Consolidation of External Tooling Commands

Document number:	P1325R0
Date:	2018-10-26
Audience:	SG15

Reply-To:	Dan Kalowsky

Introduction

A new [[tooling]] attribute is proposed. This attribute will serve to provide a unified language mechanism for communicating with external C++ tooling.

Motivation and Scope

This proposal uses the term `external tooling` to indicate an application that parses C++ source code to walk an Abstract Syntax Tree (AST) to conduct some transformation or analysis other than an Immediate Representation (IR).

Currently, there exists an ad-hoc collection of control mechanisms that form around most code-analysis tools. Replacing that ad-hoc nature and providing a uniform mechanism for communication will bring the following benefits:

- create a reserved namespace for all external tooling to utilize that clearly defines where tooling control mechanisms can be placed
- create a reserved namespace for all external tooling to utilize that clearly defines what tooling control mechanisms can be acted upon
- removal of any variations based upon compiler selection such that the requested action will be supported across all compilers (even if the external tooling does not in its current state)
- removal of any possible confusion towards the action happening for beginners to the language
- have the ability to clearly search an entire codebase for sections currently disabling external tooling

A vast industry exists around external tooling within the C++ language, with many different providers all with the goal of helping to develop error-free code. Taking an incomplete tour via an internet search results in several available. With choices from free to commercial, a developer has plenty of choices, including (but not limited to) the following:

- cppcheck (http://cppcheck.sourceforge.net/)
- clang-tidy (http://clang.llvm.org/extra/clang-tidy/)
- clang-format (https://clang.llvm.org/docs/ClangFormat.html)
- Coverity (https://scan.coverity.com/)
- oclint (http://oclint.org/)
- Klockwork (http://www.klocwork.com/products-services/klocwork)
- PVS-Studio (https://www.viva64.com/en/pvs-studio/)
- Bullseye (https://www.bullseye.com/)

There are even community created documents that consolidate lists of tools with the goal of helping developers create better code[1].

An external tool provides some level of functionality, with some trade-offs of complexity, coverage, and analysis speed. There exists in every external tooling, a custom mechanism(s) to disable or suppress the flagging of items of interest or false positives. Often, but not always, that mechanism is inline with the code itself.

Many software projects employ multiple versions of these tools, creating a precarious situation when updating tools, searching for disabled/suppressed lines, and even trying to add an external tooling control block.

There are currently four identified mechanisms for control:

- Structured comments
- Compiler specific attribute markers
- Preprocessor macros
- C++ custom attributes

Ensuring that any and all external tooling is setup properly will always be a practice in massaging the source code. In cases where code needs to work with multiple compilers, the process can become an exercise in code mangling. All this effort moves a developer from worrying about writing clear, clean, and concise code into worrying how non-standard undocumented functionality will interact with the C++ language and each compiler.

Structured Comments

Most tools provide some form of a structured comment mechanism to conduct inline external tooling control. The comment format is a collection of ad-hoc tooling dependent notations placed within a comment block for the code. For example clang-tidy disables all checks on a per line basis with `// NOLINT`:

```
for (int i = 0; i < N; ++i) // NOLINT
  cout << arr[i];
```

These inline communication mechanisms can become more complicated, by providing details towards specific features, functions, or checks to disable with varying levels of clarity. For example the following line will disable the clang-tidy specific check modernize-loop-convert only:

```
for (int i = 0; i < N; ++i) // NOLINT(modernize-loop-convert)
  cout << arr[i];
```

Other external tools also have the similar narrowing behavior. For examples:

- Coverity disables variable dereferencing with `// coverity[var_deref_op]`
- PVS-Studio disables variable dereferencing with `//-V522`

Compiler-specific Attribute Markers

Some tools implement their control mechanisms through compiler specific extensions. OCLint uses GCC's `__attribute__` command to suppress falsely flagged items. For example, an unused local variable warning is suppressed by adding:

```
__attribute__((annotate("oclint:suppress[unused local variable]")))
```

Preprocessor Macros

Another portion of tools implement their own preprocessor macro to communicate commands and control to their tooling. For example, to exclude a fragment of code from analysis with a preprocessor macro a developer would do the following:

```
  #if !defined(EXTERNAL_TOOLING)
  // Some longer code section here
  // that is to be ignored by external
  // tooling.
  #endif // !defined(EXTERNAL_TOOLING)
```

