

# constexpr std::shared\_ptr

Document #: P3037R0  
Date: 2023-11-06  
Project: Programming Language C++  
Audience: SG7 Compile-time programming, LWG Library, LEWG Library Evolution  
Reply-to: Paul Keir  
<[graham.keir@gmail.com](mailto:graham.keir@gmail.com)>

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Motivation and Scope</b>	<b>2</b>
2.1	Atomic Operations . . . . .	2
2.2	Two Memory Allocations . . . . .	3
2.3	Relational Operators . . . . .	3
2.4	Maybe Not Now, But Soon . . . . .	4
<b>3</b>	<b>Impact on the Standard</b>	<b>4</b>
<b>4</b>	<b>Implementation</b>	<b>5</b>
<b>5</b>	<b>Proposed Wording</b>	<b>5</b>
<b>6</b>	<b>Acknowledgements</b>	<b>5</b>
<b>7</b>	<b>References</b>	<b>6</b>

# 1 Introduction

Since the adoption of [P0784R7] in C++20, constant expressions can include dynamic memory allocation; yet support for smart pointers extends only to `std::unique_ptr` (since [P2273R3] in C++23). As at runtime, smart pointers can encourage hygienic memory management during constant evaluation; and with no remaining technical obstacles, parity between runtime and compile-time support for smart pointers should duly and intuitively reflect the increased maturity of language support for constant expression evaluation. We therefore propose that `std::shared_ptr` and associated class templates from 20.3 [smartptr] permit `constexpr` evaluation.

## 2 Motivation and Scope

Two proposals adopted for C++26 and C++23 can facilitate a straightforward implementation of comprehensive `constexpr` support for `std::shared_ptr`: [P2738R1] and [P2448R2]. The former allows the `get_deleter` member function to operate, given the type erasure required within the `std::shared_ptr` unary class template. The latter can allow even minor associated classes such as `std::bad_weak_ptr` to receive `constexpr` qualification, while inheriting from the currently non-`constexpr` class: `std::exception`. We furthermore propose that the relational operators of `std::unique_ptr`, which can legally operate on pointers originating from a single allocation during constant evaluation, should also adopt the `constexpr` specifier.

As with C++23 `constexpr` support for `std::unique_ptr`, bumping the value `__cpp_lib_constexpr_memory` is our requested feature macro change; yet in the discussion and implementation presented here, we adopt the macro `__cpp_lib_constexpr_shared_ptr`. We also use the `_GLIBCXX26_CONSTEXPR` macro in place of the literal `constexpr` keyword to ensure the specifier only applies when the `-std=c++26` flag is enabled.

We below elaborate on points which go beyond the simple addition of the `constexpr` specifier to the relevant member functions.

### 2.1 Atomic Operations

`std::shared_ptr` can operate within a multithreaded runtime environment; and a number of its member functions use atomic functions to ensure that shared state is updated correctly. Constant expressions must currently be evaluated by a single thread. A `constexpr std::shared_ptr` implementation can engage with the `constexpr`-friendly support for single-threaded evaluation available in atomic function definitions within standard library implementations. For example, in `libstdc++`'s interface to atomic functions, the `__is_single_threaded` function, which controls execution of both `__exchange_and_add_dispatch` and `__atomic_add_dispatch` within the `ext/atomicity.h` header file, can be changed to start as follows:

```
_GLIBCXX26_CONSTEXPR
__attribute__((__always_inline__))
inline bool
__is_single_threaded() _GLIBCXX_NOTHROW
{
#ifdef __cpp_lib_constexpr_shared_ptr
    if (std::is_constant_evaluated())
        return true;
#endif
    // ... 7 more lines here
}
```

Built-in GCC atomic functions such as `__atomic_load_n` are also used within `libstdc++`'s implementation of `std::shared_ptr`. These could similarly be updated to account for a `constexpr` single-threaded execution environment within the compiler. The approach taken within our own implementation is a local one; eliding the call to the atomic function through the predication of `std::is_constant_evaluated` (or `if constexpr`). For example, here is an updated function from `bits/shared_ptr_base.h`, used by `std::shared_ptr::use_count()` and elsewhere:

```

_GLIBCXX26_CONSTEXPR
long
_M_get_use_count() const noexcept
{
#ifdef __cpp_lib_constexpr_shared_ptr
    return std::is_constant_evaluated()
        ? _M_use_count
        : __atomic_load_n(&_M_use_count, __ATOMIC_RELAXED);
#else
    return __atomic_load_n(&_M_use_count, __ATOMIC_RELAXED);
#endif
}

```

## 2.2 Two Memory Allocations

Unlike `std::unique_ptr`, a `std::shared_ptr` must store not only the managed object, but also the type-erased deleter and allocator, as well as the number of `std::shared_ptr`s and `std::weak_ptr`s which own or refer to the managed object. This information is managed as part of a dynamically allocated object referred to as the *control block*.

