Document Number: P1460R0
Date: 2020-01-13
Reply to: Marshall Clow
mclow.lists@gmail.com
Alberto Barbati
ganesh@barbati.net

Mandating the Standard Library: Clause 20 - Utilities library

With the adoption of P0788R3, we have a new way of specifying requirements for the library clauses of the standard. This is one of a series of papers reformulating the requirements into the new format. This effort was strongly influenced by the informational paper P1369R0.

The changes in this series of papers fall into three broad categories.

— Change "participate in overload resolution" wording into "Constraints" elements

— Change "Requires" elements into either "Mandates" or "Expects", depending (mostly) on whether or not they can be checked at compile time.

— Drive-by fixes (hopefully very few)

This paper covers Clause 20 (Utilities), and is based on N4842.

This is a different paper than N1462 (Mandating the Standard Library: Clause 20 - Strings library), because when I started this project, "Utilities" was in clause 19.

Drive-by fixes:

— a couple of places that said "The constructor of Foo constructs an object of class Foo" have been removed

Help for the editors: The changes here can be viewed as latex sources with the following commands

```
git clone git@github.com:mclow/mandate.git
cd mandate
git diff master..chapter19 utilities.tex
```

20 General utilities library [utilities]

20.1 General [utilities.general]

1 This Clause describes utilities that are generally useful in C++ programs; some of these utilities are used by other elements of the C++ standard library. These utilities are summarized in Table 39.

Table 39: General utilities library summary [tab:utilities.summary]

	Subclause	Header
20.2	Utility components	`<utility>`
20.3	Compile-time integer sequences	
20.4	Pairs	
20.5	Tuples	`<tuple>`
20.6	Optional objects	`<optional>`
20.7	Variants	`<variant>`
20.8	Storage for any type	`<any>`
20.9	Fixed-size sequences of bits	`<bitset>`
20.10	Memory	`<cstdlib>`, `<memory>`
20.11	Smart pointers	`<memory>`
20.12	Memory resources	`<memory_resource>`
20.13	Scoped allocators	`<scoped_allocator>`
20.14	Function objects	`<functional>`
20.15	Type traits	`<type_traits>`
20.16	Compile-time rational arithmetic	`<ratio>`
20.17	Type indexes	`<typeindex>`
20.18	Execution policies	`<execution>`
20.19	Primitive numeric conversions	`<charconv>`
20.20	Formatting	`<format>`

20.2 Utility components [utility]

20.2.1 Header `<utility>` synopsis [utility.syn]

1 The header `<utility>` contains some basic function and class templates that are used throughout the rest of the library.

```
#include <initializer_list>      // see ??

namespace std {
  // 20.2.2, swap
  template<class T>
    constexpr void swap(T& a, T& b) noexcept(see below);
  template<class T, size_t N>
    constexpr void swap(T (&a)[N], T (&b)[N]) noexcept(is_nothrow_swappable_v<T>);

  // 20.2.3, exchange
  template<class T, class U = T>
    constexpr T exchange(T& obj, U&& new_val);

  // 20.2.4, forward/move
  template<class T>
    constexpr T&& forward(remove_reference_t<T>& t) noexcept;
  template<class T>
    constexpr T&& forward(remove_reference_t<T>&& t) noexcept;
  template<class T>
    constexpr remove_reference_t<T>&& move(T&&) noexcept;
```

```
template<class T>
  constexpr conditional_t<
      !is_nothrow_move_constructible_v<T> && is_copy_constructible_v<T>, const T&, T&&>
    move_if_noexcept(T& x) noexcept;

// 20.2.5, as_const
template<class T>
  constexpr add_const_t<T>& as_const(T& t) noexcept;
template<class T>
  void as_const(const T&&) = delete;

// 20.2.6, declval
template<class T>
  add_rvalue_reference_t<T> declval() noexcept;    // as unevaluated operand

// 20.3, Compile-time integer sequences
template<class T, T...>
  struct integer_sequence;
template<size_t... I>
  using index_sequence = integer_sequence<size_t, I...>;

template<class T, T N>
  using make_integer_sequence = integer_sequence<T, see below>;
template<size_t N>
  using make_index_sequence = make_integer_sequence<size_t, N>;

template<class... T>
  using index_sequence_for = make_index_sequence<sizeof...(T)>;

// 20.4, class template pair
template<class T1, class T2>
  struct pair;

// 20.4.3, pair specialized algorithms
template<class T1, class T2>
  constexpr void swap(pair<T1, T2>& x, pair<T1, T2>& y) noexcept(noexcept(x.swap(y)));

template<class T1, class T2>
  constexpr see below make_pair(T1&&, T2&&);

// 20.4.4, tuple-like access to pair
template<class T> struct tuple_size;
template<size_t I, class T> struct tuple_element;

template<class T1, class T2> struct tuple_size<pair<T1, T2>>;
template<size_t I, class T1, class T2> struct tuple_element<I, pair<T1, T2>>;

template<size_t I, class T1, class T2>
  constexpr tuple_element_t<I, pair<T1, T2>>& get(pair<T1, T2>&) noexcept;
template<size_t I, class T1, class T2>
  constexpr tuple_element_t<I, pair<T1, T2>>&& get(pair<T1, T2>&&) noexcept;
template<size_t I, class T1, class T2>
  constexpr const tuple_element_t<I, pair<T1, T2>>& get(const pair<T1, T2>&) noexcept;
template<size_t I, class T1, class T2>
  constexpr const tuple_element_t<I, pair<T1, T2>>&& get(const pair<T1, T2>&&) noexcept;
template<class T1, class T2>
  constexpr T1& get(pair<T1, T2>& p) noexcept;
template<class T1, class T2>
  constexpr const T1& get(const pair<T1, T2>& p) noexcept;
template<class T1, class T2>
  constexpr T1&& get(pair<T1, T2>&& p) noexcept;
template<class T1, class T2>
  constexpr const T1&& get(const pair<T1, T2>&& p) noexcept;
```

```
  template<class T2, class T1>
    constexpr T2& get(pair<T1, T2>& p) noexcept;
  template<class T2, class T1>
    constexpr const T2& get(const pair<T1, T2>& p) noexcept;
  template<class T2, class T1>
    constexpr T2&& get(pair<T1, T2>&& p) noexcept;
  template<class T2, class T1>
    constexpr const T2&& get(const pair<T1, T2>&& p) noexcept;

  // 20.4.5, pair piecewise construction
  struct piecewise_construct_t {
    explicit piecewise_construct_t() = default;
  };
  inline constexpr piecewise_construct_t piecewise_construct{};
  template<class... Types> class tuple;          // defined in <tuple> (20.5.2)

  // in-place construction
  struct in_place_t {
    explicit in_place_t() = default;
  };
  inline constexpr in_place_t in_place{};
  template<class T>
    struct in_place_type_t {
      explicit in_place_type_t() = default;
    };
  template<class T> inline constexpr in_place_type_t<T> in_place_type{};
  template<size_t I>
    struct in_place_index_t {
      explicit in_place_index_t() = default;
    };
  template<size_t I> inline constexpr in_place_index_t<I> in_place_index{};
}
```

20.2.2 swap [utility.swap]

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

1 *Remarks:* This function is a designated customization point (??) and shall not participate in overload resolution unless `is_move_constructible_v<T>` is `true` and `is_move_assignable_v<T>` is `true`. The expression inside `noexcept` is equivalent to:

```
is_nothrow_move_constructible_v<T> && is_nothrow_move_assignable_v<T>
```

Constraints: `is_move_constructible_v<T>` is `true` and `is_move_assignable_v<T>` is `true`.

2 ~~*Requires:*~~ *Preconditions:* Type `T` ~~shall be~~is *Cpp17MoveConstructible* (Table ??) and *Cpp17MoveAssignable* (Table ??).

3 *Effects:* Exchanges values stored in two locations.

Remarks: This function is a designated customization point (??). The expression inside `noexcept` is equivalent to:

```
is_nothrow_move_constructible_v<T> && is_nothrow_move_assignable_v<T>
```

```
template<class T, size_t N>
  constexpr void swap(T (&a)[N], T (&b)[N]) noexcept(is_nothrow_swappable_v<T>);
```

4 ~~*Remarks:*~~ *Constraints:* ~~This function shall not participate in overload resolution unless~~ `is_swappable_v<T>` is `true`.

5 ~~*Requires:*~~ *Preconditions:* `a[i]` ~~shall be~~is swappable with (??) `b[i]` for all `i` in the range `[0, N)`.

6 *Effects:* As if by `swap_ranges(a, a + N, b)`.

20.2.3 exchange [utility.exchange]

```
template<class T, class U = T>
```

```
constexpr T exchange(T& obj, U&& new_val);
```

1 *Effects:* Equivalent to:

```
T old_val = std::move(obj);
obj = std::forward<U>(new_val);
return old_val;
```

20.2.4 Forward/move helpers [forward]

1 The library provides templated helper functions to simplify applying move semantics to an lvalue and to simplify the implementation of forwarding functions. All functions specified in this subclause are signal-safe (**??**).

```
template<class T> constexpr T&& forward(remove_reference_t<T>& t) noexcept;
template<class T> constexpr T&& forward(remove_reference_t<T>&& t) noexcept;
```

2 *Mandates:* In the second form, `is_lvalue_reference_v<T>` is `false`.

Returns: `static_cast<T&&>(t)`.

3 ~~*Remarks:* If the second form is instantiated with an lvalue reference type, the program is ill-formed.~~

4 [*Example*:

```
template<class T, class A1, class A2>
shared_ptr<T> factory(A1&& a1, A2&& a2) {
  return shared_ptr<T>(new T(std::forward<A1>(a1), std::forward<A2>(a2)));
}

struct A {
  A(int&, const double&);
};

void g() {
  shared_ptr<A> sp1 = factory<A>(2, 1.414); // error: 2 will not bind to int&
  int i = 2;
  shared_ptr<A> sp2 = factory<A>(i, 1.414); // OK
}
```

In the first call to `factory`, `A1` is deduced as `int`, so 2 is forwarded to `A`'s constructor as an rvalue. In the second call to `factory`, `A1` is deduced as `int&`, so `i` is forwarded to `A`'s constructor as an lvalue. In both cases, `A2` is deduced as `double`, so 1.414 is forwarded to `A`'s constructor as an rvalue. — *end example*]

```
template<class T> constexpr remove_reference_t<T>&& move(T&& t) noexcept;
```

5 *Returns:* `static_cast<remove_reference_t<T>&&>(t)`.

6 [*Example*:

```
template<class T, class A1>
shared_ptr<T> factory(A1&& a1) {
  return shared_ptr<T>(new T(std::forward<A1>(a1)));
}

struct A {
  A();
  A(const A&);      // copies from lvalues
  A(A&&);           // moves from rvalues
};

void g() {
  A a;
  shared_ptr<A> sp1 = factory<A>(a);                // "a" binds to A(const A&)
  shared_ptr<A> sp1 = factory<A>(std::move(a));     // "a" binds to A(A&&)
}
```

In the first call to `factory`, `A1` is deduced as `A&`, so `a` is forwarded as a non-const lvalue. This binds to the constructor `A(const A&)`, which copies the value from `a`. In the second call to `factory`, because of

the call `std::move(a)`, `A1` is deduced as `A`, so `a` is forwarded as an rvalue. This binds to the constructor `A(A&&)`, which moves the value from `a`. — *end example*]

```
template<class T> constexpr conditional_t<
    !is_nothrow_move_constructible_v<T> && is_copy_constructible_v<T>, const T&, T&&>
  move_if_noexcept(T& x) noexcept;
```

7 *Returns:* `std::move(x)`.

20.2.5 Function template `as_const` [utility.as.const]

```
template<class T> constexpr add_const_t<T>& as_const(T& t) noexcept;
```

1 *Returns:* `t`.

20.2.6 Function template `declval` [declval]

1 The library provides the function template `declval` to simplify the definition of expressions which occur as unevaluated operands (??).

```
template<class T> add_rvalue_reference_t<T> declval() noexcept;      // as unevaluated operand
```

2 ~~*Remarks:* If this function is odr-used (??), the program is ill-formed.~~

Mandates: This function shall not be odr-used (??).

3 *Remarks:* The template parameter `T` of `declval` may be an incomplete type.

4 [*Example*:

```
template<class To, class From> decltype(static_cast<To>(declval<From>())) convert(From&&);
```

declares a function template `convert` which only participates in overloading if the type `From` can be explicitly converted to type `To`. For another example see class template `common_type` (20.15.7.6). — *end example*]

20.3 Compile-time integer sequences [intseq]

20.3.1 In general [intseq.general]

1 The library provides a class template that can represent an integer sequence. When used as an argument to a function template the template parameter pack defining the sequence can be deduced and used in a pack expansion. [*Note*: The `index_sequence` alias template is provided for the common case of an integer sequence of type `size_t`; see also 20.5.5. — *end note*]

20.3.2 Class template `integer_sequence` [intseq.intseq]

```
namespace std {
  template<class T, T... I> struct integer_sequence {
    using value_type = T;
    static constexpr size_t size() noexcept { return sizeof...(I); }
  };
}
```

1 *Mandates:* `T` ~~shall be~~is an integer type.

20.3.3 Alias template `make_integer_sequence` [intseq.make]

```
template<class T, T N>
  using make_integer_sequence = integer_sequence<T, see below>;
```

1 *Mandates:* `N >= 0`.

2 ~~If `N` is negative the program is ill-formed.~~ The alias template `make_integer_sequence` denotes a specialization of `integer_sequence` with `N` template non-type arguments. The type `make_integer_sequence<T, N>` denotes the type `integer_sequence<T, 0, 1, ..., N-1>`. [*Note*: `make_integer_sequence<int, 0>` denotes the type `integer_sequence<int>`. — *end note*]

20.4 Pairs [pairs]

20.4.1 In general [pairs.general]

1 The library provides a template for heterogeneous pairs of values. The library also provides a matching function template to simplify their construction and several templates that provide access to `pair` objects as if they were `tuple` objects (see 20.5.6 and 20.5.7).

20.4.2 Class template `pair` [pairs.pair]

```
namespace std {
  template<class T1, class T2>
  struct pair {
    using first_type  = T1;
    using second_type = T2;

    T1 first;
    T2 second;

    pair(const pair&) = default;
    pair(pair&&) = default;
    constexpr explicit(see below) pair();
    constexpr explicit(see below) pair(const T1& x, const T2& y);
    template<class U1, class U2>
      constexpr explicit(see below) pair(U1&& x, U2&& y);
    template<class U1, class U2>
      constexpr explicit(see below) pair(const pair<U1, U2>& p);
    template<class U1, class U2>
      constexpr explicit(see below) pair(pair<U1, U2>&& p);
    template<class... Args1, class... Args2>
      constexpr pair(piecewise_construct_t,
                     tuple<Args1...> first_args, tuple<Args2...> second_args);

    constexpr pair& operator=(const pair& p);
    template<class U1, class U2>
      constexpr pair& operator=(const pair<U1, U2>& p);
    constexpr pair& operator=(pair&& p) noexcept(see below);
    template<class U1, class U2>
      constexpr pair& operator=(pair<U1, U2>&& p);

    constexpr void swap(pair& p) noexcept(see below);

    // 20.4.3, pair specialized algorithms
    friend constexpr bool operator==(const pair&, const pair&) = default;
    friend constexpr bool operator==(const pair& x, const pair& y)
      requires (is_reference_v<T1> || is_reference_v<T2>)
      { return x.first == y.first && x.second == y.second; }
    friend constexpr common_comparison_category_t<synth-three-way-result<T1>,
                                                  synth-three-way-result<T2>>
      operator<=>(const pair& x, const pair& y) { see below }
  };

  template<class T1, class T2>
    pair(T1, T2) -> pair<T1, T2>;
}
```

1 Constructors and member functions of `pair` do not throw exceptions unless one of the element-wise operations specified to be called for that operation throws an exception.

2 The defaulted move and copy constructor, respectively, of `pair` is a constexpr function if and only if all required element-wise initializations for copy and move, respectively, would satisfy the requirements for a constexpr function.

3 If `(is_trivially_destructible_v<T1> && is_trivially_destructible_v<T2>)` is `true`, then the destructor of `pair` is trivial.

```
constexpr explicit(see below) pair();
```

4 *Constraints:* `is_default_constructible_v<first_type>` is `true` and `is_default_constructible_v<second_type>` is `true`.

Effects: Value-initializes `first` and `second`.

5 *Remarks:* This constructor shall not participate in overload resolution unless `is_default_constructible_v<first_type>` is `true` and `is_default_constructible_v<second_type>` is `true`. [*Note*: This behavior can be implemented by a constructor template with default template arguments. — *end note*] The expression inside `explicit` evaluates to `true` if and only if either `first_type` or `second_type` is not implicitly default-constructible. [*Note*: This behavior can be implemented with a trait that checks whether a `const first_type&` or a `const second_type&` can be initialized with `{}`. — *end note*]

```
constexpr explicit(see below) pair(const T1& x, const T2& y);
```

6 *Constraints:* `is_copy_constructible_v<first_type>` is `true` and `is_copy_constructible_v<second_type>` is `true`.

Effects: Initializes `first` with `x` and `second` with `y`.

7 *Remarks:* ~~This constructor shall not participate in overload resolution unless `is_copy_constructible_v<first_type>` is `true` and `is_copy_constructible_v<second_type>` is `true`.~~

The expression inside `explicit` is equivalent to:

```
!is_convertible_v<const first_type&, first_type> ||
  !is_convertible_v<const second_type&, second_type>
```

```
template<class U1, class U2> constexpr explicit(see below) pair(U1&& x, U2&& y);
```

8 *Constraints:* `is_constructible_v<first_type, U1&&>` is `true` and `is_constructible_v<second_type, U2&&>` is `true`.

Effects: Initializes `first` with `std::forward<U1>(x)` and `second` with `std::forward<U2>(y)`.

9 *Remarks:* ~~This constructor shall not participate in overload resolution unless `is_constructible_v<first_type, U1&&>` is `true` and `is_constructible_v<second_type, U2&&>` is `true`.~~ The expression inside `explicit` is equivalent to:

```
!is_convertible_v<U1, first_type> || !is_convertible_v<U2, second_type>
```

```
template<class U1, class U2> constexpr explicit(see below) pair(const pair<U1, U2>& p);
```

10 *Constraints:* `is_constructible_v<first_type, const U1&>` is `true` and `is_constructible_v<second_type, con` is `true`.

Effects: Initializes members from the corresponding members of the argument.

11 *Remarks:* ~~This constructor shall not participate in overload resolution unless `is_constructible_v<first_type, const U2&>` is `true`.~~ The expression inside explicit is equivalent to:

```
!is_convertible_v<const U1&, first_type> || !is_convertible_v<const U2&, second_type>
```

```
template<class U1, class U2> constexpr explicit(see below) pair(pair<U1, U2>&& p);
```

12 *Constraints:* `is_constructible_v<first_type, U1&&>` is `true` and `is_constructible_v<second_type, U2&&>` is `true`.

Effects: Initializes `first` with `std::forward<U1>(p.first)` and `second` with `std::forward<U2>(p.second)`.

13 *Remarks:* ~~This constructor shall not participate in overload resolution unless `is_constructible_v<first_type, U1&&>` is `true` and `is_constructible_v<second_type, U2&&>` is `true`.~~ The expression inside explicit is equivalent to:

```
!is_convertible_v<U1, first_type> || !is_convertible_v<U2, second_type>
```

```
template<class... Args1, class... Args2>
  constexpr pair(piecewise_construct_t,
```

```
        tuple<Args1...> first_args, tuple<Args2...> second_args);
```

14 ~~*Requires:*~~ *Preconditions:* `is_constructible_v<first_type, Args1&&...>` is `true` and `is_constructible_v<second_type, Args2&&...>` is `true`.

15 *Effects:* Initializes `first` with arguments of types `Args1...` obtained by forwarding the elements of `first_args` and initializes `second` with arguments of types `Args2...` obtained by forwarding the elements of `second_args`. (Here, forwarding an element `x` of type `U` within a `tuple` object means calling `std::forward<U>(x)`.) This form of construction, whereby constructor arguments for `first` and `second` are each provided in a separate `tuple` object, is called *piecewise construction*.

```
constexpr pair& operator=(const pair& p);
```

16 *Mandates:* `is_copy_assignable_v<first_type>` is `true` and `is_copy_assignable_v<second_type>` is `true`.

17 *Effects:* Assigns `p.first` to `first` and `p.second` to `second`.

18 ~~*Remarks:* This operator shall be defined as deleted unless `is_copy_assignable_v<first_type>` is `true` and `is_copy_assignable_v<second_type>` is `true`.~~

19 *Returns:* `*this`.

```
template<class U1, class U2> constexpr pair& operator=(const pair<U1, U2>& p);
```

20 *Constraints:* `is_assignable_v<first_type&, const U1&>` is `true` and `is_assignable_v<second_type&, const U` is `true`.

Effects: Assigns `p.first` to `first` and `p.second` to `second`.

21 ~~*Remarks:* This operator shall not participate in overload resolution unless `is_assignable_v<first_type&, const U` is `true` and `is_assignable_v<second_type&, const U2&>` is `true`.~~

22 *Returns:* `*this`.

```
constexpr pair& operator=(pair&& p) noexcept(see below);
```

23 *Constraints:* `is_move_assignable_v<first_type>` is `true` and `is_move_assignable_v<second_type>` is `true`.

Effects: Assigns to `first` with `std::forward<first_type>(p.first)` and to `second` with `std::forward<second_type>(p.second)`.

24 *Remarks:* This operator shall not participate in overload resolution unless `is_move_assignable_v<first_type>` is `true` and `is_move_assignable_v<second_type>` is `true`.

25 *Remarks:* The expression inside `noexcept` is equivalent to:

```
is_nothrow_move_assignable_v<T1> && is_nothrow_move_assignable_v<T2>
```

26 *Returns:* `*this`.

Remarks: The expression inside `noexcept` is equivalent to:

```
is_nothrow_move_assignable_v<T1> && is_nothrow_move_assignable_v<T2>
```

```
template<class U1, class U2> constexpr pair& operator=(pair<U1, U2>&& p);
```

27 *Constraints:* `is_assignable_v<first_type&, U1&&>` is `true` and `is_assignable_v<second_type&, U2&&>` is `true`.

Effects: Assigns to `first` with `std::forward<U1>(p.first)` and to `second` with `std::forward<U2>(p.second)`.

28 ~~*Remarks:* This operator shall not participate in overload resolution unless `is_assignable_v<first_type&, U1&&>` is `true` and `is_assignable_v<second_type&, U2&&>` is `true`.~~

29 *Returns:* `*this`.

```
constexpr void swap(pair& p) noexcept(see below);
```

30 ~~*Requires:*~~ *Preconditions:* `first` ~~shall be~~is swappable with (??) `p.first` and `second` ~~shall be~~is swappable with `p.second`.

31 *Effects:* Swaps `first` with `p.first` and `second` with `p.second`.

32 *Remarks:* The expression inside `noexcept` is equivalent to:

```
is_nothrow_swappable_v<first_type> && is_nothrow_swappable_v<second_type>
```

20.4.3 Specialized algorithms [pairs.spec]

```
friend constexpr
  common_comparison_category_t<synth-three-way-result<T1>, synth-three-way-result<T2>>
    operator<=>(const pair& x, const pair& y);
```

1 *Effects:* Equivalent to:

```
if (auto c = synth-three-way(x.first, y.first); c != 0) return c;
return synth-three-way(x.second, y.second);
```

```
template<class T1, class T2>
  constexpr void swap(pair<T1, T2>& x, pair<T1, T2>& y) noexcept(noexcept(x.swap(y)));
```

2 *Constraints:* `is_swappable_v<T1>` is `true` and `is_swappable_v<T2>` is `true`.

Effects: As if by `x.swap(y)`.

3 ~~*Remarks:* This function shall not participate in overload resolution unless `is_swappable_v<T1>` is `true` and `is_swappable_v<T2>` is `true`.~~

```
template<class T1, class T2>
  constexpr pair<unwrap_ref_decay_t<T1>, unwrap_ref_decay_t<T2>> make_pair(T1&& x, T2&& y);
```

4 *Returns:*

```
pair<unwrap_ref_decay_t<T1>,
     unwrap_ref_decay_t<T2>>(std::forward<T1>(x), std::forward<T2>(y))
```

5 [*Example*: In place of:

```
return pair<int, double>(5, 3.1415926);      // explicit types
```

a C++ program may contain:

```
return make_pair(5, 3.1415926);              // types are deduced
```

— *end example*]

20.4.4 Tuple-like access to pair [pair.astuple]

```
template<class T1, class T2>
  struct tuple_size<pair<T1, T2>> : integral_constant<size_t, 2> { };

template<size_t I, class T1, class T2>
  struct tuple_element<I, pair<T1, T2>> {
    using type = see below ;
  };
```

1 ~~*Requires:*~~ *Mandates:* `I < 2`. ~~The program is ill-formed if `I` is out of bounds.~~

2 *Type:* The type `T1` if `I == 0`, otherwise the type `T2`.

```
template<size_t I, class T1, class T2>
  constexpr tuple_element_t<I, pair<T1, T2>>& get(pair<T1, T2>& p) noexcept;
template<size_t I, class T1, class T2>
  constexpr const tuple_element_t<I, pair<T1, T2>>& get(const pair<T1, T2>& p) noexcept;
template<size_t I, class T1, class T2>
  constexpr tuple_element_t<I, pair<T1, T2>>&& get(pair<T1, T2>&& p) noexcept;
template<size_t I, class T1, class T2>
  constexpr const tuple_element_t<I, pair<T1, T2>>&& get(const pair<T1, T2>&& p) noexcept;
```

3 *Mandates:* `I < 2`.

Returns: If `I == 0` returns a reference to `p.first`; if `I == 1` returns a reference to `p.second`~~; otherwise the program is ill-formed~~.

```
template<class T1, class T2>
  constexpr T1& get(pair<T1, T2>& p) noexcept;
template<class T1, class T2>
  constexpr const T1& get(const pair<T1, T2>& p) noexcept;
```

```
template<class T1, class T2>
  constexpr T1&& get(pair<T1, T2>&& p) noexcept;
template<class T1, class T2>
  constexpr const T1&& get(const pair<T1, T2>&& p) noexcept;
```

4 ~~*Requires:*~~ *Mandates:* `T1` and `T2` are distinct types. ~~Otherwise, the program is ill-formed.~~

5 *Returns:* A reference to `p.first`.

```
template<class T2, class T1>
  constexpr T2& get(pair<T1, T2>& p) noexcept;
template<class T2, class T1>
  constexpr const T2& get(const pair<T1, T2>& p) noexcept;
template<class T2, class T1>
  constexpr T2&& get(pair<T1, T2>&& p) noexcept;
template<class T2, class T1>
  constexpr const T2&& get(const pair<T1, T2>&& p) noexcept;
```

6 ~~*Requires:*~~ *Mandates:* `T1` and `T2` are distinct types. ~~Otherwise, the program is ill-formed.~~

7 *Returns:* A reference to `p.second`.

20.4.5 Piecewise construction [pair.piecewise]

```
struct piecewise_construct_t {
  explicit piecewise_construct_t() = default;
};
inline constexpr piecewise_construct_t piecewise_construct{};
```

1 The `struct piecewise_construct_t` is an empty class type used as a unique type to disambiguate constructor and function overloading. Specifically, `pair` has a constructor with `piecewise_construct_t` as the first argument, immediately followed by two `tuple` (20.5) arguments used for piecewise construction of the elements of the `pair` object.

