Proposal: Modern Macros for C++

Table of Contents

Introduction

This proposal introduces a new feature called Modern Macros to the C++ programming language. Modern Macros aim to enhance the macro capabilities and enable static reflections in C++ by allowing macros to access and process syntax data and access the AST data at compile time. The proposal outlines the basic functionality, including compiler-internal functions and library functions, to achieve this goal.

Motivation

The current macro system in C++ is limited and lacks the ability to access the Abstract Syntax Tree (AST) and make meaningful decisions based on tokens passed to it during compilation. Modern Macros provide a more powerful and flexible macro system that allows developers to write more complex and safe compile-time code transformations, write their own static reflections and enforce custom rules on the code.

Design Overview

Modern Macros are invoked using the following syntax:

```
macro macro_function{syntax being passed to macro};
```

In this example, `macro_function` is a `constexpr` function to which the tokens inside the brackets are passed as character array template parameters. Each character array will include one token. The `macro_function`'s template arguments should be constructible from the array of characters that will be passed to it inside its template when invoked as a macro.

Example:

```
template <size_t N>
struct fixed_string
{
    constexpr static std::size_t size = N;
    char data[N] = {0};

    constexpr fixed_string() = default;
    constexpr fixed_string(fixed_string const &Other) { std::copy_n(Other.data, N, data); }
    constexpr fixed_string(char const (&str)[N + 1]) { std::copy_n(str, N, data); }
    constexpr auto operator[](std::size_t i) const { return data[i]; }
};

template <>
struct fixed_string<0>
{
    constexpr static std::size_t size = 0;
    char *data = nullptr;

    constexpr fixed_string() = default;
    constexpr fixed_string(fixed_string const &) = default;
    constexpr fixed_string(char const (&)[1]) {}
};

template<fixed_string... Tokens>
constexpr auto quote() {
    return quoteImpl<Tokens...>::result.data;
}

macro quote{hi there};
```

The `quote` function will be called like:

```
quote<"hi", "there">();
```

The result of the macro call will replace the macro invocation in the code.

Compiler-Internal Functions

To support the Modern Macros, a minimal set of compiler-internal functions are proposed. These functions provide basic access to the AST and the current declaration context in order to make decisions based on data included in that. The return types of these functions are `void*` to ensure safety and abstract the implementation details.

Retrieving AST Context

- `constexpr void* __get_ast_context();`
  - Returns the root context of the AST for the current compilation unit.

Node Navigation

- `constexpr void* __get_first_child(void* node);`
  - Returns the first child of the given AST node.
- `constexpr void* __get_next_sibling(void* node);`
  - Returns the next sibling of the given AST node.

Node Properties

- `constexpr const char* __get_node_name(void* node);`
  - Returns the name of the given AST node.
- `constexpr const char* __get_node_type(void* node);`
  - Returns the type of the given AST node.

Current Declaration Context

- `constexpr void* __get_current_declaration_context();`
  - Returns the AST node representing the current declaration context.

Retrieving Declarations in the Current Context

- `constexpr void* __get_first_declaration_in_context(void* context);`
  - Returns the first declaration node in the current declaration context.
- `constexpr void* __get_next_declaration(void* declaration);`
  - Returns the next declaration node in the current declaration context.

Library Functions

The following library functions are built on top of the compiler-internal functions to provide more complex and user-friendly functionality.

Node Navigation and Counting

```
namespace AST {
    constexpr size_t getNodeChildrenCount(void* node);
    constexpr void* findNodeByName(void* node, const char* name);
}
```

Querying Properties

```
namespace AST {
    constexpr bool nodeHasChildWithType(void* node, const char* type);
    constexpr bool nodeHasChildWithName(void* node, const char* name);
}
```

Complex Queries

```
namespace MyASTUtils {
    constexpr bool structHasStringMember(void* structNode);
}
```

Finding Declarations

```
namespace AST {
    constexpr void* findDeclarationByName(void* context, const char* name);
    constexpr bool contextHasDeclarationWithName(const char* name);
    constexpr bool contextHasDeclarationWithType(const char* type);
}
```

Examples

Keep in mind that these examples are not accurate and are just a demonstration of how the library functions might be used and how it would look like to enforce custom rules which uses static reflections.

Example 1: No Pointers in Context

```
template<fixed_string... Tokens>
constexpr auto noPointersInContext() {
    static_assert(!AST::contextHasDeclarationWithType("pointer"), "Pointers are not allowed in the current context.");

    return "valid_code_snippet";
}

// Invocation
macro noPointersInContext{ some_code };
```

Example 2: Structs Without String Members

```
template<String... Tokens>
constexpr auto noStructsWithStringMembers() {
    void* context = __get_current_declaration_context();

    for (void* decl = __get_first_declaration_in_context(context); decl; decl = __get_next_declaration(decl)) {
        if (std::strcmp(__get_node_type(decl), "struct") == 0 && MyASTUtils::structHasStringMember(decl)) {
            static_assert(false, "Structs with string members are not allowed.");
        }
    }

    return "valid_code_snippet";
}

// Invocation
macro noStructsWithStringMembers{ some_code };
```

Conclusion

The Modern Macros feature introduces aims to solve two problems with a single solution. By adding the modern and safe macro to C++, we will finally get passed using the old macro system which is unsafe, not-complete and enable a whole wide range of new possible programs and lay the ground work for the developer to define and design their own reflection and syntax rules.