`constexpr std::shared_ptr` and friends

Document #:	P3037R5
Date:	2025-03-27
Project:	Programming Language C++
Audience:	Library Working Group (LWG)

Reply-to:	Paul Keir <paul.keir@uws.ac.uk> Hana Dusíková <cpp@hanicka.net>

Contents

1 Revision History

- R5 2025-03-27
  - Changed proposal title for clarity based on LEWG feedback in Hagenberg
  - Included support for partial specialisations of `std::atomic` for `std::shared_ptr` and `std::weak_ptr`
  - Added full context for wording changes (Section 6)
- R4 2024-10-21
  - Added `constexpr` qualifier to the wording for `std::bad_weak_ptr` with [P3068R6]
  - Updated ClangOz reference in motivation
- R3 2024-09-03
  - Removed `constexpr` specification from `reinterpret_pointer_cast` (Section 3.4)
  - Added references to [P3309R3] and [P3068R6]
  - Added details of a second implementation based on libc++
- R2 2024-05-24
  - Added wording
  - Removed `constexpr` specification from some functions (Section 3.4)
  - 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 classes and template member functions from [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 implementation is thus 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_{};
};
```

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. In earlier revisions of this paper, standard atomic functions were not qualified as `constexpr`. Yet `constexpr` implementations of [smartptr] classes and functions could be implemented, through recognising that as constant expressions must be evaluated by a single thread, execution 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
}
```

Today, the adoption of [P3309R3] has enabled `constexpr` functionality in `std::atomic` (and `std::atomic_ref`) for C++26; and so code such as that above will soon be unnecessary. Yet support for the partial specialisations of `std::atomic` for `std::shared_ptr` and `std::weak_ptr` were not included in [P3309R3]. During discussions on that paper, it was proposed that `atomic<shared_ptr>` should be supported in `constexpr` whenever `shared_ptr` is supported in `constexpr` (whichever paper lands second should have this change). We consequent here (the second paper) propose `constexpr` support for the partial specialisations of `std::atomic` for `std::shared_ptr` and `std::weak_ptr`, as specified in [util.smartptr.atomic.shared] and [util.smartptr.atomic.weak].

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 [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 [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 [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

The functions from [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.

- [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 ([hash.requirements]). (A pointer cannot, during constant evaluation, be converted to an `std::size_t` using `reinterpret_cast`; or otherwise.)
- [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.
- [util.smartptr.shared.obs]: The recently adopted `owner_hash()` member function of `std::shared_ptr`, also cannot be defined according to the *Cpp17Hash* requirements.
- [util.smartptr.weak.obs]: The recently adopted `owner_hash()` member function of `std::weak_ptr`, also cannot be defined according to the *Cpp17Hash* requirements.
- [util.smartptr.shared.cast]: Neither of the two `reinterpret_pointer_cast` overloads can be included as their implementations will typically call `reinterpret_cast`, which is prohibited here.

We also *do not* propose any specification change for the overloads of `operator«` for `std::shared_ptr` and `std::unique_ptr`, from [util.smartptr.shared.io] and [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 by the first author, 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`).

A second implementation, by the second author, based on the "libc++" C++ Library is also available: on Github here (via commit 23217d0); and with a corresponding Compiler Explorer instance here.

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.

Pages from the *C++ Standard Draft Sources* which include wording changes proposed here are also provided on the following pages; with changes highlighted.

```
#define __cpp_lib_bind_front                        202306L // freestanding, also in <functional>
#define __cpp_lib_bit_cast                          201806L // freestanding, also in <bit>
#define __cpp_lib_bitops                            201907L // freestanding, also in <bit>
#define __cpp_lib_bitset                            202306L // also in <bitset>
#define __cpp_lib_bool_constant                     201505L // freestanding, also in <type_traits>
#define __cpp_lib_bounded_array_traits              201902L // freestanding, also in <type_traits>
#define __cpp_lib_boyer_moore_searcher              201603L // also in <functional>
#define __cpp_lib_byte                              201603L // freestanding, also in <cstddef>
#define __cpp_lib_byteswap                          202110L // freestanding, also in <bit>
#define __cpp_lib_char8_t                           201907L
  // freestanding, also in <atomic>, <filesystem>, <istream>, <limits>, <locale>, <ostream>, <string>,
  // <string_view>
#define __cpp_lib_chrono                            202306L // also in <chrono>
#define __cpp_lib_chrono_udls                       201304L // also in <chrono>
#define __cpp_lib_clamp                             201603L // also in <algorithm>
#define __cpp_lib_common_reference                  202302L // freestanding, also in <type_traits>
#define __cpp_lib_common_reference_wrapper          202302L // freestanding, also in <functional>
#define __cpp_lib_complex_udls                      201309L // also in <complex>
#define __cpp_lib_concepts                          202207L
  // freestanding, also in <concepts>, <compare>
#define __cpp_lib_constexpr_algorithms              202306L // also in <algorithm>, <utility>
#define __cpp_lib_constexpr_atomic                  202411L // also in <atomic>
#define __cpp_lib_constexpr_bitset                  202207L // also in <bitset>
#define __cpp_lib_constexpr_charconv                202207L // freestanding, also in <charconv>
#define __cpp_lib_constexpr_cmath                   202306L // also in <cmath>, <cstdlib>
#define __cpp_lib_constexpr_complex                 202306L // also in <complex>
#define __cpp_lib_constexpr_deque                   202502L // also in <deque>
#define __cpp_lib_constexpr_dynamic_alloc           201907L // also in <memory>
#define __cpp_lib_constexpr_exceptions              202502L
  // also in <exception>, <stdexcept>, <expected>, <optional>, <variant>, and <format>
#define __cpp_lib_constexpr_flat_map                202502L // also in <flat_map>
#define __cpp_lib_constexpr_flat_set                202502L // also in <flat_set>
#define __cpp_lib_constexpr_forward_list            202502L // also in <forward_list>
#define __cpp_lib_constexpr_functional              201907L // freestanding, also in <functional>
#define __cpp_lib_constexpr_inplace_vector          202502L // also in <inplace_vector>
#define __cpp_lib_constexpr_iterator                201811L // freestanding, also in <iterator>
#define __cpp_lib_constexpr_list                    202502L // also in <list>
#define __cpp_lib_constexpr_map                     202502L // also in <map>
#define __cpp_lib_constexpr_memory                  202202LYYYYMML // freestanding, also in <memory>
#define __cpp_lib_constexpr_new                     202406L // freestanding, also in <new>
#define __cpp_lib_constexpr_numeric                 201911L // also in <numeric>
#define __cpp_lib_constexpr_queue                   202502L // also in <queue>
#define __cpp_lib_constexpr_set                     202502L // also in <set>
#define __cpp_lib_constexpr_stack                   202502L // also in <stack>
#define __cpp_lib_constexpr_string                  201907L // also in <string>
#define __cpp_lib_constexpr_string_view             201811L // also in <string_view>
#define __cpp_lib_constexpr_tuple                   201811L // freestanding, also in <tuple>
#define __cpp_lib_constexpr_typeinfo                202106L // freestanding, also in <typeinfo>
#define __cpp_lib_constexpr_unordered_map           202502L // also in <unordered_map>
#define __cpp_lib_constexpr_unordered_set           202502L // also in <unordered_set>
#define __cpp_lib_constexpr_utility                 201811L // freestanding, also in <utility>
#define __cpp_lib_constexpr_vector                  201907L // also in <vector>
#define __cpp_lib_constrained_equality              202411L // freestanding,
  // also in <utility>, <tuple>, <optional>, <variant>, <expected>
#define __cpp_lib_containers_ranges                 202202L
  // also in <vector>, <list>, <forward_list>, <map>, <set>, <unordered_map>, <unordered_set>,
  // <deque>, <queue>, <stack>, <string>
#define __cpp_lib_contracts                         202502L // freestanding, also in <contracts>
#define __cpp_lib_copyable_function                 202306L // also in <functional>
#define __cpp_lib_coroutine                         201902L // freestanding, also in <coroutine>
#define __cpp_lib_debugging                         202403L // freestanding, also in <debugging>
#define __cpp_lib_destroying_delete                 201806L // freestanding, also in <new>
#define __cpp_lib_enable_shared_from_this           201603L // also in <memory>
#define __cpp_lib_endian                            201907L // freestanding, also in <bit>
```

```
  template<nothrow-input-range R>
    requires destructible<range_value_t<R>>
      constexpr borrowed_iterator_t<R> destroy(R&& r) noexcept;                    // freestanding

  template<nothrow-input-iterator I>
    requires destructible<iter_value_t<I>>
      constexpr I destroy_n(I first, iter_difference_t<I> n) noexcept;             // freestanding
}

// 20.3.1, class template unique_ptr
template<class T> struct default_delete;                                           // freestanding
template<class T> struct default_delete<T[]>;                                      // freestanding
template<class T, class D = default_delete<T>> class unique_ptr;                   // freestanding
template<class T, class D> class unique_ptr<T[], D>;                               // freestanding

template<class T, class... Args>
  constexpr unique_ptr<T> make_unique(Args&&... args);                             // T is not array
template<class T>
  constexpr unique_ptr<T> make_unique(size_t n);                                   // T is U[]
template<class T, class... Args>
  unspecified make_unique(Args&&...) = delete;                                     // T is U[N]

template<class T>
  constexpr unique_ptr<T> make_unique_for_overwrite();                             // T is not array
template<class T>
  constexpr unique_ptr<T> make_unique_for_overwrite(size_t n);                     // T is U[]
template<class T, class... Args>
  unspecified make_unique_for_overwrite(Args&&...) = delete;                       // T is U[N]

template<class T, class D>
  constexpr void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y) noexcept;          // freestanding

template<class T1, class D1, class T2, class D2>
  constexpr bool operator==(const unique_ptr<T1, D1>& x,                           // freestanding
                            const unique_ptr<T2, D2>& y);
template<class T1, class D1, class T2, class D2>
  constexpr bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);        // free-
standing
template<class T1, class D1, class T2, class D2>
  constexpr bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);        // free-
standing
template<class T1, class D1, class T2, class D2>
  constexpr bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);       // free-
standing
template<class T1, class D1, class T2, class D2>
  constexpr bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);       // free-
standing
template<class T1, class D1, class T2, class D2>
  requires three_way_comparable_with<typename unique_ptr<T1, D1>::pointer,
                                     typename unique_ptr<T2, D2>::pointer>
  constexpr compare_three_way_result_t<typename unique_ptr<T1, D1>::pointer,
                               typename unique_ptr<T2, D2>::pointer>
    operator<=>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);         // freestanding

template<class T, class D>
  constexpr bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;        // freestanding
template<class T, class D>
  constexpr bool operator<(const unique_ptr<T, D>& x, nullptr_t);                  // freestanding
template<class T, class D>
  constexpr bool operator<(nullptr_t, const unique_ptr<T, D>& y);                  // freestanding
template<class T, class D>
  constexpr bool operator>(const unique_ptr<T, D>& x, nullptr_t);                  // freestanding
template<class T, class D>
  constexpr bool operator>(nullptr_t, const unique_ptr<T, D>& y);                  // freestanding
```

```
  template<class T, class D>
    constexpr bool operator<=(const unique_ptr<T, D>& x, nullptr_t);                // freestanding
  template<class T, class D>
    constexpr bool operator<=(nullptr_t, const unique_ptr<T, D>& y);                // freestanding
  template<class T, class D>
    constexpr bool operator>=(const unique_ptr<T, D>& x, nullptr_t);                // freestanding
  template<class T, class D>
    constexpr bool operator>=(nullptr_t, const unique_ptr<T, D>& y);                // freestanding
  template<class T, class D>
    requires three_way_comparable<typename unique_ptr<T, D>::pointer>
    constexpr compare_three_way_result_t<typename unique_ptr<T, D>::pointer>
      operator<=>(const unique_ptr<T, D>& x, nullptr_t);                            // freestanding

  template<class E, class T, class Y, class D>
    basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os, const unique_ptr<Y, D>& p);

  // 20.3.2.1, class bad_weak_ptr
  class bad_weak_ptr;

  // 20.3.2.2, class template shared_ptr
  template<class T> class shared_ptr;

  // 20.3.2.2.7, shared_ptr creation
  template<class T, class... Args>
    constexpr shared_ptr<T> make_shared(Args&&... args);                            // T is not
array
  template<class T, class A, class... Args>
    constexpr shared_ptr<T> allocate_shared(const A& a, Args&&... args);            // T is not
array

  template<class T>
    constexpr shared_ptr<T> make_shared(size_t N);                                  // T is U[]
  template<class T, class A>
    constexpr shared_ptr<T> allocate_shared(const A& a, size_t N);                  // T is U[]

  template<class T>
    constexpr shared_ptr<T> make_shared();                                          // T is U[N]
  template<class T, class A>
    constexpr shared_ptr<T> allocate_shared(const A& a);                            // T is U[N]

  template<class T>
    constexpr shared_ptr<T> make_shared(size_t N, const remove_extent_t<T>& u);     // T is U[]
  template<class T, class A>
    constexpr shared_ptr<T> allocate_shared(const A& a, size_t N,
                                const remove_extent_t<T>& u);                       // T is U[]

  template<class T>
    constexpr shared_ptr<T> make_shared(const remove_extent_t<T>& u);               // T is U[N]
  template<class T, class A>
    constexpr shared_ptr<T> allocate_shared(const A& a, const remove_extent_t<T>& u);  // T is U[N]

  template<class T>
    constexpr shared_ptr<T> make_shared_for_overwrite();                            // T is not
U[]
  template<class T, class A>
    constexpr shared_ptr<T> allocate_shared_for_overwrite(const A& a);              // T is not
U[]

  template<class T>
    constexpr shared_ptr<T> make_shared_for_overwrite(size_t N);                    // T is U[]
  template<class T, class A>
    constexpr shared_ptr<T> allocate_shared_for_overwrite(const A& a, size_t N);    // T is U[]
```

```
// 20.3.2.2.8, shared_ptr comparisons
template<class T, class U>
  constexpr bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  constexpr strong_ordering operator<=>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;

template<class T>
  constexpr bool operator==(const shared_ptr<T>& x, nullptr_t) noexcept;
template<class T>
  constexpr strong_ordering operator<=>(const shared_ptr<T>& x, nullptr_t) noexcept;

// 20.3.2.2.9, shared_ptr specialized algorithms
template<class T>
  constexpr void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

// 20.3.2.2.10, shared_ptr casts
template<class T, class U>
  constexpr shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  constexpr shared_ptr<T> static_pointer_cast(shared_ptr<U>&& r) noexcept;
template<class T, class U>
  constexpr shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  constexpr shared_ptr<T> dynamic_pointer_cast(shared_ptr<U>&& r) noexcept;
template<class T, class U>
  constexpr shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  constexpr shared_ptr<T> const_pointer_cast(shared_ptr<U>&& r) noexcept;
template<class T, class U>
  shared_ptr<T> reinterpret_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> reinterpret_pointer_cast(shared_ptr<U>&& r) noexcept;

// 20.3.2.2.11, shared_ptr get_deleter
template<class D, class T>
  constexpr D* get_deleter(const shared_ptr<T>& p) noexcept;

// 20.3.2.2.12, shared_ptr I/O
template<class E, class T, class Y>
  basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os, const shared_ptr<Y>& p);

// 20.3.2.3, class template weak_ptr
template<class T> class weak_ptr;

// 20.3.2.3.7, weak_ptr specialized algorithms
template<class T> constexpr void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;

// 20.3.2.4, class template owner_less
template<class T = void> struct owner_less;

// 20.3.2.5, struct owner_hash
struct owner_hash;

// 20.3.2.6, struct owner_equal
struct owner_equal;

// 20.3.2.7, class template enable_shared_from_this
template<class T> class enable_shared_from_this;

// 20.3.3, hash support
template<class T> struct hash;                                  // freestanding
template<class T, class D> struct hash<unique_ptr<T, D>>;       // freestanding
template<class T> struct hash<shared_ptr<T>>;
```

```
  // 32.5.8.7, atomic smart pointers
  template<class T> struct atomic;                                    // freestanding
  template<class T> struct atomic<shared_ptr<T>>;
  template<class T> struct atomic<weak_ptr<T>>;

  // 20.3.4.1, class template out_ptr_t
  template<class Smart, class Pointer, class... Args>
    class out_ptr_t;                                                  // freestanding

  // 20.3.4.2, function template out_ptr
  template<class Pointer = void, class Smart, class... Args>
    constexpr auto out_ptr(Smart& s, Args&&... args);                 // free-
standing

  // 20.3.4.3, class template inout_ptr_t
  template<class Smart, class Pointer, class... Args>
    class inout_ptr_t;                                                // freestanding

  // 20.3.4.4, function template inout_ptr
  template<class Pointer = void, class Smart, class... Args>
    constexpr auto inout_ptr(Smart& s, Args&&... args);               // free-
standing

  // 20.4.1, class template indirect
  template<class T, class Allocator = allocator<T>>
    class indirect;

  // 20.4.1.10, hash support
  template<class T, class Alloc> struct hash<indirect<T, Alloc>>;

  // 20.4.2, class template polymorphic
  template<class T, class Allocator = allocator<T>>
    class polymorphic;

  namespace pmr {
    template<class T> using indirect    = indirect<T, polymorphic_allocator<T>>;
    template<class T> using polymorphic = polymorphic<T, polymorphic_allocator<T>>;
  }
}
```