20.5 Tuples [tuple]

20.5.1 In general [tuple.general]

1 Subclause 20.5 describes the tuple library that provides a tuple type as the class template `tuple` that can be instantiated with any number of arguments. Each template argument specifies the type of an element in the `tuple`. Consequently, tuples are heterogeneous, fixed-size collections of values. An instantiation of `tuple` with two arguments is similar to an instantiation of `pair` with the same two arguments. See 20.4.

20.5.2 Header <tuple> synopsis [tuple.syn]

```
namespace std {
  // 20.5.3, class template tuple
  template<class... Types>
    class tuple;

  // 20.5.4, tuple creation functions
  inline constexpr unspecified ignore;

  template<class... TTypes>
    constexpr tuple<unwrap_ref_decay_t<TTypes>...> make_tuple(TTypes&&...);

  template<class... TTypes>
    constexpr tuple<TTypes&&...> forward_as_tuple(TTypes&&...) noexcept;

  template<class... TTypes>
    constexpr tuple<TTypes&...> tie(TTypes&...) noexcept;

  template<class... Tuples>
    constexpr tuple<CTypes...> tuple_cat(Tuples&&...);

  // 20.5.5, calling a function with a tuple of arguments
  template<class F, class Tuple>
    constexpr decltype(auto) apply(F&& f, Tuple&& t);
```

```
  template<class T, class Tuple>
    constexpr T make_from_tuple(Tuple&& t);

  // 20.5.6, tuple helper classes
  template<class T> struct tuple_size;                    // not defined
  template<class T> struct tuple_size<const T>;
  template<class T> struct tuple_size<volatile T>;
  template<class T> struct tuple_size<const volatile T>;

  template<class... Types> struct tuple_size<tuple<Types...>>;

  template<size_t I, class T> struct tuple_element;       // not defined
  template<size_t I, class T> struct tuple_element<I, const T>;
  template<size_t I, class T> struct tuple_element<I, volatile T>;
  template<size_t I, class T> struct tuple_element<I, const volatile T>;

  template<size_t I, class... Types>
    struct tuple_element<I, tuple<Types...>>;

  template<size_t I, class T>
    using tuple_element_t = typename tuple_element<I, T>::type;

  // 20.5.7, element access
  template<size_t I, class... Types>
    constexpr tuple_element_t<I, tuple<Types...>>& get(tuple<Types...>&) noexcept;
  template<size_t I, class... Types>
    constexpr tuple_element_t<I, tuple<Types...>>&& get(tuple<Types...>&&) noexcept;
  template<size_t I, class... Types>
    constexpr const tuple_element_t<I, tuple<Types...>>& get(const tuple<Types...>&) noexcept;
  template<size_t I, class... Types>
    constexpr const tuple_element_t<I, tuple<Types...>>&& get(const tuple<Types...>&&) noexcept;
  template<class T, class... Types>
    constexpr T& get(tuple<Types...>& t) noexcept;
  template<class T, class... Types>
    constexpr T&& get(tuple<Types...>&& t) noexcept;
  template<class T, class... Types>
    constexpr const T& get(const tuple<Types...>& t) noexcept;
  template<class T, class... Types>
    constexpr const T&& get(const tuple<Types...>&& t) noexcept;

  // 20.5.8, relational operators
  template<class... TTypes, class... UTypes>
    constexpr bool operator==(const tuple<TTypes...>&, const tuple<UTypes...>&);
  template<class... TTypes, class... UTypes>
    constexpr common_comparison_category_t<synth-three-way-result<TTypes, UTypes>...>
      operator<=>(const tuple<TTypes...>&, const tuple<UTypes...>&);

  // 20.5.9, allocator-related traits
  template<class... Types, class Alloc>
    struct uses_allocator<tuple<Types...>, Alloc>;

  // 20.5.10, specialized algorithms
  template<class... Types>
    constexpr void swap(tuple<Types...>& x, tuple<Types...>& y) noexcept(see below);

  // 20.5.6, tuple helper classes
  template<class T>
    inline constexpr size_t tuple_size_v = tuple_size<T>::value;
}
```

20.5.3 Class template `tuple` [tuple.tuple]

```
namespace std {
  template<class... Types>
```

```
class tuple {
public:
  // 20.5.3.1, tuple construction
  constexpr explicit(see below) tuple();
  constexpr explicit(see below) tuple(const Types&...);          // only if sizeof...(Types) >= 1
  template<class... UTypes>
    constexpr explicit(see below) tuple(UTypes&&...);            // only if sizeof...(Types) >= 1

  tuple(const tuple&) = default;
  tuple(tuple&&) = default;

  template<class... UTypes>
    constexpr explicit(see below) tuple(const tuple<UTypes...>&);
  template<class... UTypes>
    constexpr explicit(see below) tuple(tuple<UTypes...>&&);

  template<class U1, class U2>
    constexpr explicit(see below) tuple(const pair<U1, U2>&);    // only if sizeof...(Types) == 2
  template<class U1, class U2>
    constexpr explicit(see below) tuple(pair<U1, U2>&&);         // only if sizeof...(Types) == 2

  // allocator-extended constructors
  template<class Alloc>
    constexpr explicit(see below)
      tuple(allocator_arg_t, const Alloc& a);
  template<class Alloc>
    constexpr explicit(see below)
      tuple(allocator_arg_t, const Alloc& a, const Types&...);
  template<class Alloc, class... UTypes>
    constexpr explicit(see below)
      tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
  template<class Alloc>
    constexpr tuple(allocator_arg_t, const Alloc& a, const tuple&);
  template<class Alloc>
    constexpr tuple(allocator_arg_t, const Alloc& a, tuple&&);
  template<class Alloc, class... UTypes>
    constexpr explicit(see below)
      tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
  template<class Alloc, class... UTypes>
    constexpr explicit(see below)
      tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
  template<class Alloc, class U1, class U2>
    constexpr explicit(see below)
      tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);
  template<class Alloc, class U1, class U2>
    constexpr explicit(see below)
      tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);

  // 20.5.3.2, tuple assignment
  constexpr tuple& operator=(const tuple&);
  constexpr tuple& operator=(tuple&&) noexcept(see below);

  template<class... UTypes>
    constexpr tuple& operator=(const tuple<UTypes...>&);
  template<class... UTypes>
    constexpr tuple& operator=(tuple<UTypes...>&&);

  template<class U1, class U2>
    constexpr tuple& operator=(const pair<U1, U2>&);             // only if sizeof...(Types) == 2
  template<class U1, class U2>
    constexpr tuple& operator=(pair<U1, U2>&&);                  // only if sizeof...(Types) == 2

  // 20.5.3.3, tuple swap
  constexpr void swap(tuple&) noexcept(see below);
```

```
  };

  template<class... UTypes>
    tuple(UTypes...) -> tuple<UTypes...>;
  template<class T1, class T2>
    tuple(pair<T1, T2>) -> tuple<T1, T2>;
  template<class Alloc, class... UTypes>
    tuple(allocator_arg_t, Alloc, UTypes...) -> tuple<UTypes...>;
  template<class Alloc, class T1, class T2>
    tuple(allocator_arg_t, Alloc, pair<T1, T2>) -> tuple<T1, T2>;
  template<class Alloc, class... UTypes>
    tuple(allocator_arg_t, Alloc, tuple<UTypes...>) -> tuple<UTypes...>;
}
```

20.5.3.1 Construction [tuple.cnstr]

1 In the descriptions that follow, let i be in the range `[0, sizeof...(Types))` in order, $\texttt{T}_i$ be the i^{th} type in `Types`, and $\texttt{U}_i$ be the i^{th} type in a template parameter pack named `UTypes`, where indexing is zero-based.

2 For each `tuple` constructor, an exception is thrown only if the construction of one of the types in `Types` throws an exception.

3 The defaulted move and copy constructor, respectively, of `tuple` is a constexpr function if and only if all required element-wise initializations for copy and move, respectively, would satisfy the requirements for a constexpr function. The defaulted move and copy constructor of `tuple<>` are constexpr functions.

4 If `is_trivially_destructible_v<`$\texttt{T}_i$`>` is `true` for all $\texttt{T}_i$, then the destructor of `tuple` is trivial.

`constexpr explicit(`*see below*`) tuple();`

5 *Constraints:* `is_default_constructible_v<`$\texttt{T}_i$`>` is `true` for all i.

6 *Effects:* Value-initializes each element.

7 *Remarks:* This constructor shall not participate in overload resolution unless `is_default_constructible_v<`$\texttt{T}_i$`>` is `true` for all i. [*Note*: This behavior can be implemented by a constructor template with default template arguments. — *end note*]

The expression inside `explicit` evaluates to `true` if and only if $\texttt{T}_i$ is not copy-list-initializable from an empty list for at least one i. [*Note*: This behavior can be implemented with a trait that checks whether a `const` $\texttt{T}_i$`&` can be initialized with `{}`. — *end note*]

`constexpr explicit(`*see below*`) tuple(const Types&...);`

8 *Constraints:* `sizeof...(Types) >= 1` and `is_copy_constructible_v<`$\texttt{T}_i$`>` is `true` for all i.

9 *Effects:* Initializes each element with the value of the corresponding parameter.

10 *Remarks:* ~~This constructor shall not participate in overload resolution unless `sizeof...(Types) >= 1` and `is_copy_constructible_v<`$\texttt{T}_i$`>` is `true` for all i.~~ The expression inside `explicit` is equivalent to:

```
!conjunction_v<is_convertible<const Types&, Types>...>
```

`template<class... UTypes> constexpr explicit(`*see below*`) tuple(UTypes&&... u);`

11 *Constraints:* `sizeof...(Types) == sizeof...(UTypes)` and `sizeof...(Types) >= 1` and `is_constructible_v<`$\texttt{T}_i$`, `$\texttt{U}_i$`&&>` is `true` for all i

12 *Effects:* Initializes the elements in the tuple with the corresponding value in `std::forward<UTypes>(u)`.

13 *Remarks:* ~~This constructor shall not participate in overload resolution unless `sizeof...(Types) == sizeof...(UTypes)` and `sizeof...(Types) >= 1` and `is_constructible_v<`$\texttt{T}_i$`, `$\texttt{U}_i$`&&>` is `true` for all i.~~ The expression inside `explicit` is equivalent to:

```
!conjunction_v<is_convertible<UTypes, Types>...>
```

`tuple(const tuple& u) = default;`

14 ~~*Requires:*~~ *Preconditions:* `is_copy_constructible_v<`$\texttt{T}_i$`>` is `true` for all i.

15 *Effects:* Initializes each element of `*this` with the corresponding element of `u`.

`tuple(tuple&& u) = default;`

16 ~~*Requires:*~~ *Preconditions:* `is_move_constructible_v<T`$_i$`>` is `true` for all i.

17 *Effects:* For all i, initializes the i^{th} element of `*this` with `std::forward<T`$_i$`>(get<`i`>(u))`.

`template<class... UTypes> constexpr explicit(`*see below*`) tuple(const tuple<UTypes...>& u);`

18 *Constraints:*

(18.1) — `sizeof...(Types) == sizeof...(UTypes)` and

(18.2) — `is_constructible_v<T`$_i$`, const U`$_i$`&>` is `true` for all i, and

(18.3) — either `sizeof...(Types) != 1`, or (when `Types...` expands to `T` and `UTypes...` expands to `U`) `is_convertible_v<const tuple<U>&, T>`, `is_constructible_v<T, const tuple<U>&>`, and `is_same_v<T, U>` are all `false`.

19 *Effects:* Initializes each element of `*this` with the corresponding element of `u`.

20 *Remarks:* This constructor shall not participate in overload resolution unless

(20.1) — `sizeof...(Types) == sizeof...(UTypes)` and

(20.2) — `is_constructible_v<T`$_i$`, const U`$_i$`&>` is `true` for all i, and

(20.3) — either `sizeof...(Types) != 1`, or (when `Types...` expands to `T` and `UTypes...` expands to `U`) `is_convertible_v<const tuple<U>&, T>`, `is_constructible_v<T, const tuple<U>&>`, and `is_same_v<T, U>` are all `false`.

The expression inside `explicit` is equivalent to:

```
!conjunction_v<is_convertible<const UTypes&, Types>...>
```

`template<class... UTypes> constexpr explicit(`*see below*`) tuple(tuple<UTypes...>&& u);`

21 *Constraints:*

(21.1) — `sizeof...(Types) == sizeof...(UTypes)`, and

(21.2) — `is_constructible_v<T`$_i$`, U`$_i$`&&>` is `true` for all i, and

(21.3) — either `sizeof...(Types) != 1`, or (when `Types...` expands to `T` and `UTypes...` expands to `U`) `is_convertible_v<tuple<U>, T>`, `is_constructible_v<T, tuple<U>>`, and `is_same_v<T, U>` are all `false`.

22 *Effects:* For all i, initializes the i^{th} element of `*this` with `std::forward<U`$_i$`>(get<`i`>(u))`.

23 *Remarks:* This constructor shall not participate in overload resolution unless

(23.1) — `sizeof...(Types) == sizeof...(UTypes)`, and

(23.2) — `is_constructible_v<T`$_i$`, U`$_i$`&&>` is `true` for all i, and

(23.3) — either `sizeof...(Types) != 1`, or (when `Types...` expands to `T` and `UTypes...` expands to `U`) `is_convertible_v<tuple<U>, T>`, `is_constructible_v<T, tuple<U>>`, and `is_same_v<T, U>` are all `false`.

The expression inside `explicit` is equivalent to:

```
!conjunction_v<is_convertible<UTypes, Types>...>
```

`template<class U1, class U2> constexpr explicit(`*see below*`) tuple(const pair<U1, U2>& u);`

24 *Constraints:* `sizeof...(Types) == 2`, `is_constructible_v<T`$_0$`, const U1&>` is `true` and `is_constructible_v<T`$_1$`, const U2&>` is `true`.

25 *Effects:* Initializes the first element with `u.first` and the second element with `u.second`.

26 *Remarks:* This constructor shall not participate in overload resolution unless `sizeof...(Types) == 2`, `is_constructible_v<T`$_0$`, const U1&>` is `true` and `is_constructible_v<T`$_1$`, const U2&>` is `true`.

27 The expression inside `explicit` is equivalent to:

```
!is_convertible_v<const U1&, T0> || !is_convertible_v<const U2&, T1>
```

`template<class U1, class U2> constexpr explicit(`*see below*`) tuple(pair<U1, U2>&& u);`

28 *Constraints:* `sizeof...(Types) == 2`, `is_constructible_v<T0, U1&&>` is `true` and `is_constructible_v<T1, U2&&>` is `true`.

29 *Effects:* Initializes the first element with `std::forward<U1>(u.first)` and the second element with `std::forward<U2>(u.second)`.

30 *Remarks:* This constructor shall not participate in overload resolution unless `sizeof...(Types) == 2`, `is_constructible_v<T0, U1&&>` is `true` and `is_constructible_v<T1, U2&&>` is `true`.

31 The expression inside `explicit` is equivalent to:

```
!is_convertible_v<U1, T0> || !is_convertible_v<U2, T1>
```

```
template<class Alloc>
  constexpr explicit(see below)
    tuple(allocator_arg_t, const Alloc& a);
template<class Alloc>
  constexpr explicit(see below)
    tuple(allocator_arg_t, const Alloc& a, const Types&...);
template<class Alloc, class... UTypes>
  constexpr explicit(see below)
    tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
template<class Alloc>
  constexpr tuple(allocator_arg_t, const Alloc& a, const tuple&);
template<class Alloc>
  constexpr tuple(allocator_arg_t, const Alloc& a, tuple&&);
template<class Alloc, class... UTypes>
  constexpr explicit(see below)
    tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
template<class Alloc, class... UTypes>
  constexpr explicit(see below)
    tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
template<class Alloc, class U1, class U2>
  constexpr explicit(see below)
    tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);
template<class Alloc, class U1, class U2>
  constexpr explicit(see below)
    tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);
```

32 ~~*Requires:*~~ *Preconditions:* `Alloc` shall meet the *Cpp17Allocator* requirements (Table **??**).

33 *Effects:* Equivalent to the preceding constructors except that each element is constructed with uses-allocator construction (20.10.8.2).

20.5.3.2 Assignment [tuple.assign]

1 For each `tuple` assignment operator, an exception is thrown only if the assignment of one of the types in `Types` throws an exception. In the function descriptions that follow, let i be in the range `[0, sizeof...(Types))` in order, T_i be the i^{th} type in `Types`, and U_i be the i^{th} type in a template parameter pack named `UTypes`, where indexing is zero-based.

```
constexpr tuple& operator=(const tuple& u);
```

2 *Mandates:* `is_copy_assignable_v<Ti>` is `true` for all i.

3 *Effects:* Assigns each element of `u` to the corresponding element of `*this`.

4 *Remarks:* This operator shall be defined as deleted unless `is_copy_assignable_v<Ti>` is `true` for all i.

5 *Returns:* `*this`.

```
constexpr tuple& operator=(tuple&& u) noexcept(see below);
```

6 *Constraints:* `is_move_assignable_v<Ti>` is `true` for all i.

7 *Effects:* For all i, assigns `std::forward<Ti>(get<i>(u))` to `get<i>(*this)`.

8 *Remarks:* This operator shall not participate in overload resolution unless `is_move_assignable_v<Ti>` is `true` for all i.

9 *Remarks:* The expression inside `noexcept` is equivalent to the logical AND of the following expressions:

`is_nothrow_move_assignable_v<T`$_i$`>`

where `T`$_i$ is the i^{th} type in `Types`.

Returns: `*this`.

```
template<class... UTypes> constexpr tuple& operator=(const tuple<UTypes...>& u);
```

Constraints: `sizeof...(Types) == sizeof...(UTypes)` and `is_assignable_v<T`$_i$`&, const U`$_i$`&>` is `true` for all i.

Effects: Assigns each element of `u` to the corresponding element of `*this`.

Remarks: This operator shall not participate in overload resolution unless `sizeof...(Types) == sizeof...(UTypes)` and `is_assignable_v<T`$_i$`&, const U`$_i$`&>` is `true` for all i.

Returns: `*this`.

```
template<class... UTypes> constexpr tuple& operator=(tuple<UTypes...>&& u);
```

Constraints: `is_assignable_v<T`$_i$`&, U`$_i$`&&> == true` for all i and `sizeof...(Types) == sizeof...(UTypes)`.

Effects: For all i, assigns `std::forward<U`$_i$`>(get<`i`>(u))` to `get<`i`>(*this)`.

Remarks: This operator shall not participate in overload resolution unless `is_assignable_v<T`$_i$`&, U`$_i$`&&> == true` for all i and `sizeof...(Types) == sizeof...(UTypes)`.

Returns: `*this`.

```
template<class U1, class U2> constexpr tuple& operator=(const pair<U1, U2>& u);
```

Constraints: `sizeof...(Types) == 2` and `is_assignable_v<T`$_0$`&, const U1&>` is `true` for the first type `T`$_0$ in `Types` and `is_assignable_v<T`$_1$`&, const U2&>` is `true` for the second type `T`$_1$ in `Types`.

Effects: Assigns `u.first` to the first element of `*this` and `u.second` to the second element of `*this`.

Remarks: This operator shall not participate in overload resolution unless `sizeof...(Types) == 2` and `is_assignable_v<T`$_0$`&, const U1&>` is `true` for the first type `T`$_0$ in `Types` and `is_assignable_v<T`$_1$`&, const U2&>` is `true` for the second type `T`$_1$ in `Types`.

Returns: `*this`.

```
template<class U1, class U2> constexpr tuple& operator=(pair<U1, U2>&& u);
```

Constraints: `sizeof...(Types) == 2` and `is_assignable_v<T`$_0$`&, U1&&>` is `true` for the first type `T`$_0$ in `Types` and `is_assignable_v<T`$_1$`&, U2&&>` is `true` for the second type `T`$_1$ in `Types`.

Effects: Assigns `std::forward<U1>(u.first)` to the first element of `*this` and `std::forward<U2>(u.second)` to the second element of `*this`.

Remarks: This operator shall not participate in overload resolution unless `sizeof...(Types) == 2` and `is_assignable_v<T`$_0$`&, U1&&>` is `true` for the first type `T`$_0$ in `Types` and `is_assignable_v<T`$_1$`&, U2&&>` is `true` for the second type `T`$_1$ in `Types`.

Returns: `*this`.

20.5.3.3 `swap` [tuple.swap]

```
constexpr void swap(tuple& rhs) noexcept(see below);
```

~~*Requires:*~~ *Preconditions:* Each element in `*this` shall be swappable with (**??**) the corresponding element in `rhs`.

Effects: Calls `swap` for each element in `*this` and its corresponding element in `rhs`.

Remarks: The expression inside `noexcept` is equivalent to the logical AND of the following expressions:

`is_nothrow_swappable_v<T`$_i$`>`

where `T`$_i$ is the i^{th} type in `Types`.

Throws: Nothing unless one of the element-wise `swap` calls throws an exception.

20.5.4 Tuple creation functions [tuple.creation]

1 In the function descriptions that follow, the members of a template parameter pack *X*`Types` are denoted by X_i for i in `[0, sizeof...(`*X*`Types))` in order, where indexing is zero-based.

```
template<class... TTypes>
  constexpr tuple<unwrap_ref_decay_t<TTypes>...> make_tuple(TTypes&&... t);
```

2 *Returns:* `tuple<unwrap_ref_decay_t<TTypes>...>(std::forward<TTypes>(t)...)`.

3 [*Example*:

```
int i; float j;
make_tuple(1, ref(i), cref(j))
```

creates a tuple of type `tuple<int, int&, const float&>`. — *end example*]

```
template<class... TTypes>
  constexpr tuple<TTypes&&...> forward_as_tuple(TTypes&&... t) noexcept;
```

4 *Effects:* Constructs a tuple of references to the arguments in `t` suitable for forwarding as arguments to a function. Because the result may contain references to temporary objects, a program shall ensure that the return value of this function does not outlive any of its arguments (e.g., the program should typically not store the result in a named variable).

5 *Returns:* `tuple<TTypes&&...>(std::forward<TTypes>(t)...)`.

```
template<class... TTypes>
  constexpr tuple<TTypes&...> tie(TTypes&... t) noexcept;
```

6 *Returns:* `tuple<TTypes&...>(t...)`. When an argument in `t` is `ignore`, assigning any value to the corresponding tuple element has no effect.

7 [*Example*: `tie` functions allow one to create tuples that unpack tuples into variables. `ignore` can be used for elements that are not needed:

```
int i; std::string s;
tie(i, ignore, s) = make_tuple(42, 3.14, "C++");
// i == 42, s == "C++"
```

— *end example*]

```
template<class... Tuples>
  constexpr tuple<CTypes...> tuple_cat(Tuples&&... tpls);
```

8 In the following paragraphs, let $\texttt{T}_i$ be the i^{th} type in `Tuples`, $\texttt{U}_i$ be `remove_reference_t<`$\texttt{T}_i$`>`, and $\texttt{tp}_i$ be the i^{th} parameter in the function parameter pack `tpls`, where all indexing is zero-based.

9 ~~*Requires:*~~ *Preconditions:* For all i, $\texttt{U}_i$ shall be the type *cv*$_i$ `tuple<Args`$_i$`...>`, where *cv*$_i$ is the (possibly empty) i^{th} *cv-qualifier-seq* and $\texttt{Args}_i$ is the template parameter pack representing the element types in $\texttt{U}_i$. Let $\texttt{A}_{ik}$ be the k^{th} type in $\texttt{Args}_i$. For all $\texttt{A}_{ik}$ the following requirements shall be met:

(9.1) — If $\texttt{T}_i$ is deduced as an lvalue reference type, then `is_constructible_v<`$\texttt{A}_{ik}$`,` *cv*$_i$ $\texttt{A}_{ik}$`&> == true`, otherwise

(9.2) — `is_constructible_v<`$\texttt{A}_{ik}$`,` *cv*$_i$ $\texttt{A}_{ik}$`&&> == true`.

10 *Remarks:* The types in `CTypes` shall be equal to the ordered sequence of the extended types $\texttt{Args}_0\texttt{...}$, $\texttt{Args}_1\texttt{...}$, ..., $\texttt{Args}_{n-1}\texttt{...}$, where n is equal to `sizeof...(Tuples)`. Let $\texttt{e}_i\texttt{...}$ be the i^{th} ordered sequence of tuple elements of the resulting `tuple` object corresponding to the type sequence $\texttt{Args}_i$.

11 *Returns:* A `tuple` object constructed by initializing the $k_i{}^{\text{th}}$ type element $\texttt{e}_{ik}$ in $\texttt{e}_i\texttt{...}$ with

`get<`k_i`>(std::forward<`$\texttt{T}_i$`>(`$\texttt{tp}_i$`))`

for each valid k_i and each group $\texttt{e}_i$ in order.

12 [*Note*: An implementation may support additional types in the template parameter pack `Tuples` that support the `tuple`-like protocol, such as `pair` and `array`. — *end note*]

20.5.5 Calling a function with a `tuple` of arguments [tuple.apply]

```
template<class F, class Tuple>
```

```
constexpr decltype(auto) apply(F&& f, Tuple&& t);
```

1 *Effects:* Given the exposition-only function:

```
template<class F, class Tuple, size_t... I>
constexpr decltype(auto) apply-impl(F&& f, Tuple&& t, index_sequence<I...>) {
                                                                    // exposition only
  return INVOKE(std::forward<F>(f), std::get<I>(std::forward<Tuple>(t))...);  // see 20.14.3
}
```

Equivalent to:

```
return apply-impl(std::forward<F>(f), std::forward<Tuple>(t),
                  make_index_sequence<tuple_size_v<remove_reference_t<Tuple>>>{});
```

```
template<class T, class Tuple>
  constexpr T make_from_tuple(Tuple&& t);
```

2 *Effects:* Given the exposition-only function:

```
template<class T, class Tuple, size_t... I>
constexpr T make-from-tuple-impl(Tuple&& t, index_sequence<I...>) {     // exposition only
  return T(get<I>(std::forward<Tuple>(t))...);
}
```

Equivalent to:

```
return make-from-tuple-impl<T>(
           forward<Tuple>(t),
           make_index_sequence<tuple_size_v<remove_reference_t<Tuple>>>{});
```

[*Note*: The type of `T` must be supplied as an explicit template parameter, as it cannot be deduced from the argument list. — *end note*]

20.5.6 Tuple helper classes [tuple.helper]

```
template<class T> struct tuple_size;
```

1 ~~*Remarks:*~~ *Preconditions:* All specializations of `tuple_size` ~~shall~~ meet the *Cpp17UnaryTypeTrait* requirements (20.15.1) with a base characteristic of `integral_constant<size_t, N>` for some `N`.

```
template<class... Types>
  struct tuple_size<tuple<Types...>> : public integral_constant<size_t, sizeof...(Types)> { };

template<size_t I, class... Types>
  struct tuple_element<I, tuple<Types...>> {
    using type = TI;
  };
```

2 ~~*Requires:*~~ *Mandates:* `I < sizeof...(Types)`. ~~The program is ill-formed if I is out of bounds.~~

3 *Type:* `TI` is the type of the $\texttt{I}^{\text{th}}$ element of `Types`, where indexing is zero-based.

```
template<class T> struct tuple_size<const T>;
template<class T> struct tuple_size<volatile T>;
template<class T> struct tuple_size<const volatile T>;
```

4 Let `TS` denote `tuple_size<T>` of the *cv*-unqualified type `T`. If the expression `TS::value` is well-formed when treated as an unevaluated operand, then each of the three templates shall meet the *Cpp17UnaryTypeTrait* requirements (20.15.1) with a base characteristic of

```
integral_constant<size_t, TS::value>
```

Otherwise, they shall have no member `value`.

