Contents

1 Revision History 2
2 Introduction 2
3 Motivation and Scope 2
   3.1 Atomic Operations ...................................................... 3
   3.2 Two Memory Allocations ................................................ 3
   3.3 Relational Operators ..................................................... 4
   3.4 Maybe Not Now, But Soon .............................................. 5
4 Impact on the Standard 5
5 Implementation 5
6 Proposed Wording 5
7 Acknowledgements 11
8 References 12
1 Revision History

— R2 2024-05-24
  — Added wording
  — Removed constexpr specification from some functions (see below here)
  — Removed SG7 from Audience (post 2024 Spring meeting in Tokyo)
— 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 appropriate class template member functions 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. The impact of adding constexpr functionality to the SYCL implementation is therefore minimised.

```cpp
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_{};
};
```

Adopted C++26 proposal [P2738R1] facilitates a straightforward implementation of comprehensive constexpr support for `std::shared_ptr`, allowing the `get_deleter` member function to operate, given the type erasure required within the `std::shared_ptr` unary class template. 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:

```cpp
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 elided similarly. In `__memory/shared_ptr.h`, libc++ makes calls to the atomic intrinsic `__atomic_load_n`, only via the in-line 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_DEC`) 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:

```cpp
contexpr 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:

```cpp
contexpr 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.

```cpp
contexpr 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

The functions from 20.3 [smartptr] listed below cannot possibly be evaluated within a constant expression. We do not propose that their specifications should change. While C++23’s [P2448R2] allows such functions to be annotated as constexpr, we suggest that in this instance the C++ community will be served better by a future update; when their constant evaluation becomes possible.

— 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.
— 20.3.2.3.6 [util.smartptr.weak.obs]: The recently adopted owner_hash() member function of std::weak_ptr, also cannot be defined according to the Cpp17Hash requirements.

We also do not propose any specification change for the overloads of operator<< for std::shared_ptr and std::ptr, from 20.3.2.12 [util.smartptr.shared.io] and 20.3.1.7 [unique.ptr.io]. Unlike the functions above, a constexpr implementation for the overloads could today use a vendor-specific extension; do nothing; or simply report an error. But such possibilities should be discussed in a separate proposal focused on I/O.

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 [P2738R1]; __cpp_constexpr >= 202306L).

6 Proposed Wording

The following wording changes apply to [N4981] and can also be viewed on Github via a fork of the C++ Standard Draft Sources repository here.

Add to 17.3.2 [version.syn] (Header <version> synopsis):

```
- #define __cpp_lib_constexpr_memory 202202L // freestanding, also in <memory>
+ #define __cpp_lib_constexpr_memory YYYYMML // freestanding, also in <memory>
```

Add to 20.2.2 [memory.syn] (Header <memory> synopsis):

```cpp
constexpr bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
constexpr bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
constexpr bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
constexpr bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
constexpr compare_three_way_result_t<typename unique_ptr<T1, D1>::pointer, 
constexpr shared_ptr<T> make_shared(Args&&... args);
constexpr shared_ptr<T> allocate_shared(const A& a, Args&&... args);
constexpr shared_ptr<T> make_shared(size_t N);
```

5
constexpr shared_ptr<T> make_shared();
constexpr shared_ptr<T> allocate_shared(const A& a);
constexpr shared_ptr<T> make_shared(size_t N, const remove_extent_t<T>& u);
constexpr shared_ptr<T> allocate_shared(const A& a, const remove_extent_t<T>& u);
constexpr shared_ptr<T> make_shared_for_overwrite();
constexpr shared_ptr<T> allocate_shared_for_overwrite(const A& a);
constexpr shared_ptr<T> make_shared_for_overwrite(const remove_extent_t<T>& u);
constexpr shared_ptr<T> allocate_shared_for_overwrite(const A& a, const remove_extent_t<T>& u);
constexpr shared_ptr<T> make_shared_for_overwrite();
constexpr shared_ptr<T> allocate_shared_for_overwrite(const A& a, size_t N);
constexpr shared_ptr<T> make_shared_for_overwrite(size_t N);
constexpr bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
constexpr strong_ordering operator<= (const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
constexpr bool operator==(const shared_ptr<T>& x, nullptr_t) noexcept;
constexpr strong_ordering operator<= (const shared_ptr<T>& x, nullptr_t) noexcept;
constexpr void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;
constexpr shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
constexpr shared_ptr<T> static_pointer_cast(shared_ptr<U>&& r) noexcept;
constexpr shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
constexpr shared_ptr<T> dynamic_pointer_cast(shared_ptr<U>&& r) noexcept;
constexpr shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;
constexpr shared_ptr<T> const_pointer_cast(shared_ptr<U>&& r) noexcept;
constexpr shared_ptr<T> reinterpret_pointer_cast(const shared_ptr<U>& r) noexcept;
constexpr shared_ptr<T> reinterpret_pointer_cast(shared_ptr<U>&& r) noexcept;
constexpr D* get_deleter(const shared_ptr<T>& p) noexcept;

template<class T> constexpr void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;
constexpr auto out_ptr(Smart& s, Args&&... args);
constexpr auto inout_ptr(Smart& s, Args&&... args);