20.2.3 Pointer traits [pointer.traits]

20.2.3.1 General [pointer.traits.general]

1 The class template `pointer_traits` supplies a uniform interface to certain attributes of pointer-like types.

```
namespace std {
  template<class Ptr> struct pointer_traits {
    see below;
  };

  template<class T> struct pointer_traits<T*> {
    using pointer         = T*;
    using element_type    = T;
    using difference_type = ptrdiff_t;

    template<class U> using rebind = U*;

    static constexpr pointer pointer_to(see below r) noexcept;
  };
}
```

20.2.3.2 Member types [pointer.traits.types]

1 The definitions in this subclause make use of the following exposition-only class template and concept:

```
template<class T1, class D1, class T2, class D2>
  constexpr bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

4 Let `CT` denote

```
common_type_t<typename unique_ptr<T1, D1>::pointer,
              typename unique_ptr<T2, D2>::pointer>
```

5 *Mandates*:

(5.1) — `unique_ptr<T1, D1>::pointer` is implicitly convertible to `CT` and

(5.2) — `unique_ptr<T2, D2>::pointer` is implicitly convertible to `CT`.

6 *Preconditions*: The specialization `less<CT>` is a function object type (22.10) that induces a strict weak ordering (26.8) on the pointer values.

7 *Returns*: `less<CT>()(x.get(), y.get())`.

```
template<class T1, class D1, class T2, class D2>
  constexpr bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

8 *Returns*: `y < x`.

```
template<class T1, class D1, class T2, class D2>
  constexpr bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

9 *Returns*: `!(y < x)`.

```
template<class T1, class D1, class T2, class D2>
  constexpr bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

10 *Returns*: `!(x < y)`.

```
template<class T1, class D1, class T2, class D2>
  requires three_way_comparable_with<typename unique_ptr<T1, D1>::pointer,
                                     typename unique_ptr<T2, D2>::pointer>
  constexpr compare_three_way_result_t<typename unique_ptr<T1, D1>::pointer,
                               typename unique_ptr<T2, D2>::pointer>
    operator<=>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

11 *Returns*: `compare_three_way()(x.get(), y.get())`.

```
template<class T, class D>
  constexpr bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;
```

12 *Returns*: `!x`.

```
template<class T, class D>
  constexpr bool operator<(const unique_ptr<T, D>& x, nullptr_t);
template<class T, class D>
  constexpr bool operator<(nullptr_t, const unique_ptr<T, D>& x);
```

13 *Preconditions*: The specialization `less<unique_ptr<T, D>::pointer>` is a function object type (22.10) that induces a strict weak ordering (26.8) on the pointer values.

14 *Returns*: The first function template returns

```
less<unique_ptr<T, D>::pointer>()(x.get(), nullptr)
```

The second function template returns

```
less<unique_ptr<T, D>::pointer>()(nullptr, x.get())
```

```
template<class T, class D>
  constexpr bool operator>(const unique_ptr<T, D>& x, nullptr_t);
template<class T, class D>
  constexpr bool operator>(nullptr_t, const unique_ptr<T, D>& x);
```

15 *Returns*: The first function template returns `nullptr < x`. The second function template returns `x < nullptr`.

```
template<class T, class D>
  constexpr bool operator<=(const unique_ptr<T, D>& x, nullptr_t);
```

```
template<class T, class D>
  constexpr bool operator<=(nullptr_t, const unique_ptr<T, D>& x);
```

16 *Returns*: The first function template returns `!(nullptr < x)`. The second function template returns `!(x < nullptr)`.

```
template<class T, class D>
  constexpr bool operator>=(const unique_ptr<T, D>& x, nullptr_t);
template<class T, class D>
  constexpr bool operator>=(nullptr_t, const unique_ptr<T, D>& x);
```

17 *Returns*: The first function template returns `!(x < nullptr)`. The second function template returns `!(nullptr < x)`.

```
template<class T, class D>
  requires three_way_comparable<typename unique_ptr<T, D>::pointer>
  constexpr compare_three_way_result_t<typename unique_ptr<T, D>::pointer>
    operator<=>(const unique_ptr<T, D>& x, nullptr_t);
```

18 *Returns*:

```
compare_three_way()(x.get(), static_cast<typename unique_ptr<T, D>::pointer>(nullptr)).
```

20.3.1.7 I/O [unique.ptr.io]

```
template<class E, class T, class Y, class D>
  basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os, const unique_ptr<Y, D>& p);
```

1 *Constraints*: `os << p.get()` is a valid expression.

2 *Effects*: Equivalent to: `os << p.get();`

3 *Returns*: `os`.

20.3.2 Shared-ownership pointers [util.sharedptr]

20.3.2.1 Class `bad_weak_ptr` [util.smartptr.weak.bad]

```
namespace std {
  class bad_weak_ptr : public exception {
  public:
    // see 17.9.3 for the specification of the special member functions
    constexpr const char* what() const noexcept override;
  };
}
```

1 An exception of type `bad_weak_ptr` is thrown by the `shared_ptr` constructor taking a `weak_ptr`.

```
constexpr const char* what() const noexcept override;
```

2 *Returns*: An implementation-defined NTBS.

20.3.2.2 Class template `shared_ptr` [util.smartptr.shared]

20.3.2.2.1 General [util.smartptr.shared.general]

1 The `shared_ptr` class template stores a pointer, usually obtained via `new`. `shared_ptr` implements semantics of shared ownership; the last remaining owner of the pointer is responsible for destroying the object, or otherwise releasing the resources associated with the stored pointer. A `shared_ptr` is said to be empty if it does not own a pointer.

