P3676R0: Enhanced `inline` Keyword with Configurable Inlining Levels

Author: Stephen Berry, Khalil Estell
Date: 2025-4-17
Audience: C++ Standards Committee

Introduction

The C++ language provides the `inline` keyword as a hint to the compiler to inline a function, but it does not guarantee that the function will be inlined. Modern performance-critical applications and libraries often need more explicit control over inlining behavior. Currently, developers must rely on compiler-specific attributes or pragmas to enforce inlining (`__forceinline`, `__attribute__((always_inline))`, `[[clang::always_inline]]` etc.), which harms portability and leads to conditional compilation based on the target compiler. Compilers also have compiler-specific attributes for avoiding inlining code (`__declspec(noinline)`, `__attribute__((noinline))`), and this proposal would standardize the no-inline syntax to `inline(0)` or `inline(std::noinline)`.

This proposal aims to enhance the `inline` keyword to accept a parameter that communicates the programmer's desired inlining behavior:

- `inline(0)` or `inline(false)` would mean "noinline" (i.e., a request to not inline the function).
- `inline(1)` or `inline(true)` would retain the current behavior, serving as a hint rather than a strict requirement. `inline` without parentheses would be equivalent, maintaining backwards compatibility.
- `inline(2)` would strongly indicate "always inline" – instructing the compiler to inline the function whenever possible.

To improve clarity and self-documentation of code, the C++ standard library can introduce named integers (e.g., `std::noinline`, `std::normal_inline`, and `std::always_inline`) so that developers can write `inline(std::always_inline)` instead of raw integers. However, raw integers work well with template meta-programming.

Examples

Performance-Critical Code:

```
inline(2) int multiply(int a, int b) {
    return a * b;
}
```

This enforces that `multiply` is always inlined for maximum performance.

Toggling Inlining Modes:

```
template <int Mode>
inline(Mode) int heavy_function(int x) {
    return complex_calculation(x);
}

int forced = heavy_function<2>(10);    // strong request to always inline
int optional = heavy_function<1>(10);  // normal inline hint
int none = heavy_function<0>(10);      // request no inlining
```

Using std Names for Clarity:

```
inline(std::always_inline) int add(int a, int b) {
    return a + b;
}

inline(std::noinline) int slow_function(int x) {
    // Some complex logic we don't want inlined
    return complex_calculation(x);
}
```

Motivation

1. **Portability and Standardization:** Current solutions rely on non-standard attributes. By providing a standardized set of inlining modes through the `inline` keyword, code becomes more portable and less reliant on compiler-specific extensions.
2. **Explicit Control Over Inlining:** Developers who require guaranteed inlining for performance-critical sections can use `inline(2)`, while those who want to prevent inlining can use `inline(0)`. This provides developers with direct control and removes guesswork and reliance on the compiler's heuristics.
3. **Compile-Time Configuration:** By allowing a `constexpr` integral value for the inlining mode, developers can conditionally choose inlining behavior at compile time without resorting to macros or multiple function definitions. Libraries that depend upon template meta-programming currently have no way to conditionally enable `noinline` or `always_inline` attributes.
4. **Consistency and Familiarity:** The existing `inline` keyword's behavior remains intact and behaves the same with `inline(1)`. With `inline(0)` and `inline(2)`, we simply extend the existing concept in a manner akin to how `noexcept` can take a boolean. The extension is intuitive, backward compatible, and uses established language features.

Macros cannot be exported with C++20 modules. This poses a serious issue with always inline declarations, because there is no cross-platform solution without macros. Hence, it is more difficult and requires more ugly code to write performant C++20 modules.

Motivating Library Development Experience

The author of this paper develops the C++ Glaze library. Performance improvements from always inline code are often 10% - 30% faster for reading and writing JSON. However, build times can be significantly affected (MSVC build times can increase ten-fold) by always inlining. This feature would be extremely helpful to provide compile time options for users to either choose peak performance or faster compilation times and smaller binaries. The desire is for developers to be able to opt into and out of peak performance where desired.

```
// This proposal would allow performance options that might take longer to build
write_json<opts{.peak_performance = true}>(...);
```

Consider a function to serialize an integer into a character buffer. If code is serializing large arrays of integers then we typically want to inline this function to avoid the function call overhead. But, this often results in significantly more binary across the codebase and probably doesn't need to be inlined everywhere in the code (some use cases might only be serializing a single integer). If we can selectively turn on and off the force inlining of a function, then we can choose to only force inline where it is necessary, and thus avoid the extra binary and compilation costs of inlining this serializaing function everywhere in the codebase.

Template Interations with Specifiers

This proposal allows `inline` arguments to use template parameters in the same manner as `noexcept`.

The code below showcases the current valid C++ mechanism for changing `noexcect` behavior based on template parameters.

```
// Template function with a boolean non-type template parameter
// that controls whether the function is noexcept
template <bool IsNoexcept>
void may_throw_function() noexcept(IsNoexcept) {
    if (!IsNoexcept) {
        throw std::runtime_error("Exception thrown");
    }
    std::cout << "Executed without throwing\n";
}
```

Proposed Changes

Syntax

The `inline` keyword is extended to accept an integer parameter describing the inlining mode:

- `inline(0)` — This requests that the compiler not inline the function (similar to `[[noinline]]` attributes in some compilers).
- `inline(1)` — This is the default behavior of `inline` as we know it today, serving as a hint rather than a guarantee.
- `inline(2)` — This requests that the compiler attempt to always inline the function, making non-inlining scenarios exceptional and potentially warranting diagnostics.