Add to 20.3.1.6 [unique.ptr.special] (Specialized algorithms):

constexpr bool operator< (const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
constexpr bool operator> (const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
constexpr bool operator<= (const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
constexpr bool operator>= (const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
constexpr compare_three_way_result_t<typename unique_ptr<T1, D1>::pointer,

Add to 20.3.2.2.1 [util.smartptr.shared.general] (General):

constexpr explicit shared_ptr(Y* p);
constexpr shared_ptr(Y* p, D d);
constexpr shared_ptr(Y* p, D d, A a);
constexpr shared_ptr(nullptr_t p, D d);
constexpr shared_ptr(nullptr_t p, D d, A a);
template<class Y> constexpr shared_ptr(const shared_ptr<Y>& r, element_type* p) noexcept;
template<class Y> constexpr shared_ptr(shared_ptr<Y>&& r, element_type* p) noexcept;
constexpr shared_ptr(const shared_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
constexpr shared_ptr(const shared_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
constexpr shared_ptr(const shared_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
constexpr shared_ptr(const shared_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
constexpr shared_ptr(const weak_ptr<Y>& r);
constexpr shared_ptr(unique_ptr<Y, D>&& r);
constexpr -shared_ptr();
constexpr shared_ptr& operator=(const shared_ptr& r) noexcept;
template<class Y> constexpr shared_ptr& operator=(const shared_ptr<Y>&& r) noexcept;
constexpr shared_ptr& operator=(shared_ptr& r) noexcept;
template<class Y> constexpr shared_ptr& operator=(shared_ptr<Y>&& r) noexcept;
constexpr shared_ptr& operator=(unique_ptr<Y, D>&& r);
constexpr void swap(shared_ptr& r) noexcept;
constexpr void reset() noexcept;
constexpr void reset(Y* p);
constexpr void reset(Y* p, D d);
constexpr void reset(Y* p, D d, A a);
constexpr element_type* get() const noexcept;
constexpr T& operator*() const noexcept;
constexpr T* operator->() const noexcept;
constexpr element_type& operator[](ptrdiff_t i) const;
constexpr long use_count() const noexcept;
constexpr explicit operator bool() const noexcept;

Add to 20.3.2.2.2 [util.smartptr.shared.const] (Constructors):

template<class Y> constexpr explicit shared_ptr(Y* p);
template<class Y, class D> constexpr shared_ptr(Y* p, D d);
template<class Y, class D, class A> constexpr shared_ptr(Y* p, D d, A a);
template<class D> constexpr shared_ptr(nullptr_t p, D d);
template<class D, class A> constexpr shared_ptr(nullptr_t p, D d, A a);
template<class Y> constexpr shared_ptr(const shared_ptr<Y>& r, element_type* p) noexcept;
template<class Y> constexpr shared_ptr(const shared_ptr<Y>& r) noexcept;
template<class Y, class D> constexpr shared_ptr(const shared_ptr<Y>& r) noexcept;
template<class Y> constexpr shared_ptr(shared_ptr<Y>&& r) noexcept;
template<class Y> constexpr shared_ptr(const shared_ptr& r) noexcept;
template<class Y> constexpr shared_ptr(shared_ptr r) noexcept;
template<class Y> constexpr shared_ptr(nullptr_t p, D d, A a);

Add to 20.3.2.2.3 [util.smartptr.shared.dest] (Destructor):

constexpr ~shared_ptr();

Add to 20.3.2.2.4 [util.smartptr.shared.assign] (Assignment):

constexpr shared_ptr& operator=(const shared_ptr<Y>&& r) noexcept;
template<class Y> constexpr shared_ptr& operator=(const shared_ptr<Y>&& r) noexcept;
constexpr shared_ptr& operator=(shared_ptr& r) noexcept;
template<class Y> constexpr shared_ptr& operator=(shared_ptr<Y>&& r) noexcept;
template<class Y> constexpr shared_ptr& operator=(unique_ptr<Y, D>&& r);

