constexpr for `std::chrono::system_clock`

Document #:	P0000R0
Date:	2025-02-25
Project:	Programming Language C++
Audience:	Library Evolution Working Group (LEWG)
Reply-to:	Nikolaos D. Bougalis <nikb@bougalis.net>

1 Revision History

Rev	Date	Changes
P0000R0	2025-02-25	Initial version

2 Introduction

This paper proposes introducing a compile-time usable clock by making `std::chrono::system_clock::now()`, `std::chrono::system_clock::from_time_t()`, and `std::chrono::system_clock::to_time_t()` callable in constant evaluation contexts by marking them `constexpr`. This would allow make it possible for developers to use the familiar, expressive and safe `std::chrono` toolkit to measure durations, generate `time_points` and `durations` and make it possible to implement functionality that is otherwise not available without the use of third-party tools and non-portable solutions.

3 Motivation and Scope

3.1 Motivating Examples

3.1.1 Time-Based UUIDs

One use case would be for generating v1 or v7 UUIDs at compile time. Doing this now means either generating UUIDs using a tool (e.g. Microsoft's `uuidgen.exe`) manually or as part of the build process, or writing macros that use the C preprocessor to attempt to generate UUIDs during the preprocessing pass.

With this proposal, the following would be possible:

```
constexpr uuid_t generate_uuid(int version, std::chonoe::system_clock::time_point tp)
{
    // ... implementation omitted
}

constexpr auto uuid1 = generate_uuid(1, std::chrono::system_clock::now());
constexpr auto uuid2 = generate_uuid(2, std::chrono::system_clock::now());
```

3.1.2 Additional Use Cases

— Automatic time-based seeding of compile-time random number generators
— Compile-time generation of build timestamps for versioning
— Simplified testing of time-dependent code
— Troubleshooting/debugging compile-time code bottlenecks

3.2 Impact On the Standard

This proposal suggests a pure library extension that does not break existing code. It adds `constexpr` to existing functions without changing their behavior in runtime contexts.

3.2.1 Core Language Impact

This proposal does not require core language changes. The existing constant evaluation rules adequately support the proposed functionality.

3.2.2 Library Impact

The changes are limited to adding `constexpr` to the declarations of three `std::chrono::system_clock` functions.

4 Design Decisions

4.1 Semantics of Compile-Time Evaluation

When evaluated during compilation, `system_clock::now()` will return the time point corresponding to the moment of constant evaluation. This aligns with the existing semantics where `now()` returns the current time when called.

4.2 Portability Considerations

Different compilers may evaluate the function at different times in the compilation process, leading to different timestamp values. This is acceptable since:

— The primary use case is obtaining a deterministic value for a specific build
— This is consistent with current `constexpr` functions that may depend on compilation environment factors
— The exact time is not important

4.3 Implementation Considerations

Implementations would need to provide a mechanism for the compiler to determine the current system time during constant evaluation. This would likely take the form of a built-in or intrinsic function: a function that is declared by the compiler and implemented in a special way by the compiler itself. The only difference would be that this special function would, itself, be `constexpr` (i.e. evaluable at compile time).

4.4 Alternatives Considered

4.4.1 New Function Instead of Modifying Existing One

We considered adding a new function such as `constexpr_now()` instead of adding `constexpr` to the existing function. This was rejected because:

— It creates unnecessary API duplication
— The existing function name clearly expresses the intent
— Making the existing function `constexpr` follows the pattern used for other functions in the standard library

5 Technical Specifications

The functions `std::chrono::system_clock::now`, `std::chrono::system_clock::from_time_t` and `std::chrono::system_clock::to_time_t` shall be callable in constant evaluation contexts.

```
namespace std::chrono {
  class system_clock {
  public:
    // ...
    static constexpr time_point now() noexcept;

    static constexpr std::time_t to_time_t(const time_point& t) noexcept;
    static constexpr time_point from_time_t(std::time_t t) noexcept;
    // ...
  };
}
```

[*Note:* For brevity and clarity, only the declaration of `system_clock` is modified in this document to mark functions as `constexpr`, but all the functions so marked must be similarly marked as `constexpr` in any detailed descriptions elsewhere in the standard. — *end note*]

5.1 Effects

5.1.1 `std::chrono::system_clock::now`

When called during constant evaluation, the function shall return a `std::chrono::system_clock::time_point` representing the time at which the evaluation occurs.