Existing runtime implementations of `std::make_shared`, `std::allocate_shared`, `std::make_shared_for_overwrite`, and `std::allocate_shared_for_overwrite`, allocate memory for both the control block, *and* the managed object, from a single dynamic memory allocation; via `reinterpret_cast`. This practise aligns with a remark at 20.3.2.2.7 [util.smartptr.shared.create]; quoted below:

- (7.1) — Implementations should perform no more than one memory allocation.  
 — [Note 1: This provides efficiency equivalent to an intrusive smart pointer. — end note]

As `reinterpret_cast` is not permitted within a constant expression, an alternative approach is required for `std::make_shared`, `std::allocate_shared`, `std::make_shared_for_overwrite`, and `std::allocate_shared_for_overwrite`. A straightforward solution is to create the object first, and pass its address to the appropriate `std::shared_ptr` constructor. Considering the control block, this approach amounts to two dynamic memory allocations; albeit at compile-time. Assuming that the runtime implementation need not change, the remark quoted above could either be removed, or changed to “Implementations should perform no more than one runtime memory allocation.”

## 2.3 Relational Operators

Comparing dynamically allocated pointers within a constant expression is legal, provided the result of the comparison is not unspecified. Such comparisons are defined in terms of a partial order, applicable to pointers which either point “to different elements of the same array, or to subobjects thereof”; or to “different non-static data members of the same object, or to subobjects of such members, recursively...”; from paragraph 4 of 7.6.9 [expr.rel]. A simple example program is shown below:

```

constexpr bool ptr_compare()
{
    int* p = new int[2]{};
    bool b = &p[0] < &p[1];
    delete [] p;
    return b;
}

static_assert(ptr_compare());

```

It is therefore unsurprising that we include the `std::shared_ptr` relational operators within the scope of our proposal to apply `constexpr` to all functions within 20.3 [smartptr]; the `std::shared_ptr` aliasing constructor

makes this especially simple to configure:

```
constexpr bool sptr_compare()
{
    double *arr = new double[2];
    std::shared_ptr p{&arr[0]}, q{p, p.get() + 1};
    return p < q;
}

static_assert(sptr_compare());
```

Furthermore, in the interests of `constexpr` consistency, we propose that the relational operators of `std::unique_ptr` *also* now include support for constant evaluation. As discussed above, the results of such comparisons are very often well defined.

It may be argued that a `std::unique_ptr` which is the sole owner of an array, or an object with data members, presents less need for relational operators. Yet we must consider that a custom deleter can easily change the operational semantics; as demonstrated in the example below. A `std::unique_ptr` should also be legally comparable with itself.

```
constexpr bool uptr_compare()
{
    short* p = new short[2]{};
    auto del = [] (short*){};
    std::unique_ptr<short[]> a{p+0};
    std::unique_ptr<short[],decltype(del)> b{p+1, del};
    return a < b;
}

static_assert(uptr_compare());
```

## 2.4 Maybe Not Now, But Soon

A core message of C++23's [P2448R2] is that the C++ community is served better by including the language version alongside the tuple of possible inputs (i.e. function and template arguments) considered for a `constexpr` function invocation within a constant expression. Consequently, while there are some functions in 20.3 [smartptr] which cannot possibly be so evaluated *today*, we propose that these should also be specified with the `constexpr` keyword. The following lists all such functions or classes:

- 20.3.2.1 [util.smartptr.weak.bad]: `std::bad_weak_ptr` cannot be constructed as it inherits from a class, `std::exception`, which has no `constexpr` member functions.
- 20.3.3 [util.smartptr.hash]: The `operator()` member of the class template specialisations for `std::hash<std::unique_ptr<T,D>>` and `std::hash<std::shared_ptr<T>>` cannot be defined according to the *Cpp17Hash* requirements (16.4.4.5 [hash.requirements]). (A pointer cannot, during constant evaluation, be converted to an `std::size_t` using `reinterpret_cast`; or otherwise.)
- 20.3.2.5 [util.smartptr.owner.hash]: The two `operator()` member functions of the recently adopted `owner_hash` class, also cannot be defined according to the *Cpp17Hash* requirements.
- 20.3.2.2.6 [util.smartptr.shared.obs]: The recently adopted `owner_hash()` member function of `std::shared_ptr`, also cannot be defined according to the *Cpp17Hash* requirements.

## 3 Impact on the Standard

This proposal is a pure library extension, and does not require any new language features.

## 4 Implementation

An implementation based on the GNU C++ Library (libstdc++) can be found [here](#). A comprehensive test suite is included there within `tests/shared_ptr_constexpr_tests.cpp`; alongside a standalone bash script to run it. All tests pass with recent GCC and Clang (i.e. versions supporting P2738; `__cpp_constexpr`  $\geq$  202306L).

## 5 Proposed Wording

## 6 Acknowledgements

Thanks to all of the following:

- (In alphabetical order by last name) Thiago Macieira, Arthur O’Dwyer, and everyone else who contributed to the online forum discussions.

## 7 References

- [P0784R7] Daveed Vandevoorde. 2019. More constexpr containers.  
<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0784r7.html>
- [P2273R3] Andreas Fertig. 2021. Making `std::unique_ptr` constexpr.  
<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2273r3.pdf>
- [P2448R2] Barry Revzin. 2022. Relaxing some `constexpr` restrictions.  
<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2448r2.html>
- [P2738R1] David Ledger. 2023. `constexpr` cast from `void*`: towards `constexpr` type-erasure.  
<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2738r1.pdf>