Other tools utilize preprocessor macros a little differently. For example, to exclude a line of code some tools use `pragma` commands. A developer would setup the compiler `pragma` operative like so:

```
#pragma EXTERNAL_TOOLING_PRAGMA ignore
if (p != nullptr) {
  // do something interesting
}
```

C++ Attributes

Some tools have already embraced the use of the C++ attributes. For example, the CppCoreGuidelines[3] also establishes tools that "implement these rules shall respect the following syntax to explicitly suppress a rule:

```
[[gsl::suppress(tag)]]
```

Impact On the Standard

This proposal has minimal impact on the current standard as it proposes a new attribute which does not change program semantics.

As of C++17 there are 6 standard attributes in the C++ language. They are:

- [[noreturn]]
- [[carries_dependency]]
- [[deprecated]]
- [[unused]]
- [[nodiscard]]
- [[fallthrough]]

The proposed format is: `[[tooling::$action("$tooling::$tag")]]`

attribute [[tooling]]

The proposed tooling attribute is essentially a namespace for actions directed towards any external tooling processes. The `$action` is a defined subset of behaviors that an external tool must take when encountering the line.

The value of `$tool` represents the name of a specific external tool that the attribute is directing an action towards. For example, `$tool` can be `cppcheck` or `coverity`, which directs any commands afterwards are to be parsed and acted upon by that specific tool.

The value of `$action` represents the specific behavior to take by the external tool.

1. The attribute can be used to alter external tool functionality for sections of code when used on a standalone line and defined as the `$action`. Proposed options for `action` are:
    1. `enable` - Informs external tooling to start any and all processing from this point forward.
        - Example: `[[tooling::enable("clang-format")]]`
        - Example: `[[tooling::enable("clang-format pvs-studio")]]`
    2. `disable` - Informs external tooling to stop any and all processing from this point forward.
        - Example: `[[tooling::disable("pvs-studio")]]`
        - Example: `[[tooling::disable("pvs-studio clang-tidy")]]`
    3. Combining the `$action` `enable` or `disable` functionality with an empty `$tool` value, creates a mechanism to communicate to all ISO C++ compliant external tools parsing over a section of code.
        - Example: `[[tooling::disable()]]`
2. The optional `$action` attribute is used to mark various names, entities, and expression statements that cause an external tool to flag a line. Proposed options for `action` attribute are:
    1. `suppress` - Informs external tooling to disable a specific action(s) on the line, based upon a code provided by the external tooling vendor.

- Example: `[[tooling::suppress("clang-tidy::google-explicit-constructor")]]`
- Example with multiple entries: `[[tooling::suppress("coverity::var_deref_op pvs-studio::-v522")]]`

3. The attribute may be applied to the declaration of a class, a typedefname, a variable, a nonstatic data member, a function, an enumeration, a template specialization, or a non-null expression statement.

Design Considerations

This paper is not attempting to define what are and what are not valid control codes for external tooling. The focus must be on how to communicate with the already established and future control codes to external tooling.

For an attribute solution to work universally, there are a few requirements:

- To work within the current grammar of attributes[2], all external tooling must parse through the string representation of their name and search for "scoping" to define the operations.
- Attribute placement needs to be standardized across compilers. For example, clang does not (4.x) support attributes on constexpr function return statements, while MSVC (2017) has difficulty with attributes in templates with SFINAE.

Why a new attribute?

With the introduction of attributes in C++, there exists the ability to define within the language a mechanism to unify a way to communicate with external tooling and still provide proper compiler aware behaviors. There is some precedent for the use of attributes to control external tooling.

A majority of external tools use a command mechanism based off of a C or C++ comment block. This was historically a best practice to provide a control mechanism that would not impact any compiler, as a comment was ignored during the parsing process. This results in code modifications that are unable to be validated by a standard compiler. Further complicating the process, many software projects adopt a maximum column width per line of code, forcing the introduction of further heuristics to the control mechanisms to indicate a line lower or above that the behavior should apply to.