5 Access checking is performed as if in a context unrelated to `TS` and `T`. Only the validity of the immediate context of the expression is considered. [*Note*: The compilation of the expression can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. — *end note*]

6 In addition to being available via inclusion of the `<tuple>` header, the three templates are available when any of the headers `<array>` (??), `<ranges>` (??), `<span>` (??), or `<utility>` (20.2.1) are included.

```
template<size_t I, class T> struct tuple_element<I, const T>;
template<size_t I, class T> struct tuple_element<I, volatile T>;
template<size_t I, class T> struct tuple_element<I, const volatile T>;
```

7 Let `TE` denote `tuple_element_t<I, T>` of the *cv*-unqualified type `T`. Then each of the three templates shall meet the *Cpp17TransformationTrait* requirements (20.15.1) with a member typedef `type` that names the following type:

(7.1) — for the first specialization, `add_const_t<TE>`,

(7.2) — for the second specialization, `add_volatile_t<TE>`, and

(7.3) — for the third specialization, `add_cv_t<TE>`.

8 In addition to being available via inclusion of the `<tuple>` header, the three templates are available when any of the headers `<array>` (??), `<ranges>` (??), `<span>` (??), or `<utility>` (20.2.1) are included.

20.5.7 Element access [tuple.elem]

```
template<size_t I, class... Types>
  constexpr tuple_element_t<I, tuple<Types...>>&
    get(tuple<Types...>& t) noexcept;
template<size_t I, class... Types>
  constexpr tuple_element_t<I, tuple<Types...>>&&
    get(tuple<Types...>&& t) noexcept;        // Note A
template<size_t I, class... Types>
  constexpr const tuple_element_t<I, tuple<Types...>>&
    get(const tuple<Types...>& t) noexcept;   // Note B
template<size_t I, class... Types>
  constexpr const tuple_element_t<I, tuple<Types...>>&& get(const tuple<Types...>&& t) noexcept;
```

1 ~~*Requires:*~~ *Mandates:* `I < sizeof...(Types)`. ~~The program is ill-formed if `I` is out of bounds.~~

2 *Returns:* A reference to the $\texttt{I}^{\text{th}}$ element of `t`, where indexing is zero-based.

3 [*Note A*: If a type `T` in `Types` is some reference type `X&`, the return type is `X&`, not `X&&`. However, if the element type is a non-reference type `T`, the return type is `T&&`. — *end note*]

4 [*Note B*: Constness is shallow. If a type `T` in `Types` is some reference type `X&`, the return type is `X&`, not `const X&`. However, if the element type is a non-reference type `T`, the return type is `const T&`. This is consistent with how constness is defined to work for member variables of reference type. — *end note*]

```
template<class T, class... Types>
  constexpr T& get(tuple<Types...>& t) noexcept;
template<class T, class... Types>
  constexpr T&& get(tuple<Types...>&& t) noexcept;
template<class T, class... Types>
  constexpr const T& get(const tuple<Types...>& t) noexcept;
template<class T, class... Types>
  constexpr const T&& get(const tuple<Types...>&& t) noexcept;
```

5 ~~*Requires:*~~ *Mandates:* The type `T` occurs exactly once in `Types`. ~~Otherwise, the program is ill-formed.~~

6 *Returns:* A reference to the element of `t` corresponding to the type `T` in `Types`.

7 [*Example*:

```
const tuple<int, const int, double, double> t(1, 2, 3.4, 5.6);
const int& i1 = get<int>(t);                    // OK, i1 has value 1
const int& i2 = get<const int>(t);              // OK, i2 has value 2
const double& d = get<double>(t);               // error: type double is not unique within t
```

— *end example*]

8 [*Note*: The reason `get` is a non-member function is that if this functionality had been provided as a member function, code where the type depended on a template parameter would have required using the `template` keyword. — *end note*]

20.5.8 Relational operators [tuple.rel]

```
template<class... TTypes, class... UTypes>
```

```
constexpr bool operator==(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

1 ~~*Requires:*~~ *Preconditions:* For all `i`, where `0 <= i` and `i < sizeof...(TTypes)`, `get<i>(t) == get<i>(u)` is a valid expression returning a type that is convertible to `bool`. `sizeof...(TTypes) == sizeof...(UTypes)`.

2 *Returns:* `true` if `get<i>(t) == get<i>(u)` for all `i`, otherwise `false`. For any two zero-length tuples `e` and `f`, `e == f` returns `true`.

3 *Effects:* The elementary comparisons are performed in order from the zeroth index upwards. No comparisons or element accesses are performed after the first equality comparison that evaluates to `false`.

```
template<class... TTypes, class... UTypes>
  constexpr common_comparison_category_t<synth-three-way-result<TTypes, UTypes>...>
    operator<=>(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

4 *Effects:* Performs a lexicographical comparison between `t` and `u`. For any two zero-length tuples `t` and `u`, `t <=> u` returns `strong_ordering::equal`. Otherwise, equivalent to:

```
if (auto c = synth-three-way(get<0>(t), get<0>(u)); c != 0) return c;
return t_tail <=> u_tail;
```

where $\texttt{r}_{\texttt{tail}}$ for some tuple `r` is a tuple containing all but the first element of `r`.

5 [*Note*: The above definition does not require $\texttt{t}_{\texttt{tail}}$ (or $\texttt{u}_{\texttt{tail}}$) to be constructed. It may not even be possible, as `t` and `u` are not required to be copy constructible. Also, all comparison functions are short circuited; they do not perform element accesses beyond what is required to determine the result of the comparison. — *end note*]

20.5.9 Tuple traits [tuple.traits]

```
template<class... Types, class Alloc>
  struct uses_allocator<tuple<Types...>, Alloc> : true_type { };
```

1 ~~*Requires:*~~ *Preconditions:* `Alloc` shall meet the *Cpp17Allocator* requirements (Table ??).

2 [*Note*: Specialization of this trait informs other library components that `tuple` can be constructed with an allocator, even though it does not have a nested `allocator_type`. — *end note*]

20.5.10 Tuple specialized algorithms [tuple.special]

```
template<class... Types>
  constexpr void swap(tuple<Types...>& x, tuple<Types...>& y) noexcept(see below);
```

1 *Constraints:* `is_swappable_v<T>` is `true` for every type `T` in `Types`.

2 *Remarks:* This function shall not participate in overload resolution unless `is_swappable_v<T>` is `true` for every type `T` in `Types`.

The expression inside `noexcept` is equivalent to:

```
noexcept(x.swap(y))
```

3 *Effects:* As if by `x.swap(y)`.

20.6 Optional objects [optional]

20.6.1 In general [optional.general]

1 Subclause 20.6 describes class template `optional` that represents optional objects. An *optional object* is an object that contains the storage for another object and manages the lifetime of this contained object, if any. The contained object may be initialized after the optional object has been initialized, and may be destroyed before the optional object has been destroyed. The initialization state of the contained object is tracked by the optional object.

20.6.2 Header <optional> synopsis [optional.syn]

```
namespace std {
  // 20.6.3, class template optional
  template<class T>
    class optional;
```

```
  // 20.6.4, no-value state indicator
  struct nullopt_t{see below};
  inline constexpr nullopt_t nullopt(unspecified);

  // 20.6.5, class bad_optional_access
  class bad_optional_access;

  // 20.6.6, relational operators
  template<class T, class U>
    constexpr bool operator==(const optional<T>&, const optional<U>&);
  template<class T, class U>
    constexpr bool operator!=(const optional<T>&, const optional<U>&);
  template<class T, class U>
    constexpr bool operator<(const optional<T>&, const optional<U>&);
  template<class T, class U>
    constexpr bool operator>(const optional<T>&, const optional<U>&);
  template<class T, class U>
    constexpr bool operator<=(const optional<T>&, const optional<U>&);
  template<class T, class U>
    constexpr bool operator>=(const optional<T>&, const optional<U>&);
  template<class T, three_way_comparable_with<T> U>
    constexpr compare_three_way_result_t<T,U>
      operator<=>(const optional<T>&, const optional<U>&);

  // 20.6.7, comparison with nullopt
  template<class T> constexpr bool operator==(const optional<T>&, nullopt_t) noexcept;
  template<class T>
    constexpr strong_ordering operator<=>(const optional<T>&, nullopt_t) noexcept;

  // 20.6.8, comparison with T
  template<class T, class U> constexpr bool operator==(const optional<T>&, const U&);
  template<class T, class U> constexpr bool operator==(const T&, const optional<U>&);
  template<class T, class U> constexpr bool operator!=(const optional<T>&, const U&);
  template<class T, class U> constexpr bool operator!=(const T&, const optional<U>&);
  template<class T, class U> constexpr bool operator<(const optional<T>&, const U&);
  template<class T, class U> constexpr bool operator<(const T&, const optional<U>&);
  template<class T, class U> constexpr bool operator>(const optional<T>&, const U&);
  template<class T, class U> constexpr bool operator>(const T&, const optional<U>&);
  template<class T, class U> constexpr bool operator<=(const optional<T>&, const U&);
  template<class T, class U> constexpr bool operator<=(const T&, const optional<U>&);
  template<class T, class U> constexpr bool operator>=(const optional<T>&, const U&);
  template<class T, class U> constexpr bool operator>=(const T&, const optional<U>&);
  template<class T, three_way_comparable_with<T> U>
    constexpr compare_three_way_result_t<T,U>
      operator<=>(const optional<T>&, const U&);

  // 20.6.9, specialized algorithms
  template<class T>
    void swap(optional<T>&, optional<T>&) noexcept(see below);

  template<class T>
    constexpr optional<see below> make_optional(T&&);
  template<class T, class... Args>
    constexpr optional<T> make_optional(Args&&... args);
  template<class T, class U, class... Args>
    constexpr optional<T> make_optional(initializer_list<U> il, Args&&... args);

  // 20.6.10, hash support
  template<class T> struct hash;
  template<class T> struct hash<optional<T>>;
}
```

20.6.3 Class template `optional` [optional.optional]

```
namespace std {
  template<class T>
  class optional {
  public:
    using value_type = T;

    // 20.6.3.1, constructors
    constexpr optional() noexcept;
    constexpr optional(nullopt_t) noexcept;
    constexpr optional(const optional&);
    constexpr optional(optional&&) noexcept(see below);
    template<class... Args>
      constexpr explicit optional(in_place_t, Args&&...);
    template<class U, class... Args>
      constexpr explicit optional(in_place_t, initializer_list<U>, Args&&...);
    template<class U = T>
      explicit(see below) constexpr optional(U&&);
    template<class U>
      explicit(see below) optional(const optional<U>&);
    template<class U>
      explicit(see below) optional(optional<U>&&);

    // 20.6.3.2, destructor
    ~optional();

    // 20.6.3.3, assignment
    optional& operator=(nullopt_t) noexcept;
    constexpr optional& operator=(const optional&);
    constexpr optional& operator=(optional&&) noexcept(see below);
    template<class U = T> optional& operator=(U&&);
    template<class U> optional& operator=(const optional<U>&);
    template<class U> optional& operator=(optional<U>&&);
    template<class... Args> T& emplace(Args&&...);
    template<class U, class... Args> T& emplace(initializer_list<U>, Args&&...);

    // 20.6.3.4, swap
    void swap(optional&) noexcept(see below);

    // 20.6.3.5, observers
    constexpr const T* operator->() const;
    constexpr T* operator->();
    constexpr const T& operator*() const&;
    constexpr T& operator*() &;
    constexpr T&& operator*() &&;
    constexpr const T&& operator*() const&&;
    constexpr explicit operator bool() const noexcept;
    constexpr bool has_value() const noexcept;
    constexpr const T& value() const&;
    constexpr T& value() &;
    constexpr T&& value() &&;
    constexpr const T&& value() const&&;
    template<class U> constexpr T value_or(U&&) const&;
    template<class U> constexpr T value_or(U&&) &&;

    // 20.6.3.6, modifiers
    void reset() noexcept;

  private:
    T *val;          // exposition only
  };
```

```
  template<class T>
    optional(T) -> optional<T>;
}
```

1 Any instance of `optional<T>` at any given time either contains a value or does not contain a value. When an instance of `optional<T>` *contains a value*, it means that an object of type `T`, referred to as the optional object's *contained value*, is allocated within the storage of the optional object. Implementations are not permitted to use additional storage, such as dynamic memory, to allocate its contained value. The contained value shall be allocated in a region of the `optional<T>` storage suitably aligned for the type `T`. When an object of type `optional<T>` is contextually converted to `bool`, the conversion returns `true` if the object contains a value; otherwise the conversion returns `false`.

2 Member `val` is provided for exposition only. When an `optional<T>` object contains a value, `val` points to the contained value.

3 `T` shall be a type other than *cv* `in_place_t` or *cv* `nullopt_t` that meets the *Cpp17Destructible* requirements (Table ??).

20.6.3.1 Constructors [optional.ctor]

```
constexpr optional() noexcept;
constexpr optional(nullopt_t) noexcept;
```

1 *Postconditions:* `*this` does not contain a value.

2 *Remarks:* No contained value is initialized. For every object type `T` these constructors shall be constexpr constructors (??).

```
constexpr optional(const optional& rhs);
```

3 *Mandates:* `is_copy_constructible_v<T>` is `true`.

4 *Effects:* If `rhs` contains a value, initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `*rhs`.

5 *Postconditions:* `bool(rhs) == bool(*this)`.

6 *Throws:* Any exception thrown by the selected constructor of `T`.

7 *Remarks:* ~~This constructor shall be defined as deleted unless `is_copy_constructible_v<T>` is `true`.~~ If `is_trivially_copy_constructible_v<T>` is `true`, this constructor is trivial.

```
constexpr optional(optional&& rhs) noexcept(see below);
```

8 *Constraints:* `is_move_constructible_v<T>` is `true`

9 *Effects:* If `rhs` contains a value, initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `std::move(*rhs)`. `bool(rhs)` is unchanged.

10 *Postconditions:* `bool(rhs) == bool(*this)`.

11 *Throws:* Any exception thrown by the selected constructor of `T`.

12 *Remarks:* The expression inside `noexcept` is equivalent to `is_nothrow_move_constructible_v<T>`. ~~This constructor shall not participate in overload resolution unless `is_move_constructible_v<T>` is `true`.~~ If `is_trivially_move_constructible_v<T>` is `true`, this constructor is trivial.

```
template<class... Args> constexpr explicit optional(in_place_t, Args&&... args);
```

13 *Constraints:* `is_constructible_v<T, Args...>` is `true`

14 *Effects:* Initializes the contained value as if direct-non-list-initializing an object of type `T` with the arguments `std::forward<Args>(args)...`.

15 *Postconditions:* `*this` contains a value.

16 *Throws:* Any exception thrown by the selected constructor of `T`.

17 *Remarks:* If `T`'s constructor selected for the initialization is a constexpr constructor, this constructor shall be a constexpr constructor. ~~This constructor shall not participate in overload resolution unless `is_constructible_v<T, Args...>` is `true`.~~

```
template<class U, class... Args>
  constexpr explicit optional(in_place_t, initializer_list<U> il, Args&&... args);
```

18 *Constraints:* `is_constructible_v<T, initializer_list<U>&, Args&&...>` is `true`.

19 *Effects:* Initializes the contained value as if direct-non-list-initializing an object of type `T` with the arguments `il, std::forward<Args>(args)...`.

20 *Postconditions:* `*this` contains a value.

21 *Throws:* Any exception thrown by the selected constructor of `T`.

22 *Remarks:* ~~This constructor shall not participate in overload resolution unless `is_constructible_v<T, initializer_`~~ ~~`is true`.~~ If `T`'s constructor selected for the initialization is a constexpr constructor, this constructor shall be a constexpr constructor.

```
template<class U = T> explicit(see below) constexpr optional(U&& v);
```

23 *Constraints:* `is_constructible_v<T, U&&>` is `true`, `is_same_v<remove_cvref_t<U>, in_place_t>` is `false`, and `is_same_v<remove_cvref_t<U>, optional>` is `false`.

24 *Effects:* Initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `std::forward<U>(v)`.

25 *Postconditions:* `*this` contains a value.

26 *Throws:* Any exception thrown by the selected constructor of `T`.

27 *Remarks:* If `T`'s selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor. ~~This constructor shall not participate in overload resolution unless `is_constructible_v<T, U&&>` is `true`, `is_same_v<remove_cvref_t<U>, in_place_t>` is `false`, and `is_same_v<remove_cvref_t<U>, optional>` is `false`.~~ The expression inside `explicit` is equivalent to:

```
!is_convertible_v<U, T>
```

```
template<class U> explicit(see below) optional(const optional<U>& rhs);
```

28 *Constraints:*

(28.1) — `is_constructible_v<T, const U&>` is `true`,

(28.2) — `is_constructible_v<T, optional<U>&>` is `false`,

(28.3) — `is_constructible_v<T, optional<U>&&>` is `false`,

(28.4) — `is_constructible_v<T, const optional<U>&>` is `false`,

(28.5) — `is_constructible_v<T, const optional<U>&&>` is `false`,

(28.6) — `is_convertible_v<optional<U>&, T>` is `false`,

(28.7) — `is_convertible_v<optional<U>&&, T>` is `false`,

(28.8) — `is_convertible_v<const optional<U>&, T>` is `false`, and

(28.9) — `is_convertible_v<const optional<U>&&, T>` is `false`.

29 *Effects:* If `rhs` contains a value, initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `*rhs`.

30 *Postconditions:* `bool(rhs) == bool(*this)`.

31 *Throws:* Any exception thrown by the selected constructor of `T`.

32 *Remarks:* This constructor shall not participate in overload resolution unless

(32.1) — `is_constructible_v<T, const U&>` is `true`,

(32.2) — `is_constructible_v<T, optional<U>&>` is `false`,

(32.3) — `is_constructible_v<T, optional<U>&&>` is `false`,

(32.4) — `is_constructible_v<T, const optional<U>&>` is `false`,

(32.5) — `is_constructible_v<T, const optional<U>&&>` is `false`,

(32.6) — `is_convertible_v<optional<U>&, T>` is `false`,

(32.7) — `is_convertible_v<optional<U>&&, T>` is `false`,

(32.8) — `is_convertible_v<const optional<U>&, T>` is `false`, and

(32.9) — `is_convertible_v<const optional<U>&&, T>` is `false`.

The expression inside `explicit` is equivalent to:

```
!is_convertible_v<const U&, T>
```

```
template<class U> explicit(see below) optional(optional<U>&& rhs);
```

33 *Constraints:*

(33.1) — `is_constructible_v<T, U&&>` is true,

(33.2) — `is_constructible_v<T, optional<U>&>` is `false`,

(33.3) — `is_constructible_v<T, optional<U>&&>` is `false`,

(33.4) — `is_constructible_v<T, const optional<U>&>` is `false`,

(33.5) — `is_constructible_v<T, const optional<U>&&>` is `false`,

(33.6) — `is_convertible_v<optional<U>&, T>` is `false`,

(33.7) — `is_convertible_v<optional<U>&&, T>` is `false`,

(33.8) — `is_convertible_v<const optional<U>&, T>` is `false`, and

(33.9) — `is_convertible_v<const optional<U>&&, T>` is `false`.

34 *Effects:* If `rhs` contains a value, initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `std::move(*rhs)`. `bool(rhs)` is unchanged.

35 *Postconditions:* `bool(rhs)` == `bool(*this)`.

36 *Throws:* Any exception thrown by the selected constructor of `T`.

37 *Remarks:* This constructor shall not participate in overload resolution unless

(37.1) — `is_constructible_v<T, U&&>` is true,

(37.2) — `is_constructible_v<T, optional<U>&>` is `false`,

(37.3) — `is_constructible_v<T, optional<U>&&>` is `false`,

(37.4) — `is_constructible_v<T, const optional<U>&>` is `false`,

(37.5) — `is_constructible_v<T, const optional<U>&&>` is `false`,

(37.6) — `is_convertible_v<optional<U>&, T>` is `false`,

(37.7) — `is_convertible_v<optional<U>&&, T>` is `false`,

(37.8) — `is_convertible_v<const optional<U>&, T>` is `false`, and

(37.9) — `is_convertible_v<const optional<U>&&, T>` is `false`.

The expression inside `explicit` is equivalent to:

```
!is_convertible_v<U, T>
```

20.6.3.2 Destructor [optional.dtor]

```
~optional();
```

1 *Effects:* If `is_trivially_destructible_v<T> != true` and `*this` contains a value, calls

```
val->T::~T()
```

2 *Remarks:* If `is_trivially_destructible_v<T>` is `true`, then this destructor is trivial.

20.6.3.3 Assignment [optional.assign]

```
optional<T>& operator=(nullopt_t) noexcept;
```

1 *Effects:* If `*this` contains a value, calls `val->T::~T()` to destroy the contained value; otherwise no effect.

2 *Postconditions:* `*this` does not contain a value.

3 *Returns:* `*this`.

4 *Postconditions:* `*this` does not contain a value.

```
constexpr optional<T>& operator=(const optional& rhs);
```

5 *Mandates:* `is_copy_constructible_v<T>` is `true` and `is_copy_assignable_v<T>` is `true`.

6 *Effects:* See Table 40.

Table 40: `optional::operator=(const optional&)` effects [tab:optional.assign.copy]

	***this contains a value**	***this does not contain a value**
rhs contains a value	assigns `*rhs` to the contained value	initializes the contained value as if direct-non-list-initializing an object of type `T` with `*rhs`
rhs does not contain a value	destroys the contained value by calling `val->T::~T()`	no effect

7 *Postconditions:* `bool(rhs) == bool(*this)`.

8 *Returns:* `*this`.

9 *Postconditions:* `bool(rhs) == bool(*this)`.

10 *Remarks:* If any exception is thrown, the result of the expression `bool(*this)` remains unchanged. If an exception is thrown during the call to `T`'s copy constructor, no effect. If an exception is thrown during the call to `T`'s copy assignment, the state of its contained value is as defined by the exception safety guarantee of `T`'s copy assignment. ~~This operator shall be defined as deleted unless `is_copy_constructible_v<T>` is `true` and `is_copy_assignable_v<T>` is `true`.~~ If `is_trivially_copy_constructible_v<T> && is_trivially_copy_assignable_v<T> && is_trivially_destructible_v<T>` is `true`, this assignment operator is trivial.

```
constexpr optional& operator=(optional&& rhs) noexcept(see below);
```

11 *Constraints:* `is_move_constructible_v<T>` is `true` and `is_move_assignable_v<T>` is `true`.

12 *Effects:* See Table 41. The result of the expression `bool(rhs)` remains unchanged.

Table 41: `optional::operator=(optional&&)` effects [tab:optional.assign.move]

	***this contains a value**	***this does not contain a value**
rhs contains a value	assigns `std::move(*rhs)` to the contained value	initializes the contained value as if direct-non-list-initializing an object of type `T` with `std::move(*rhs)`
rhs does not contain a value	destroys the contained value by calling `val->T::~T()`	no effect

13 *Postconditions:* `bool(rhs) == bool(*this)`.

14 *Returns:* `*this`.

15 *Postconditions:* `bool(rhs) == bool(*this)`.

16 *Remarks:* The expression inside `noexcept` is equivalent to:

```
is_nothrow_move_assignable_v<T> && is_nothrow_move_constructible_v<T>
```

17 If any exception is thrown, the result of the expression `bool(*this)` remains unchanged. If an exception is thrown during the call to `T`'s move constructor, the state of `*rhs.val` is determined by the exception safety guarantee of `T`'s move constructor. If an exception is thrown during the call to `T`'s move assignment, the state of `*val` and `*rhs.val` is determined by the exception safety guarantee of `T`'s move assignment. ~~This operator shall not participate in overload resolution unless `is_move_constructible_v<T>` is `true` and `is_move_assignable_v<T>` is `true`.~~ If `is_trivially_move_constructible_v<T> && is_trivially_move_assignable_v<T> && is_trivially_destructible_v<T>` is `true`, this assignment operator is trivial.

```
template<class U = T> optional<T>& operator=(U&& v);
```

18 *Constraints:* `is_same_v<remove_cvref_t<U>, optional>` is `false`, `conjunction_v<is_scalar<T>, is_same<T, decay_t<U>>>` is `false`, `is_constructible_v<T, U>` is `true`, and `is_assignable_v<T&, U>` is `true`.

19 *Effects:* If `*this` contains a value, assigns `std::forward<U>(v)` to the contained value; otherwise initializes the contained value as if direct-non-list-initializing object of type `T` with `std::forward<U>(v)`.

20 *Postconditions:* `*this` contains a value.

21 *Returns:* `*this`.

22 *Postconditions:* `*this` contains a value.

23 *Remarks:* If any exception is thrown, the result of the expression `bool(*this)` remains unchanged. If an exception is thrown during the call to `T`'s constructor, the state of `v` is determined by the exception safety guarantee of `T`'s constructor. If an exception is thrown during the call to `T`'s assignment, the state of `*val` and `v` is determined by the exception safety guarantee of `T`'s assignment. ~~This function shall not participate in overload resolution unless `is_same_v<remove_cvref_t<U>, optional>` is `false`, `conjunction_v<is_scalar<T>, is_same<T, decay_t<U>>>` is `false`, `is_constructible_v<T, U>` is `true`, and `is_assignable_v<T&, U>` is `true`.~~

```
template<class U> optional<T>& operator=(const optional<U>& rhs);
```

24 *Constraints:*

(24.1) — `is_constructible_v<T, const U&>` is `true`,

(24.2) — `is_assignable_v<T&, const U&>` is `true`,

(24.3) — `is_constructible_v<T, optional<U>&>` is `false`,

(24.4) — `is_constructible_v<T, optional<U>&&>` is `false`,

(24.5) — `is_constructible_v<T, const optional<U>&>` is `false`,

(24.6) — `is_constructible_v<T, const optional<U>&&>` is `false`,

(24.7) — `is_convertible_v<optional<U>&, T>` is `false`,

(24.8) — `is_convertible_v<optional<U>&&, T>` is `false`,

(24.9) — `is_convertible_v<const optional<U>&, T>` is `false`,

(24.10) — `is_convertible_v<const optional<U>&&, T>` is `false`,

(24.11) — `is_assignable_v<T&, optional<U>&>` is `false`,

(24.12) — `is_assignable_v<T&, optional<U>&&>` is `false`,

(24.13) — `is_assignable_v<T&, const optional<U>&>` is `false`, and

(24.14) — `is_assignable_v<T&, const optional<U>&&>` is `false`.

25 *Effects:* See Table 42.

Table 42: `optional::operator=(const optional<U>&)` effects [tab:optional.assign.copy.templ]

	`*this` contains a value	**`*this` does not contain a value**
`rhs` contains a value	assigns `*rhs` to the contained value	initializes the contained value as if direct-non-list-initializing an object of type `T` with `*rhs`
`rhs` does not contain a value	destroys the contained value by calling `val->T::~T()`	no effect

26 *Postconditions:* `bool(rhs) == bool(*this)`.