```
namespace std {
  template<class T> class shared_ptr {
  public:
    using element_type = remove_extent_t<T>;
    using weak_type    = weak_ptr<T>;

    // 20.3.2.2.2, constructors
    constexpr shared_ptr() noexcept;
    constexpr shared_ptr(nullptr_t) noexcept : shared_ptr() { }
    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(shared_ptr<Y>&& r, element_type* p) noexcept;
    constexpr shared_ptr(const shared_ptr& r) noexcept;
    template<class Y> constexpr shared_ptr(const shared_ptr<Y>& r) noexcept;
    constexpr shared_ptr(shared_ptr&& r) noexcept;
    template<class Y> constexpr shared_ptr(shared_ptr<Y>&& r) noexcept;
    template<class Y> constexpr explicit shared_ptr(const weak_ptr<Y>& r);
    template<class Y, class D>
      constexpr shared_ptr(unique_ptr<Y, D>&& r);

    // 20.3.2.2.3, destructor
    constexpr ~shared_ptr();

    // 20.3.2.2.4, assignment
    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;
    template<class Y, class D>
      constexpr shared_ptr& operator=(unique_ptr<Y, D>&& r);

    // 20.3.2.2.5, 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);

    // 20.3.2.2.6, observers
    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;
    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;
    size_t owner_hash() const noexcept;
    template<class U>
      constexpr bool owner_equal(const shared_ptr<U>& b) const noexcept;
    template<class U>
      constexpr bool owner_equal(const weak_ptr<U>& b) const noexcept;
  };

  template<class T>
    shared_ptr(weak_ptr<T>) -> shared_ptr<T>;
  template<class T, class D>
    shared_ptr(unique_ptr<T, D>) -> shared_ptr<T>;
}
```

[2] Specializations of `shared_ptr` shall be *Cpp17CopyConstructible*, *Cpp17CopyAssignable*, and *Cpp17LessThanComparable*, allowing their use in standard containers. Specializations of `shared_ptr` shall be contextually convertible to `bool`, allowing their use in boolean expressions and declarations in conditions.

3 The template parameter `T` of `shared_ptr` may be an incomplete type.

[*Note 1*: `T` can be a function type. — *end note*]

4 [*Example 1*:

```
if (shared_ptr<X> px = dynamic_pointer_cast<X>(py)) {
  // do something with px
}
```

— *end example*]

5 For purposes of determining the presence of a data race, member functions shall access and modify only the `shared_ptr` and `weak_ptr` objects themselves and not objects they refer to. Changes in `use_count()` do not reflect modifications that can introduce data races.

6 For the purposes of 20.3, a pointer type `Y*` is said to be *compatible with* a pointer type `T*` when either `Y*` is convertible to `T*` or `Y` is `U[N]` and `T` is *cv* `U[]`.

20.3.2.2.2 Constructors [util.smartptr.shared.const]

1 In the constructor definitions below, enables `shared_from_this` with `p`, for a pointer `p` of type `Y*`, means that if `Y` has an unambiguous and accessible base class that is a specialization of `enable_shared_from_this` (20.3.2.2.7), then `remove_cv_t<Y>*` shall be implicitly convertible to `T*` and the constructor evaluates the statement:

```
if (p != nullptr && p->weak-this.expired())
  p->weak-this = shared_ptr<remove_cv_t<Y>>(*this, const_cast<remove_cv_t<Y>*>(p));
```

The assignment to the *`weak-this`* member is not atomic and conflicts with any potentially concurrent access to the same object (6.9.2).

```
constexpr shared_ptr() noexcept;
```

2 *Postconditions*: `use_count() == 0 && get() == nullptr`.

```
template<class Y> constexpr explicit shared_ptr(Y* p);
```

3 *Constraints*: When `T` is an array type, the expression `delete[] p` is well-formed and either `T` is `U[N]` and `Y(*)[N]` is convertible to `T*`, or `T` is `U[]` and `Y(*)[]` is convertible to `T*`. When `T` is not an array type, the expression `delete p` is well-formed and `Y*` is convertible to `T*`.

4 *Mandates*: `Y` is a complete type.

5 *Preconditions*: The expression `delete[] p`, when `T` is an array type, or `delete p`, when `T` is not an array type, has well-defined behavior, and does not throw exceptions.

6 *Effects*: When `T` is not an array type, constructs a `shared_ptr` object that owns the pointer `p`. Otherwise, constructs a `shared_ptr` that owns `p` and a deleter of an unspecified type that calls `delete[] p`. When `T` is not an array type, enables `shared_from_this` with `p`. If an exception is thrown, `delete p` is called when `T` is not an array type, `delete[] p` otherwise.

7 *Postconditions*: `use_count() == 1 && get() == p`.

8 *Throws*: `bad_alloc`, or an implementation-defined exception when a resource other than memory cannot be obtained.

```
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);
```

9 *Constraints*: `is_move_constructible_v<D>` is `true`, and `d(p)` is a well-formed expression. For the first two overloads:

(9.1) — If `T` is an array type, then either `T` is `U[N]` and `Y(*)[N]` is convertible to `T*`, or `T` is `U[]` and `Y(*)[]` is convertible to `T*`.

(9.2) — If `T` is not an array type, then `Y*` is convertible to `T*`.

10 *Preconditions*: Construction of `d` and a deleter of type `D` initialized with `std::move(d)` do not throw exceptions. The expression `d(p)` has well-defined behavior and does not throw exceptions. `A` meets the *Cpp17Allocator* requirements (16.4.4.6.1).

11 *Effects*: Constructs a `shared_ptr` object that owns the object `p` and the deleter `d`. When `T` is not an array type, the first and second constructors enable `shared_from_this` with `p`. The second and fourth constructors shall use a copy of `a` to allocate memory for internal use. If an exception is thrown, `d(p)` is called.

12 *Postconditions*: `use_count() == 1 && get() == p`.

13 *Throws*: `bad_alloc`, or an implementation-defined exception when a resource other than memory cannot be obtained.

```
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;
```

14 *Effects*: Constructs a `shared_ptr` instance that stores `p` and shares ownership with the initial value of `r`.

15 *Postconditions*: `get() == p`. For the second overload, `r` is empty and `r.get() == nullptr`.

16 [*Note 1*: Use of this constructor leads to a dangling pointer unless `p` remains valid at least until the ownership group of `r` is destroyed. — *end note*]

17 [*Note 2*: This constructor allows creation of an empty `shared_ptr` instance with a non-null stored pointer. — *end note*]

```
constexpr shared_ptr(const shared_ptr& r) noexcept;
template<class Y> constexpr shared_ptr(const shared_ptr<Y>& r) noexcept;
```

18 *Constraints*: For the second constructor, `Y*` is compatible with `T*`.

19 *Effects*: If `r` is empty, constructs an empty `shared_ptr` object; otherwise, constructs a `shared_ptr` object that shares ownership with `r`.

20 *Postconditions*: `get() == r.get() && use_count() == r.use_count()`.

```
constexpr shared_ptr(shared_ptr&& r) noexcept;
template<class Y> constexpr shared_ptr(shared_ptr<Y>&& r) noexcept;
```

21 *Constraints*: For the second constructor, `Y*` is compatible with `T*`.

22 *Effects*: Move constructs a `shared_ptr` instance from `r`.

23 *Postconditions*: `*this` contains the old value of `r`. `r` is empty, and `r.get() == nullptr`.

```
template<class Y> constexpr explicit shared_ptr(const weak_ptr<Y>& r);
```

24 *Constraints*: `Y*` is compatible with `T*`.

25 *Effects*: Constructs a `shared_ptr` object that shares ownership with `r` and stores a copy of the pointer stored in `r`. If an exception is thrown, the constructor has no effect.

26 *Postconditions*: `use_count() == r.use_count()`.

27 *Throws*: `bad_weak_ptr` when `r.expired()`.

```
template<class Y, class D> constexpr shared_ptr(unique_ptr<Y, D>&& r);
```

28 *Constraints*: `Y*` is compatible with `T*` and `unique_ptr<Y, D>::pointer` is convertible to `element_type*`.

29 *Effects*: If `r.get() == nullptr`, equivalent to `shared_ptr()`. Otherwise, if `D` is not a reference type, equivalent to `shared_ptr(r.release(), std::move(r.get_deleter()))`. Otherwise, equivalent to `shared_ptr(r.release(), ref(r.get_deleter()))`. If an exception is thrown, the constructor has no effect.

20.3.2.2.3 Destructor [util.smartptr.shared.dest]

```
constexpr ~shared_ptr();
```

1 *Effects*:

(1.1) — If `*this` is empty or shares ownership with another `shared_ptr` instance (`use_count() > 1`), there are no side effects.

(1.2) — Otherwise, if `*this` owns an object `p` and a deleter `d`, `d(p)` is called.

(1.3) — Otherwise, `*this` owns a pointer `p`, and `delete p` is called.

2 [*Note 1*: Since the destruction of `*this` decreases the number of instances that share ownership with `*this` by one, after `*this` has been destroyed all `shared_ptr` instances that shared ownership with `*this` will report a `use_count()` that is one less than its previous value. — *end note*]

20.3.2.2.4 Assignment [util.smartptr.shared.assign]

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

1 *Effects*: Equivalent to `shared_ptr(r).swap(*this)`.

2 *Returns*: `*this`.

3 [*Note 1*: The use count updates caused by the temporary object construction and destruction are not observable side effects, so the implementation can meet the effects (and the implied guarantees) via different means, without creating a temporary. In particular, in the example:

```
shared_ptr<int> p(new int);
shared_ptr<void> q(p);
p = p;
q = p;
```

both assignments can be no-ops. — *end note*]

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

4 *Effects*: Equivalent to `shared_ptr(std::move(r)).swap(*this)`.

5 *Returns*: `*this`.

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

6 *Effects*: Equivalent to `shared_ptr(std::move(r)).swap(*this)`.

7 *Returns*: `*this`.

20.3.2.2.5 Modifiers [util.smartptr.shared.mod]

```
constexpr void swap(shared_ptr& r) noexcept;
```

1 *Effects*: Exchanges the contents of `*this` and `r`.

```
constexpr void reset() noexcept;
```

2 *Effects*: Equivalent to `shared_ptr().swap(*this)`.

```
template<class Y> constexpr void reset(Y* p);
```

3 *Effects*: Equivalent to `shared_ptr(p).swap(*this)`.

```
template<class Y, class D> constexpr void reset(Y* p, D d);
```

4 *Effects*: Equivalent to `shared_ptr(p, d).swap(*this)`.

```
template<class Y, class D, class A> constexpr void reset(Y* p, D d, A a);
```

5 *Effects*: Equivalent to `shared_ptr(p, d, a).swap(*this)`.

20.3.2.2.6 Observers [util.smartptr.shared.obs]

```
constexpr element_type* get() const noexcept;
```

1 *Returns*: The stored pointer.

```
constexpr T& operator*() const noexcept;
```

2 *Preconditions*: `get() != nullptr`.

3 *Returns*: `*get()`.

4 *Remarks*: When `T` is an array type or *cv* `void`, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well-formed.

```
constexpr T* operator->() const noexcept;
```

5 *Preconditions*: `get() != nullptr`.

6 *Returns*: `get()`.

7 *Remarks*: When `T` is an array type, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well-formed.

```
constexpr element_type& operator[](ptrdiff_t i) const;
```

8 *Preconditions*: `get() != nullptr && i >= 0`. If `T` is `U[N]`, `i < N`.

9 *Returns*: `get()[i]`.

10 *Throws*: Nothing.

11 *Remarks*: When `T` is not an array type, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well-formed.