Add to 20.3.2.2.5 [util.smartptr.shared.mod] (Modifiers):

constexpr void swap(shared_ptr& r) noexcept;
constexpr void reset() noexcept;
template<class Y> constexpr void reset(Y* p);
template<class Y, class D> constexpr void reset(Y* p, D d);
template<class Y, class D, class A> constexpr void reset(Y* p, D d, A a);

Add to 20.3.2.2.6 [util.smartptr.shared.obs] (Observers):

Add to 20.3.2.2.5 [util.smartptr.shared.mod] (Modifiers):

constexpr void swap(shared_ptr& r) noexcept;
constexpr void reset() noexcept;
template<class Y> constexpr void reset(Y* p);
template<class Y, class D> constexpr void reset(Y* p, D d);
template<class Y, class D, class A> constexpr void reset(Y* p, D d, A a);
constexpr element_type* get() const noexcept;
constexpr T& operator*() const noexcept;
constexpr T* operator->() const noexcept;
constexpr element_type& operator[](ptrdiff_t i) const;
constexpr long use_count() const noexcept;
constexpr explicit operator bool() const noexcept;

Add to 20.3.2.2.7 [util.smartptr.shared.create] (Creation):
constexpr shared_ptr<T> make_shared(argc);
constexpr shared_ptr<T> allocate_shared(const A& a, argc);
constexpr shared_ptr<T> make_shared_for_overwrite(argc);
constexpr shared_ptr<T> allocate_shared_for_overwrite(const A& a, argc);
constexpr shared_ptr<T> make_shared(const Args&&... args);
constexpr shared_ptr<T> allocate_shared(const A& a, Args&&... args);
constexpr make_shared(size_t N);
constexpr shared_ptr<T> allocate_shared(const A& a, size_t N);
constexpr shared_ptr<T> make_shared();
constexpr shared_ptr<T> allocate_shared(const A& a);
constexpr shared_ptr<T> make_shared(size_t N, const remove_extent_t<T>& u);
constexpr shared_ptr<T> allocate_shared(size_t N, const remove_extent_t<T>& u);
constexpr shared_ptr<T> make_shared_for_overwrite();
constexpr shared_ptr<T> allocate_shared_for_overwrite(const A& a);
constexpr shared_ptr<T> make_shared_for_overwrite(size_t N);
constexpr shared_ptr<T> allocate_shared_for_overwrite(const A& a, size_t N);

Add to 20.3.2.2.8 [util.smartptr.shared.cmp] (Comparison):
constexpr bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) const noexcept;
template<class T> constexpr bool operator==(const shared_ptr<T>& a, nullptr_t) const noexcept;
constexpr strong_ordering operator<=>(const shared_ptr<T>& a, const shared_ptr<U>& b) const noexcept;
constexpr strong_ordering operator<=>(const shared_ptr<T>& a, nullptr_t) const noexcept;

Add to 20.3.2.2.9 [util.smartptr.shared.spec] (Specialized algorithms):
constexpr void swap(shared_ptr<T>& a, shared_ptr<T>& b) const noexcept;

Add to 20.3.2.2.10 [util.smartptr.shared.cast] (Casts):
constexpr shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) const noexcept;
constexpr shared_ptr<T> static_pointer_cast(shared_ptr<U>& r) const noexcept;
constexpr shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) const noexcept;
constexpr shared_ptr<T> dynamic_pointer_cast(shared_ptr<U>&& r) const noexcept;
constexpr shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) const noexcept;
constexpr shared_ptr<T> const_pointer_cast(shared_ptr<U>& r) const noexcept;
constexpr shared_ptr<T> reinterpret_pointer_cast(const shared_ptr<U>& r) const noexcept;
constexpr shared_ptr<T> reinterpret_pointer_cast(shared_ptr<U>&& r) const noexcept;

Add to 20.3.2.2.11 [util.smartptr.getdeleter] (get_deleter):
Add to 20.3.2.3.1 [util.smartptr.weak.general] (General):

```cpp
constexpr weak_ptr(const shared_ptr<Y>& r) noexcept;
constexpr weak_ptr(const weak_ptr& r) noexcept;
constexpr weak_ptr(const weak_ptr<Y>& r) noexcept;
constexpr weak_ptr(weak_ptr&& r) noexcept;
constexpr weak_ptr(weak_ptr<Y>&& r) noexcept;
constexpr ~weak_ptr();
constexpr weak_ptr& operator=(const weak_ptr& r) noexcept;
constexpr weak_ptr& operator=(const weak_ptr<Y>& r) noexcept;
constexpr weak_ptr& operator=(const shared_ptr<Y>& r) noexcept;
constexpr weak_ptr& operator=(weak_ptr&& r) noexcept;
constexpr weak_ptr& operator=(weak_ptr<Y>&& r) noexcept;
```