27 *Returns:* `*this`.

28 *Postconditions:* `bool(rhs) == bool(*this)`.

29 *Remarks:* If any exception is thrown, the result of the expression `bool(*this)` remains unchanged. If an exception is thrown during the call to `T`'s constructor, the state of `*rhs.val` is determined by the exception safety guarantee of `T`'s constructor. If an exception is thrown during the call to `T`'s assignment, the state of `*val` and `*rhs.val` is determined by the exception safety guarantee of `T`'s assignment. This function shall not participate in overload resolution unless

(29.1) — `is_constructible_v<T, const U&>` is `true`,

(29.2) — `is_assignable_v<T&, const U&>` is `true`,

(29.3) — `is_constructible_v<T, optional<U>&>` is `false`,

(29.4) — `is_constructible_v<T, optional<U>&&>` is `false`,

(29.5) — `is_constructible_v<T, const optional<U>&>` is `false`,

(29.6) — `is_constructible_v<T, const optional<U>&&>` is `false`,

(29.7) — `is_convertible_v<optional<U>&, T>` is `false`,

(29.8) — `is_convertible_v<optional<U>&&, T>` is `false`,

(29.9) — `is_convertible_v<const optional<U>&, T>` is `false`,

(29.10) — `is_convertible_v<const optional<U>&&, T>` is `false`,

(29.11) — `is_assignable_v<T&, optional<U>&>` is `false`,

(29.12) — `is_assignable_v<T&, optional<U>&&>` is `false`,

(29.13) — `is_assignable_v<T&, const optional<U>&>` is `false`, and

(29.14) — `is_assignable_v<T&, const optional<U>&&>` is `false`.

```
template<class U> optional<T>& operator=(optional<U>&& rhs);
```

30 *Constraints:*

`is_constructible_v<T, U>` is `true`,

`is_assignable_v<T&, U>` is `true`,

`is_constructible_v<T, optional<U>&>` is `false`,

`is_constructible_v<T, optional<U>&&>` is `false`,

`is_constructible_v<T, const optional<U>&>` is `false`,

`is_constructible_v<T, const optional<U>&&>` is `false`,

`is_convertible_v<optional<U>&, T>` is `false`,

`is_convertible_v<optional<U>&&, T>` is `false`,

`is_convertible_v<const optional<U>&, T>` is `false`,

`is_convertible_v<const optional<U>&&, T>` is `false`,

`is_assignable_v<T&, optional<U>&>` is `false`,

`is_assignable_v<T&, optional<U>&&>` is `false`,

`is_assignable_v<T&, const optional<U>&>` is `false`, and

`is_assignable_v<T&, const optional<U>&&>` is `false`.

31 *Effects:* See Table 43. The result of the expression `bool(rhs)` remains unchanged.

32 *Postconditions:* `bool(rhs) == bool(*this)`.

33 *Returns:* `*this`.

34 *Postconditions:* `bool(rhs) == bool(*this)`.

35 *Remarks:* If any exception is thrown, the result of the expression `bool(*this)` remains unchanged. If an exception is thrown during the call to `T`'s constructor, the state of `*rhs.val` is determined by the exception safety guarantee of `T`'s constructor. If an exception is thrown during the call to `T`'s assignment, the state of `*val` and `*rhs.val` is determined by the exception safety guarantee of `T`'s assignment. This function shall not participate in overload resolution unless

(35.1) — `is_constructible_v<T, U>` is `true`,

Table 43: `optional::operator=(optional<U>&&)` effects [tab:optional.assign.move.templ]

	`*this` contains a value	**`*this` does not contain a value**
`rhs` contains a value	assigns `std::move(*rhs)` to the contained value	initializes the contained value as if direct-non-list-initializing an object of type `T` with `std::move(*rhs)`
`rhs` does not contain a value	destroys the contained value by calling `val->T::~T()`	no effect

(35.2) — `is_assignable_v<T&, U>` is `true`,

(35.3) — `is_constructible_v<T, optional<U>&>` is `false`,

(35.4) — `is_constructible_v<T, optional<U>&&>` is `false`,

(35.5) — `is_constructible_v<T, const optional<U>&>` is `false`,

(35.6) — `is_constructible_v<T, const optional<U>&&>` is `false`,

(35.7) — `is_convertible_v<optional<U>&, T>` is `false`,

(35.8) — `is_convertible_v<optional<U>&&, T>` is `false`,

(35.9) — `is_convertible_v<const optional<U>&, T>` is `false`,

(35.10) — `is_convertible_v<const optional<U>&&, T>` is `false`,

(35.11) — `is_assignable_v<T&, optional<U>&>` is `false`,

(35.12) — `is_assignable_v<T&, optional<U>&&>` is `false`,

(35.13) — `is_assignable_v<T&, const optional<U>&>` is `false`, and

(35.14) — `is_assignable_v<T&, const optional<U>&&>` is `false`.

```
template<class... Args> T& emplace(Args&&... args);
```

36 ~~*Requires:*~~ *Preconditions:* `is_constructible_v<T, Args&&...>` is `true`.

37 *Effects:* Calls `*this = nullopt`. Then initializes the contained value as if direct-non-list-initializing an object of type `T` with the arguments `std::forward<Args>(args)...`.

38 *Postconditions:* `*this` contains a value.

39 *Returns:* A reference to the new contained value.

40 *Throws:* Any exception thrown by the selected constructor of `T`.

41 *Remarks:* If an exception is thrown during the call to `T`'s constructor, `*this` does not contain a value, and the previous `*val` (if any) has been destroyed.

```
template<class U, class... Args> T& emplace(initializer_list<U> il, Args&&... args);
```

42 *Constraints:* `is_constructible_v<T, initializer_list<U>&, Args&&...>` is `true`.

43 *Effects:* Calls `*this = nullopt`. Then initializes the contained value as if direct-non-list-initializing an object of type `T` with the arguments `il, std::forward<Args>(args)...`.

44 *Postconditions:* `*this` contains a value.

45 *Returns:* A reference to the new contained value.

46 *Throws:* Any exception thrown by the selected constructor of `T`.

47 *Remarks:* If an exception is thrown during the call to `T`'s constructor, `*this` does not contain a value, and the previous `*val` (if any) has been destroyed. ~~This function shall not participate in overload resolution unless `is_constructible_v<T, initializer_list<U>&, Args&&...>` is `true`.~~

20.6.3.4 Swap [optional.swap]

```
void swap(optional& rhs) noexcept(see below);
```

1 ~~*Requires:*~~ *Preconditions:* Lvalues of type `T` shall be swappable and `is_move_constructible_v<T>` is `true`.

2 *Effects:* See Table 44.

Table 44: `optional::swap(optional&)` effects [tab:optional.swap]

	***this contains a value**	***this does not contain a value**
rhs contains a value	calls `swap(*(*this), *rhs)`	initializes the contained value of `*this` as if direct-non-list-initializing an object of type `T` with the expression `std::move(*rhs)`, followed by `rhs.val->T::~T()`; postcondition is that `*this` contains a value and `rhs` does not contain a value
rhs does not contain a value	initializes the contained value of `rhs` as if direct-non-list-initializing an object of type `T` with the expression `std::move(*(*this))`, followed by `val->T::~T()`; postcondition is that `*this` does not contain a value and `rhs` contains a value	no effect

3 *Throws:* Any exceptions thrown by the operations in the relevant part of Table 44.

4 *Remarks:* The expression inside `noexcept` is equivalent to:

```
is_nothrow_move_constructible_v<T> && is_nothrow_swappable_v<T>
```

If any exception is thrown, the results of the expressions `bool(*this)` and `bool(rhs)` remain unchanged. If an exception is thrown during the call to function `swap`, the state of `*val` and `*rhs.val` is determined by the exception safety guarantee of `swap` for lvalues of `T`. If an exception is thrown during the call to `T`'s move constructor, the state of `*val` and `*rhs.val` is determined by the exception safety guarantee of `T`'s move constructor.

20.6.3.5 Observers [optional.observe]

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

1 ~~*Requires:*~~ *Preconditions:* `*this` contains a value.

2 *Returns:* `val`.

3 *Throws:* Nothing.

4 *Remarks:* These functions shall be constexpr functions.

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

5 ~~*Requires:*~~ *Preconditions:* `*this` contains a value.

6 *Returns:* `*val`.

7 *Throws:* Nothing.

8 *Remarks:* These functions shall be constexpr functions.

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

9 ~~*Requires:*~~ *Preconditions:* `*this` contains a value.

10 *Effects:* Equivalent to: `return std::move(*val);`

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

11 *Returns:* `true` if and only if `*this` contains a value.

12 *Remarks:* This function shall be a constexpr function.

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

13 *Returns:* `true` if and only if `*this` contains a value.

14 *Remarks:* This function shall be a constexpr function.

```
constexpr const T& value() const&;
constexpr T& value() &;
```

15 *Effects:* Equivalent to:

```
return bool(*this) ? *val : throw bad_optional_access();
```

```
constexpr T&& value() &&;
constexpr const T&& value() const&&;
```

16 *Effects:* Equivalent to:

```
return bool(*this) ? std::move(*val) : throw bad_optional_access();
```

```
template<class U> constexpr T value_or(U&& v) const&;
```

17 *Mandates:* `is_copy_constructible_v<T> && is_convertible_v<U&&, T>` is `false`.

18 *Effects:* Equivalent to:

```
return bool(*this) ? **this : static_cast<T>(std::forward<U>(v));
```

19 *Remarks:* If `is_copy_constructible_v<T> && is_convertible_v<U&&, T>` is `false`, the program is ill-formed.

```
template<class U> constexpr T value_or(U&& v) &&;
```

20 *Mandates:* `is_move_constructible_v<T> && is_convertible_v<U&&, T>` is `false`.

21 *Effects:* Equivalent to:

```
return bool(*this) ? std::move(**this) : static_cast<T>(std::forward<U>(v));
```

22 *Remarks:* If `is_move_constructible_v<T> && is_convertible_v<U&&, T>` is `false`, the program is ill-formed.

20.6.3.6 Modifiers [optional.mod]

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

1 *Effects:* If `*this` contains a value, calls `val->T::~T()` to destroy the contained value; otherwise no effect.

2 *Postconditions:* `*this` does not contain a value.

20.6.4 No-value state indicator [optional.nullopt]

```
struct nullopt_t{see below};
inline constexpr nullopt_t nullopt(unspecified);
```

1 The struct `nullopt_t` is an empty class type used as a unique type to indicate the state of not containing a value for `optional` objects. In particular, `optional<T>` has a constructor with `nullopt_t` as a single argument; this indicates that an optional object not containing a value shall be constructed.

2 Type `nullopt_t` shall not have a default constructor or an initializer-list constructor, and shall not be an aggregate.

20.6.5 Class `bad_optional_access` [optional.bad.access]

```
class bad_optional_access : public exception {
public:
  // see ?? for the specification of the special member functions
  const char* what() const noexcept override;
};
```

1 The class `bad_optional_access` defines the type of objects thrown as exceptions to report the situation where an attempt is made to access the value of an optional object that does not contain a value.

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

2 *Effects:* Constructs an object of class `bad_optional_access`.

3 *Postconditions:* `what()` returns an implementation-defined NTBS.

Returns: An implementation-defined NTBS.

20.6.6 Relational operators [optional.relops]

`template<class T, class U> constexpr bool operator==(const optional<T>& x, const optional<U>& y);`

1 ~~*Requires:*~~ *Preconditions:* The expression `*x == *y` shall be well-formed and its result shall be convertible to `bool`. [*Note*: `T` need not be *Cpp17EqualityComparable*. — *end note*]

2 *Returns:* If `bool(x) != bool(y)`, `false`; otherwise if `bool(x) == false`, `true`; otherwise `*x == *y`.

3 *Remarks:* Specializations of this function template for which `*x == *y` is a core constant expression shall be constexpr functions.

`template<class T, class U> constexpr bool operator!=(const optional<T>& x, const optional<U>& y);`

4 ~~*Requires:*~~ *Preconditions:* The expression `*x != *y` shall be well-formed and its result shall be convertible to `bool`.

5 *Returns:* If `bool(x) != bool(y)`, `true`; otherwise, if `bool(x) == false`, `false`; otherwise `*x != *y`.

6 *Remarks:* Specializations of this function template for which `*x != *y` is a core constant expression shall be constexpr functions.

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

7 ~~*Requires:*~~ *Preconditions:* `*x < *y` shall be well-formed and its result shall be convertible to `bool`.

8 *Returns:* If `!y`, `false`; otherwise, if `!x`, `true`; otherwise `*x < *y`.

9 *Remarks:* Specializations of this function template for which `*x < *y` is a core constant expression shall be constexpr functions.

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

10 ~~*Requires:*~~ *Preconditions:* The expression `*x > *y` shall be well-formed and its result shall be convertible to `bool`.

11 *Returns:* If `!x`, `false`; otherwise, if `!y`, `true`; otherwise `*x > *y`.

12 *Remarks:* Specializations of this function template for which `*x > *y` is a core constant expression shall be constexpr functions.

`template<class T, class U> constexpr bool operator<=(const optional<T>& x, const optional<U>& y);`

13 ~~*Requires:*~~ *Preconditions:* The expression `*x <= *y` shall be well-formed and its result shall be convertible to `bool`.

14 *Returns:* If `!x`, `true`; otherwise, if `!y`, `false`; otherwise `*x <= *y`.

15 *Remarks:* Specializations of this function template for which `*x <= *y` is a core constant expression shall be constexpr functions.

`template<class T, class U> constexpr bool operator>=(const optional<T>& x, const optional<U>& y);`

16 ~~*Requires:*~~ *Preconditions:* The expression `*x >= *y` shall be well-formed and its result shall be convertible to `bool`.

17 *Returns:* If `!y`, `true`; otherwise, if `!x`, `false`; otherwise `*x >= *y`.

18 *Remarks:* Specializations of this function template for which `*x >= *y` is a core constant expression shall be constexpr functions.

```
template<class T, three_way_comparable_with<T> U>
  constexpr compare_three_way_result_t<T,U>
    operator<=>(const optional<T>& x, const optional<U>& y);
```

19 *Returns:* If `x && y`, `*x <=> *y`; otherwise `bool(x) <=> bool(y)`.

20 *Remarks:* Specializations of this function template for which `*x <=> *y` is a core constant expression shall be constexpr functions.

20.6.7 Comparison with `nullopt` [optional.nullops]

```
template<class T> constexpr bool operator==(const optional<T>& x, nullopt_t) noexcept;
```

1 *Returns:* `!x`.

```
template<class T> constexpr strong_ordering operator<=>(const optional<T>& x, nullopt_t) noexcept;
```

2 *Returns:* `bool(x) <=> false`.

20.6.8 Comparison with `T` [optional.comp.with.t]

```
template<class T, class U> constexpr bool operator==(const optional<T>& x, const U& v);
```

1 ~~*Requires:*~~ *Preconditions:* The expression `*x == v` shall be well-formed and its result shall be convertible to `bool`. [*Note*: `T` need not be *Cpp17EqualityComparable*. — *end note*]

2 *Effects:* Equivalent to: `return bool(x) ? *x == v : false;`

```
template<class T, class U> constexpr bool operator==(const T& v, const optional<U>& x);
```

3 ~~*Requires:*~~ *Preconditions:* The expression `v == *x` shall be well-formed and its result shall be convertible to `bool`.

4 *Effects:* Equivalent to: `return bool(x) ? v == *x : false;`

```
template<class T, class U> constexpr bool operator!=(const optional<T>& x, const U& v);
```

5 ~~*Requires:*~~ *Preconditions:* The expression `*x != v` shall be well-formed and its result shall be convertible to `bool`.

6 *Effects:* Equivalent to: `return bool(x) ? *x != v : true;`

```
template<class T, class U> constexpr bool operator!=(const T& v, const optional<U>& x);
```

7 ~~*Requires:*~~ *Preconditions:* The expression `v != *x` shall be well-formed and its result shall be convertible to `bool`.

8 *Effects:* Equivalent to: `return bool(x) ? v != *x : true;`

```
template<class T, class U> constexpr bool operator<(const optional<T>& x, const U& v);
```

9 ~~*Requires:*~~ *Preconditions:* The expression `*x < v` shall be well-formed and its result shall be convertible to `bool`.

10 *Effects:* Equivalent to: `return bool(x) ? *x < v : true;`

```
template<class T, class U> constexpr bool operator<(const T& v, const optional<U>& x);
```

11 ~~*Requires:*~~ *Preconditions:* The expression `v < *x` shall be well-formed and its result shall be convertible to `bool`.

12 *Effects:* Equivalent to: `return bool(x) ? v < *x : false;`

```
template<class T, class U> constexpr bool operator>(const optional<T>& x, const U& v);
```

13 ~~*Requires:*~~ *Preconditions:* The expression `*x > v` shall be well-formed and its result shall be convertible to `bool`.

14 *Effects:* Equivalent to: `return bool(x) ? *x > v : false;`

```
template<class T, class U> constexpr bool operator>(const T& v, const optional<U>& x);
```

15 ~~*Requires:*~~ *Preconditions:* The expression `v > *x` shall be well-formed and its result shall be convertible to `bool`.

16 *Effects:* Equivalent to: `return bool(x) ? v > *x : true;`

```
template<class T, class U> constexpr bool operator<=(const optional<T>& x, const U& v);
```

17 ~~*Requires:*~~ *Preconditions:* The expression `*x <= v` shall be well-formed and its result shall be convertible to `bool`.

18 *Effects:* Equivalent to: `return bool(x) ? *x <= v : true;`

```
template<class T, class U> constexpr bool operator<=(const T& v, const optional<U>& x);
```

19 ~~*Requires:*~~ *Preconditions:* The expression `v <= *x` shall be well-formed and its result shall be convertible to `bool`.

20 *Effects:* Equivalent to: `return bool(x) ? v <= *x : false;`

```
template<class T, class U> constexpr bool operator>=(const optional<T>& x, const U& v);
```

21 ~~*Requires:*~~ *Preconditions:* The expression `*x >= v` shall be well-formed and its result shall be convertible to `bool`.

22 *Effects:* Equivalent to: `return bool(x) ? *x >= v : false;`

```
template<class T, class U> constexpr bool operator>=(const T& v, const optional<U>& x);
```

23 ~~*Requires:*~~ *Preconditions:* The expression `v >= *x` shall be well-formed and its result shall be convertible to `bool`.

24 *Effects:* Equivalent to: `return bool(x) ? v >= *x : true;`

```
template<class T, three_way_comparable_with<T> U>
  constexpr compare_three_way_result_t<T,U>
    operator<=>(const optional<T>& x, const U& v);
```

25 *Effects:* Equivalent to: `return bool(x) ? *x <=> v : strong_ordering::less;`

20.6.9 Specialized algorithms [optional.specalg]

```
template<class T> void swap(optional<T>& x, optional<T>& y) noexcept(noexcept(x.swap(y)));
```

1 *Constraints:* `is_move_constructible_v<T>` is `true` and `is_swappable_v<T>` is `true`.

2 *Effects:* Calls `x.swap(y)`.

3 *Remarks:* This function shall not participate in overload resolution unless `is_move_constructible_v<T>` is `true` and `is_swappable_v<T>` is `true`.

```
template<class T> constexpr optional<decay_t<T>> make_optional(T&& v);
```

4 *Returns:* `optional<decay_t<T>>(std::forward<T>(v))`.

```
template<class T, class...Args>
  constexpr optional<T> make_optional(Args&&... args);
```

5 *Effects:* Equivalent to: `return optional<T>(in_place, std::forward<Args>(args)...);`

```
template<class T, class U, class... Args>
  constexpr optional<T> make_optional(initializer_list<U> il, Args&&... args);
```

6 *Effects:* Equivalent to: `return optional<T>(in_place, il, std::forward<Args>(args)...);`

20.6.10 Hash support [optional.hash]

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

1 The specialization `hash<optional<T>>` is enabled (20.14.18) if and only if `hash<remove_const_t<T>>` is enabled. When enabled, for an object `o` of type `optional<T>`, if `bool(o) == true`, then `hash<optional<T>>()(o)` shall evaluate to the same value as `hash<remove_const_t<T>>()(*o)`; otherwise it evaluates to an unspecified value. The member functions are not guaranteed to be `noexcept`.

20.7 Variants [variant]

20.7.1 In general [variant.general]

1 A variant object holds and manages the lifetime of a value. If the `variant` holds a value, that value's type has to be one of the template argument types given to variant. These template arguments are called alternatives.

20.7.2 Header <variant> synopsis [variant.syn]

```
namespace std {
  // 20.7.3, class template variant
  template<class... Types>
    class variant;

  // 20.7.4, variant helper classes
  template<class T> struct variant_size;                          // not defined
  template<class T> struct variant_size<const T>;
  template<class T> struct variant_size<volatile T>;
  template<class T> struct variant_size<const volatile T>;
  template<class T>
    inline constexpr size_t variant_size_v = variant_size<T>::value;

  template<class... Types>
    struct variant_size<variant<Types...>>;

  template<size_t I, class T> struct variant_alternative;         // not defined
  template<size_t I, class T> struct variant_alternative<I, const T>;
  template<size_t I, class T> struct variant_alternative<I, volatile T>;
  template<size_t I, class T> struct variant_alternative<I, const volatile T>;
  template<size_t I, class T>
    using variant_alternative_t = typename variant_alternative<I, T>::type;

  template<size_t I, class... Types>
    struct variant_alternative<I, variant<Types...>>;

  inline constexpr size_t variant_npos = -1;

  // 20.7.5, value access
  template<class T, class... Types>
    constexpr bool holds_alternative(const variant<Types...>&) noexcept;

  template<size_t I, class... Types>
    constexpr variant_alternative_t<I, variant<Types...>>& get(variant<Types...>&);
  template<size_t I, class... Types>
    constexpr variant_alternative_t<I, variant<Types...>>&& get(variant<Types...>&&);
  template<size_t I, class... Types>
    constexpr const variant_alternative_t<I, variant<Types...>>& get(const variant<Types...>&);
  template<size_t I, class... Types>
    constexpr const variant_alternative_t<I, variant<Types...>>&& get(const variant<Types...>&&);

  template<class T, class... Types>
    constexpr T& get(variant<Types...>&);
  template<class T, class... Types>
    constexpr T&& get(variant<Types...>&&);
  template<class T, class... Types>
    constexpr const T& get(const variant<Types...>&);
  template<class T, class... Types>
    constexpr const T&& get(const variant<Types...>&&);

  template<size_t I, class... Types>
    constexpr add_pointer_t<variant_alternative_t<I, variant<Types...>>>
      get_if(variant<Types...>*) noexcept;
  template<size_t I, class... Types>
    constexpr add_pointer_t<const variant_alternative_t<I, variant<Types...>>>
      get_if(const variant<Types...>*) noexcept;
```

```
  template<class T, class... Types>
    constexpr add_pointer_t<T>
      get_if(variant<Types...>*) noexcept;
  template<class T, class... Types>
    constexpr add_pointer_t<const T>
      get_if(const variant<Types...>*) noexcept;

  // 20.7.6, relational operators
  template<class... Types>
    constexpr bool operator==(const variant<Types...>&, const variant<Types...>&);
  template<class... Types>
    constexpr bool operator!=(const variant<Types...>&, const variant<Types...>&);
  template<class... Types>
    constexpr bool operator<(const variant<Types...>&, const variant<Types...>&);
  template<class... Types>
    constexpr bool operator>(const variant<Types...>&, const variant<Types...>&);
  template<class... Types>
    constexpr bool operator<=(const variant<Types...>&, const variant<Types...>&);
  template<class... Types>
    constexpr bool operator>=(const variant<Types...>&, const variant<Types...>&);
  template<class... Types> requires (three_way_comparable<Types> && ...)
    constexpr common_comparison_category_t<compare_three_way_result_t<Types>...>
      operator<=>(const variant<Types...>&, const variant<Types...>&);

  // 20.7.7, visitation
  template<class Visitor, class... Variants>
    constexpr see below visit(Visitor&&, Variants&&...);
  template<class R, class Visitor, class... Variants>
    constexpr R visit(Visitor&&, Variants&&...);

  // 20.7.8, class monostate
  struct monostate;

  // 20.7.9, monostate relational operators
  constexpr bool operator==(monostate, monostate) noexcept;
  constexpr strong_ordering operator<=>(monostate, monostate) noexcept;

  // 20.7.10, specialized algorithms
  template<class... Types>
    void swap(variant<Types...>&, variant<Types...>&) noexcept(see below);

  // 20.7.11, class bad_variant_access
  class bad_variant_access;

  // 20.7.12, hash support
  template<class T> struct hash;
  template<class... Types> struct hash<variant<Types...>>;
  template<> struct hash<monostate>;
}
```

20.7.3 Class template variant [variant.variant]

```
namespace std {
  template<class... Types>
  class variant {
  public:
    // 20.7.3.1, constructors
    constexpr variant() noexcept(see below);
    constexpr variant(const variant&);
    constexpr variant(variant&&) noexcept(see below);

    template<class T>
      constexpr variant(T&&) noexcept(see below);
```

```
    template<class T, class... Args>
      constexpr explicit variant(in_place_type_t<T>, Args&&...);
    template<class T, class U, class... Args>
      constexpr explicit variant(in_place_type_t<T>, initializer_list<U>, Args&&...);

    template<size_t I, class... Args>
      constexpr explicit variant(in_place_index_t<I>, Args&&...);
    template<size_t I, class U, class... Args>
      constexpr explicit variant(in_place_index_t<I>, initializer_list<U>, Args&&...);

    // 20.7.3.2, destructor
    ~variant();

    // 20.7.3.3, assignment
    constexpr variant& operator=(const variant&);
    constexpr variant& operator=(variant&&) noexcept(see below);

    template<class T> variant& operator=(T&&) noexcept(see below);

    // 20.7.3.4, modifiers
    template<class T, class... Args>
      T& emplace(Args&&...);
    template<class T, class U, class... Args>
      T& emplace(initializer_list<U>, Args&&...);
    template<size_t I, class... Args>
      variant_alternative_t<I, variant<Types...>>& emplace(Args&&...);
    template<size_t I, class U, class... Args>
      variant_alternative_t<I, variant<Types...>>& emplace(initializer_list<U>, Args&&...);

    // 20.7.3.5, value status
    constexpr bool valueless_by_exception() const noexcept;
    constexpr size_t index() const noexcept;

    // 20.7.3.6, swap
    void swap(variant&) noexcept(see below);
  };
}
```

1 Any instance of `variant` at any given time either holds a value of one of its alternative types or holds no value. When an instance of `variant` holds a value of alternative type `T`, it means that a value of type `T`, referred to as the `variant` object's *contained value*, is allocated within the storage of the `variant` object. Implementations are not permitted to use additional storage, such as dynamic memory, to allocate the contained value. The contained value shall be allocated in a region of the `variant` storage suitably aligned for all types in `Types`.