Additionally, the standard library may provide named integers such as:

```
namespace std {
  constexpr int noinline = 0;
  constexpr int normal_inline = 1;
  constexpr int always_inline = 2;
}
```

Developers could then write:

```
inline(std::always_inline) int add(int a, int b) {
    return a + b;
}
```

Semantics

1. `inline(0)` **(No Inline):** This mode requests the compiler not to inline the function, effectively negating any other inlining requests. It aligns with some compilers' `noinline` attributes. The compiler can still decide to inline if mandated by other rules (unlikely in practice), but this mode strongly suggests that no inlining should occur.
2. `inline(1)` **(Normal Inline):** This remains unchanged from the current meaning of `inline` – a suggestion (not a demand) to the compiler that inlining may be beneficial.
3. `inline(2)` **(Always Inline):** The compiler is instructed to inline the function at every call site where possible. If it cannot inline the function (due to technical limitations like recursion, address-taking, or linker constraints), the compiler should be encouraged to emit a diagnostic. This behavior is similar to non-standard `__forceinline` or `__attribute__((always_inline))`.
4. **Compile-Time Toggling:** Much like `noexcept(expr)`, we can write:

   ```
   constexpr int mode = 2;

   inline(mode) int critical_function(int x, int y) {
       return x * y;
   }
   ```

   Changing `mode` changes the inlining strategy without modifying the function's body or resorting to macros.

Diagnostics

If a function specified as `inline(2)` cannot be inlined, compilers are encouraged (though not required) to emit a warning.

Backward and Forward Compatibility

- **Backward Compatibility:** Existing code using `inline` without arguments or just `inline` keyword behaves as `inline(1)`. There is no breaking change to existing code.
- **Forward Compatibility:** A feature-test macro (e.g., `__cpp_inline_modes`) can be introduced to allow libraries and codebases to conditionally use this feature.

Relationship with Existing Mechanisms

- **`inline` vs. `inline(2)`:**
  `inline` without parameters remains a suggestion. `inline(2)` elevates this to a requirement for the compiler to inline the function when possible, turning what was once a weak hint into a strong directive.
- **No Need for Compiler-Specific Attributes:**
  Standardizing an inlining mode removes the need for `__attribute__((always_inline))`, `__forceinline`, or other vendor-specific methods.

Consideration of inline for Linkage

The `inline` keyword is used for linkage control to avoid ODR violations. If In these header scenarios `inline(0)` should behave in the same terms of linkage as the current `inline`. ODR violations should be prevented, but the compiler should take this as a request to call the function and not insert (inline) the code at the call site.

Global inline Variables

Global inline variables must also respect the inline arguments.

```
// A request to directly embed the table rather than access via a memory lookup
inline(2) constexpr std::array<int8_t, 4> table{ 5, 6, 7, 8 };
```

A global lambda's inline arguments must apply to the `operator()()` call.

```
// A request to always inline the caller's contents wherever invoked
inline(std::always_inline) constexpr auto caller = []{
  // some logic
};
```

Compiler Extensions?

All positive integers are reserved by the standard for the inline argument. Negative integers may be used by compiler vendors to add experimental inlining features (e.g. `inline(-5)`). Implementations may error on any integer arguments other than 0, 1, and 2, and should produce warnings for invalid inputs.

How Often Is Always Inline Used?

Almost every C++ library that is found on most popular lists uses always inline macros (just search the codebase for `always_inline`). A small sample of popular libraries that use always inline macros:

- bitcoin (82K+ stars) `ALWAYS_INLINE` macro
- godot (95K+ stars) `_ALWAYS_INLINE_` macro
- llama.cpp (77K+ stars) `ALWAYS_INLINE` macro
- opencv (81K+ stars) `CV_ALWAYS_INLINE` macro
- react-native (121K+ stars) `RCTREQUIRED_INLINE` macro
- tensorflow (189K+ stars) `EIGEN_ALWAYS_INLINE` macro
- terminal (97K+ stars) `__attribute__((always_inline))` attribute

Typically the macro is like that of llama.cpp:

```
#if (defined(_WIN32) || defined(_WIN64))
#define ALWAYS_INLINE __forceinline
#elif __has_attribute(always_inline) || defined(__GNUC__)
#define ALWAYS_INLINE __attribute__((__always_inline__)) inline
#else
```

Comparison with Attribute Approach

While `[[always_inline]]` and `[[noinline]]` attributes could be standardized, the proposed `inline(N)` syntax offers several advantages. It is more consistent with the current use of `inline`. It allows compile time customization through template parameters like `noexcept`, which is critical to achieve full inlining control. And, it is more natural syntax, being similar to `noexcept` syntax.

Conclusion

Enhancing the `inline` keyword to accept an integral inlining mode (0, 1, 2) provides a portable, standardized, and expressive way to control function inlining. This change preserves backward compatibility, aligns with existing language design patterns, and eliminates the need for non-standard compiler-specific attributes.

By adopting this proposal, developers gain improved portability, clearer intent, and the ability to fine-tune their code's performance characteristics without resorting to macros or vendor lock-in.