```
constexpr long use_count() const noexcept;
```

12 *Synchronization*: None.

13 *Returns*: The number of `shared_ptr` objects, `*this` included, that share ownership with `*this`, or `0` when `*this` is empty.

14 [*Note 1*: `get() == nullptr` does not imply a specific return value of `use_count()`. — *end note*]

15 [*Note 2*: `weak_ptr<T>::lock()` can affect the return value of `use_count()`. — *end note*]

16 [*Note 3*: When multiple threads might affect the return value of `use_count()`, the result is approximate. In particular, `use_count() == 1` does not imply that accesses through a previously destroyed `shared_ptr` have in any sense completed. — *end note*]

```
constexpr explicit operator bool() const noexcept;
```

17 *Returns*: `get() != nullptr`.

```
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;
```

18 *Returns*: An unspecified value such that

(18.1) — `owner_before(b)` defines a strict weak ordering as defined in 26.8;

(18.2) — `!owner_before(b) && !b.owner_before(*this)` is `true` if and only if `owner_equal(b)` is `true`.

```
size_t owner_hash() const noexcept;
```

19 *Returns*: An unspecified value such that, for any object `x` where `owner_equal(x)` is `true`, `owner_hash() == x.owner_hash()` is `true`.

```
template<class U>
  constexpr bool owner_equal(const shared_ptr<U>& b) const noexcept;
template<class U>
  constexpr bool owner_equal(const weak_ptr<U>& b) const noexcept;
```

20 *Returns*: `true` if and only if `*this` and `b` share ownership or are both empty. Otherwise returns `false`.

21 *Remarks*: `owner_equal` is an equivalence relation.

20.3.2.2.7 Creation [util.smartptr.shared.create]

1 The common requirements that apply to all `make_shared`, `allocate_shared`, `make_shared_for_overwrite`, and `allocate_shared_for_overwrite` overloads, unless specified otherwise, are described below.

```
template<class T, ...>
  constexpr shared_ptr<T> make_shared(args);
template<class T, class A, ...>
  constexpr shared_ptr<T> allocate_shared(const A& a, args);
template<class T, ...>
  constexpr shared_ptr<T> make_shared_for_overwrite(args);
```

```
template<class T, class A, ...>
  constexpr shared_ptr<T> allocate_shared_for_overwrite(const A& a, args);
```

2 *Preconditions*: `A` meets the *Cpp17Allocator* requirements (16.4.4.6.1).

3 *Effects*: Allocates memory for an object of type `T` (or `U[N]` when `T` is `U[]`, where `N` is determined from *args* as specified by the concrete overload). The object is initialized from *args* as specified by the concrete overload. The `allocate_shared` and `allocate_shared_for_overwrite` templates use a copy of `a` (rebound for an unspecified `value_type`) to allocate memory. If an exception is thrown, the functions have no effect.

4 *Postconditions*: `r.get() != nullptr && r.use_count() == 1`, where `r` is the return value.

5 *Returns*: A `shared_ptr` instance that stores and owns the address of the newly constructed object.

6 *Throws*: `bad_alloc`, or an exception thrown from `allocate` or from the initialization of the object.

7 *Remarks*:

(7.1) — Implementations should perform no more than one memory allocation.

[*Note 1*: This provides efficiency equivalent to an intrusive smart pointer. — *end note*]

(7.2) — When an object of an array type `U` is specified to have an initial value of `u` (of the same type), this shall be interpreted to mean that each array element of the object has as its initial value the corresponding element from `u`.

(7.3) — When an object of an array type is specified to have a default initial value, this shall be interpreted to mean that each array element of the object has a default initial value.

(7.4) — When a (sub)object of a non-array type `U` is specified to have an initial value of `v`, or `U(l...)`, where `l...` is a list of constructor arguments, `make_shared` shall initialize this (sub)object via the expression `::new(pv) U(v)` or `::new(pv) U(l...)` respectively, where `pv` has type `void*` and points to storage suitable to hold an object of type `U`.

(7.5) — When a (sub)object of a non-array type `U` is specified to have an initial value of `v`, or `U(l...)`, where `l...` is a list of constructor arguments, `allocate_shared` shall initialize this (sub)object via the expression

(7.5.1) — `allocator_traits<A2>::construct(a2, pu, v)` or

(7.5.2) — `allocator_traits<A2>::construct(a2, pu, l...)`

respectively, where `pu` is a pointer of type `remove_cv_t<U>*` pointing to storage suitable to hold an object of type `remove_cv_t<U>` and `a2` of type `A2` is a potentially rebound copy of the allocator `a` passed to `allocate_shared`.

(7.6) — When a (sub)object of non-array type `U` is specified to have a default initial value, `make_shared` shall initialize this (sub)object via the expression `::new(pv) U()`, where `pv` has type `void*` and points to storage suitable to hold an object of type `U`.

(7.7) — When a (sub)object of non-array type `U` is specified to have a default initial value, `allocate_shared` initializes this (sub)object via the expression `allocator_traits<A2>::construct(a2, pu)`, where `pu` is a pointer of type `remove_cv_t<U>*` pointing to storage suitable to hold an object of type `remove_cv_t<U>` and `a2` of type `A2` is a potentially rebound copy of the allocator `a` passed to `allocate_shared`.

(7.8) — When a (sub)object of non-array type `U` is initialized by `make_shared_for_overwrite` or `allocate_shared_for_overwrite`, it is initialized via the expression `::new(pv) U`, where `pv` has type `void*` and points to storage suitable to hold an object of type `U`.

(7.9) — Array elements are initialized in ascending order of their addresses.

(7.10) — When the lifetime of the object managed by the return value ends, or when the initialization of an array element throws an exception, the initialized elements are destroyed in the reverse order of their original construction.

(7.11) — When a (sub)object of non-array type `U` that was initialized by `make_shared`, `make_shared_for_overwrite`, or `allocate_shared_for_overwrite` is to be destroyed, it is destroyed via the expression `pu->~U()` where `pu` points to that object of type `U`.

(7.12) — When a (sub)object of non-array type `U` that was initialized by `allocate_shared` is to be destroyed, it is destroyed via the expression `allocator_traits<A2>::destroy(a2, pu)` where `pu` is a pointer

of type `remove_cv_t<U>*` pointing to that object of type `remove_cv_t<U>` and `a2` of type `A2` is a potentially rebound copy of the allocator `a` passed to `allocate_shared`.

[*Note 2*: These functions will typically allocate more memory than `sizeof(T)` to allow for internal bookkeeping structures such as reference counts. — *end note*]

```
template<class T, class... Args>
  constexpr shared_ptr<T> make_shared(Args&&... args);                    // T is not array
template<class T, class A, class... Args>
  constexpr shared_ptr<T> allocate_shared(const A& a, Args&&... args);    // T is not array
```

8 *Constraints*: `T` is not an array type.

9 *Returns*: A `shared_ptr` to an object of type `T` with an initial value `T(std::forward<Args>(args)...)`.

10 *Remarks*: The `shared_ptr` constructors called by these functions enable `shared_from_this` with the address of the newly constructed object of type `T`.

11 [*Example 1*:

```
shared_ptr<int> p = make_shared<int>(); // shared_ptr to int()
shared_ptr<vector<int>> q = make_shared<vector<int>>(16, 1);
  // shared_ptr to vector of 16 elements with value 1
```

— *end example*]

```
template<class T> shared_ptr<T>
  constexpr make_shared(size_t N);                                        // T is U[]
template<class T, class A>
  constexpr shared_ptr<T> allocate_shared(const A& a, size_t N);          // T is U[]
```

12 *Constraints*: `T` is of the form `U[]`.

13 *Returns*: A `shared_ptr` to an object of type `U[N]` with a default initial value, where `U` is `remove_extent_t<T>`.

14 [*Example 2*:

```
shared_ptr<double[]> p = make_shared<double[]>(1024);
  // shared_ptr to a value-initialized double[1024]
shared_ptr<double[][2][2]> q = make_shared<double[][2][2]>(6);
  // shared_ptr to a value-initialized double[6][2][2]
```

— *end example*]

```
template<class T>
  constexpr shared_ptr<T> make_shared();                                  // T is U[N]
template<class T, class A>
  constexpr shared_ptr<T> allocate_shared(const A& a);                    // T is U[N]
```

15 *Constraints*: `T` is of the form `U[N]`.

16 *Returns*: A `shared_ptr` to an object of type `T` with a default initial value.

17 [*Example 3*:

```
shared_ptr<double[1024]> p = make_shared<double[1024]>();
  // shared_ptr to a value-initialized double[1024]
shared_ptr<double[6][2][2]> q = make_shared<double[6][2][2]>();
  // shared_ptr to a value-initialized double[6][2][2]
```

— *end example*]

```
template<class T>
  constexpr shared_ptr<T> make_shared(size_t N, const remove_extent_t<T>& u);       // T is U[]
template<class T, class A>
  constexpr shared_ptr<T> allocate_shared(const A& a, size_t N, const remove_extent_t<T>& u);   // T is U[]
```

18 *Constraints*: `T` is of the form `U[]`.

19 *Returns*: A `shared_ptr` to an object of type `U[N]`, where `U` is `remove_extent_t<T>` and each array element has an initial value of `u`.

20 [*Example 4*:

```
shared_ptr<double[]> p = make_shared<double[]>(1024, 1.0);
  // shared_ptr to a double[1024], where each element is 1.0
shared_ptr<double[][2]> q = make_shared<double[][2]>(6, {1.0, 0.0});
  // shared_ptr to a double[6][2], where each double[2] element is {1.0, 0.0}
shared_ptr<vector<int>[]> r = make_shared<vector<int>[]>(4, {1, 2});
  // shared_ptr to a vector<int>[4], where each vector has contents {1, 2}
```

— *end example*]

```
template<class T>
  constexpr shared_ptr<T> make_shared(const remove_extent_t<T>& u);       // T is U[N]
template<class T, class A>
  constexpr shared_ptr<T> allocate_shared(const A& a, const remove_extent_t<T>& u);   // T is U[N]
```

21 *Constraints*: `T` is of the form `U[N]`.

22 *Returns*: A `shared_ptr` to an object of type `T`, where each array element of type `remove_extent_t<T>` has an initial value of `u`.

23 [*Example 5*:

```
shared_ptr<double[1024]> p = make_shared<double[1024]>(1.0);
  // shared_ptr to a double[1024], where each element is 1.0
shared_ptr<double[6][2]> q = make_shared<double[6][2]>({1.0, 0.0});
  // shared_ptr to a double[6][2], where each double[2] element is {1.0, 0.0}
shared_ptr<vector<int>[4]> r = make_shared<vector<int>[4]>({1, 2});
  // shared_ptr to a vector<int>[4], where each vector has contents {1, 2}
```

— *end example*]

```
template<class T>
  constexpr shared_ptr<T> make_shared_for_overwrite();
template<class T, class A>
  constexpr shared_ptr<T> allocate_shared_for_overwrite(const A& a);
```

24 *Constraints*: `T` is not an array of unknown bound.

25 *Returns*: A `shared_ptr` to an object of type `T`.

26 [*Example 6*:

```
struct X { double data[1024]; };
shared_ptr<X> p = make_shared_for_overwrite<X>();
  // shared_ptr to a default-initialized X, where each element in X::data has an indeterminate value

shared_ptr<double[1024]> q = make_shared_for_overwrite<double[1024]>();
  // shared_ptr to a default-initialized double[1024], where each element has an indeterminate value
```

— *end example*]

```
template<class T>
  constexpr shared_ptr<T> make_shared_for_overwrite(size_t N);
template<class T, class A>
  constexpr shared_ptr<T> allocate_shared_for_overwrite(const A& a, size_t N);
```

27 *Constraints*: `T` is an array of unknown bound.

28 *Returns*: A `shared_ptr` to an object of type `U[N]`, where `U` is `remove_extent_t<T>`.

29 [*Example 7*:

```
shared_ptr<double[]> p = make_shared_for_overwrite<double[]>(1024);
  // shared_ptr to a default-initialized double[1024], where each element has an indeterminate value
```

— *end example*]

20.3.2.2.8 Comparison [util.smartptr.shared.cmp]

```
template<class T, class U>
  constexpr bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
```

1 *Returns*: `a.get() == b.get()`.