2 All types in `Types` shall meet the *Cpp17Destructible* requirements (Table **??**).

3 A program that instantiates the definition of `variant` with no template arguments is ill-formed.

20.7.3.1 Constructors [variant.ctor]

1 In the descriptions that follow, let i be in the range `[0, sizeof...(Types))`, and $\texttt{T}_i$ be the i^{th} type in `Types`.

```
constexpr variant() noexcept(see below);
```

2 *Constraints:* `is_default_constructible_v<`$\texttt{T}_0$`>` is `true`.

3 *Effects:* Constructs a `variant` holding a value-initialized value of type $\texttt{T}_0$.

4 *Postconditions:* `valueless_by_exception()` is `false` and `index()` is `0`.

5 *Throws:* Any exception thrown by the value-initialization of $\texttt{T}_0$.

6 *Remarks:* This function shall be `constexpr` if and only if the value-initialization of the alternative type $\texttt{T}_0$ would satisfy the requirements for a constexpr function. The expression inside `noexcept` is equivalent to `is_nothrow_default_constructible_v<`$\texttt{T}_0$`>`. ~~This function shall not participate in overload~~

~~resolution unless `is_default_constructible_v<T0>` is `true`.~~ [*Note*: See also class `monostate`. — *end note*]

`constexpr variant(const variant& w);`

7 *Mandates:* `is_copy_constructible_v<Ti>` is `true` for all i.

8 *Effects:* If `w` holds a value, initializes the `variant` to hold the same alternative as `w` and direct-initializes the contained value with `get<j>(w)`, where `j` is `w.index()`. Otherwise, initializes the `variant` to not hold a value.

9 *Throws:* Any exception thrown by direct-initializing any `Ti` for all i.

10 *Remarks:* ~~This constructor shall be defined as deleted unless `is_copy_constructible_v<Ti>` is `true` for all i.~~ If `is_trivially_copy_constructible_v<Ti>` is `true` for all i, this constructor is trivial.

`constexpr variant(variant&& w) noexcept(`*see below*`);`

11 *Constraints:* `is_move_constructible_v<Ti>` is `true` for all i.

12 *Effects:* If `w` holds a value, initializes the `variant` to hold the same alternative as `w` and direct-initializes the contained value with `get<j>(std::move(w))`, where `j` is `w.index()`. Otherwise, initializes the `variant` to not hold a value.

13 *Throws:* Any exception thrown by move-constructing any `Ti` for all i.

14 *Remarks:* The expression inside `noexcept` is equivalent to the logical AND of `is_nothrow_move_constructible_v<Ti>` for all i. ~~This function shall not participate in overload resolution unless `is_move_constructible_v<Ti>` is `true` for all i.~~ If `is_trivially_move_constructible_v<Ti>` is `true` for all i, this constructor is trivial.

`template<class T> constexpr variant(T&& t) noexcept(`*see below*`);`

15 Let `Tj` be a type that is determined as follows: build an imaginary function *FUN*`(Ti)` for each alternative type `Ti` for which `Ti x[] = {std::forward<T>(t)};` is well-formed for some invented variable `x` and, if `Ti` is *cv* `bool`, `remove_cvref_t<T>` is `bool`. The overload *FUN*`(Tj)` selected by overload resolution for the expression *FUN*`(std::forward<T>(t))` defines the alternative `Tj` which is the type of the contained value after construction.

16 *Constraints:*

(16.1) — `sizeof...(Types)` is nonzero,

(16.2) — `is_same_v<remove_cvref_t<T>, variant>` is `false`,

(16.3) — `remove_cvref_t<T>` is neither a specialization of `in_place_type_t` nor a specialization of `in_place_index_t`,

(16.4) — `is_constructible_v<Tj, T>` is `true`, and

(16.5) — the expression *FUN*`(std::forward<T>(t))` (with *FUN* being the above-mentioned set of imaginary functions) is well-formed.

17 *Effects:* Initializes `*this` to hold the alternative type `Tj` and direct-initializes the contained value as if direct-non-list-initializing it with `std::forward<T>(t)`.

18 *Postconditions:* `holds_alternative<Tj>(*this)` is `true`.

19 *Throws:* Any exception thrown by the initialization of the selected alternative `Tj`.

20 *Remarks:* This function shall not participate in overload resolution unless

(20.1) — `sizeof...(Types)` is nonzero,

(20.2) — `is_same_v<remove_cvref_t<T>, variant>` is `false`,

(20.3) — `remove_cvref_t<T>` is neither a specialization of `in_place_type_t` nor a specialization of `in_place_index_t`,

(20.4) — `is_constructible_v<Tj, T>` is `true`, and

(20.5) — the expression *FUN*`(std::forward<T>(t))` (with *FUN* being the above-mentioned set of imaginary functions) is well-formed.

21 *Remarks:* [*Note*:

```
variant<string, string> v("abc");
```

is ill-formed, as both alternative types have an equally viable constructor for the argument. — *end note*]

22 The expression inside `noexcept` is equivalent to `is_nothrow_constructible_v<`T_j`, T>`. If T_j's selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

```
template<class T, class... Args> constexpr explicit variant(in_place_type_t<T>, Args&&... args);
```

23 *Constraints:* there is exactly one occurrence of `T` in `Types...` and `is_constructible_v<T, Args...>` is `true`.

24 *Effects:* Initializes the contained value as if direct-non-list-initializing an object of type `T` with the arguments `std::forward<Args>(args)...`.

25 *Postconditions:* `holds_alternative<T>(*this)` is `true`.

26 *Throws:* Any exception thrown by calling the selected constructor of `T`.

27 *Remarks:* ~~This function shall not participate in overload resolution unless there is exactly one occurrence of `T` in `Types` and `is_constructible_v<T, Args...>` is `true`.~~ If `T`'s selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

```
template<class T, class U, class... Args>
  constexpr explicit variant(in_place_type_t<T>, initializer_list<U> il, Args&&... args);
```

28 *Constraints:* there is exactly one occurrence of `T` in `Types...` and `is_constructible_v<T, initializer_list<U>&, Args...>` is `true`.

29 *Effects:* Initializes the contained value as if direct-non-list-initializing an object of type `T` with the arguments `il, std::forward<Args>(args)...`.

30 *Postconditions:* `holds_alternative<T>(*this)` is `true`.

31 *Throws:* Any exception thrown by calling the selected constructor of `T`.

32 *Remarks:* ~~This function shall not participate in overload resolution unless there is exactly one occurrence of `T` in `Types` and `is_constructible_v<T, initializer_list<U>&, Args...>` is `true`.~~ If `T`'s selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

```
template<size_t I, class... Args> constexpr explicit variant(in_place_index_t<I>, Args&&... args);
```

33 *Constraints:*

(33.1) — `I` is less than `sizeof...(Types)` and

(33.2) — `is_constructible_v<`T_I`, Args...>` is `true`.

34 *Effects:* Initializes the contained value as if direct-non-list-initializing an object of type T_I with the arguments `std::forward<Args>(args)...`.

35 *Postconditions:* `index()` is `I`.

36 *Throws:* Any exception thrown by calling the selected constructor of T_I.

37 *Remarks:* This function shall not participate in overload resolution unless

(37.1) — `I` is less than `sizeof...(Types)` and

(37.2) — `is_constructible_v<`T_I`, Args...>` is `true`.

If T_I's selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

```
template<size_t I, class U, class... Args>
  constexpr explicit variant(in_place_index_t<I>, initializer_list<U> il, Args&&... args);
```

38 *Constraints:*

(38.1) — `I` is less than `sizeof...(Types)` and

(38.2) — `is_constructible_v<`T_I`, initializer_list<U>&, Args...>` is `true`.

39 *Effects:* Initializes the contained value as if direct-non-list-initializing an object of type T_I with the arguments `il, std::forward<Args>(args)...`.

40 *Postconditions:* `index()` is `I`.

41 *Remarks:* This function shall not participate in overload resolution unless

(41.1) — `I` is less than `sizeof...(Types)` and

(41.2) — `is_constructible_v<`T_I`, initializer_list<U>&, Args...>` is `true`.

If T_I's selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

20.7.3.2 Destructor [variant.dtor]

```
~variant();
```

1 *Effects:* If `valueless_by_exception()` is `false`, destroys the currently contained value.

2 *Remarks:* If `is_trivially_destructible_v<`T_i`>` is `true` for all T_i, then this destructor is trivial.

20.7.3.3 Assignment [variant.assign]

```
constexpr variant& operator=(const variant& rhs);
```

1 Let j be `rhs.index()`.

2 *Mandates:* This operator shall be defined as deleted unless `is_copy_constructible_v<`T_i`> && is_copy_assignable_v<`T_i`>` is `true` for all i.

3 *Effects:*

(3.1) — If neither `*this` nor `rhs` holds a value, there is no effect.

(3.2) — Otherwise, if `*this` holds a value but `rhs` does not, destroys the value contained in `*this` and sets `*this` to not hold a value.

(3.3) — Otherwise, if `index() ==` j, assigns the value contained in `rhs` to the value contained in `*this`.

(3.4) — Otherwise, if either `is_nothrow_copy_constructible_v<`T_j`>` is `true` or `is_nothrow_move_constructible_v<`T_j`>` is `false`, equivalent to `emplace<`j`>(get<`j`>(rhs))`.

(3.5) — Otherwise, equivalent to `operator=(variant(rhs))`.

4 *Returns:* `*this`.

5 *Postconditions:* `index() == rhs.index()`.

6 *Remarks:* ~~This operator shall be defined as deleted unless `is_copy_constructible_v<`T_i`> && is_copy_assignable_v<`T_i`>` is `true` for all i.~~ If `is_trivially_copy_constructible_v<`T_i`> && is_trivially_copy_assignable_v<`T_i`> && is_trivially_destructible_v<`T_i`>` is true for all i, this assignment operator is trivial.

```
constexpr variant& operator=(variant&& rhs) noexcept(see below);
```

7 Let j be `rhs.index()`.

8 *Constraints:* `is_move_constructible_v<`T_i`> && is_move_assignable_v<`T_i`>` is `true` for all i.

9 *Effects:*

(9.1) — If neither `*this` nor `rhs` holds a value, there is no effect.

(9.2) — Otherwise, if `*this` holds a value but `rhs` does not, destroys the value contained in `*this` and sets `*this` to not hold a value.

(9.3) — Otherwise, if `index() ==` j, assigns `get<`j`>(std::move(rhs))` to the value contained in `*this`.

(9.4) — Otherwise, equivalent to `emplace<`j`>(get<`j`>(std::move(rhs)))`.

10 *Returns:* `*this`.

11 *Remarks:* ~~This function shall not participate in overload resolution unless `is_move_constructible_v<`T_i`> && is_move_assignable_v<`T_i`>` is `true` for all i.~~ If `is_trivially_move_constructible_v<`T_i`> && is_trivially_move_assignable_v<`T_i`> && is_trivially_destructible_v<`T_i`>` is true for all i, this assignment operator is trivial. The expression inside `noexcept` is equivalent to: `is_nothrow_move_constructible_v<`T_i`> && is_nothrow_move_assignable_v<`T_i`>` for all i.

(11.1) — If an exception is thrown during the call to T_j's move construction (with j being `rhs.index()`), the `variant` will hold no value.

(11.2) — If an exception is thrown during the call to T_j's move assignment, the state of the contained value is as defined by the exception safety guarantee of T_j's move assignment; `index()` will be j.

```
template<class T> variant& operator=(T&& t) noexcept(see below);
```

12 Let T_j be a type that is determined as follows: build an imaginary function *FUN*(`T`$_i$) for each alternative type `T`$_i$ for which `T`$_i$ `x[] = {std::forward<T>(t)};` is well-formed for some invented variable `x` and, if `T`$_i$ is *cv* `bool`, `remove_cvref_t<T>` is `bool`. The overload *FUN*(`T`$_j$) selected by overload resolution for the expression *FUN*`(std::forward<T>(t))` defines the alternative `T`$_j$ which is the type of the contained value after assignment.

13 *Constraints:*

(13.1) — `is_same_v<remove_cvref_t<T>, variant>` is `false`,

(13.2) — `is_assignable_v<T`$_j$`&, T> && is_constructible_v<T`$_j$`, T>` is `true`, and

(13.3) — the expression *FUN*`(std::forward<T>(t))` (with *FUN* being the above-mentioned set of imaginary functions) is well-formed.

14 *Effects:*

(14.1) — If `*this` holds a `T`$_j$, assigns `std::forward<T>(t)` to the value contained in `*this`.

(14.2) — Otherwise, if `is_nothrow_constructible_v<T`$_j$`, T> || !is_nothrow_move_constructible_v<T`$_j$`>` is `true`, equivalent to `emplace<`j`>(std::forward<T>(t))`.

(14.3) — Otherwise, equivalent to `operator=(variant(std::forward<T>(t)))`.

15 *Postconditions:* `holds_alternative<T`$_j$`>(*this)` is `true`, with `T`$_j$ selected by the imaginary function overload resolution described above.

16 *Returns:* `*this`.

17 *Remarks:* This function shall not participate in overload resolution unless

(17.1) — `is_same_v<remove_cvref_t<T>, variant>` is `false`,

(17.2) — `is_assignable_v<T`$_j$`&, T> && is_constructible_v<T`$_j$`, T>` is `true`, and

(17.3) — the expression *FUN*`(std::forward<T>(t))` (with *FUN* being the above-mentioned set of imaginary functions) is well-formed.

18 *Remarks:* [*Note*:

```
variant<string, string> v;
v = "abc";
```

is ill-formed, as both alternative types have an equally viable constructor for the argument. — *end note*]

19 The expression inside `noexcept` is equivalent to:

```
is_nothrow_assignable_v<T_j&, T> && is_nothrow_constructible_v<T_j, T>
```

(19.1) — If an exception is thrown during the assignment of `std::forward<T>(t)` to the value contained in `*this`, the state of the contained value and `t` are as defined by the exception safety guarantee of the assignment expression; `valueless_by_exception()` will be `false`.

(19.2) — If an exception is thrown during the initialization of the contained value, the `variant` object might not hold a value.

20.7.3.4 Modifiers [variant.mod]

```
template<class T, class... Args> T& emplace(Args&&... args);
```

1 Let I be the zero-based index of `T` in `Types`.

2 *Constraints:* `is_constructible_v<T, Args...>` is `true`, and `T` occurs exactly once in `Types`.

3 *Effects:* Equivalent to: `return emplace<`I`>(std::forward<Args>(args)...);`

4 *Remarks:* This function shall not participate in overload resolution unless `is_constructible_v<T, Args...>` is `true`, and `T` occurs exactly once in `Types`.

```
template<class T, class U, class... Args> T& emplace(initializer_list<U> il, Args&&... args);
```

5 Let I be the zero-based index of `T` in `Types`.

6 *Constraints:* `is_constructible_v<T, initializer_list<U>&, Args...>` is `true`, and `T` occurs exactly once in `Types...`.

7 *Effects:* Equivalent to: `return emplace<I>(il, std::forward<Args>(args)...);`

8 *Remarks:* This function shall not participate in overload resolution unless `is_constructible_v<T, initializer_list<U>&, Args...>` is `true`, and `T` occurs exactly once in `Types`.

```
template<size_t I, class... Args>
  variant_alternative_t<I, variant<Types...>>& emplace(Args&&... args);
```

9 ~~*Requires:*~~ *Mandates:* `I < sizeof...(Types)`.

10 *Constraints:* `is_constructible_v<T`$_I$`, Args...>` is `true`.

11 *Effects:* Destroys the currently contained value if `valueless_by_exception()` is `false`. Then initializes the contained value as if direct-non-list-initializing a value of type `T`$_I$ with the arguments `std::forward<Args>(args)...`.

12 *Postconditions:* `index()` is `I`.

13 *Returns:* A reference to the new contained value.

14 *Throws:* Any exception thrown during the initialization of the contained value.

15 *Remarks:* This function shall not participate in overload resolution unless `is_constructible_v<T`$_I$`, Args...>` is `true`. If an exception is thrown during the initialization of the contained value, the `variant` might not hold a value.

```
template<size_t I, class U, class... Args>
  variant_alternative_t<I, variant<Types...>>& emplace(initializer_list<U> il, Args&&... args);
```

16 ~~*Requires:*~~ *Mandates:* `I < sizeof...(Types)`.

17 *Constraints:* `is_constructible_v<T`$_I$`, initializer_list<U>&, Args...>` is `true`.

18 *Effects:* Destroys the currently contained value if `valueless_by_exception()` is `false`. Then initializes the contained value as if direct-non-list-initializing a value of type `T`$_I$ with the arguments `il, std::forward<Args>(args)...`.

19 *Postconditions:* `index()` is `I`.

20 *Returns:* A reference to the new contained value.

21 *Throws:* Any exception thrown during the initialization of the contained value.

22 *Remarks:* This function shall not participate in overload resolution unless `is_constructible_v<T`$_I$`, initializer_list<U>&, Args...>` is `true`. If an exception is thrown during the initialization of the contained value, the `variant` might not hold a value.

20.7.3.5 Value status [variant.status]

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

1 *Effects:* Returns `false` if and only if the `variant` holds a value.

2 [*Note*: A `variant` might not hold a value if an exception is thrown during a type-changing assignment or emplacement. The latter means that even a `variant<float, int>` can become `valueless_by_exception()`, for instance by

```
struct S { operator int() { throw 42; }};
variant<float, int> v{12.f};
v.emplace<1>(S());
```

— *end note*]

```
constexpr size_t index() const noexcept;
```

3 *Effects:* If `valueless_by_exception()` is `true`, returns `variant_npos`. Otherwise, returns the zero-based index of the alternative of the contained value.

20.7.3.6 Swap [variant.swap]

```
void swap(variant& rhs) noexcept(see below);
```

1 ~~*Requires:*~~ *Mandates:* Lvalues of type `T`$_i$ shall be swappable (**??**) and `is_move_constructible_v<T`$_i$`>` shall be `true` for all i.

2 *Effects:*

(2.1) — If `valueless_by_exception() && rhs.valueless_by_exception()` no effect.

(2.2) — Otherwise, if `index() == rhs.index()`, calls `swap(get<`*i*`>(*this), get<`*i*`>(rhs))` where *i* is `index()`.

(2.3) — Otherwise, exchanges values of `rhs` and `*this`.

3 *Throws:* If `index() == rhs.index()`, any exception thrown by `swap(get<`*i*`>(*this), get<`*i*`>(rhs))` with *i* being `index()`. Otherwise, any exception thrown by the move constructor of $\texttt{T}_i$ or $\texttt{T}_j$ with *i* being `index()` and *j* being `rhs.index()`.

4 *Remarks:* If an exception is thrown during the call to function `swap(get<`*i*`>(*this), get<`*i*`>(rhs))`, the states of the contained values of `*this` and of `rhs` are determined by the exception safety guarantee of `swap` for lvalues of $\texttt{T}_i$ with *i* being `index()`. If an exception is thrown during the exchange of the values of `*this` and `rhs`, the states of the values of `*this` and of `rhs` are determined by the exception safety guarantee of `variant`'s move constructor. The expression inside `noexcept` is equivalent to the logical AND of `is_nothrow_move_constructible_v<`$\texttt{T}_i$`> && is_nothrow_swappable_v<`$\texttt{T}_i$`>` for all *i*.

20.7.4 `variant` helper classes [variant.helper]

```
template<class T> struct variant_size;
```

1 ~~*Remarks:*~~ *Preconditions:* All specializations of `variant_size` shall meet the *Cpp17UnaryTypeTrait* requirements (20.15.1) with a base characteristic of `integral_constant<size_t, N>` for some `N`.

```
template<class T> class variant_size<const T>;
template<class T> class variant_size<volatile T>;
template<class T> class variant_size<const volatile T>;
```

2 *Preconditions:* Let `VS` denote `variant_size<T>` of the cv-unqualified type `T`. Then each of the three templates shall meet the *Cpp17UnaryTypeTrait* requirements (20.15.1) with a base characteristic of `integral_constant<size_t, VS::value>`.

```
template<class... Types>
  struct variant_size<variant<Types...>> : integral_constant<size_t, sizeof...(Types)> { };

template<size_t I, class T> class variant_alternative<I, const T>;
template<size_t I, class T> class variant_alternative<I, volatile T>;
template<size_t I, class T> class variant_alternative<I, const volatile T>;
```

3 *Preconditions:* Let `VA` denote `variant_alternative<I, T>` of the cv-unqualified type `T`. Then each of the three templates shall meet the *Cpp17TransformationTrait* requirements (20.15.1) with a member typedef `type` that names the following type:

(3.1) — for the first specialization, `add_const_t<VA::type>`,

(3.2) — for the second specialization, `add_volatile_t<VA::type>`, and

(3.3) — for the third specialization, `add_cv_t<VA::type>`.

```
variant_alternative<I, variant<Types...>>::type
```

4 ~~*Requires:*~~ *Mandates:* `I < sizeof...(Types)`. ~~The program is ill-formed if `I` is out of bounds.~~

5 *Value:* The type $\texttt{T}_I$.

20.7.5 Value access [variant.get]

```
template<class T, class... Types>
  constexpr bool holds_alternative(const variant<Types...>& v) noexcept;
```

1 ~~*Requires:*~~ *Mandates:* The type `T` occurs exactly once in `Types`. ~~Otherwise, the program is ill-formed.~~

2 *Returns:* `true` if `index()` is equal to the zero-based index of `T` in `Types`.

```
template<size_t I, class... Types>
  constexpr variant_alternative_t<I, variant<Types...>>& get(variant<Types...>& v);
template<size_t I, class... Types>
  constexpr variant_alternative_t<I, variant<Types...>>&& get(variant<Types...>&& v);
template<size_t I, class... Types>
  constexpr const variant_alternative_t<I, variant<Types...>>& get(const variant<Types...>& v);
```

```
template<size_t I, class... Types>
  constexpr const variant_alternative_t<I, variant<Types...>>&& get(const variant<Types...>&& v);
```

3 ~~*Requires:*~~ *Mandates:* `I < sizeof...(Types)`. ~~Otherwise, the program is ill-formed.~~

4 *Effects:* If `v.index()` is `I`, returns a reference to the object stored in the `variant`. Otherwise, throws an exception of type `bad_variant_access`.

```
template<class T, class... Types> constexpr T& get(variant<Types...>& v);
template<class T, class... Types> constexpr T&& get(variant<Types...>&& v);
template<class T, class... Types> constexpr const T& get(const variant<Types...>& v);
template<class T, class... Types> constexpr const T&& get(const variant<Types...>&& v);
```

5 ~~*Requires:*~~ *Mandates:* The type `T` occurs exactly once in `Types`. ~~Otherwise, the program is ill-formed.~~

6 *Effects:* If `v` holds a value of type `T`, returns a reference to that value. Otherwise, throws an exception of type `bad_variant_access`.

```
template<size_t I, class... Types>
  constexpr add_pointer_t<variant_alternative_t<I, variant<Types...>>>
    get_if(variant<Types...>* v) noexcept;
template<size_t I, class... Types>
  constexpr add_pointer_t<const variant_alternative_t<I, variant<Types...>>>
    get_if(const variant<Types...>* v) noexcept;
```

7 ~~*Requires:*~~ *Mandates:* `I < sizeof...(Types)`. ~~Otherwise, the program is ill-formed.~~

8 *Returns:* A pointer to the value stored in the `variant`, if `v != nullptr` and `v->index() == I`. Otherwise, returns `nullptr`.

```
template<class T, class... Types>
  constexpr add_pointer_t<T>
    get_if(variant<Types...>* v) noexcept;
template<class T, class... Types>
  constexpr add_pointer_t<const T>
    get_if(const variant<Types...>* v) noexcept;
```

9 ~~*Requires:*~~ *Mandates:* The type `T` occurs exactly once in `Types`. ~~Otherwise, the program is ill-formed.~~

10 *Effects:* Equivalent to: `return get_if<`*i*`>(v);` with *i* being the zero-based index of `T` in `Types`.

20.7.6 Relational operators [variant.relops]

```
template<class... Types>
  constexpr bool operator==(const variant<Types...>& v, const variant<Types...>& w);
```

1 ~~*Requires:*~~ *Preconditions:* `get<`*i*`>(v) == get<`*i*`>(w)` is a valid expression returning a type that is convertible to `bool`, for all *i*.

2 *Returns:* If `v.index() != w.index()`, `false`; otherwise if `v.valueless_by_exception()`, `true`; otherwise `get<`*i*`>(v) == get<`*i*`>(w)` with *i* being `v.index()`.

```
template<class... Types>
  constexpr bool operator!=(const variant<Types...>& v, const variant<Types...>& w);
```

3 ~~*Requires:*~~ *Preconditions:* `get<`*i*`>(v) != get<`*i*`>(w)` is a valid expression returning a type that is convertible to `bool`, for all *i*.

4 *Returns:* If `v.index() != w.index()`, `true`; otherwise if `v.valueless_by_exception()`, `false`; otherwise `get<`*i*`>(v) != get<`*i*`>(w)` with *i* being `v.index()`.

```
template<class... Types>
  constexpr bool operator<(const variant<Types...>& v, const variant<Types...>& w);
```

5 ~~*Requires:*~~ *Preconditions:* `get<`*i*`>(v) < get<`*i*`>(w)` is a valid expression returning a type that is convertible to `bool`, for all *i*.

6 *Returns:* If `w.valueless_by_exception()`, `false`; otherwise if `v.valueless_by_exception()`, `true`; otherwise, if `v.index() < w.index()`, `true`; otherwise if `v.index() > w.index()`, `false`; otherwise `get<`*i*`>(v) < get<`*i*`>(w)` with *i* being `v.index()`.

```
template<class... Types>
  constexpr bool operator>(const variant<Types...>& v, const variant<Types...>& w);
```

7 ~~*Requires:*~~ *Preconditions:* `get<`*i*`>(v) > get<`*i*`>(w)` is a valid expression returning a type that is convertible to `bool`, for all *i*.

8 *Returns:* If `v.valueless_by_exception()`, `false`; otherwise if `w.valueless_by_exception()`, `true`; otherwise, if `v.index() > w.index()`, `true`; otherwise if `v.index() < w.index()`, `false`; otherwise `get<`*i*`>(v) > get<`*i*`>(w)` with *i* being `v.index()`.

```
template<class... Types>
  constexpr bool operator<=(const variant<Types...>& v, const variant<Types...>& w);
```

9 ~~*Requires:*~~ *Preconditions:* `get<`*i*`>(v) <= get<`*i*`>(w)` is a valid expression returning a type that is convertible to `bool`, for all *i*.

10 *Returns:* If `v.valueless_by_exception()`, `true`; otherwise if `w.valueless_by_exception()`, `false`; otherwise, if `v.index() < w.index()`, `true`; otherwise if `v.index() > w.index()`, `false`; otherwise `get<`*i*`>(v) <= get<`*i*`>(w)` with *i* being `v.index()`.

```
template<class... Types>
  constexpr bool operator>=(const variant<Types...>& v, const variant<Types...>& w);
```

11 ~~*Requires:*~~ *Preconditions:* `get<`*i*`>(v) >= get<`*i*`>(w)` is a valid expression returning a type that is convertible to `bool`, for all *i*.