Add to 20.3.2.3.3 [util.smartptr.weak.assign] (Assignment):

```cpp
constexpr weak_ptr& operator=(const weak_ptr& r) noexcept;
template<class Y> constexpr weak_ptr& operator=(const weak_ptr<Y>& r) noexcept;
template<class Y> constexpr weak_ptr& operator=(const shared_ptr<Y>& r) noexcept;
constexpr weak_ptr& operator=(weak_ptr&& r) noexcept;
template<class Y> constexpr weak_ptr& operator=(weak_ptr<Y>&& r) noexcept;
```

Add to 20.3.2.3.5 [util.smartptr.weak.mod] (Modifiers):

```cpp
constexpr void swap(weak_ptr& r) noexcept;
constexpr void reset() noexcept;
```

Add to 20.3.2.3.6 [util.smartptr.weak.obs] (Observers):

```cpp
constexpr long use_count() const noexcept;
constexpr bool expired() const noexcept;
constexpr shared_ptr<T> lock() const noexcept;
template<class U> constexpr bool owner_before(const shared_ptr<U>& b) const noexcept;
template<class U> constexpr bool owner_before(const weak_ptr<U>& b) const noexcept;
constexpr bool owner_equal(const shared_ptr<U>& b) const noexcept;
```
Add to 20.3.2.7 [util.smartptr.weak.spec] (Specialized algorithms):

```cpp
constexpr bool owner_equal(const weak_ptr<U>& b) const noexcept;
```

Add to 20.3.2.4 [util.smartptr.ownerless] (Class template owner_less):

```cpp
constexpr bool operator()(const shared_ptr<T>&, const shared_ptr<T>&) const noexcept;
constexpr bool operator()(const shared_ptr<T>&, const weak_ptr<T>&) const noexcept;
constexpr bool operator()(const weak_ptr<T>&, const shared_ptr<T>&) const noexcept;
constexpr bool operator()(const weak_ptr<T>&, const weak_ptr<T>&) const noexcept;
constexpr bool operator()(const shared_ptr<T>&, const weak_ptr<U>&) const noexcept;
constexpr bool operator()(const weak_ptr<T>&, const shared_ptr<U>&) const noexcept;
constexpr bool operator()(const weak_ptr<T>&, const weak_ptr<U>&) const noexcept;
```

Add to 20.3.2.6 [util.smartptr.owner.equal] (Struct owner_equal):

```cpp
constexpr bool operator()(const shared_ptr<T>&, const shared_ptr<U>&) const noexcept;
constexpr bool operator()(const shared_ptr<T>&, const weak_ptr<U>&) const noexcept;
constexpr bool operator()(const weak_ptr<T>&, const shared_ptr<U>&) const noexcept;
constexpr bool operator()(const weak_ptr<T>&, const weak_ptr<U>&) const noexcept;
```

Add to 20.3.2.7 [util.smartptr.enab] (Class template enable_shared_from_this):

```cpp
constexpr enable_shared_from_this(const enable_shared_from_this&) noexcept;
constexpr ~enable_shared_from_this();
constexpr operator Pointer*() const noexcept;
constexpr operator void**() const noexcept;
constexpr explicit out_ptr_t(Smart&, Args...);
constexpr ~out_ptr_t();
constexpr operator Pointer*() const noexcept;
constexpr operator void**() const noexcept;
```

Add to 20.3.4.1 [out.ptr.t] (Class template out_ptr_t):

```cpp
constexpr explicit out_ptr_t(T& smart, Args... args);
```
Add to 20.3.4.2 [out.ptr] (Function template out_ptr):

```cpp
constexpr auto out_ptr(Smart& s, Args&&... args);
```

Add to 20.3.4.3 [inout.ptr.t] (Class template inout_ptr_t):

```cpp
constexpr explicit inout_ptr_t(Smart&, Args...);
constexpr ~inout_ptr_t();
constexpr operator Pointer*() const noexcept;
constexpr operator void**() const noexcept;
constexpr explicit inout_ptr_t(Smart& smart, Args... args);
constexpr ~inout_ptr_t();
constexpr operator Pointer*() const noexcept;
constexpr operator void**() const noexcept;
```

Add to 20.3.4.4 [inout.ptr] (Function template inout_ptr):

```cpp
constexpr auto inout_ptr(Smart& s, Args&&... args);
```

## 7 Acknowledgements

Thanks to all of the following:

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

https://doi.org/10.1007/978-3-031-51075-5_6

https://wg21.link/n4981

https://wg21.link/p0784

https://wg21.link/p2273

https://wg21.link/p2448

https://wg21.link/p2738