```
template<class T> constexpr bool operator==(const shared_ptr<T>& a, nullptr_t) noexcept;
```

2 *Returns*: `!a`.

```
template<class T, class U>
  constexpr strong_ordering operator<=>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
```

3 *Returns*: `compare_three_way()(a.get(), b.get())`.

4 [*Note 1*: Defining a comparison operator function allows `shared_ptr` objects to be used as keys in associative containers. — *end note*]

```
template<class T>
  constexpr strong_ordering operator<=>(const shared_ptr<T>& a, nullptr_t) noexcept;
```

5 *Returns*:

```
compare_three_way()(a.get(), static_cast<typename shared_ptr<T>::element_type*>(nullptr))
```

20.3.2.2.9 Specialized algorithms [util.smartptr.shared.spec]

```
template<class T>
  constexpr void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;
```

1 *Effects*: Equivalent to `a.swap(b)`.

20.3.2.2.10 Casts [util.smartptr.shared.cast]

```
template<class T, class U>
  constexpr shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  constexpr shared_ptr<T> static_pointer_cast(shared_ptr<U>&& r) noexcept;
```

1 *Mandates*: The expression `static_cast<T*>((U*)nullptr)` is well-formed.

2 *Returns*:

```
shared_ptr<T>(R, static_cast<typename shared_ptr<T>::element_type*>(r.get()))
```

where *R* is `r` for the first overload, and `std::move(r)` for the second.

3 [*Note 1*: The seemingly equivalent expression `shared_ptr<T>(static_cast<T*>(r.get()))` can result in undefined behavior, attempting to delete the same object twice. — *end note*]

```
template<class T, class U>
  constexpr shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  constexpr shared_ptr<T> dynamic_pointer_cast(shared_ptr<U>&& r) noexcept;
```

4 *Mandates*: The expression `dynamic_cast<T*>((U*)nullptr)` is well-formed. The expression `dynamic_cast<typename shared_ptr<T>::element_type*>(r.get())` is well-formed.

5 *Preconditions*: The expression `dynamic_cast<typename shared_ptr<T>::element_type*>(r.get())` has well-defined behavior.

6 *Returns*:

(6.1) — When `dynamic_cast<typename shared_ptr<T>::element_type*>(r.get())` returns a non-null value `p`, `shared_ptr<T>(R, p)`, where *R* is `r` for the first overload, and `std::move(r)` for the second.

(6.2) — Otherwise, `shared_ptr<T>()`.

7 [*Note 2*: The seemingly equivalent expression `shared_ptr<T>(dynamic_cast<T*>(r.get()))` can result in undefined behavior, attempting to delete the same object twice. — *end note*]

```
template<class T, class U>
  constexpr shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  constexpr shared_ptr<T> const_pointer_cast(shared_ptr<U>&& r) noexcept;
```

8 *Mandates*: The expression `const_cast<T*>((U*)nullptr)` is well-formed.

9 *Returns*:

```
shared_ptr<T>(R, const_cast<typename shared_ptr<T>::element_type*>(r.get()))
```

where *R* is `r` for the first overload, and `std::move(r)` for the second.

10 [*Note 3*: The seemingly equivalent expression `shared_ptr<T>(const_cast<T*>(r.get()))` can result in undefined behavior, attempting to delete the same object twice. — *end note*]

```
template<class T, class U>
  shared_ptr<T> reinterpret_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> reinterpret_pointer_cast(shared_ptr<U>&& r) noexcept;
```

11 *Mandates*: The expression `reinterpret_cast<T*>((U*)nullptr)` is well-formed.

12 *Returns*:

```
shared_ptr<T>(R, reinterpret_cast<typename shared_ptr<T>::element_type*>(r.get()))
```

where *R* is `r` for the first overload, and `std::move(r)` for the second.

13 [*Note 4*: The seemingly equivalent expression `shared_ptr<T>(reinterpret_cast<T*>(r.get()))` can result in undefined behavior, attempting to delete the same object twice. — *end note*]

20.3.2.2.11 `get_deleter` [util.smartptr.getdeleter]

```
template<class D, class T>
  constexpr D* get_deleter(const shared_ptr<T>& p) noexcept;
```

1 *Returns*: If `p` owns a deleter `d` of type cv-unqualified `D`, returns `addressof(d)`; otherwise returns `nullptr`. The returned pointer remains valid as long as there exists a `shared_ptr` instance that owns `d`.

[*Note 1*: It is unspecified whether the pointer remains valid longer than that. This can happen if the implementation doesn't destroy the deleter until all `weak_ptr` instances that share ownership with `p` have been destroyed. — *end note*]

20.3.2.2.12 I/O [util.smartptr.shared.io]

```
template<class E, class T, class Y>
  basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os, const shared_ptr<Y>& p);
```

1 *Effects*: As if by: `os << p.get();`

2 *Returns*: `os`.

20.3.2.3 Class template `weak_ptr` [util.smartptr.weak]

20.3.2.3.1 General [util.smartptr.weak.general]

1 The `weak_ptr` class template stores a weak reference to an object that is already managed by a `shared_ptr`. To access the object, a `weak_ptr` can be converted to a `shared_ptr` using the member function `lock`.

```
namespace std {
  template<class T> class weak_ptr {
  public:
    using element_type = remove_extent_t<T>;

    // 20.3.2.3.2, constructors
    constexpr weak_ptr() noexcept;
    template<class Y>
      constexpr weak_ptr(const shared_ptr<Y>& r) noexcept;
    constexpr weak_ptr(const weak_ptr& r) noexcept;
    template<class Y>
      constexpr weak_ptr(const weak_ptr<Y>& r) noexcept;
    constexpr weak_ptr(weak_ptr&& r) noexcept;
    template<class Y>
      constexpr weak_ptr(weak_ptr<Y>&& r) noexcept;

    // 20.3.2.3.3, destructor
    constexpr ~weak_ptr();

    // 20.3.2.3.4, assignment
    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;

    // 20.3.2.3.5, modifiers
    constexpr void swap(weak_ptr& r) noexcept;
    constexpr void reset() noexcept;

    // 20.3.2.3.6, observers
    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;
    size_t owner_hash() const noexcept;
    template<class U>
      constexpr bool owner_equal(const shared_ptr<U>& b) const noexcept;
    template<class U>
      constexpr bool owner_equal(const weak_ptr<U>& b) const noexcept;
  };

  template<class T>
    weak_ptr(shared_ptr<T>) -> weak_ptr<T>;
}
```

2 Specializations of `weak_ptr` shall be *Cpp17CopyConstructible* and *Cpp17CopyAssignable*, allowing their use in standard containers. The template parameter `T` of `weak_ptr` may be an incomplete type.

20.3.2.3.2 Constructors [util.smartptr.weak.const]

```
constexpr weak_ptr() noexcept;
```

1 *Effects*: Constructs an empty `weak_ptr` object that stores a null pointer value.

2 *Postconditions*: `use_count() == 0`.

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

3 *Constraints*: For the second and third constructors, `Y*` is compatible with `T*`.

4 *Effects*: If `r` is empty, constructs an empty `weak_ptr` object that stores a null pointer value; otherwise, constructs a `weak_ptr` object that shares ownership with `r` and stores a copy of the pointer stored in `r`.

5 *Postconditions*: `use_count() == r.use_count()`.

```
constexpr weak_ptr(weak_ptr&& r) noexcept;
template<class Y> constexpr weak_ptr(weak_ptr<Y>&& r) noexcept;
```

6 *Constraints*: For the second constructor, `Y*` is compatible with `T*`.

7 *Effects*: Move constructs a `weak_ptr` instance from `r`.

8 *Postconditions*: `*this` contains the old value of `r`. `r` is empty, stores a null pointer value, and `r.use_count() == 0`.

20.3.2.3.3 Destructor [util.smartptr.weak.dest]

```
constexpr ~weak_ptr();
```

1 *Effects*: Destroys this `weak_ptr` object but has no effect on the object its stored pointer points to.

20.3.2.3.4 Assignment [util.smartptr.weak.assign]

```
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;
```

1 *Effects*: Equivalent to `weak_ptr(r).swap(*this)`.

2 *Returns*: `*this`.

3 *Remarks*: The implementation may meet the effects (and the implied guarantees) via different means, without creating a temporary object.

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

4 *Effects*: Equivalent to `weak_ptr(std::move(r)).swap(*this)`.

5 *Returns*: `*this`.

20.3.2.3.5 Modifiers [util.smartptr.weak.mod]

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

1 *Effects*: Exchanges the contents of `*this` and `r`.

```
constexpr void reset() noexcept;
```

2 *Effects*: Equivalent to `weak_ptr().swap(*this)`.

20.3.2.3.6 Observers [util.smartptr.weak.obs]

```
constexpr long use_count() const noexcept;
```

1 *Returns*: `0` if `*this` is empty; otherwise, the number of `shared_ptr` instances that share ownership with `*this`.

```
constexpr bool expired() const noexcept;
```

2 *Returns*: `use_count() == 0`.

```
constexpr shared_ptr<T> lock() const noexcept;
```

3 *Returns*: `expired() ? shared_ptr<T>() : shared_ptr<T>(*this)`, executed atomically.

```
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;
```

4 *Returns*: An unspecified value such that

(4.1) — `owner_before(b)` defines a strict weak ordering as defined in 26.8;

(4.2) — `!owner_before(b) && !b.owner_before(*this)` is `true` if and only if `owner_equal(b)` is `true`.

```
size_t owner_hash() const noexcept;
```

5 *Returns*: An unspecified value such that, for any object `x` where `owner_equal(x)` is `true`, `owner_hash() == x.owner_hash()` is `true`.

```
template<class U>
  constexpr bool owner_equal(const shared_ptr<U>& b) const noexcept;
template<class U>
  constexpr bool owner_equal(const weak_ptr<U>& b) const noexcept;
```

6 *Returns*: `true` if and only if `*this` and `b` share ownership or are both empty. Otherwise returns `false`.

7 *Remarks*: `owner_equal` is an equivalence relation.

20.3.2.3.7 Specialized algorithms [util.smartptr.weak.spec]

```
template<class T>
  constexpr void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;
```

1 *Effects*: Equivalent to `a.swap(b)`.

20.3.2.4 Class template `owner_less` [util.smartptr.ownerless]

1 The class template `owner_less` allows ownership-based mixed comparisons of shared and weak pointers.