12 *Returns:* If `w.valueless_by_exception()`, `true`; otherwise if `v.valueless_by_exception()`, `false`; otherwise, if `v.index() > w.index()`, `true`; otherwise if `v.index() < w.index()`, `false`; otherwise `get<`*i*`>(v) >= get<`*i*`>(w)` with *i* being `v.index()`.

```
template<class... Types> requires (three_way_comparable<Types> && ...)
  constexpr common_comparison_category_t<compare_three_way_result_t<Types>...>
    operator<=>(const variant<Types...>& v, const variant<Types...>& w);
```

13 *Effects:* Equivalent to:

```
if (v.valueless_by_exception() && w.valueless_by_exception())
  return strong_ordering::equal;
if (v.valueless_by_exception()) return strong_ordering::less;
if (w.valueless_by_exception()) return strong_ordering::greater;
if (auto c = v.index() <=> w.index(); c != 0) return c;
return get<i>(v) <=> get<i>(w);
```

with *i* being `v.index()`.

20.7.7 Visitation [variant.visit]

```
template<class Visitor, class... Variants>
  constexpr see below visit(Visitor&& vis, Variants&&... vars);
template<class R, class Visitor, class... Variants>
  constexpr R visit(Visitor&& vis, Variants&&... vars);
```

1 Let n be `sizeof...(Variants)`. Let `m` be a pack of n values of type `size_t`. Such a pack is called valid if $0 \le \mathtt{m}_i < \mathtt{variant\_size\_v<remove\_reference\_t<Variants}_i\mathtt{>>}$ for all $0 \le i < n$. For each valid pack `m`, let e(`m`) denote the expression:

```
INVOKE(std::forward<Visitor>(vis), get<m>(std::forward<Variants>(vars))...) // see 20.14.3
```

for the first form and

```
INVOKE<R>(std::forward<Visitor>(vis), get<m>(std::forward<Variants>(vars))...) // see 20.14.3
```

for the second form.

2 ~~*Requires:*~~ *Mandates:* For each valid pack `m`, e(`m`) shall be a valid expression. All such expressions shall be of the same type and value category~~; otherwise, the program is ill-formed~~.

3 *Returns:* e(`m`), where `m` is the pack for which $\mathtt{m}_i$ is $\mathtt{vars}_i\mathtt{.index()}$ for all $0 \le i < n$. The return type is `decltype(`e`(m))` for the first form.

4 *Throws:* `bad_variant_access` if any `variant` in `vars` is `valueless_by_exception()`.

5 *Complexity:* For $n \leq 1$, the invocation of the callable object is implemented in constant time, i.e., for $n = 1$, it does not depend on the number of alternative types of `Variants`$_0$. For $n > 1$, the invocation of the callable object has no complexity requirements.

20.7.8 Class `monostate` [variant.monostate]

```
struct monostate{};
```

1 The class `monostate` can serve as a first alternative type for a `variant` to make the `variant` type default constructible.

20.7.9 `monostate` relational operators [variant.monostate.relops]

```
constexpr bool operator==(monostate, monostate) noexcept { return true; }
constexpr strong_ordering operator<=>(monostate, monostate) noexcept
{ return strong_ordering::equal; }
```

1 [*Note*: `monostate` objects have only a single state; they thus always compare equal. — *end note*]

20.7.10 Specialized algorithms [variant.specalg]

```
template<class... Types>
  void swap(variant<Types...>& v, variant<Types...>& w) noexcept(see below);
```

1 *Constraints:* `is_move_constructible_v<`T_i`> && is_swappable_v<`T_i`>` is `true` for all i.

2 *Effects:* Equivalent to `v.swap(w)`.

3 *Remarks:* ~~This function shall not participate in overload resolution unless `is_move_constructible_v<`T_i`> && is_swa`~~ ~~`is_true` for all i.~~ The expression inside `noexcept` is equivalent to `noexcept(v.swap(w))`.

20.7.11 Class `bad_variant_access` [variant.bad.access]

```
  class bad_variant_access : public exception {
  public:
    // see ?? for the specification of the special member functions
    const char* what() const noexcept override;
  };
```

1 Objects of type `bad_variant_access` are thrown to report invalid accesses to the value of a `variant` object.

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

2 *Returns:* An implementation-defined NTBS.

20.7.12 Hash support [variant.hash]

```
template<class... Types> struct hash<variant<Types...>>;
```

1 The specialization `hash<variant<Types...>>` is enabled (20.14.18) if and only if every specialization in `hash<remove_const_t<Types>>...` is enabled. The member functions are not guaranteed to be `noexcept`.

```
template<> struct hash<monostate>;
```

2 The specialization is enabled (20.14.18).

20.8 Storage for any type [any]

1 This subclause describes components that C++ programs may use to perform operations on objects of a discriminated type.

2 [*Note*: The discriminated type may contain values of different types but does not attempt conversion between them, i.e., `5` is held strictly as an `int` and is not implicitly convertible either to `"5"` or to `5.0`. This indifference to interpretation but awareness of type effectively allows safe, generic containers of single values, with no scope for surprises from ambiguous conversions. — *end note*]

20.8.1 Header <any> synopsis [any.synop]

```
namespace std {
  // 20.8.2, class bad_any_cast
  class bad_any_cast;
```

```
  // 20.8.3, class any
  class any;

  // 20.8.4, non-member functions
  void swap(any& x, any& y) noexcept;

  template<class T, class... Args>
    any make_any(Args&& ...args);
  template<class T, class U, class... Args>
    any make_any(initializer_list<U> il, Args&& ...args);

  template<class T>
    T any_cast(const any& operand);
  template<class T>
    T any_cast(any& operand);
  template<class T>
    T any_cast(any&& operand);

  template<class T>
    const T* any_cast(const any* operand) noexcept;
  template<class T>
    T* any_cast(any* operand) noexcept;
}
```

20.8.2 Class `bad_any_cast` [any.bad.any.cast]

```
class bad_any_cast : public bad_cast {
public:
  // see ?? for the specification of the special member functions
  const char* what() const noexcept override;
};
```

1 Objects of type `bad_any_cast` are thrown by a failed `any_cast` (20.8.4).

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

2 *Returns:* An implementation-defined NTBS.

20.8.3 Class any [any.class]

```
namespace std {
  class any {
  public:
    // 20.8.3.1, construction and destruction
    constexpr any() noexcept;

    any(const any& other);
    any(any&& other) noexcept;

    template<class T>
      any(T&& value);

    template<class T, class... Args>
      explicit any(in_place_type_t<T>, Args&&...);
    template<class T, class U, class... Args>
      explicit any(in_place_type_t<T>, initializer_list<U>, Args&&...);

    ~any();

    // 20.8.3.2, assignments
    any& operator=(const any& rhs);
    any& operator=(any&& rhs) noexcept;

    template<class T>
      any& operator=(T&& rhs);
```

```
    // 20.8.3.3, modifiers
    template<class T, class... Args>
      decay_t<T>& emplace(Args&& ...);
    template<class T, class U, class... Args>
      decay_t<T>& emplace(initializer_list<U>, Args&&...);
    void reset() noexcept;
    void swap(any& rhs) noexcept;

    // 20.8.3.4, observers
    bool has_value() const noexcept;
    const type_info& type() const noexcept;
  };
}
```

1 An object of class `any` stores an instance of any type that meets the constructor requirements or it has no value, and this is referred to as the *state* of the class `any` object. The stored instance is called the *contained value*. Two states are equivalent if either they both have no value, or they both have a value and the contained values are equivalent.

2 The non-member `any_cast` functions provide type-safe access to the contained value.

3 Implementations should avoid the use of dynamically allocated memory for a small contained value. However, any such small-object optimization shall only be applied to types `T` for which `is_nothrow_move_constructible_v<T>` is `true`. [*Example*: A contained value of type `int` could be stored in an internal buffer, not in separately-allocated memory. — *end example*]

20.8.3.1 Construction and destruction [any.cons]

```
constexpr any() noexcept;
```

1 *Postconditions:* `has_value()` is `false`.

```
any(const any& other);
```

2 *Effects:* If `other.has_value()` is `false`, constructs an object that has no value. Otherwise, equivalent to `any(in_place_type<T>, any_cast<const T&>(other))` where `T` is the type of the contained value.

3 *Throws:* Any exceptions arising from calling the selected constructor for the contained value.

```
any(any&& other) noexcept;
```

4 *Effects:* If `other.has_value()` is `false`, constructs an object that has no value. Otherwise, constructs an object of type `any` that contains either the contained value of `other`, or contains an object of the same type constructed from the contained value of `other` considering that contained value as an rvalue.

```
template<class T>
  any(T&& value);
```

5 Let `VT` be `decay_t<T>`.

6 *Requires:* `VT` shall meet the *Cpp17CopyConstructible* requirements.

7 *Constraints:* `VT` is not the same type as `any`, `VT` is not a specialization of `in_place_type_t`, and `is_copy_constructible_v<VT>` is `true`.

8 *Preconditions:* `VT` meets the *Cpp17CopyConstructible* requirements.

9 *Effects:* Constructs an object of type `any` that contains an object of type `VT` direct-initialized with `std::forward<T>(value)`.

10 *Remarks:* This constructor shall not participate in overload resolution unless `VT` is not the same type as `any`, `VT` is not a specialization of `in_place_type_t`, and `is_copy_constructible_v<VT>` is `true`.

11 *Throws:* Any exception thrown by the selected constructor of `VT`.

```
template<class T, class... Args>
  explicit any(in_place_type_t<T>, Args&&... args);
```

12 Let `VT` be `decay_t<T>`.

13 *Requires:* `VT` shall meet the *Cpp17CopyConstructible* requirements.

Constraints: `is_copy_constructible_v<VT>` is `true` and `is_constructible_v<VT, Args...>` is `true`.

Preconditions: `VT` meets the *Cpp17CopyConstructible* requirements.

Effects: Initializes the contained value as if direct-non-list-initializing an object of type `VT` with the arguments `std::forward<Args>(args)...`.

Postconditions: `*this` contains a value of type `VT`.

Throws: Any exception thrown by the selected constructor of `VT`.

Remarks: This constructor shall not participate in overload resolution unless `is_copy_constructible_v<VT>` is `true` and `is_constructible_v<VT, Args...>` is `true`.

```
template<class T, class U, class... Args>
  explicit any(in_place_type_t<T>, initializer_list<U> il, Args&&... args);
```

Let `VT` be `decay_t<T>`.

Constraints: `is_copy_constructible_v<VT>` is `true` and `is_constructible_v<VT, initializer_list<U>&, Args...>` is `true`.

~~*Requires:*~~ *Preconditions:* `VT` ~~shall meet~~meets the *Cpp17CopyConstructible* requirements.

Effects: Initializes the contained value as if direct-non-list-initializing an object of type `VT` with the arguments `il, std::forward<Args>(args)...`.

Postconditions: `*this` contains a value.

Throws: Any exception thrown by the selected constructor of `VT`.

Remarks: This constructor shall not participate in overload resolution unless `is_copy_constructible_v<VT>` is `true` and `is_constructible_v<VT, initializer_list<U>&, Args...>` is `true`.

```
~any();
```

Effects: As if by `reset()`.

20.8.3.2 Assignment [any.assign]

```
any& operator=(const any& rhs);
```

Effects: As if by `any(rhs).swap(*this)`. No effects if an exception is thrown.

Returns: `*this`.

Throws: Any exceptions arising from the copy constructor for the contained value.

```
any& operator=(any&& rhs) noexcept;
```

Effects: As if by `any(std::move(rhs)).swap(*this)`.

Returns: `*this`.

Postconditions: The state of `*this` is equivalent to the original state of `rhs`.

```
template<class T>
  any& operator=(T&& rhs);
```

Let `VT` be `decay_t<T>`.

Constraints: `VT` is not the same type as `any` and `is_copy_constructible_v<VT>` is `true`.

~~*Requires:*~~ *Preconditions:* `VT` ~~shall meet~~meets the *Cpp17CopyConstructible* requirements.

Effects: Constructs an object `tmp` of type `any` that contains an object of type `VT` direct-initialized with `std::forward<T>(rhs)`, and `tmp.swap(*this)`. No effects if an exception is thrown.

Returns: `*this`.

Remarks: This operator shall not participate in overload resolution unless `VT` is not the same type as `any` and `is_copy_constructible_v<VT>` is `true`.

Throws: Any exception thrown by the selected constructor of `VT`.

20.8.3.3 Modifiers [any.modifiers]

```
template<class T, class... Args>
  decay_t<T>& emplace(Args&&... args);
```

1 Let VT be decay_t<T>.

2 *Constraints:* is_copy_constructible_v<VT> is true and is_constructible_v<VT, Args...> is true.

~~*Requires:*~~ *Preconditions:* VT ~~shall meet~~meets the *Cpp17CopyConstructible* requirements.

3 *Effects:* Calls reset(). Then initializes the contained value as if direct-non-list-initializing an object of type VT with the arguments std::forward<Args>(args)....

4 *Postconditions:* *this contains a value.

5 *Returns:* A reference to the new contained value.

6 *Throws:* Any exception thrown by the selected constructor of VT.

7 *Remarks:* If an exception is thrown during the call to VT's constructor, *this does not contain a value, and any previously contained value has been destroyed. This function shall not participate in overload resolution unless is_copy_constructible_v<VT> is true and is_constructible_v<VT, Args...> is true.

```
template<class T, class U, class... Args>
  decay_t<T>& emplace(initializer_list<U> il, Args&&... args);
```

8 Let VT be decay_t<T>.

9 *Constraints:* is_copy_constructible_v<VT> is true and is_constructible_v<VT, initializer_list<U>&, Args...> is true.

10 ~~*Requires:*~~ *Preconditions:* VT ~~shall meet~~meets the *Cpp17CopyConstructible* requirements.

11 *Effects:* Calls reset(). Then initializes the contained value as if direct-non-list-initializing an object of type VT with the arguments il, std::forward<Args>(args)....

12 *Postconditions:* *this contains a value.

13 *Returns:* A reference to the new contained value.

14 *Throws:* Any exception thrown by the selected constructor of VT.

15 *Remarks:* If an exception is thrown during the call to VT's constructor, *this does not contain a value, and any previously contained value has been destroyed. The function shall not participate in overload resolution unless is_copy_constructible_v<VT> is true and is_constructible_v<VT, initializer_list<U>&, Args...> is true.

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

16 *Effects:* If has_value() is true, destroys the contained value.

17 *Postconditions:* has_value() is false.

```
void swap(any& rhs) noexcept;
```

18 *Effects:* Exchanges the states of *this and rhs.

20.8.3.4 Observers [any.observers]

```
bool has_value() const noexcept;
```

1 *Returns:* true if *this contains an object, otherwise false.

```
const type_info& type() const noexcept;
```

2 *Returns:* typeid(T) if *this has a contained value of type T, otherwise typeid(void).

3 [*Note*: Useful for querying against types known either at compile time or only at runtime. — *end note*]

20.8.4 Non-member functions [any.nonmembers]

```
void swap(any& x, any& y) noexcept;
```

1 *Effects:* As if by x.swap(y).

```
template<class T, class... Args>
  any make_any(Args&& ...args);
```

2 *Effects:* Equivalent to: `return any(in_place_type<T>, std::forward<Args>(args)...);`

```
template<class T, class U, class... Args>
  any make_any(initializer_list<U> il, Args&& ...args);
```

3 *Effects:* Equivalent to: `return any(in_place_type<T>, il, std::forward<Args>(args)...);`

```
template<class T>
  T any_cast(const any& operand);
template<class T>
  T any_cast(any& operand);
template<class T>
  T any_cast(any&& operand);
```

4 Let `U` be the type `remove_cvref_t<T>`.

5 ~~*Requires:*~~ *Mandates:* For the first overload, `is_constructible_v<T, const U&>` is `true`. For the second overload, `is_constructible_v<T, U&>` is `true`. For the third overload, `is_constructible_v<T, U>` is `true`. ~~Otherwise the program is ill-formed.~~

6 *Returns:* For the first and second overload, `static_cast<T>(*any_cast<U>(&operand))`. For the third overload, `static_cast<T>(std::move(*any_cast<U>(&operand)))`.

7 *Throws:* `bad_any_cast` if `operand.type() != typeid(remove_reference_t<T>)`.

8 [*Example*:

```
any x(5);                                   // x holds int
assert(any_cast<int>(x) == 5);              // cast to value
any_cast<int&>(x) = 10;                     // cast to reference
assert(any_cast<int>(x) == 10);

x = "Meow";                                 // x holds const char*
assert(strcmp(any_cast<const char*>(x), "Meow") == 0);
any_cast<const char*&>(x) = "Harry";
assert(strcmp(any_cast<const char*>(x), "Harry") == 0);

x = string("Meow");                         // x holds string
string s, s2("Jane");
s = move(any_cast<string&>(x));             // move from any
assert(s == "Meow");
any_cast<string&>(x) = move(s2);            // move to any
assert(any_cast<const string&>(x) == "Jane");

string cat("Meow");
const any y(cat);                           // const y holds string
assert(any_cast<const string&>(y) == cat);

any_cast<string&>(y);                       // error; cannot
                                            // any_cast away const
```

— *end example*]

```
template<class T>
  const T* any_cast(const any* operand) noexcept;
template<class T>
  T* any_cast(any* operand) noexcept;
```

9 *Returns:* If `operand != nullptr && operand->type() == typeid(T)`, a pointer to the object contained by `operand`; otherwise, `nullptr`.

10 [*Example*:

```
bool is_string(const any& operand) {
  return any_cast<string>(&operand) != nullptr;
}
```

— *end example*]

20.9 Bitsets [bitset]

20.9.1 Header <bitset> synopsis [bitset.syn]

1 The header <bitset> defines a class template and several related functions for representing and manipulating fixed-size sequences of bits.

```
#include <string>
#include <iosfwd>    // for istream (??), ostream (??), see ??

namespace std {
  template<size_t N> class bitset;

  // 20.9.4, bitset operators
  template<size_t N>
    bitset<N> operator&(const bitset<N>&, const bitset<N>&) noexcept;
  template<size_t N>
    bitset<N> operator|(const bitset<N>&, const bitset<N>&) noexcept;
  template<size_t N>
    bitset<N> operator^(const bitset<N>&, const bitset<N>&) noexcept;
  template<class charT, class traits, size_t N>
    basic_istream<charT, traits>&
      operator>>(basic_istream<charT, traits>& is, bitset<N>& x);
  template<class charT, class traits, size_t N>
    basic_ostream<charT, traits>&
      operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);
}
```

20.9.2 Class template bitset [template.bitset]

```
namespace std {
  template<size_t N> class bitset {
  public:
    // bit reference
    class reference {
      friend class bitset;
      reference() noexcept;

    public:
      reference(const reference&) = default;
      ~reference();
      reference& operator=(bool x) noexcept;             // for b[i] = x;
      reference& operator=(const reference&) noexcept;   // for b[i] = b[j];
      bool operator~() const noexcept;                   // flips the bit
      operator bool() const noexcept;                    // for x = b[i];
      reference& flip() noexcept;                        // for b[i].flip();
    };

    // 20.9.2.1, constructors
    constexpr bitset() noexcept;
    constexpr bitset(unsigned long long val) noexcept;
    template<class charT, class traits, class Allocator>
      explicit bitset(
        const basic_string<charT, traits, Allocator>& str,
        typename basic_string<charT, traits, Allocator>::size_type pos = 0,
        typename basic_string<charT, traits, Allocator>::size_type n
          = basic_string<charT, traits, Allocator>::npos,
        charT zero = charT('0'),
        charT one = charT('1'));
    template<class charT>
      explicit bitset(
        const charT* str,
        typename basic_string<charT>::size_type n = basic_string<charT>::npos,
        charT zero = charT('0'),
        charT one = charT('1'));
```

```
    // 20.9.2.2, bitset operations
    bitset<N>& operator&=(const bitset<N>& rhs) noexcept;
    bitset<N>& operator|=(const bitset<N>& rhs) noexcept;
    bitset<N>& operator^=(const bitset<N>& rhs) noexcept;
    bitset<N>& operator<<=(size_t pos) noexcept;
    bitset<N>& operator>>=(size_t pos) noexcept;
    bitset<N>& set() noexcept;
    bitset<N>& set(size_t pos, bool val = true);
    bitset<N>& reset() noexcept;
    bitset<N>& reset(size_t pos);
    bitset<N>  operator~() const noexcept;
    bitset<N>& flip() noexcept;
    bitset<N>& flip(size_t pos);

    // element access
    constexpr bool operator[](size_t pos) const;        // for b[i];
    reference operator[](size_t pos);                   // for b[i];

    unsigned long to_ulong() const;
    unsigned long long to_ullong() const;
    template<class charT = char,
             class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
      basic_string<charT, traits, Allocator>
        to_string(charT zero = charT('0'), charT one = charT('1')) const;

    size_t count() const noexcept;
    constexpr size_t size() const noexcept;
    bool operator==(const bitset<N>& rhs) const noexcept;
    bool test(size_t pos) const;
    bool all() const noexcept;
    bool any() const noexcept;
    bool none() const noexcept;
    bitset<N> operator<<(size_t pos) const noexcept;
    bitset<N> operator>>(size_t pos) const noexcept;
  };

  // 20.9.3, hash support
  template<class T> struct hash;
  template<size_t N> struct hash<bitset<N>>;
}
```

1 The class template `bitset<N>` describes an object that can store a sequence consisting of a fixed number of bits, `N`.

2 Each bit represents either the value zero (reset) or one (set). To *toggle* a bit is to change the value zero to one, or the value one to zero. Each bit has a non-negative position `pos`. When converting between an object of class `bitset<N>` and a value of some integral type, bit position `pos` corresponds to the *bit value* `1 << pos`. The integral value corresponding to two or more bits is the sum of their bit values.

3 The functions described in this subclause can report three kinds of errors, each associated with a distinct exception:

(3.1) — an *invalid-argument* error is associated with exceptions of type `invalid_argument` (??);

(3.2) — an *out-of-range* error is associated with exceptions of type `out_of_range` (??);

(3.3) — an *overflow* error is associated with exceptions of type `overflow_error` (??).

20.9.2.1 Constructors [bitset.cons]

```
constexpr bitset() noexcept;
```

1 *Effects:* Initializes all bits in `*this` to zero.

```
constexpr bitset(unsigned long long val) noexcept;
```

2 *Effects:* Initializes the first `M` bit positions to the corresponding bit values in `val`. `M` is the smaller of `N` and the number of bits in the value representation (??) of `unsigned long long`. If `M < N`, the remaining bit positions are initialized to zero.

```
template<class charT, class traits, class Allocator>
  explicit bitset(
    const basic_string<charT, traits, Allocator>& str,
    typename basic_string<charT, traits, Allocator>::size_type pos = 0,
    typename basic_string<charT, traits, Allocator>::size_type n
      = basic_string<charT, traits, Allocator>::npos,
    charT zero = charT('0'),
    charT one = charT('1'));
```

3 *Throws:* `out_of_range` if `pos > str.size()` or `invalid_argument` if an invalid character is found (see below).

4 *Effects:* Determines the effective length `rlen` of the initializing string as the smaller of `n` and `str.size() - pos`.

The function then throws `invalid_argument` if any of the `rlen` characters in `str` beginning at position `pos` is other than `zero` or `one`. The function uses `traits::eq()` to compare the character values.

~~Otherwise, t~~The function constructs an object of class `bitset<N>`, initializing the first `M` bit positions to values determined from the corresponding characters in the string `str`. `M` is the smaller of `N` and `rlen`.

5 An element of the constructed object has value zero if the corresponding character in `str`, beginning at position `pos`, is `zero`. Otherwise, the element has the value one. Character position `pos + M - 1` corresponds to bit position zero. Subsequent decreasing character positions correspond to increasing bit positions.

6 If `M < N`, remaining bit positions are initialized to zero.

7 *Throws:* `out_of_range` if `pos > str.size()` or `invalid_argument` if any of the `rlen` characters in `str` beginning at position `pos` is other than `zero` or `one`. The function uses `traits::eq()` to compare the character values.

```
template<class charT>
  explicit bitset(
    const charT* str,
    typename basic_string<charT>::size_type n = basic_string<charT>::npos,
    charT zero = charT('0'),
    charT one = charT('1'));
```

8 *Effects:* As if by:

```
  bitset(n == basic_string<charT>::npos
            ? basic_string<charT>(str)
            : basic_string<charT>(str, n),
         0, n, zero, one)
```

20.9.2.2 Members [bitset.members]

```
bitset<N>& operator&=(const bitset<N>& rhs) noexcept;
```

1 *Effects:* Clears each bit in `*this` for which the corresponding bit in `rhs` is clear, and leaves all other bits unchanged.

2 *Returns:* `*this`.

```
bitset<N>& operator|=(const bitset<N>& rhs) noexcept;
```

3 *Effects:* Sets each bit in `*this` for which the corresponding bit in `rhs` is set, and leaves all other bits unchanged.

4 *Returns:* `*this`.

`bitset<N>& operator^=(const bitset<N>& rhs) noexcept;`

5 *Effects:* Toggles each bit in `*this` for which the corresponding bit in `rhs` is set, and leaves all other bits unchanged.

6 *Returns:* `*this`.

`bitset<N>& operator<<=(size_t pos) noexcept;`

7 *Effects:* Replaces each bit at position `I` in `*this` with a value determined as follows:

(7.1) — If `I < pos`, the new value is zero;

(7.2) — If `I >= pos`, the new value is the previous value of the bit at position `I - pos`.

8 *Returns:* `*this`.

`bitset<N>& operator>>=(size_t pos) noexcept;`

9 *Effects:* Replaces each bit at position `I` in `*this` with a value determined as follows:

(9.1) — If `pos >= N - I`, the new value is zero;

(9.2) — If `pos < N - I`, the new value is the previous value of the bit at position `I + pos`.

10 *Returns:* `*this`.

`bitset<N>& set() noexcept;`

11 *Effects:* Sets all bits in `*this`.

12 *Returns:* `*this`.

`bitset<N>& set(size_t pos, bool val = true);`

13 *Throws:* `out_of_range` if `pos` does not correspond to a valid bit position.

14 *Effects:* Stores a new value in the bit at position `pos` in `*this`. If `val` is `true`, the stored value is one, otherwise it is zero.

15 *Returns:* `*this`.

16 *Throws:* `out_of_range` if `pos` does not correspond to a valid bit position.

`bitset<N>& reset() noexcept;`

17 *Effects:* Resets all bits in `*this`.

18 *Returns:* `*this`.

`bitset<N>& reset(size_t pos);`

19 *Throws:* `out_of_range` if `pos` does not correspond to a valid bit position.

20 *Effects:* Resets the bit at position `pos` in `*this`.

21 *Returns:* `*this`.

22 *Throws:* `out_of_range` if `pos` does not correspond to a valid bit position.

`bitset<N> operator~() const noexcept;`

23 *Effects:* Constructs an object `x` of class `bitset<N>` and initializes it with `*this`.

24 *Returns:* `x.flip()`.

`bitset<N>& flip() noexcept;`

25 *Effects:* Toggles all bits in `*this`.

26 *Returns:* `*this`.

`bitset<N>& flip(size_t pos);`

27 *Throws:* `out_of_range` if `pos` does not correspond to a valid bit position.

28 *Effects:* Toggles the bit at position `pos` in `*this`.

29 *Returns:* `*this`.

30 *Throws:* `out_of_range` if `pos` does not correspond to a valid bit position.

```
unsigned long to_ulong() const;
```

31 *Throws:* `overflow_error` if the integral value `x` corresponding to the bits in `*this` cannot be represented as type `unsigned long`.

32 *Returns:* `x`.

