() << '(' << left << ") != "
            << assert_equal.param<1>().c_str() << '(' << right << ')';
    }
}
```

Questions / Suggested Straw Polls:

1. We want ability to capture as string, call-site expressions used in parameters.
    a. We want to be able to piece-wise capture these strings.
        i. We like markup of the parameter to stringize
        ii. We like specifying stringized parameters and their relation to the corresponding parameter to stringize
    b. We want to be able to capture all these strings in one expression. (capture N strings with 1 declaration)
        i. We want that capture to also capture source context

## Acknowledgements

Thanks to Jay Miller, Jason Smith, Alex Kondratskiy, and the rest of KCG for their input and support.