```
namespace std {
  template<class T = void> struct owner_less;

  template<class T> struct owner_less<shared_ptr<T>> {
    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;
  };

  template<class T> struct owner_less<weak_ptr<T>> {
    constexpr bool operator()(const weak_ptr<T>&, const weak_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;
  };

  template<> struct owner_less<void> {
    template<class T, class U>
      constexpr bool operator()(const shared_ptr<T>&, const shared_ptr<U>&) const noexcept;
    template<class T, class U>
      constexpr bool operator()(const shared_ptr<T>&, const weak_ptr<U>&) const noexcept;
    template<class T, class U>
      constexpr bool operator()(const weak_ptr<T>&, const shared_ptr<U>&) const noexcept;
    template<class T, class U>
      constexpr bool operator()(const weak_ptr<T>&, const weak_ptr<U>&) const noexcept;

    using is_transparent = unspecified;
  };
}
```

2 `operator()(x, y)` returns `x.owner_before(y)`.

[*Note 1*: Note that

(2.1) — `operator()` defines a strict weak ordering as defined in 26.8;

(2.2) — `!operator()(a, b) && !operator()(b, a)` is `true` if and only if `a.owner_equal(b)` is `true`.

— *end note*]

20.3.2.5 Struct `owner_hash` [util.smartptr.owner.hash]

1 The class `owner_hash` provides ownership-based hashing.

```
namespace std {
  struct owner_hash {
    template<class T>
      size_t operator()(const shared_ptr<T>&) const noexcept;

    template<class T>
      size_t operator()(const weak_ptr<T>&) const noexcept;

    using is_transparent = unspecified;
  };
}

template<class T>
  size_t operator()(const shared_ptr<T>& x) const noexcept;
template<class T>
  size_t operator()(const weak_ptr<T>& x) const noexcept;
```

2 *Returns*: `x.owner_hash()`.

3 [*Note 1*: For any object `y` where `x.owner_equal(y)` is `true`, `x.owner_hash() == y.owner_hash()` is `true`. — *end note*]

20.3.2.6 Struct `owner_equal` [util.smartptr.owner.equal]

1 The class `owner_equal` provides ownership-based mixed equality comparisons of shared and weak pointers.

```
namespace std {
  struct owner_equal {
    template<class T, class U>
      constexpr bool operator()(const shared_ptr<T>&, const shared_ptr<U>&) const noexcept;
    template<class T, class U>
      constexpr bool operator()(const shared_ptr<T>&, const weak_ptr<U>&) const noexcept;
    template<class T, class U>
      constexpr bool operator()(const weak_ptr<T>&, const shared_ptr<U>&) const noexcept;
    template<class T, class U>
      constexpr bool operator()(const weak_ptr<T>&, const weak_ptr<U>&) const noexcept;

    using is_transparent = unspecified;
  };
}

template<class T, class U>
  constexpr bool operator()(const shared_ptr<T>& x, const shared_ptr<U>& y) const noexcept;
template<class T, class U>
  constexpr bool operator()(const shared_ptr<T>& x, const weak_ptr<U>& y) const noexcept;
template<class T, class U>
  constexpr bool operator()(const weak_ptr<T>& x, const shared_ptr<U>& y) const noexcept;
template<class T, class U>
  constexpr bool operator()(const weak_ptr<T>& x, const weak_ptr<U>& y) const noexcept;
```

2 *Returns*: `x.owner_equal(y)`.

3 [*Note 1*: `x.owner_equal(y)` is `true` if and only if `x` and `y` share ownership or are both empty. — *end note*]

20.3.2.7 Class template `enable_shared_from_this` [util.smartptr.enab]

1 A class `T` can inherit from `enable_shared_from_this<T>` to inherit the `shared_from_this` member functions that obtain a `shared_ptr` instance pointing to `*this`.

2 [*Example 1*:

```
struct X: public enable_shared_from_this<X> { };

int main() {
  shared_ptr<X> p(new X);
  shared_ptr<X> q = p->shared_from_this();
  assert(p == q);
  assert(p.owner_equal(q));                          // p and q share ownership
}
```

— *end example*]

```
namespace std {
  template<class T> class enable_shared_from_this {
  protected:
    constexpr enable_shared_from_this() noexcept;
    constexpr enable_shared_from_this(const enable_shared_from_this&) noexcept;
    constexpr enable_shared_from_this& operator=(const enable_shared_from_this&) noexcept;
    constexpr ~enable_shared_from_this();

  public:
    constexpr shared_ptr<T> shared_from_this();
    constexpr shared_ptr<T const> shared_from_this() const;
    constexpr weak_ptr<T> weak_from_this() noexcept;
    constexpr weak_ptr<T const> weak_from_this() const noexcept;

  private:
    mutable weak_ptr<T> weak-this;  // exposition only
  };
}
```

3 The template parameter `T` of `enable_shared_from_this` may be an incomplete type.

```
constexpr enable_shared_from_this() noexcept;
constexpr enable_shared_from_this(const enable_shared_from_this<T>&) noexcept;
```

4 *Effects*: Value-initializes *weak-this*.

```
constexpr enable_shared_from_this<T>& operator=(const enable_shared_from_this<T>&) noexcept;
```

5 *Returns*: `*this`.

6 [*Note 1*: *weak-this* is not changed. — *end note*]

```
constexpr shared_ptr<T>       shared_from_this();
constexpr shared_ptr<T const> shared_from_this() const;
```

7 *Returns*: `shared_ptr<T>(`*weak-this*`)`.

```
constexpr weak_ptr<T>       weak_from_this() noexcept;
constexpr weak_ptr<T const> weak_from_this() const noexcept;
```

8 *Returns*: *weak-this*.

20.3.3 Smart pointer hash support [util.smartptr.hash]

```
template<class T, class D> struct hash<unique_ptr<T, D>>;
```

1 Letting `UP` be `unique_ptr<T, D>`, the specialization `hash<UP>` is enabled (22.10.19) if and only if `hash<typename UP::pointer>` is enabled. When enabled, for an object `p` of type `UP`, `hash<UP>()(p)` evaluates to the same value as `hash<typename UP::pointer>()(p.get())`. The member functions are not guaranteed to be `noexcept`.

```
template<class T> struct hash<shared_ptr<T>>;
```

2 For an object `p` of type `shared_ptr<T>`, `hash<shared_ptr<T>>()(p)` evaluates to the same value as `hash<typename shared_ptr<T>::element_type*>()(p.get())`.

20.3.4 Smart pointer adaptors [smartptr.adapt]

20.3.4.1 Class template `out_ptr_t` [out.ptr.t]

1 `out_ptr_t` is a class template used to adapt types such as smart pointers (20.3) for functions that use output pointer parameters.

2 [*Example 1*:

```
#include <memory>
#include <cstdio>

int fopen_s(std::FILE** f, const char* name, const char* mode);

struct fclose_deleter {
  void operator()(std::FILE* f) const noexcept {
    std::fclose(f);
  }
};

int main(int, char*[]) {
  constexpr const char* file_name = "ow.o";
  std::unique_ptr<std::FILE, fclose_deleter> file_ptr;
  int err = fopen_s(std::out_ptr<std::FILE*>(file_ptr), file_name, "r+b");
  if (err != 0)
    return 1;
  // *file_ptr is valid
  return 0;
}
```

`unique_ptr` can be used with `out_ptr` to be passed into an output pointer-style function, without needing to hold onto an intermediate pointer value and manually delete it on error or failure. — *end example*]

```
namespace std {
  template<class Smart, class Pointer, class... Args>
  class out_ptr_t {
  public:
    constexpr explicit out_ptr_t(Smart&, Args...);
    out_ptr_t(const out_ptr_t&) = delete;

    constexpr ~out_ptr_t();

    constexpr operator Pointer*() const noexcept;
    constexpr operator void**() const noexcept;

  private:
    Smart& s;                    // exposition only
    tuple<Args...> a;            // exposition only
    Pointer p;                   // exposition only
  };
}
```

3 `Pointer` shall meet the *Cpp17NullablePointer* requirements. If `Smart` is a specialization of `shared_ptr` and `sizeof...(Args) == 0`, the program is ill-formed.

[*Note 1*: It is typically a user error to reset a `shared_ptr` without specifying a deleter, as `shared_ptr` will replace a custom deleter upon usage of `reset`, as specified in 20.3.2.2.5. — *end note*]

4 Program-defined specializations of `out_ptr_t` that depend on at least one program-defined type need not meet the requirements for the primary template.

5 Evaluations of the conversion functions on the same object may conflict (6.9.2.2).

`constexpr explicit out_ptr_t(Smart& smart, Args... args);`

6 *Effects*: Initializes `s` with `smart`, `a` with `std::forward<Args>(args)...`, and value-initializes `p`. Then, equivalent to:

(6.1) — `s.reset();`

if the expression `s.reset()` is well-formed;

(6.2) — otherwise,

```
s = Smart();
```

if `is_constructible_v<Smart>` is `true`;

(6.3) — otherwise, the program is ill-formed.

7 [*Note 2*: The constructor is not `noexcept` to allow for a variety of non-terminating and safe implementation strategies. For example, an implementation can allocate a `shared_ptr`'s internal node in the constructor and let implementation-defined exceptions escape safely. The destructor can then move the allocated control block in directly and avoid any other exceptions. — *end note*]

`constexpr ~out_ptr_t();`

8 Let `SP` be *`POINTER_OF_OR`*`(Smart, Pointer)` (20.2.1).

9 *Effects*: Equivalent to:

(9.1) —
```
if (p) {
  apply([&](auto&&... args) {
    s.reset(static_cast<SP>(p), std::forward<Args>(args)...); }, std::move(a));
}
```

if the expression `s.reset(static_cast<SP>(p), std::forward<Args>(args)...)` is well-formed;

(9.2) — otherwise,

```
if (p) {
  apply([&](auto&&... args) {
    s = Smart(static_cast<SP>(p), std::forward<Args>(args)...); }, std::move(a));
}
```

if `is_constructible_v<Smart, SP, Args...>` is `true`;

(9.3) — otherwise, the program is ill-formed.

```
constexpr operator Pointer*() const noexcept;
```

10 *Preconditions*: `operator void**()` has not been called on `*this`.

11 *Returns*: `addressof(const_cast<Pointer&>(p))`.

```
constexpr operator void**() const noexcept;
```

12 *Constraints*: `is_same_v<Pointer, void*>` is `false`.

13 *Mandates*: `is_pointer_v<Pointer>` is `true`.

14 *Preconditions*: `operator Pointer*()` has not been called on `*this`.

15 *Returns*: A pointer value `v` such that:

(15.1) — the initial value `*v` is equivalent to `static_cast<void*>(p)` and

(15.2) — any modification of `*v` that is not followed by a subsequent modification of `*this` affects the value of `p` during the destruction of `*this`, such that `static_cast<void*>(p) == *v`.

16 *Remarks*: Accessing `*v` outside the lifetime of `*this` has undefined behavior.

17 [*Note 3*: `reinterpret_cast<void**>(static_cast<Pointer*>(*this))` can be a viable implementation strategy for some implementations. — *end note*]

20.3.4.2 Function template `out_ptr` [out.ptr]

```
template<class Pointer = void, class Smart, class... Args>
  constexpr auto out_ptr(Smart& s, Args&&... args);
```

1 Let `P` be `Pointer` if `is_void_v<Pointer>` is `false`, otherwise *`POINTER_OF`*`(Smart)`.

2 *Returns*: `out_ptr_t<Smart, P, Args&&...>(s, std::forward<Args>(args)...)`

20.3.4.3 Class template `inout_ptr_t` [inout.ptr.t]

1 `inout_ptr_t` is a class template used to adapt types such as smart pointers (20.3) for functions that use output pointer parameters whose dereferenced values may first be deleted before being set to another allocated value.

2 [*Example 1*:

```
#include <memory>

struct star_fish* star_fish_alloc();
int star_fish_populate(struct star_fish** ps, const char* description);

struct star_fish_deleter {
  void operator() (struct star_fish* c) const noexcept;
};

using star_fish_ptr = std::unique_ptr<star_fish, star_fish_deleter>;

int main(int, char*[]) {
  star_fish_ptr peach(star_fish_alloc());
  // ...
  // used, need to re-make
  int err = star_fish_populate(std::inout_ptr(peach), "caring clown-fish liker");
  return err;
}
```

A `unique_ptr` can be used with `inout_ptr` to be passed into an output pointer-style function. The original value will be properly deleted according to the function it is used with and a new value reset in its place. — *end example*]

```
namespace std {
  template<class Smart, class Pointer, class... Args>
  class inout_ptr_t {
  public:
    constexpr explicit inout_ptr_t(Smart&, Args...);
    inout_ptr_t(const inout_ptr_t&) = delete;
```

```
    constexpr ~inout_ptr_t();

    constexpr operator Pointer*() const noexcept;
    constexpr operator void**() const noexcept;

  private:
    Smart& s;                    // exposition only
    tuple<Args...> a;            // exposition only
    Pointer p;                   // exposition only
  };
}
```

3 `Pointer` shall meet the *Cpp17NullablePointer* requirements. If `Smart` is a specialization of `shared_ptr`, the program is ill-formed.

[*Note 1*: It is impossible to properly acquire unique ownership of the managed resource from a `shared_ptr` given its shared ownership model. — *end note*]

4 Program-defined specializations of `inout_ptr_t` that depend on at least one program-defined type need not meet the requirements for the primary template.

5 Evaluations of the conversion functions on the same object may conflict (6.9.2.2).

