`constexpr std::shared_ptr`

Document #:	P3037R1
Date:	2024-03-05
Project:	Programming Language C++
Audience:	SG7 Compile-time programming, LEWG Library Evolution

Reply-to:	Paul Keir <graham.keir@gmail.com>

Contents

1 Revision History

- R1 2024-03-05
  - Added a motivating example
  - Included libc++ & MSVC STL in atomic operation considerations
- R0 2023-11-06
  - Original Proposal

2 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 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.

3 Motivation and Scope

It is convenient when the same C++ code can be deployed both at runtime and compile time. Our recent project investigates performance scaling of *parallel* constant expression evaluation in an experimental Clang compiler [ClangOz]. As well as C++17 parallel algorithms, a prototype `constexpr` implementation of the Khronos SYCL API was utilised, where a SYCL `buffer` class abstracts over device and/or host memory. In the simplified code excerpt below, the `std::shared_ptr` data member ensures memory is properly deallocated upon the `buffer`'s destruction, according to its owner status. This is a common approach for runtime code, and a `constexpr` `std::shared_ptr` class implementation helpfully bypasses thoughts of raw pointers, and preprocessor macros here; and the impact of adding `constexpr` functionality to the SYCL implementation is minimised.

```
template <class T, int dims = 1>
struct buffer
{
  constexpr buffer(const range<dims> &r)
    : range_{ r }, data_{ new T[r.size()], [this](auto* p){ delete [] p; } } { }

  constexpr buffer(T* hostData, const range<dims>& r)
    : range_{ r }, data_{ hostData, [](auto){} } { }

  const range<dims> range_{};
  std::shared_ptr<T[]> data_{};
};
```

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

3.1 Atomic Operations

The existing `std::shared_ptr` class can operate within a multithreaded runtime environment. A number of its member functions may therefore be defined using atomic functions; so ensuring that shared state is updated correctly. Atomic functions are not qualified as `constexpr`; but as constant expressions must be evaluated by a single thread, a `constexpr` `std::shared_ptr` implementation can safely skip calls to atomic functions through the predication of `std::is_constant_evaluated` (or `if consteval`). For example, here is a modified function from GCC's libstdc++, called from `std::shared_ptr::use_count()` and elsewhere:

```
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
}
```

The use of atomic intrinsics within Clang's libc++ and MSVC's STL can be similarly elided. In `__memory/shared_ptr.h`, libc++ makes calls to the atomic intrinsic `__atomic_load_n`, only via the inline C++ functions `__libcpp_relaxed_load` and `__libcpp_acquire_load`; while `__atomic_add_fetch` is accessed only via `__libcpp_atomic_refcount_increment` and `__libcpp_atomic_refcount_decrement`. Each of these four functions is comprised only of return statement pairs, predicated upon *object-like* macros including `_LIBCPP_HAS_NO_THREADS`; and so could easily be modified to involve `std::is_constant_evaluated` as above.

In `stl/inc/memory`, the `std::shared_ptr` of MSVC's STL inherits a `_Ref_count_base` member through `_Ptr_base`. `_Ref_count_base` has two `_Atomic_counter_t` members (aliases of `unsigned long`), updated atomically using the `_InterlockedCompareExchange`; `_InterlockedIncrement` (via the macro `_MT_INCR`); or `_InterlockedDecrement` (via the macro `_MT_DECR`) atomic intrinsics. All the (five) functions invoking these intrinsics can again make use of `std::is_constant_evaluated` to avoid the atomic operations.

Adding `constexpr` support to an implementation of `std::shared_ptr` built directly upon an `std::atomic` instance would need to take an alternative approach; likely involving the modification of its `std::atomic` definition.

3.2 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 can be left unchanged; as this is only a recommendation, not a requirement.

3.3 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());
```

3.4 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.

4 Impact on the Standard

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

5 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 >= 202306L`).

6 Proposed Wording

7 Acknowledgements

Thanks to all of the following:

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

8 References

[ClangOz] Paul Keir. 2024. Performance Analysis of Compiler Support for Parallel Evaluation of C++ Constant Expressions.
https://doi.org/10.1007/978-3-031-51075-5_6

[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