33 *Throws:* `overflow_error` if the integral value `x` corresponding to the bits in `*this` cannot be represented as type `unsigned long`.

```
unsigned long long to_ullong() const;
```

34 *Throws:* `overflow_error` if the integral value `x` corresponding to the bits in `*this` cannot be represented as type `unsigned long long`.

35 *Returns:* `x`.

36 *Throws:* `overflow_error` if the integral value `x` corresponding to the bits in `*this` cannot be represented as type `unsigned long long`.

```
template<class charT = char,
         class traits = char_traits<charT>,
         class Allocator = allocator<charT>>
  basic_string<charT, traits, Allocator>
    to_string(charT zero = charT('0'), charT one = charT('1')) const;
```

37 *Effects:* Constructs a string object of the appropriate type and initializes it to a string of length `N` characters. Each character is determined by the value of its corresponding bit position in `*this`. Character position `N - 1` corresponds to bit position zero. Subsequent decreasing character positions correspond to increasing bit positions. Bit value zero becomes the character `zero`, bit value one becomes the character `one`.

38 *Returns:* The created object.

```
size_t count() const noexcept;
```

39 *Returns:* A count of the number of bits set in `*this`.

```
constexpr size_t size() const noexcept;
```

40 *Returns:* `N`.

```
bool operator==(const bitset<N>& rhs) const noexcept;
```

41 *Returns:* `true` if the value of each bit in `*this` equals the value of the corresponding bit in `rhs`.

```
bool test(size_t pos) const;
```

42 *Throws:* `out_of_range` if `pos` does not correspond to a valid bit position.

43 *Returns:* `true` if the bit at position `pos` in `*this` has the value one.

44 *Throws:* `out_of_range` if `pos` does not correspond to a valid bit position.

```
bool all() const noexcept;
```

45 *Returns:* `count() == size()`.

```
bool any() const noexcept;
```

46 *Returns:* `count() != 0`.

```
bool none() const noexcept;
```

47 *Returns:* `count() == 0`.

```
bitset<N> operator<<(size_t pos) const noexcept;
```

48 *Returns:* `bitset<N>(*this) <<= pos`.

```
bitset<N> operator>>(size_t pos) const noexcept;
```

49 *Returns:* `bitset<N>(*this) >>= pos`.

```
constexpr bool operator[](size_t pos) const;
```

50 ~~*Requires:*~~ *Preconditions:* `pos` shall be valid.

51 *Returns:* `true` if the bit at position `pos` in `*this` has the value one, otherwise `false`.

52 *Throws:* Nothing.

```
bitset<N>::reference operator[](size_t pos);
```

53 ~~*Requires:*~~ *Preconditions:* `pos` shall be valid.

54 *Returns:* An object of type `bitset<N>::reference` such that `(*this)[pos] == this->test(pos)`, and such that `(*this)[pos] = val` is equivalent to `this->set(pos, val)`.

55 *Throws:* Nothing.

56 *Remarks:* For the purpose of determining the presence of a data race (**??**), any access or update through the resulting reference potentially accesses or modifies, respectively, the entire underlying bitset.

20.9.3 `bitset` hash support [bitset.hash]

```
template<size_t N> struct hash<bitset<N>>;
```

1 The specialization is enabled (20.14.18).

20.9.4 `bitset` operators [bitset.operators]

```
bitset<N> operator&(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;
```

1 *Returns:* `bitset<N>(lhs) &= rhs`.

```
bitset<N> operator|(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;
```

2 *Returns:* `bitset<N>(lhs) |= rhs`.

```
bitset<N> operator^(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;
```

3 *Returns:* `bitset<N>(lhs) ^= rhs`.

```
template<class charT, class traits, size_t N>
  basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is, bitset<N>& x);
```

4 A formatted input function (**??**).

5 *Effects:* Extracts up to `N` characters from `is`. Stores these characters in a temporary object `str` of type `basic_string<charT, traits>`, then evaluates the expression `x = bitset<N>(str)`. Characters are extracted and stored until any of the following occurs:

(5.1) — `N` characters have been extracted and stored;

(5.2) — end-of-file occurs on the input sequence;

(5.3) — the next input character is neither `is.widen('0')` nor `is.widen('1')` (in which case the input character is not extracted).

6 If `N > 0` and no characters are stored in `str`, calls `is.setstate(ios_base::failbit)` (which may throw `ios_base::failure` (**??**)).

7 *Returns:* `is`.

```
template<class charT, class traits, size_t N>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);
```

8 *Returns:*

```
os << x.template to_string<charT, traits, allocator<charT>>(
  use_facet<ctype<charT>>(os.getloc()).widen('0'),
  use_facet<ctype<charT>>(os.getloc()).widen('1'))
```

(see **??**).

20.10 Memory [memory]

20.10.1 In general [memory.general]

1 Subclause 20.10 describes the contents of the header `<memory>` (20.10.2) and some of the contents of the header `<cstdlib>` (??).

20.10.2 Header `<memory>` synopsis [memory.syn]

1 The header `<memory>` defines several types and function templates that describe properties of pointers and pointer-like types, manage memory for containers and other template types, destroy objects, and construct multiple objects in uninitialized memory buffers (20.10.3–20.10.11). The header also defines the templates `unique_ptr`, `shared_ptr`, `weak_ptr`, and various function templates that operate on objects of these types (20.11).

```
namespace std {
  // 20.10.3, pointer traits
  template<class Ptr> struct pointer_traits;
  template<class T> struct pointer_traits<T*>;

  // 20.10.4, pointer conversion
  template<class T>
    constexpr T* to_address(T* p) noexcept;
  template<class Ptr>
    auto to_address(const Ptr& p) noexcept;

  // 20.10.5, pointer safety
  enum class pointer_safety { relaxed, preferred, strict };
  void declare_reachable(void* p);
  template<class T>
    T* undeclare_reachable(T* p);
  void declare_no_pointers(char* p, size_t n);
  void undeclare_no_pointers(char* p, size_t n);
  pointer_safety get_pointer_safety() noexcept;

  // 20.10.6, pointer alignment
  void* align(size_t alignment, size_t size, void*& ptr, size_t& space);
  template<size_t N, class T>
    [[nodiscard]] constexpr T* assume_aligned(T* ptr);

  // 20.10.7, allocator argument tag
  struct allocator_arg_t { explicit allocator_arg_t() = default; };
  inline constexpr allocator_arg_t allocator_arg{};

  // 20.10.8, uses_allocator
  template<class T, class Alloc> struct uses_allocator;

  // 20.10.8.1, uses_allocator
  template<class T, class Alloc>
    inline constexpr bool uses_allocator_v = uses_allocator<T, Alloc>::value;

  // 20.10.8.2, uses-allocator construction
  template<class T, class Alloc, class... Args>
    constexpr auto uses_allocator_construction_args(const Alloc& alloc,
                                                    Args&&... args) noexcept -> see below;
  template<class T, class Alloc, class Tuple1, class Tuple2>
    constexpr auto uses_allocator_construction_args(const Alloc& alloc, piecewise_construct_t,
                                                    Tuple1&& x, Tuple2&& y)
                                                    noexcept ->  see below;
  template<class T, class Alloc>
    constexpr auto uses_allocator_construction_args(const Alloc& alloc) noexcept -> see below;
  template<class T, class Alloc, class U, class V>
    constexpr auto uses_allocator_construction_args(const Alloc& alloc,
                                                    U&& u, V&& v) noexcept -> see below;
```

```
template<class T, class Alloc, class U, class V>
  constexpr auto uses_allocator_construction_args(const Alloc& alloc,
                                                  const pair<U,V>& pr) noexcept -> see below;
template<class T, class Alloc, class U, class V>
  constexpr auto uses_allocator_construction_args(const Alloc& alloc,
                                                  pair<U,V>&& pr) noexcept -> see below;
template<class T, class Alloc, class... Args>
  constexpr T make_obj_using_allocator(const Alloc& alloc, Args&&... args);
template<class T, class Alloc, class... Args>
  constexpr T* uninitialized_construct_using_allocator(T* p, const Alloc& alloc,
                                                       Args&&... args);

// 20.10.9, allocator traits
template<class Alloc> struct allocator_traits;

// 20.10.10, the default allocator
template<class T> class allocator;
template<class T, class U>
  constexpr bool operator==(const allocator<T>&, const allocator<U>&) noexcept;

// 20.10.11, specialized algorithms
// 20.10.11.1, special memory concepts
template<class I>
  concept no-throw-input-iterator = see below;     // exposition only
template<class I>
  concept no-throw-forward-iterator = see below;   // exposition only
template<class S, class I>
  concept no-throw-sentinel = see below;           // exposition only
template<class R>
  concept no-throw-input-range = see below;        // exposition only
template<class R>
  concept no-throw-forward-range = see below;      // exposition only

template<class T>
  constexpr T* addressof(T& r) noexcept;
template<class T>
  const T* addressof(const T&&) = delete;
template<class ForwardIterator>
  void uninitialized_default_construct(ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  void uninitialized_default_construct(ExecutionPolicy&& exec,        // see ??
                                       ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Size>
  ForwardIterator uninitialized_default_construct_n(ForwardIterator first, Size n);
template<class ExecutionPolicy, class ForwardIterator, class Size>
  ForwardIterator uninitialized_default_construct_n(ExecutionPolicy&& exec,   // see ??
                                                    ForwardIterator first, Size n);

namespace ranges {
  template<no-throw-forward-iterator I, no-throw-sentinel<I> S>
    requires default_constructible<iter_value_t<I>>
      I uninitialized_default_construct(I first, S last);
  template<no-throw-forward-range R>
    requires default_constructible<range_value_t<R>>
      safe_iterator_t<R> uninitialized_default_construct(R&& r);

  template<no-throw-forward-iterator I>
    requires default_constructible<iter_value_t<I>>
      I uninitialized_default_construct_n(I first, iter_difference_t<I> n);
}

template<class ForwardIterator>
  void uninitialized_value_construct(ForwardIterator first, ForwardIterator last);
```

```
template<class ExecutionPolicy, class ForwardIterator>
  void uninitialized_value_construct(ExecutionPolicy&& exec,  // see ??
                                     ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Size>
  ForwardIterator uninitialized_value_construct_n(ForwardIterator first, Size n);
template<class ExecutionPolicy, class ForwardIterator, class Size>
  ForwardIterator uninitialized_value_construct_n(ExecutionPolicy&& exec, // see ??
                                                  ForwardIterator first, Size n);

namespace ranges {
  template<no-throw-forward-iterator I, no-throw-sentinel<I> S>
    requires default_constructible<iter_value_t<I>>
      I uninitialized_value_construct(I first, S last);
  template<no-throw-forward-range R>
    requires default_constructible<range_value_t<R>>
      safe_iterator_t<R> uninitialized_value_construct(R&& r);

  template<no-throw-forward-iterator I>
    requires default_constructible<iter_value_t<I>>
      I uninitialized_value_construct_n(I first, iter_difference_t<I> n);
}

template<class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                     ForwardIterator result);
template<class ExecutionPolicy, class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_copy(ExecutionPolicy&& exec,  // see ??
                                     InputIterator first, InputIterator last,
                                     ForwardIterator result);
template<class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(InputIterator first, Size n,
                                       ForwardIterator result);
template<class ExecutionPolicy, class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(ExecutionPolicy&& exec,        // see ??
                                       InputIterator first, Size n,
                                       ForwardIterator result);

namespace ranges {
  template<class I, class O>
  using uninitialized_copy_result = copy_result<I, O>;
  template<input_iterator I, sentinel_for<I> S1,
           no-throw-forward-iterator O, no-throw-sentinel<O> S2>
    requires constructible_from<iter_value_t<O>, iter_reference_t<I>>
      uninitialized_copy_result<I, O>
        uninitialized_copy(I ifirst, S1 ilast, O ofirst, S2 olast);
  template<input_range IR, no-throw-forward-range OR>
    requires constructible_from<range_value_t<OR>, range_reference_t<IR>>
      uninitialized_copy_result<safe_iterator_t<IR>, safe_iterator_t<OR>>
        uninitialized_copy(IR&& in_range, OR&& out_range);

  template<class I, class O>
    using uninitialized_copy_n_result = uninitialized_copy_result<I, O>;
  template<input_iterator I, no-throw-forward-iterator O, no-throw-sentinel<O> S>
    requires constructible_from<iter_value_t<O>, iter_reference_t<I>>
      uninitialized_copy_n_result<I, O>
        uninitialized_copy_n(I ifirst, iter_difference_t<I> n, O ofirst, S olast);
}

template<class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_move(InputIterator first, InputIterator last,
                                     ForwardIterator result);
template<class ExecutionPolicy, class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_move(ExecutionPolicy&& exec,  // see ??
                                     InputIterator first, InputIterator last,
```

```
                                    ForwardIterator result);
template<class InputIterator, class Size, class ForwardIterator>
  pair<InputIterator, ForwardIterator> uninitialized_move_n(InputIterator first, Size n,
                                                            ForwardIterator result);
template<class ExecutionPolicy, class InputIterator, class Size, class ForwardIterator>
  pair<InputIterator, ForwardIterator>
    uninitialized_move_n(ExecutionPolicy&& exec,              // see ??
                         InputIterator first, Size n, ForwardIterator result);

namespace ranges {
  template<class I, class O>
    using uninitialized_move_result = uninitialized_copy_result<I, O>;
  template<input_iterator I, sentinel_for<I> S1,
           no-throw-forward-iterator O, no-throw-sentinel<O> S2>
    requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
      uninitialized_move_result<I, O>
        uninitialized_move(I ifirst, S1 ilast, O ofirst, S2 olast);
  template<input_range IR, no-throw-forward-range OR>
    requires constructible_from<range_value_t<OR>, range_rvalue_reference_t<IR>>
      uninitialized_move_result<safe_iterator_t<IR>, safe_iterator_t<OR>>
        uninitialized_move(IR&& in_range, OR&& out_range);

  template<class I, class O>
    using uninitialized_move_n_result = uninitialized_copy_result<I, O>;
  template<input_iterator I,
           no-throw-forward-iterator O, no-throw-sentinel<O> S>
    requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
      uninitialized_move_n_result<I, O>
        uninitialized_move_n(I ifirst, iter_difference_t<I> n, O ofirst, S olast);
}

template<class ForwardIterator, class T>
  void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x);
template<class ExecutionPolicy, class ForwardIterator, class T>
  void uninitialized_fill(ExecutionPolicy&& exec,             // see ??
                          ForwardIterator first, ForwardIterator last, const T& x);
template<class ForwardIterator, class Size, class T>
  ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
template<class ExecutionPolicy, class ForwardIterator, class Size, class T>
  ForwardIterator uninitialized_fill_n(ExecutionPolicy&& exec,        // see ??
                                       ForwardIterator first, Size n, const T& x);

namespace ranges {
  template<no-throw-forward-iterator I, no-throw-sentinel<I> S, class T>
    requires constructible_from<iter_value_t<I>, const T&>
      I uninitialized_fill(I first, S last, const T& x);
  template<no-throw-forward-range R, class T>
    requires constructible_from<range_value_t<R>, const T&>
      safe_iterator_t<R> uninitialized_fill(R&& r, const T& x);

  template<no-throw-forward-iterator I, class T>
    requires constructible_from<iter_value_t<I>, const T&>
      I uninitialized_fill_n(I first, iter_difference_t<I> n, const T& x);
}

// 20.10.11.8, construct_at
template<class T, class... Args>
  constexpr T* construct_at(T* location, Args&&... args);

namespace ranges {
  template<class T, class... Args>
    constexpr T* construct_at(T* location, Args&&... args);
}
```

```
// 20.10.11.9, destroy
template<class T>
  constexpr void destroy_at(T* location);
template<class ForwardIterator>
  constexpr void destroy(ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  constexpr void destroy(ExecutionPolicy&& exec,              // see ??
                         ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Size>
  constexpr ForwardIterator destroy_n(ForwardIterator first, Size n);
template<class ExecutionPolicy, class ForwardIterator, class Size>
  constexpr ForwardIterator destroy_n(ExecutionPolicy&& exec, // see ??
                                      ForwardIterator first, Size n);

namespace ranges {
  template<destructible T>
    constexpr void destroy_at(T* location) noexcept;

  template<no-throw-input-iterator I, no-throw-sentinel<I> S>
    requires destructible<iter_value_t<I>>
      constexpr I destroy(I first, S last) noexcept;
  template<no-throw-input-range R>
    requires destructible<range_value_t<R>>
      constexpr safe_iterator_t<R> destroy(R&& r) noexcept;

  template<no-throw-input-iterator I>
    requires destructible<iter_value_t<I>>
      constexpr I destroy_n(I first, iter_difference_t<I> n) noexcept;
}

// 20.11.1, class template unique_ptr
template<class T> struct default_delete;
template<class T> struct default_delete<T[]>;
template<class T, class D = default_delete<T>> class unique_ptr;
template<class T, class D> class unique_ptr<T[], D>;

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

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

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

template<class T, class D>
  bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;
template<class T, class D>
  bool operator<(const unique_ptr<T, D>& x, nullptr_t);
template<class T, class D>
  bool operator<(nullptr_t, const unique_ptr<T, D>& y);
template<class T, class D>
  bool operator>(const unique_ptr<T, D>& x, nullptr_t);
template<class T, class D>
  bool operator>(nullptr_t, const unique_ptr<T, D>& y);
template<class T, class D>
  bool operator<=(const unique_ptr<T, D>& x, nullptr_t);
template<class T, class D>
  bool operator<=(nullptr_t, const unique_ptr<T, D>& y);
template<class T, class D>
  bool operator>=(const unique_ptr<T, D>& x, nullptr_t);
template<class T, class D>
  bool operator>=(nullptr_t, const unique_ptr<T, D>& y);
template<class T, class D>
  requires three_way_comparable_with<typename unique_ptr<T, D>::pointer, nullptr_t>
  compare_three_way_result_t<typename unique_ptr<T, D>::pointer, nullptr_t>
    operator<=>(const unique_ptr<T, D>& x, nullptr_t);

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);

// 20.11.2, class bad_weak_ptr
class bad_weak_ptr;

// 20.11.3, class template shared_ptr
template<class T> class shared_ptr;

// 20.11.3.6, shared_ptr creation
template<class T, class... Args>
  shared_ptr<T> make_shared(Args&&... args);                          // T is not array
template<class T, class A, class... Args>
  shared_ptr<T> allocate_shared(const A& a, Args&&... args);          // T is not array

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

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

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

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

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

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

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

// 20.11.3.7, shared_ptr comparisons
template<class T, class U>
  bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  strong_ordering operator<=>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;

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

// 20.11.3.8, shared_ptr specialized algorithms
template<class T>
  void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

// 20.11.3.9, shared_ptr casts
template<class T, class U>
  shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> static_pointer_cast(shared_ptr<U>&& r) noexcept;
template<class T, class U>
  shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> dynamic_pointer_cast(shared_ptr<U>&& r) noexcept;
template<class T, class U>
  shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  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;

// 20.11.3.10, shared_ptr get_deleter
template<class D, class T>
  D* get_deleter(const shared_ptr<T>& p) noexcept;

// 20.11.3.11, 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);

// 20.11.4, class template weak_ptr
template<class T> class weak_ptr;

// 20.11.4.6, weak_ptr specialized algorithms
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;

// 20.11.5, class template owner_less
template<class T = void> struct owner_less;
```

```
  // 20.11.6, class template enable_shared_from_this
  template<class T> class enable_shared_from_this;

  // 20.11.7, hash support
  template<class T> struct hash;
  template<class T, class D> struct hash<unique_ptr<T, D>>;
  template<class T> struct hash<shared_ptr<T>>;

  // ??, atomic smart pointers
  template<class T> struct atomic;
  template<class T> struct atomic<shared_ptr<T>>;
  template<class T> struct atomic<weak_ptr<T>>;
}
```

20.10.3 Pointer traits [pointer.traits]

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 {
    using pointer         = Ptr;
    using element_type    = see below;
    using difference_type = see below;

    template<class U> using rebind = see below;

    static pointer pointer_to(see below r);
  };

  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;
  };
}
```

20.10.3.1 Member types [pointer.traits.types]

`using element_type = see below;`

1 *Type:* `Ptr::element_type` if the *qualified-id* `Ptr::element_type` is valid and denotes a type (??); otherwise, `T` if `Ptr` is a class template instantiation of the form `SomePointer<T, Args>`, where `Args` is zero or more type arguments; otherwise, the specialization is ill-formed.

`using difference_type = see below;`

2 *Type:* `Ptr::difference_type` if the *qualified-id* `Ptr::difference_type` is valid and denotes a type (??); otherwise, `ptrdiff_t`.

`template<class U> using rebind = see below;`

3 *Alias template:* `Ptr::rebind<U>` if the *qualified-id* `Ptr::rebind<U>` is valid and denotes a type (??); otherwise, `SomePointer<U, Args>` if `Ptr` is a class template instantiation of the form `SomePointer<T, Args>`, where `Args` is zero or more type arguments; otherwise, the instantiation of `rebind` is ill-formed.

20.10.3.2 Member functions [pointer.traits.functions]

```
static pointer pointer_traits::pointer_to(see below r);
static constexpr pointer pointer_traits<T*>::pointer_to(see below r) noexcept;
```

1 *Mandates:* For the first member function, `Ptr::pointer_to(r)` is well-formed.

2 *Preconditions:* For the first member function, `Ptr::pointer_to(r)` returns a pointer to `r` through which indirection is valid.

3 *Returns:* The first member function returns `Ptr::pointer_to(r)`. The second member function returns `addressof(r)`.

4 *Remarks:* If `element_type` is *cv* `void`, the type of `r` is unspecified; otherwise, it is `element_type&`.

5 *Returns:* The first member function returns a pointer to `r` obtained by calling `Ptr::pointer_to(r)` through which indirection is valid; an instantiation of this function is ill-formed if `Ptr` does not have a matching `pointer_to` static member function. The second member function returns `addressof(r)`.

20.10.3.3 Optional members [pointer.traits.optmem]

1 Specializations of `pointer_traits` may define the member declared in this subclause to customize the behavior of the standard library.

```
static element_type* to_address(pointer p) noexcept;
```

2 *Returns:* A pointer of type `element_type*` that references the same location as the argument `p`.

3 [*Note*: This function should be the inverse of `pointer_to`. If defined, it customizes the behavior of the non-member function `to_address` (20.10.4). — *end note*]

20.10.4 Pointer conversion [pointer.conversion]

```
template<class T> constexpr T* to_address(T* p) noexcept;
```

1 ~~*Requires:*~~ *Mandates:* `T` is not a function type. ~~Otherwise the program is ill-formed.~~

2 *Returns:* `p`.

```
template<class Ptr> auto to_address(const Ptr& p) noexcept;
```

3 *Returns:* `pointer_traits<Ptr>::to_address(p)` if that expression is well-formed (see 20.10.3.3), otherwise `to_address(p.operator->())`.

20.10.5 Pointer safety [util.dynamic.safety]

1 A complete object is *declared reachable* while the number of calls to `declare_reachable` with an argument referencing the object exceeds the number of calls to `undeclare_reachable` with an argument referencing the object.

```
void declare_reachable(void* p);
```

2 ~~*Requires:*~~ *Preconditions:* `p` shall be a safely-derived pointer (**??**) or a null pointer value.

3 *Effects:* If `p` is not null, the complete object referenced by `p` is subsequently declared reachable (**??**).

4 *Throws:* May throw `bad_alloc` if the system cannot allocate additional memory that may be required to track objects declared reachable.

```
template<class T> T* undeclare_reachable(T* p);
```

5 ~~*Requires:*~~ *Preconditions:* If `p` is not null, the complete object referenced by `p` shall have been previously declared reachable, and shall be live (**??**) from the time of the call until the last `undeclare_reachable(p)` call on the object.

6 *Returns:* A safely derived copy of `p` which shall compare equal to `p`.

7 *Throws:* Nothing.

8 [*Note*: It is expected that calls to `declare_reachable(p)` will consume a small amount of memory in addition to that occupied by the referenced object until the matching call to `undeclare_reachable(p)` is encountered. Long running programs should arrange that calls are matched. — *end note*]

```
void declare_no_pointers(char* p, size_t n);
```

9 ~~*Requires:*~~ *Preconditions:* No bytes in the specified range are currently registered with `declare_no_pointers()`. If the specified range is in an allocated object, then it shall be entirely within a single allocated object. The object shall be live until the corresponding `undeclare_no_pointers()` call. [*Note*: In a garbage-collecting implementation, the fact that a region in an object is registered with `declare_no_pointers()` should not prevent the object from being collected. — *end note*]

10 *Effects:* The `n` bytes starting at `p` no longer contain traceable pointer locations, independent of their type. Hence indirection through a pointer located there is undefined if the object it points to was

created by global `operator new` and not previously declared reachable. [*Note*: This may be used to inform a garbage collector or leak detector that this region of memory need not be traced. — *end note*]

11 *Throws:* Nothing.

12 [*Note*: Under some conditions implementations may need to allocate memory. However, the request can be ignored if memory allocation fails. — *end note*]

```
void undeclare_no_pointers(char* p, size_t n);
```

13 ~~*Requires:*~~ *Preconditions:* The same range shall previously have been passed to `declare_no_pointers()`.

14 *Effects:* Unregisters a range registered with `declare_no_pointers()` for destruction. It shall be called before the lifetime of the object ends.

15 *Throws:* Nothing.

```
pointer_safety get_pointer_safety() noexcept;
```

16 *Returns:* `pointer_safety::strict` if the implementation has strict pointer safety (**??**). It is implementation-defined whether `get_pointer_safety` returns `pointer_safety::relaxed` or `pointer_safety::preferred` if the implementation has relaxed pointer safety.[220]

20.10.6 Pointer alignment [ptr.align]

```
void* align(size_t alignment, size_t size, void*& ptr, size_t& space);
```

1 *Preconditions:*

(1.1) — `alignment` is a power of two

(1.2) — `ptr` represents the address of contiguous storage of at least `space` bytes

2 *Effects:* If it is possible to fit `size` bytes of storage aligned by `alignment` into the buffer pointed to by `ptr` with length `space`, the function updates `ptr` to represent the first possible address of such storage and decreases `space` by the number of bytes used for alignment. Otherwise, the function does nothing.

3 *Requires:*

(3.1) — `alignment` shall be a power of two

(3.2) — `ptr` shall represent the address of contiguous storage of at least `space` bytes

4 *Returns:* A null pointer if the requested aligned buffer would not fit into the available space, otherwise the adjusted value of `ptr`.

5 [*Note*: The function updates its `ptr` and `space` arguments so that it can be called repeatedly with possibly different `alignment` and `size` arguments for the same buffer. — *end note*]

```
template<size_t N, class T>
  [[nodiscard]] constexpr T* assume_aligned(T* ptr);
```

6 *Mandates:* `N` is a power of two.

7 *Preconditions:* `ptr` points to an object `X` of a type similar (**??**) to `T`, where `X` has alignment `N` (**??**).

8 *Returns:* `ptr`.

9 *Throws:* Nothing.

10 [*Note*: The alignment assumption on an object `X` expressed by a call to `assume_aligned` may result in generation of more efficient code. It is up to the program to ensure that the assumption actually holds. The call does not cause the compiler to verify or enforce this. An implementation might only make the assumption for those operations on `X` that access `X` through the pointer returned by `assume_aligned`. — *end note*]

20.10.7 Allocator argument tag [allocator.tag]

```
namespace std {
  struct allocator_arg_t { explicit allocator_arg_t() = default; };
```

220) `pointer_safety::preferred` might be returned to indicate that a leak detector is running so that the program can avoid spurious leak reports.