When called at runtime, the function's behavior remains unchanged from the current standard.

5.1.2 `std::chrono::system_clock::to_time_t`

When called during constant evaluation, the function shall return a `std::time_t` corresponding to the specified `std::chrono::system_clock::time_point` input.

When called at runtime, the function's behavior remains unchanged from the current standard.

5.1.3 `std::chrono::system_clock::from_time_t`

When called during constant evaluation, the function shall return a `std::chrono::system_clock::time_point` corresponding to the specified `std::time_t` input.

When called at runtime, the function's behavior remains unchanged from the current standard.

6 Comparison with Existing Alternatives

6.1 Limitations of `__DATE__` and `__TIME__` Macros

While some might suggest that `__DATE__` and `__TIME__` preprocessor macros already provide equivalent compile-time timestamp functionality, these macros have significant limitations that make them inadequate replacements for a `constexpr std::chrono::system_clock::now()`:

— **Likely impossible to implement a `Clock` using `__DATE__` and `__TIME__`:** Using the `__DATE__` and `__TIME__` to implement a `Clock` is likely impossible. This proposal would, automatically, make such a `Clock` available on every standard-compliant platform.

[*Note:* The author believes that it is *actually* impossible to implement a `Clock` using the `__DATE__` and `__TIME__` macros but concedes that there are people whose knowledge of ancient runes and arcane words of power is much greater and who may be able to create a `Clock` but questions whether such madness should be undertaken. — *end note*]

— **Timezone Ambiguity**: The timezone in which `__DATE__` and `__TIME__` macros are specified is not well-defined and differs from platform to platform. This inconsistency makes it difficult to reliably construct a `std::chrono::time_point` from these values.

— **String Format Limitations**: The macros expand to string literals requiring parsing:

  — `__DATE__` expands to a string literal in the form "Mmm DD YYYY"
  — `__TIME__` expands to a string literal in the form "HH:MM:SS"

  Converting these strings to a proper time point requires additional parsing logic, introducing complexity and potential errors.

— **Granularity Restrictions**: The `__TIME__` macro only provides second-level granularity, whereas `std::chrono::system_clock::now()` typically offers much finer resolution.

— **Standard Library Integration**: Using macros requires converting strings to time types, whereas `constexpr std::chrono::system_clock::now()` would directly provide a properly typed `time_point` that integrates seamlessly with the rest of the chronology library.

7 Potential Concerns

7.1 Consistency Between Multiple Compilations

Different compilation runs will naturally produce different timestamps. This is expected and aligned with the function's purpose of providing the current time.

For use cases requiring consistent values across builds (e.g., deterministic builds), developers should continue using alternative approaches such as injecting timestamp values through build system variables.

7.2 Impact on Build Caching Systems

Build caching systems like ccache might be affected because identical source files could produce different object files due to embedded timestamps. However:

— This only affects code that actually uses `system_clock::now()` in constant expressions
— Caching systems already must handle time-dependent macros like `__TIME__`
— Many caching systems provide mechanisms to ignore specific time-dependent elements

8 Implementation Experience

No experimental implementations exist at this time, but it is expected that this feature can be added with reasonable effort and without negative performance impacts.

9 Conclusion

Making `std::chrono::system_clock::now`, `std::chrono::system_clock::from_time_t` and `std::chrono::system_clock`
usable in constant evaluation contexts provides useful capabilities for C++ developers while maintaining backward compatibility. The suggested change aligns with the direction of increasing `constexpr` support throughout the standard library and enables new compile-time patterns.

10 Acknowledgements

Thanks to Howard Hinnant, Alexander Popovich and Scott Determan for their encouragement and valuable input and feedback on this proposal.

11 References

[1] ISO/IEC 14882:2023 Programming Language C++ (C++23 DIS)