```
constexpr explicit inout_ptr_t(Smart& smart, Args... args);
```

6 *Effects*: Initializes `s` with `smart`, `a` with `std::forward<Args>(args)...`, and `p` to either

(6.1) — `smart` if `is_pointer_v<Smart>` is `true`,

(6.2) — otherwise, `smart.get()`.

7 *Remarks*: An implementation can call `s.release()`.

8 [*Note 2*: The constructor is not `noexcept` to allow for a variety of non-terminating and safe implementation strategies. For example, an intrusive pointer implementation with a control block can allocate in the constructor and safely fail with an exception. — *end note*]

```
constexpr ~inout_ptr_t();
```

9 Let `SP` be *`POINTER_OF_OR`*`(Smart, Pointer)` (20.2.1).

10 Let *`release-statement`* be `s.release();` if an implementation does not call `s.release()` in the constructor. Otherwise, it is empty.

11 *Effects*: Equivalent to:

(11.1) —
```
apply([&](auto&&... args) {
  s = Smart(static_cast<SP>(p), std::forward<Args>(args)...); }, std::move(a));
```
if `is_pointer_v<Smart>` is `true`;

(11.2) — otherwise,
```
release-statement;
if (p) {
  apply([&](auto&&... args) {
    s.reset(static_cast<SP>(p), std::forward<Args>(args)...); }, std::move(a));
}
```
if the expression `s.reset(static_cast<SP>(p), std::forward<Args>(args)...)` is well-formed;

(11.3) — otherwise,
```
release-statement;
if (p) {
  apply([&](auto&&... args) {
    s = Smart(static_cast<SP>(p), std::forward<Args>(args)...); }, std::move(a));
}
```
if `is_constructible_v<Smart, SP, Args...>` is `true`;

(11.4) — otherwise, the program is ill-formed.

```
constexpr operator Pointer*() const noexcept;
```

12 *Preconditions*: `operator void**()` has not been called on `*this`.

13 *Returns*: `addressof(const_cast<Pointer&>(p))`.

```
constexpr operator void**() const noexcept;
```

14 *Constraints*: `is_same_v<Pointer, void*>` is `false`.

15 *Mandates*: `is_pointer_v<Pointer>` is `true`.

16 *Preconditions*: `operator Pointer*()` has not been called on `*this`.

17 *Returns*: A pointer value `v` such that:

(17.1) — the initial value `*v` is equivalent to `static_cast<void*>(p)` and

(17.2) — any modification of `*v` that is not followed by subsequent modification of `*this` affects the value of `p` during the destruction of `*this`, such that `static_cast<void*>(p) == *v`.

18 *Remarks*: Accessing `*v` outside the lifetime of `*this` has undefined behavior.

19 [*Note 3*: `reinterpret_cast<void**>(static_cast<Pointer*>(*this))` can be a viable implementation strategy for some implementations. — *end note*]

20.3.4.4 Function template `inout_ptr` [inout.ptr]

```
template<class Pointer = void, class Smart, class... Args>
  constexpr auto inout_ptr(Smart& s, Args&&... args);
```

1 Let `P` be `Pointer` if `is_void_v<Pointer>` is `false`, otherwise *`POINTER_OF`*`(Smart)`.

2 *Returns*: `inout_ptr_t<Smart, P, Args&&...>(s, std::forward<Args>(args)...)`.

20.4 Types for composite class design [mem.composite.types]

20.4.1 Class template `indirect` [indirect]

20.4.1.1 General [indirect.general]

1 An indirect object manages the lifetime of an owned object. An indirect object is *valueless* if it has no owned object. An indirect object may become valueless only after it has been moved from.

2 In every specialization `indirect<T, Allocator>`, if the type `allocator_traits<Allocator>::value_type` is not the same type as `T`, the program is ill-formed. Every object of type `indirect<T, Allocator>` uses an object of type `Allocator` to allocate and free storage for the owned object as needed.

3 Constructing an owned object with `args...` using the allocator `a` means calling `allocator_traits<Allocator>::construct(a, `*`p`*`, args...)` where `args` is an expression pack, `a` is an allocator, and *`p`* is a pointer obtained by calling `allocator_traits<Allocator>::allocate`.

4 The member *`alloc`* is used for any memory allocation and element construction performed by member functions during the lifetime of each indirect object. The allocator *`alloc`* may be replaced only via assignment or `swap()`. `Allocator` replacement is performed by copy assignment, move assignment, or swapping of the allocator only if (23.2.2.2):

(4.1) — `allocator_traits<Allocator>::propagate_on_container_copy_assignment::value`, or

(4.2) — `allocator_traits<Allocator>::propagate_on_container_move_assignment::value`, or

(4.3) — `allocator_traits<Allocator>::propagate_on_container_swap::value`

is `true` within the implementation of the corresponding `indirect` operation.

5 A program that instantiates the definition of the template `indirect<T, Allocator>` with a type for the `T` parameter that is a non-object type, an array type, `in_place_t`, a specialization of `in_place_type_t`, or a cv-qualified type is ill-formed.

6 The template parameter `T` of `indirect` may be an incomplete type.

7 The template parameter `Allocator` of `indirect` shall meet the Cpp17Allocator requirements.

8 If a program declares an explicit or partial specialization of `indirect`, the behavior is undefined.

[3] [*Example 1*:

```
template<typename T> class atomic_list {
  struct node {
    T t;
    shared_ptr<node> next;
  };
  atomic<shared_ptr<node>> head;

public:
  shared_ptr<node> find(T t) const {
    auto p = head.load();
    while (p && p->t != t)
      p = p->next;

    return p;
  }

  void push_front(T t) {
    auto p = make_shared<node>();
    p->t = t;
    p->next = head;
    while (!head.compare_exchange_weak(p->next, p)) {}
  }
};
```

— *end example*]

32.5.8.7.2 Partial specialization for shared_ptr [util.smartptr.atomic.shared]

```
namespace std {
  template<class T> struct atomic<shared_ptr<T>> {
    using value_type = shared_ptr<T>;

    static constexpr bool is_always_lock_free = implementation-defined;
    bool is_lock_free() const noexcept;

    constexpr atomic() noexcept;
    constexpr atomic(nullptr_t) noexcept : atomic() { }
    constexpr atomic(shared_ptr<T> desired) noexcept;
    atomic(const atomic&) = delete;
    void operator=(const atomic&) = delete;

    constexpr shared_ptr<T> load(memory_order order = memory_order::seq_cst) const noexcept;
    constexpr operator shared_ptr<T>() const noexcept;
    constexpr void store(shared_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
    constexpr void operator=(shared_ptr<T> desired) noexcept;
    constexpr void operator=(nullptr_t) noexcept;

    constexpr shared_ptr<T> exchange(shared_ptr<T> desired,
                                     memory_order order = memory_order::seq_cst) noexcept;
    constexpr bool compare_exchange_weak(shared_ptr<T>& expected, shared_ptr<T> desired,
                                         memory_order success, memory_order failure) noexcept;
    constexpr bool compare_exchange_strong(shared_ptr<T>& expected, shared_ptr<T> desired,
                                           memory_order success, memory_order failure) noexcept;
    constexpr bool compare_exchange_weak(shared_ptr<T>& expected, shared_ptr<T> desired,
                                         memory_order order = memory_order::seq_cst) noexcept;
    constexpr bool compare_exchange_strong(shared_ptr<T>& expected, shared_ptr<T> desired,
                                           memory_order order = memory_order::seq_cst) noexcept;

    constexpr void wait(shared_ptr<T> old, memory_order order = memory_order::seq_cst) const noexcept;
    constexpr void notify_one() noexcept;
    constexpr void notify_all() noexcept;

  private:
    shared_ptr<T> p;            // exposition only
```

```
  };
}
```

```
constexpr atomic() noexcept;
```

1 *Effects*: Value-initializes `p`.

```
constexpr atomic(shared_ptr<T> desired) noexcept;
```

2 *Effects*: Initializes the object with the value `desired`. Initialization is not an atomic operation (6.9.2).

[*Note 1*: It is possible to have an access to an atomic object `A` race with its construction, for example, by communicating the address of the just-constructed object `A` to another thread via `memory_order::relaxed` operations on a suitable atomic pointer variable, and then immediately accessing `A` in the receiving thread. This results in undefined behavior. — *end note*]

```
constexpr void store(shared_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
```

3 *Preconditions*: `order` is `memory_order::relaxed`, `memory_order::release`, or `memory_order::seq_cst`.

4 *Effects*: Atomically replaces the value pointed to by `this` with the value of `desired` as if by `p.swap(desired)`. Memory is affected according to the value of `order`.

```
constexpr void operator=(shared_ptr<T> desired) noexcept;
```

5 *Effects*: Equivalent to `store(desired)`.

```
constexpr void operator=(nullptr_t) noexcept;
```

6 *Effects*: Equivalent to `store(nullptr)`.

```
constexpr shared_ptr<T> load(memory_order order = memory_order::seq_cst) const noexcept;
```

7 *Preconditions*: `order` is `memory_order::relaxed`, `memory_order::acquire`, or `memory_order::seq_cst`.

8 *Effects*: Memory is affected according to the value of `order`.

9 *Returns*: Atomically returns `p`.

```
constexpr operator shared_ptr<T>() const noexcept;
```

10 *Effects*: Equivalent to: `return load();`

```
constexpr shared_ptr<T> exchange(shared_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
```

11 *Effects*: Atomically replaces `p` with `desired` as if by `p.swap(desired)`. Memory is affected according to the value of `order`. This is an atomic read-modify-write operation (6.9.2.2).

12 *Returns*: Atomically returns the value of `p` immediately before the effects.