The use of preprocessor macros and compiler specific attribute extensions, unnecessarily increases the complexity of reading the code for any human. When multiple tools are utilized, a series of convoluted logic jumps may exist at the tooling intersections, that is easily broken and not realized until much later. For example, it is possible with multiple tools to create a code sample similar to:

```
#if !defined(EXTERNAL_TOOLING_1)
  // code line 1
  // code line 2
#if !defined(EXTERNAL_TOOLING_2)
  // code line 3
#endif // !defined(EXTERNAL_TOOLING_2)
#endif // !defined(EXTERNAL_TOOLING_1)
#if !defined(EXTERNAL_TOOLING_2)
  // code line 4
#endif // !defined(EXTERNAL_TOOLING_2)
```

With the proposed tooling attribute this sample would be changed to:

```
[[tooling:disable("EXTERNAL_TOOLING_1")]]
  // code line 1
  // code line 2
[[tooling:disable("EXTERNAL_TOOLING_2")]]
  // code line 3
[[tooling:enable("EXTERNAL_TOOLING_1")]]
```

```
  // code line 4
[[tooling:enable("EXTERNAL_TOOLING_2")]]
```

The use of the compiler `pragma` operators introduces other challenges. A compiler configured to issue warnings on unknown pragmas will now encounter the external tooling line and throw a warning. The unknown pragma warning is enabled in both GCC and clang as part of their `-Wall` configuration, and Visual Studio as part of the Level 1 warning series.

- clang/gcc : `warning: unknown pragma ignored [-Wunknown-pragmas]`
- Visual Studio : `warning C4068: unknown pragma`

When a project is configured to build with warnings as errors, this pragma will now generate a compile error. This can be solved with the use of another preprocessor definition to obscure the line from other external tools. For example, Bullseye entries can look like:

```
#if defined(EXTERNAL_TOOLING_PRAGMA)
  #pragma EXTERNAL_TOOLING_PRAGMA ignore
#endif
  if (p != nullptr) {
    // do something interesting
  }
```

Even with the explicit attribute based ruling there exist some challenges where different compilers do not completely support the same format, resulting in variations to accomplish the same behavior with the attribute. For example, the GSL supports the more popular C++ compilers, which disagree on implemntation support for the attribute syntax:

- MSVC understands `[[gsl::suppress(tag.x)]]`
- clang understands `[[gsl::suppress("tag.x")]]`

A macro based solution to this problem is highlighted throughout Microsoft's GSL implementation. See the gsl_assert[4] file as an example. This solution will only grow more complex as other C++ compilers are supported.

Even with this attribute precident, many compilers issue a warning or error (depending upon configuration) when encountering an unknown attribute. The inclusion of a standards specific attribute to communicate to external tooling would remove conflicts on all flavors of compilers, while still providing the same benefits already established by efforts like the CppCoreGuidelines.

Open Questions

1. Given earlier statements about some code bases having set a hard requirement on line length, should the `$action` list also include a `suppress_next`?

2. Given earlier statements about some code bases having set a hard requirement on line length, should the `$action` list also include a `suppress_last`?

Technical Specifications

9.11.N tooling attribute [dcl.attr.tooling]

The attribute-token tooling specifies a control mechanism aimed at external tooling.

```
 tooling-attribute-specifier:
   [[ tooling::action_opt ]]
```

The attribute may be applied to all statements and empty lines.

Example: Implementation where the return result of a function is explicitly ignored:

```
foo() [[tooling::suppress("clang-tidy::bugprone-unused-return-value")]];
```

Example: Implementation to suppress a warning on a case of using else statements after a return.

```
if (foo()) {
  do_something();
  return;
} else { [[tooling::suppress("clang-tidy::readability-else-after-return")]]
  do_something_else();
  return;
}
```

Acknowledgements

This proposal would not be complete without acknowledging contributions from Dalton Woodard, Anna Gringauze, and various members of the C++ community who helped by answering questions and providing input.

References

- [1] A sampling of some sites:
  - https://github.com/mre/awesome-static-analysis
  - https://github.com/fffaraz/awesome-cpp#static-code-analysis
- [2] http://eel.is/c++draft/dcl.attr#:attribute
- [3] http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#inforce-enforcement
- [4] https://github.com/Microsoft/GSL/blob/1995e86d1ad70519465374fb4876c6ef7c9f8c61/include/gsl/gsl_assert#L27

Revision History