```
constexpr bool compare_exchange_weak(shared_ptr<T>& expected, shared_ptr<T> desired,
                                     memory_order success, memory_order failure) noexcept;
constexpr bool compare_exchange_strong(shared_ptr<T>& expected, shared_ptr<T> desired,
                                       memory_order success, memory_order failure) noexcept;
```

13 *Preconditions*: `failure` is `memory_order::relaxed`, `memory_order::acquire`, or `memory_order::seq_cst`.

14 *Effects*: If `p` is equivalent to `expected`, assigns `desired` to `p` and has synchronization semantics corresponding to the value of `success`, otherwise assigns `p` to `expected` and has synchronization semantics corresponding to the value of `failure`.

15 *Returns*: `true` if `p` was equivalent to `expected`, `false` otherwise.

16 *Remarks*: Two `shared_ptr` objects are equivalent if they store the same pointer value and either share ownership or are both empty. The weak form may fail spuriously. See 32.5.8.2.

17 If the operation returns `true`, `expected` is not accessed after the atomic update and the operation is an atomic read-modify-write operation (6.9.2) on the memory pointed to by `this`. Otherwise, the operation is an atomic load operation on that memory, and `expected` is updated with the existing value read from the atomic object in the attempted atomic update. The `use_count` update corresponding to

the write to `expected` is part of the atomic operation. The write to `expected` itself is not required to be part of the atomic operation.

```
constexpr bool compare_exchange_weak(shared_ptr<T>& expected, shared_ptr<T> desired,
                                     memory_order order = memory_order::seq_cst) noexcept;
```

18 *Effects*: Equivalent to:

```
return compare_exchange_weak(expected, desired, order, fail_order);
```

where `fail_order` is the same as `order` except that a value of `memory_order::acq_rel` shall be replaced by the value `memory_order::acquire` and a value of `memory_order::release` shall be replaced by the value `memory_order::relaxed`.

```
constexpr bool compare_exchange_strong(shared_ptr<T>& expected, shared_ptr<T> desired,
                                       memory_order order = memory_order::seq_cst) noexcept;
```

19 *Effects*: Equivalent to:

```
return compare_exchange_strong(expected, desired, order, fail_order);
```

where `fail_order` is the same as `order` except that a value of `memory_order::acq_rel` shall be replaced by the value `memory_order::acquire` and a value of `memory_order::release` shall be replaced by the value `memory_order::relaxed`.

```
constexpr void wait(shared_ptr<T> old, memory_order order = memory_order::seq_cst) const noexcept;
```

20 *Preconditions*: `order` is `memory_order::relaxed`, `memory_order::acquire`, or `memory_order::seq_cst`.

21 *Effects*: Repeatedly performs the following steps, in order:

(21.1) — Evaluates `load(order)` and compares it to `old`.

(21.2) — If the two are not equivalent, returns.

(21.3) — Blocks until it is unblocked by an atomic notifying operation or is unblocked spuriously.

22 *Remarks*: Two `shared_ptr` objects are equivalent if they store the same pointer and either share ownership or are both empty. This function is an atomic waiting operation (32.5.6).

```
constexpr void notify_one() noexcept;
```

23 *Effects*: Unblocks the execution of at least one atomic waiting operation that is eligible to be unblocked (32.5.6) by this call, if any such atomic waiting operations exist.

24 *Remarks*: This function is an atomic notifying operation (32.5.6).

```
constexpr void notify_all() noexcept;
```

25 *Effects*: Unblocks the execution of all atomic waiting operations that are eligible to be unblocked (32.5.6) by this call.

26 *Remarks*: This function is an atomic notifying operation (32.5.6).

32.5.8.7.3 Partial specialization for `weak_ptr` [util.smartptr.atomic.weak]

```
namespace std {
  template<class T> struct atomic<weak_ptr<T>> {
    using value_type = weak_ptr<T>;

    static constexpr bool is_always_lock_free = implementation-defined;
    bool is_lock_free() const noexcept;

    constexpr atomic() noexcept;
    constexpr atomic(weak_ptr<T> desired) noexcept;
    atomic(const atomic&) = delete;
    constexpr void operator=(const atomic&) = delete;

    constexpr weak_ptr<T> load(memory_order order = memory_order::seq_cst) const noexcept;
    constexpr operator weak_ptr<T>() const noexcept;
    constexpr void store(weak_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
    constexpr void operator=(weak_ptr<T> desired) noexcept;
```

```
    constexpr weak_ptr<T> exchange(weak_ptr<T> desired,
                                   memory_order order = memory_order::seq_cst) noexcept;
    constexpr bool compare_exchange_weak(weak_ptr<T>& expected, weak_ptr<T> desired,
                                         memory_order success, memory_order failure) noexcept;
    constexpr bool compare_exchange_strong(weak_ptr<T>& expected, weak_ptr<T> desired,
                                           memory_order success, memory_order failure) noexcept;
    constexpr bool compare_exchange_weak(weak_ptr<T>& expected, weak_ptr<T> desired,
                                         memory_order order = memory_order::seq_cst) noexcept;
    constexpr bool compare_exchange_strong(weak_ptr<T>& expected, weak_ptr<T> desired,
                                           memory_order order = memory_order::seq_cst) noexcept;

    constexpr void wait(weak_ptr<T> old, memory_order order = memory_order::seq_cst) const noexcept;
    constexpr void notify_one() noexcept;
    constexpr void notify_all() noexcept;

  private:
    weak_ptr<T> p;              // exposition only
  };
}
```

`constexpr atomic() noexcept;`

1 *Effects*: Value-initializes `p`.

`constexpr atomic(weak_ptr<T> desired) noexcept;`

2 *Effects*: Initializes the object with the value `desired`. Initialization is not an atomic operation (6.9.2).

[*Note 1*: It is possible to have an access to an atomic object `A` race with its construction, for example, by communicating the address of the just-constructed object `A` to another thread via `memory_order::relaxed` operations on a suitable atomic pointer variable, and then immediately accessing `A` in the receiving thread. This results in undefined behavior. — *end note*]

`constexpr void store(weak_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;`

3 *Preconditions*: `order` is `memory_order::relaxed`, `memory_order::release`, or `memory_order::seq_cst`.

4 *Effects*: Atomically replaces the value pointed to by `this` with the value of `desired` as if by `p.swap(desired)`. Memory is affected according to the value of `order`.

`constexpr void operator=(weak_ptr<T> desired) noexcept;`

5 *Effects*: Equivalent to `store(desired)`.

`constexpr weak_ptr<T> load(memory_order order = memory_order::seq_cst) const noexcept;`

6 *Preconditions*: `order` is `memory_order::relaxed`, `memory_order::acquire`, or `memory_order::seq_cst`.

7 *Effects*: Memory is affected according to the value of `order`.

8 *Returns*: Atomically returns `p`.

`constexpr operator weak_ptr<T>() const noexcept;`

9 *Effects*: Equivalent to: `return load();`

`constexpr weak_ptr<T> exchange(weak_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;`

10 *Effects*: Atomically replaces `p` with `desired` as if by `p.swap(desired)`. Memory is affected according to the value of `order`. This is an atomic read-modify-write operation (6.9.2.2).

11 *Returns*: Atomically returns the value of `p` immediately before the effects.

```
constexpr bool compare_exchange_weak(weak_ptr<T>& expected, weak_ptr<T> desired,
                                     memory_order success, memory_order failure) noexcept;
constexpr bool compare_exchange_strong(weak_ptr<T>& expected, weak_ptr<T> desired,
                                       memory_order success, memory_order failure) noexcept;
```

12 *Preconditions*: `failure` is `memory_order::relaxed`, `memory_order::acquire`, or `memory_order::seq_cst`.

13 *Effects*: If `p` is equivalent to `expected`, assigns `desired` to `p` and has synchronization semantics corresponding to the value of `success`, otherwise assigns `p` to `expected` and has synchronization semantics corresponding to the value of `failure`.

14 *Returns*: `true` if `p` was equivalent to `expected`, `false` otherwise.

15 *Remarks*: Two `weak_ptr` objects are equivalent if they store the same pointer value and either share ownership or are both empty. The weak form may fail spuriously. See 32.5.8.2.

16 If the operation returns `true`, `expected` is not accessed after the atomic update and the operation is an atomic read-modify-write operation (6.9.2) on the memory pointed to by `this`. Otherwise, the operation is an atomic load operation on that memory, and `expected` is updated with the existing value read from the atomic object in the attempted atomic update. The `use_count` update corresponding to the write to `expected` is part of the atomic operation. The write to `expected` itself is not required to be part of the atomic operation.

```
constexpr bool compare_exchange_weak(weak_ptr<T>& expected, weak_ptr<T> desired,
                             memory_order order = memory_order::seq_cst) noexcept;
```

17 *Effects*: Equivalent to:

```
return compare_exchange_weak(expected, desired, order, fail_order);
```

where `fail_order` is the same as `order` except that a value of `memory_order::acq_rel` shall be replaced by the value `memory_order::acquire` and a value of `memory_order::release` shall be replaced by the value `memory_order::relaxed`.

```
constexpr bool compare_exchange_strong(weak_ptr<T>& expected, weak_ptr<T> desired,
                               memory_order order = memory_order::seq_cst) noexcept;
```

18 *Effects*: Equivalent to:

```
return compare_exchange_strong(expected, desired, order, fail_order);
```

where `fail_order` is the same as `order` except that a value of `memory_order::acq_rel` shall be replaced by the value `memory_order::acquire` and a value of `memory_order::release` shall be replaced by the value `memory_order::relaxed`.

```
constexpr void wait(weak_ptr<T> old, memory_order order = memory_order::seq_cst) const noexcept;
```

19 *Preconditions*: `order` is `memory_order::relaxed`, `memory_order::acquire`, or `memory_order::seq_cst`.

20 *Effects*: Repeatedly performs the following steps, in order:

(20.1) — Evaluates `load(order)` and compares it to `old`.

(20.2) — If the two are not equivalent, returns.

(20.3) — Blocks until it is unblocked by an atomic notifying operation or is unblocked spuriously.

21 *Remarks*: Two `weak_ptr` objects are equivalent if they store the same pointer and either share ownership or are both empty. This function is an atomic waiting operation (32.5.6).

```
constexpr void notify_one() noexcept;
```

22 *Effects*: Unblocks the execution of at least one atomic waiting operation that is eligible to be unblocked (32.5.6) by this call, if any such atomic waiting operations exist.

23 *Remarks*: This function is an atomic notifying operation (32.5.6).

```
constexpr void notify_all() noexcept;
```

24 *Effects*: Unblocks the execution of all atomic waiting operations that are eligible to be unblocked (32.5.6) by this call.

25 *Remarks*: This function is an atomic notifying operation (32.5.6).

32.5.9 Non-member functions [atomics.nonmembers]

1 A non-member function template whose name matches the pattern `atomic_`*f* or the pattern `atomic_`*f*`_explicit` invokes the member function *f*, with the value of the first parameter as the object expression and the values of the remaining parameters (if any) as the arguments of the member function call, in order. An

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 BSI Panel, SG7, LEWG, and online forum discussions.

8 References

[P0784R7] Peter Dimov, Louis Dionne, Nina Ranns, Richard Smith, Daveed Vandevoorde, *More constexpr containers* (2019)
https://wg21.link/P0784R7

[P2738R1] Corentin Jabot, David Ledger, *`constexpr` cast from `void*`: towards `constexpr` type-erasure* (2023)
https://wg21.link/P2738R1

[P2448R2] Barry Revzin, *Relaxing some `constexpr` restrictions* (2022)
https://wg21.link/P2448R2

[P2273R3] Andreas Fertig, *Making `std::unique_ptr` constexpr* (2021)
https://wg21.link/P2273R3

[P3309R3] Hana Dusíková, *constexpr atomic<T> and atomic_ref<T>* (2024)
https://wg21.link/P3309R3

[P3068R6] Hana Dusíková, *Allowing exception throwing in constant-evaluation* (2024)
https://wg21.link/P3068R6

[ClangOz] Paul Keir, Andrew Gozillon, *ClangOz: Parallel constant evaluation of C++ map and reduce operations* (2024)
https://doi.org/10.1016/j.cola.2024.101298