```
  inline constexpr allocator_arg_t allocator_arg{};
}
```

1 The `allocator_arg_t` struct is an empty class type used as a unique type to disambiguate constructor and function overloading. Specifically, several types (see `tuple` 20.5) have constructors with `allocator_arg_t` as the first argument, immediately followed by an argument of a type that meets the *Cpp17Allocator* requirements (Table **??**).

20.10.8 `uses_allocator` [allocator.uses]

20.10.8.1 `uses_allocator` trait [allocator.uses.trait]

```
template<class T, class Alloc> struct uses_allocator;
```

1 *Remarks:* Automatically detects whether `T` has a nested `allocator_type` that is convertible from `Alloc`. Meets the *Cpp17BinaryTypeTrait* requirements (20.15.1). The implementation shall provide a definition that is derived from `true_type` if the *qualified-id* `T::allocator_type` is valid and denotes a type (**??**) and `is_convertible_v<Alloc, T::allocator_type> != false`, otherwise it shall be derived from `false_type`. A program may specialize this template to derive from `true_type` for a program-defined type `T` that does not have a nested `allocator_type` but nonetheless can be constructed with an allocator where either:

(1.1) — the first argument of a constructor has type `allocator_arg_t` and the second argument has type `Alloc` or

(1.2) — the last argument of a constructor has type `Alloc`.

20.10.8.2 Uses-allocator construction [allocator.uses.construction]

1 *Uses-allocator construction* with allocator `alloc` and constructor arguments `args...` refers to the construction of an object of type `T` such that `alloc` is passed to the constructor of `T` if `T` uses an allocator type compatible with `alloc`. When applied to the construction of an object of type `T`, it is equivalent to initializing it with the value of the expression `make_obj_using_allocator<T>(alloc, args...)`, described below.

2 The following utility functions support three conventions for passing `alloc` to a constructor:

(2.1) — If `T` does not use an allocator compatible with `alloc`, then `alloc` is ignored.

(2.2) — Otherwise, if `T` has a constructor invocable as `T(allocator_arg, alloc, args...)` (leading-allocator convention), then uses-allocator construction chooses this constructor form.

(2.3) — Otherwise, if `T` has a constructor invocable as `T(args..., alloc)` (trailing-allocator convention), then uses-allocator construction chooses this constructor form.

3 The `uses_allocator_construction_args` function template takes an allocator and argument list and produces (as a tuple) a new argument list matching one of the above conventions. Additionally, overloads are provided that treat specializations of `pair` such that uses-allocator construction is applied individually to the `first` and `second` data members. The `make_obj_using_allocator` and `uninitialized_construct_using_allocator` function templates apply the modified constructor arguments to construct an object of type `T` as a return value or in-place, respectively. [*Note*: For `uses_allocator_construction_args` and `make_obj_using_allocator`, type `T` is not deduced and must therefore be specified explicitly by the caller. — *end note*]

```
template<class T, class Alloc, class... Args>
  constexpr auto uses_allocator_construction_args(const Alloc& alloc,
                                                  Args&&... args) noexcept -> see below;
```

4 *Constraints:* `T` is not a specialization of `pair`.

5 *Returns:* A `tuple` value determined as follows:

(5.1) — If `uses_allocator_v<T, Alloc>` is `false` and `is_constructible_v<T, Args...>` is `true`, return `forward_as_tuple(std::forward<Args>(args)...)`.

(5.2) — Otherwise, if `uses_allocator_v<T, Alloc>` is `true` and `is_constructible_v<T, allocator_arg_t, const Alloc&, Args...>` is `true`, return

```
tuple<allocator_arg_t, const Alloc&, Args&&...>(
  allocator_arg, alloc, std::forward<Args>(args)...)
```

(5.3) — Otherwise, if `uses_allocator_v<T, Alloc>` is `true` and `is_constructible_v<T, Args..., const Alloc&>` is `true`, return `forward_as_tuple(std::forward<Args>(args)..., alloc)`.

(5.4) — Otherwise, the program is ill-formed.

[*Note*: This definition prevents a silent failure to pass the allocator to a constructor of a type for which `uses_allocator_v<T, Alloc>` is `true`. — *end note*]

```
template<class T, class Alloc, class Tuple1, class Tuple2>
  constexpr auto uses_allocator_construction_args(const Alloc& alloc, piecewise_construct_t,
                                                  Tuple1&& x, Tuple2&& y)
                                                  noexcept -> see below;
```

6 *Constraints:* `T` is a specialization of `pair`.

7 *Effects:* For `T` specified as `pair<T1, T2>`, equivalent to:

```
return make_tuple(
  piecewise_construct,
  apply([&alloc](auto&&... args1) {
          return uses_allocator_construction_args<T1>(
            alloc, std::forward<decltype(args1)>(args1)...);
        }, std::forward<Tuple1>(x)),
  apply([&alloc](auto&&... args2) {
          return uses_allocator_construction_args<T2>(
            alloc, std::forward<decltype(args2)>(args2)...);
        }, std::forward<Tuple2>(y)));
```

```
template<class T, class Alloc>
  constexpr auto uses_allocator_construction_args(const Alloc& alloc) noexcept -> see below;
```

8 *Constraints:* `T` is a specialization of `pair`.

9 *Effects:* Equivalent to:

```
return uses_allocator_construction_args<T>(alloc, piecewise_construct,
                                           tuple<>{}, tuple<>{});
```

```
template<class T, class Alloc, class U, class V>
  constexpr auto uses_allocator_construction_args(const Alloc& alloc,
                                                  U&& u, V&& v) noexcept -> see below;
```

10 *Constraints:* `T` is a specialization of `pair`.

11 *Effects:* Equivalent to:

```
return uses_allocator_construction_args<T>(alloc, piecewise_construct,
                                           forward_as_tuple(std::forward<U>(u)),
                                           forward_as_tuple(std::forward<V>(v)));
```

```
template<class T, class Alloc, class U, class V>
  constexpr auto uses_allocator_construction_args(const Alloc& alloc,
                                                  const pair<U,V>& pr) noexcept -> see below;
```

12 *Constraints:* `T` is a specialization of `pair`.

13 *Effects:* Equivalent to:

```
return uses_allocator_construction_args<T>(alloc, piecewise_construct,
                                           forward_as_tuple(pr.first),
                                           forward_as_tuple(pr.second));
```

```
template<class T, class Alloc, class U, class V>
  constexpr auto uses_allocator_construction_args(const Alloc& alloc,
                                                  pair<U,V>&& pr) noexcept -> see below;
```

14 *Constraints:* `T` is a specialization of `pair`.

15 *Effects:* Equivalent to:

```
return uses_allocator_construction_args<T>(alloc, piecewise_construct,
                                           forward_as_tuple(std::move(pr).first),
                                           forward_as_tuple(std::move(pr).second));
```

```
template<class T, class Alloc, class... Args>
  constexpr T make_obj_using_allocator(const Alloc& alloc, Args&&... args);
```

16 *Effects:* Equivalent to:

```
return make_from_tuple<T>(uses_allocator_construction_args<T>(
                            alloc, std::forward<Args>(args)...));
```

```
template<class T, class Alloc, class... Args>
  constexpr T* uninitialized_construct_using_allocator(T* p, const Alloc& alloc, Args&&... args);
```

17 *Effects:* Equivalent to:

```
return ::new(static_cast<void*>(p))
  T(make_obj_using_allocator<T>(alloc, std::forward<Args>(args)...));
```

20.10.9 Allocator traits [allocator.traits]

1 The class template `allocator_traits` supplies a uniform interface to all allocator types. An allocator cannot be a non-class type, however, even if `allocator_traits` supplies the entire required interface. [*Note*: Thus, it is always possible to create a derived class from an allocator. — *end note*]

```
namespace std {
  template<class Alloc> struct allocator_traits {
    using allocator_type     = Alloc;

    using value_type         = typename Alloc::value_type;

    using pointer            = see below;
    using const_pointer      = see below;
    using void_pointer       = see below;
    using const_void_pointer = see below;

    using difference_type    = see below;
    using size_type          = see below;

    using propagate_on_container_copy_assignment = see below;
    using propagate_on_container_move_assignment = see below;
    using propagate_on_container_swap            = see below;
    using is_always_equal                        = see below;

    template<class T> using rebind_alloc = see below;
    template<class T> using rebind_traits = allocator_traits<rebind_alloc<T>>;

    [[nodiscard]] static constexpr pointer allocate(Alloc& a, size_type n);
    [[nodiscard]] static constexpr pointer allocate(Alloc& a, size_type n,
                                                    const_void_pointer hint);

    static constexpr void deallocate(Alloc& a, pointer p, size_type n);

    template<class T, class... Args>
      static constexpr void construct(Alloc& a, T* p, Args&&... args);

    template<class T>
      static constexpr void destroy(Alloc& a, T* p);

    static constexpr size_type max_size(const Alloc& a) noexcept;

    static constexpr Alloc select_on_container_copy_construction(const Alloc& rhs);
  };
}
```

20.10.9.1 Member types [allocator.traits.types]

```
using pointer = see below;
```

1 *Type:* `Alloc::pointer` if the *qualified-id* `Alloc::pointer` is valid and denotes a type (**??**); otherwise, `value_type*`.

```
using const_pointer = see below;
```

2 *Type:* `Alloc::const_pointer` if the *qualified-id* `Alloc::const_pointer` is valid and denotes a type (**??**); otherwise, `pointer_traits<pointer>::rebind<const value_type>`.

```
using void_pointer = see below;
```

3 *Type:* `Alloc::void_pointer` if the *qualified-id* `Alloc::void_pointer` is valid and denotes a type (**??**); otherwise, `pointer_traits<pointer>::rebind<void>`.

```
using const_void_pointer = see below;
```

4 *Type:* `Alloc::const_void_pointer` if the *qualified-id* `Alloc::const_void_pointer` is valid and denotes a type (**??**); otherwise, `pointer_traits<pointer>::rebind<const void>`.

```
using difference_type = see below;
```

5 *Type:* `Alloc::difference_type` if the *qualified-id* `Alloc::difference_type` is valid and denotes a type (**??**); otherwise, `pointer_traits<pointer>::difference_type`.

```
using size_type = see below;
```

6 *Type:* `Alloc::size_type` if the *qualified-id* `Alloc::size_type` is valid and denotes a type (**??**); otherwise, `make_unsigned_t<difference_type>`.

```
using propagate_on_container_copy_assignment = see below;
```

7 *Type:* `Alloc::propagate_on_container_copy_assignment` if the *qualified-id* `Alloc::propagate_on_container_copy_assignment` is valid and denotes a type (**??**); otherwise `false_type`.

```
using propagate_on_container_move_assignment = see below;
```

8 *Type:* `Alloc::propagate_on_container_move_assignment` if the *qualified-id* `Alloc::propagate_on_container_move_assignment` is valid and denotes a type (**??**); otherwise `false_type`.

```
using propagate_on_container_swap = see below;
```

9 *Type:* `Alloc::propagate_on_container_swap` if the *qualified-id* `Alloc::propagate_on_container_swap` is valid and denotes a type (**??**); otherwise `false_type`.

```
using is_always_equal = see below;
```

10 *Type:* `Alloc::is_always_equal` if the *qualified-id* `Alloc::is_always_equal` is valid and denotes a type (**??**); otherwise `is_empty<Alloc>::type`.

```
template<class T> using rebind_alloc = see below;
```

11 *Alias template:* `Alloc::rebind<T>::other` if the *qualified-id* `Alloc::rebind<T>::other` is valid and denotes a type (**??**); otherwise, `Alloc<T, Args>` if `Alloc` is a class template instantiation of the form `Alloc<U, Args>`, where `Args` is zero or more type arguments; otherwise, the instantiation of `rebind_alloc` is ill-formed.

20.10.9.2 Static member functions [allocator.traits.members]

```
[[nodiscard]] static constexpr pointer allocate(Alloc& a, size_type n);
```

1 *Returns:* `a.allocate(n)`.

```
[[nodiscard]] static constexpr pointer allocate(Alloc& a, size_type n, const_void_pointer hint);
```

2 *Returns:* `a.allocate(n, hint)` if that expression is well-formed; otherwise, `a.allocate(n)`.

```
static constexpr void deallocate(Alloc& a, pointer p, size_type n);
```

3 *Effects:* Calls `a.deallocate(p, n)`.

4 *Throws:* Nothing.

```
template<class T, class... Args>
  static constexpr void construct(Alloc& a, T* p, Args&&... args);
```

5 *Effects:* Calls `a.construct(p, std::forward<Args>(args)...)` if that call is well-formed; otherwise, invokes `construct_at(p, std::forward<Args>(args)...)`.

```
template<class T>
  static constexpr void destroy(Alloc& a, T* p);
```

6 *Effects:* Calls `a.destroy(p)` if that call is well-formed; otherwise, invokes `destroy_at(p)`.

```
static constexpr size_type max_size(const Alloc& a) noexcept;
```

7 *Returns:* `a.max_size()` if that expression is well-formed; otherwise, `numeric_limits<size_type>::max()/sizeof(value_type)`.

```
static constexpr Alloc select_on_container_copy_construction(const Alloc& rhs);
```

8 *Returns:* `rhs.select_on_container_copy_construction()` if that expression is well-formed; otherwise, `rhs`.

20.10.10 The default allocator [default.allocator]

1 All specializations of the default allocator meet the allocator completeness requirements (**??**).

```
namespace std {
  template<class T> class allocator {
   public:
    using value_type                             = T;
    using size_type                              = size_t;
    using difference_type                        = ptrdiff_t;
    using propagate_on_container_move_assignment = true_type;
    using is_always_equal                        = true_type;

    constexpr allocator() noexcept;
    constexpr allocator(const allocator&) noexcept;
    template<class U> constexpr allocator(const allocator<U>&) noexcept;
    constexpr ~allocator();
    constexpr allocator& operator=(const allocator&) = default;

    [[nodiscard]] constexpr T* allocate(size_t n);
    constexpr void deallocate(T* p, size_t n);
  };
}
```

20.10.10.1 Members [allocator.members]

1 Except for the destructor, member functions of the default allocator shall not introduce data races (**??**) as a result of concurrent calls to those member functions from different threads. Calls to these functions that allocate or deallocate a particular unit of storage shall occur in a single total order, and each such deallocation call shall happen before the next allocation (if any) in this order.

```
[[nodiscard]] constexpr T* allocate(size_t n);
```

2 *Returns:* A pointer to the initial element of an array of storage of size `n * sizeof(T)`, aligned appropriately for objects of type `T`.

3 *Remarks:* the storage is obtained by calling `::operator new` (**??**), but it is unspecified when or how often this function is called.

4 *Throws:* `bad_array_new_length` if `SIZE_MAX / sizeof(T) < n`, or `bad_alloc` if the storage cannot be obtained.

```
constexpr void deallocate(T* p, size_t n);
```

5 ~~*Requires:*~~ *Preconditions:* `p` shall be a pointer value obtained from `allocate()`. `n` shall equal the value passed as the first argument to the invocation of allocate which returned `p`.

6 *Effects:* Deallocates the storage referenced by `p` .

7 *Remarks:* Uses `::operator delete` (**??**), but it is unspecified when this function is called.

20.10.10.2 Operators [allocator.globals]

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

1 *Returns:* `true`.

20.10.11 Specialized algorithms [specialized.algorithms]

1 Throughout this subclause, the names of template parameters are used to express type requirements for those algorithms defined directly in namespace `std`.

(1.1) — If an algorithm's template parameter is named `InputIterator`, the template argument shall meet the *Cpp17InputIterator* requirements (**??**).

(1.2) — If an algorithm's template parameter is named `ForwardIterator`, the template argument shall meet the *Cpp17ForwardIterator* requirements (**??**), and is required to have the property that no exceptions are thrown from increment, assignment, comparison, or indirection through valid iterators.

2 Unless otherwise specified, if an exception is thrown in the following algorithms, objects constructed by a placement *new-expression* (**??**) are destroyed in an unspecified order before allowing the exception to propagate.

3 In the description of the algorithms, operator `-` is used for some of the iterator categories for which it need not be defined. In these cases, the value of the expression `b - a` is the number of increments of `a` needed to make `bool(a == b)` be `true`.

4 The following additional requirements apply for those algorithms defined in namespace `std::ranges`:

(4.1) — The entities defined in the `std::ranges` namespace in this subclause are not found by argument-dependent name lookup (**??**). When found by unqualified (**??**) name lookup for the *postfix-expression* in a function call (**??**), they inhibit argument-dependent name lookup.

(4.2) — Overloads of algorithms that take `range` arguments (**??**) behave as if they are implemented by calling `ranges::begin` and `ranges::end` on the `range`(s) and dispatching to the overload that takes separate iterator and sentinel arguments.

(4.3) — The number and order of deducible template parameters for algorithm declarations is unspecified, except where explicitly stated otherwise. [*Note*: Consequently, these algorithms may not be called with explicitly-specified template argument lists. — *end note*]

5 [*Note*: When invoked on ranges of potentially overlapping subobjects (**??**), the algorithms specified in this subclause 20.10.11 result in undefined behavior. — *end note*]

6 Some algorithms defined in this clause make use of the exposition-only function *`voidify`*:

```
template<class T>
  constexpr void* voidify(T& obj) noexcept {
    return const_cast<void*>(static_cast<const volatile void*>(addressof(obj)));
  }
```

20.10.11.1 Special memory concepts [special.mem.concepts]

1 Some algorithms in this subclause are constrained with the following exposition-only concepts:

```
template<class I>
concept no-throw-input-iterator = // exposition only
  input_iterator<I> &&
  is_lvalue_reference_v<iter_reference_t<I>> &&
  same_as<remove_cvref_t<iter_reference_t<I>>, iter_value_t<I>>;
```

2 A type `I` models *`no-throw-input-iterator`* only if no exceptions are thrown from increment, copy construction, move construction, copy assignment, move assignment, or indirection through valid iterators.

3 [*Note*: This concept allows some `input_iterator` (**??**) operations to throw exceptions. — *end note*]

```
template<class S, class I>
concept no-throw-sentinel = sentinel_for<S, I>; // exposition only
```

4 Types `S` and `I` model *`no-throw-sentinel`* only if no exceptions are thrown from copy construction, move construction, copy assignment, move assignment, or comparisons between valid values of type `I` and `S`.

5 [*Note*: This concept allows some `sentinel_for` (**??**) operations to throw exceptions. — *end note*]

```
template<class R>
concept no-throw-input-range = // exposition only
  range<R> &&
  no-throw-input-iterator<iterator_t<R>> &&
  no-throw-sentinel<sentinel_t<R>, iterator_t<R>>;
```

6 A type `R` models *`no-throw-input-range`* only if no exceptions are thrown from calls to `ranges::begin` and `ranges::end` on an object of type `R`.

```
template<class I>
concept no-throw-forward-iterator = // exposition only
  no-throw-input-iterator<I> &&
  forward_iterator<I> &&
  no-throw-sentinel<I, I>;
```

7 [*Note*: This concept allows some `forward_iterator` (**??**) operations to throw exceptions. — *end note*]

```
template<class R>
concept no-throw-forward-range = // exposition only
  no-throw-input-range<R> &&
  no-throw-forward-iterator<iterator_t<R>>;
```

20.10.11.2 `addressof` [specialized.addressof]

```
template<class T> constexpr T* addressof(T& r) noexcept;
```

1 *Returns:* The actual address of the object or function referenced by `r`, even in the presence of an overloaded `operator&`.

2 *Remarks:* An expression `addressof(E)` is a constant subexpression (**??**) if `E` is an lvalue constant subexpression.

20.10.11.3 `uninitialized_default_construct` [uninitialized.construct.default]

```
template<class ForwardIterator>
  void uninitialized_default_construct(ForwardIterator first, ForwardIterator last);
```

1 *Effects:* Equivalent to:

```
for (; first != last; ++first)
  ::new (voidify(*first))
    typename iterator_traits<ForwardIterator>::value_type;
```

```
namespace ranges {
  template<no-throw-forward-iterator I, no-throw-sentinel<I> S>
      requires default_constructible<iter_value_t<I>>
    I uninitialized_default_construct(I first, S last);
  template<no-throw-forward-range R>
      requires default_constructible<range_value_t<R>>
    safe_iterator_t<R> uninitialized_default_construct(R&& r);
}
```

2 *Effects:* Equivalent to:

```
for (; first != last; ++first)
  ::new (voidify(*first)) remove_reference_t<iter_reference_t<I>>;
return first;
```

```
template<class ForwardIterator, class Size>
  ForwardIterator uninitialized_default_construct_n(ForwardIterator first, Size n);
```

3 *Effects:* Equivalent to:

```
for (; n > 0; (void)++first, --n)
  ::new (voidify(*first))
    typename iterator_traits<ForwardIterator>::value_type;
return first;
```

```
namespace ranges {
  template<no-throw-forward-iterator I>
      requires default_constructible<iter_value_t<I>>
    I uninitialized_default_construct_n(I first, iter_difference_t<I> n);
}
```

4 *Effects:* Equivalent to:

```
return uninitialized_default_construct(counted_iterator(first, n),
                                       default_sentinel).base();
```

20.10.11.4 `uninitialized_value_construct` [uninitialized.construct.value]

```
template<class ForwardIterator>
  void uninitialized_value_construct(ForwardIterator first, ForwardIterator last);
```

1 *Effects:* Equivalent to:

```
for (; first != last; ++first)
  ::new (voidify(*first))
    typename iterator_traits<ForwardIterator>::value_type();
```

```
namespace ranges {
  template<no-throw-forward-iterator I, no-throw-sentinel<I> S>
      requires default_constructible<iter_value_t<I>>
    I uninitialized_value_construct(I first, S last);
  template<no-throw-forward-range R>
      requires default_constructible<range_value_t<R>>
    safe_iterator_t<R> uninitialized_value_construct(R&& r);
}
```

2 *Effects:* Equivalent to:

```
for (; first != last; ++first)
  ::new (voidify(*first)) remove_reference_t<iter_reference_t<I>>();
return first;
```

```
template<class ForwardIterator, class Size>
  ForwardIterator uninitialized_value_construct_n(ForwardIterator first, Size n);
```

3 *Effects:* Equivalent to:

```
for (; n > 0; (void)++first, --n)
  ::new (voidify(*first))
    typename iterator_traits<ForwardIterator>::value_type();
return first;
```

```
namespace ranges {
  template<no-throw-forward-iterator I>
      requires default_constructible<iter_value_t<I>>
    I uninitialized_value_construct_n(I first, iter_difference_t<I> n);
}
```

4 *Effects:* Equivalent to:

```
return uninitialized_value_construct(counted_iterator(first, n),
                                     default_sentinel).base();
```

20.10.11.5 `uninitialized_copy` [uninitialized.copy]

```
template<class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                     ForwardIterator result);
```

1 *Preconditions:* `[result, (last - first))` shall not overlap with `[first, last)`.

2 *Effects:* Equivalent to:

```
for (; first != last; ++result, (void) ++first)
  ::new (voidify(*result))
    typename iterator_traits<ForwardIterator>::value_type(*first);
```

3 *Returns:* `result`.

```
namespace ranges {
  template<input_iterator I, sentinel_for<I> S1,
           no-throw-forward-iterator O, no-throw-sentinel<O> S2>
      requires constructible_from<iter_value_t<O>, iter_reference_t<I>>
    uninitialized_copy_result<I, O>
      uninitialized_copy(I ifirst, S1 ilast, O ofirst, S2 olast);
  template<input_range IR, no-throw-forward-range OR>
      requires constructible_from<range_value_t<OR>, range_reference_t<IR>>
    uninitialized_copy_result<safe_iterator_t<IR>, safe_iterator_t<OR>>
      uninitialized_copy(IR&& in_range, OR&& out_range);
}
```

4 *Preconditions:* `[ofirst, olast)` shall not overlap with `[ifirst, ilast)`.

5 *Effects:* Equivalent to:

```
for (; ifirst != ilast && ofirst != olast; ++ofirst, (void)++ifirst) {
  ::new (voidify(*ofirst)) remove_reference_t<iter_reference_t<O>>(*ifirst);
}
return {ifirst, ofirst};
```

```
template<class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(InputIterator first, Size n, ForwardIterator result);
```

6 *Preconditions:* `[result, n)` shall not overlap with `[first, n)`.

7 *Effects:* Equivalent to:

```
for ( ; n > 0; ++result, (void) ++first, --n) {
  ::new (voidify(*result))
    typename iterator_traits<ForwardIterator>::value_type(*first);
}
```

8 *Returns:* `result`.

```
namespace ranges {
  template<input_iterator I, no-throw-forward-iterator O, no-throw-sentinel<O> S>
      requires constructible_from<iter_value_t<O>, iter_reference_t<I>>
    uninitialized_copy_n_result<I, O>
      uninitialized_copy_n(I ifirst, iter_difference_t<I> n, O ofirst, S olast);
}
```

9 *Preconditions:* `[ofirst, olast)` shall not overlap with `[ifirst, n)`.

10 *Effects:* Equivalent to:

```
auto t = uninitialized_copy(counted_iterator(ifirst, n),
                            default_sentinel, ofirst, olast);
return {t.in.base(), t.out};
```

20.10.11.6 `uninitialized_move` [**uninitialized.move**]

```
template<class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_move(InputIterator first, InputIterator last,
                                     ForwardIterator result);
```

1 *Preconditions:* `[result, (last - first))` shall not overlap with `[first, last)`.

2 *Effects:* Equivalent to:

```
for (; first != last; (void)++result, ++first)
  ::new (voidify(*result))
    typename iterator_traits<ForwardIterator>::value_type(std::move(*first));
return result;
```

```
namespace ranges {
  template<input_iterator I, sentinel_for<I> S1,
           no-throw-forward-iterator O, no-throw-sentinel<O> S2>
      requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
    uninitialized_move_result<I, O>
      uninitialized_move(I ifirst, S1 ilast, O ofirst, S2 olast);
```

```
  template<input_range IR, no-throw-forward-range OR>
      requires constructible_from<range_value_t<OR>, range_rvalue_reference_t<IR>>
    uninitialized_move_result<safe_iterator_t<IR>, safe_iterator_t<OR>>
      uninitialized_move(IR&& in_range, OR&& out_range);
}
```

3 *Preconditions:* `[ofirst, olast)` shall not overlap with `[ifirst, ilast)`.

4 *Effects:* Equivalent to:

```
for (; ifirst != ilast && ofirst != olast; ++ofirst, (void)++ifirst) {
  ::new (voidify(*ofirst))
    remove_reference_t<iter_reference_t<O>>(ranges::iter_move(ifirst));
}
return {ifirst, ofirst};
```

5 [*Note*: If an exception is thrown, some objects in the range `[first, last)` are left in a valid, but unspecified state. — *end note*]

```
template<class InputIterator, class Size, class ForwardIterator>
  pair<InputIterator, ForwardIterator>
    uninitialized_move_n(InputIterator first, Size n, ForwardIterator result);
```

6 *Preconditions:* `[result, n)` shall not overlap with `[first, n)`.

7 *Effects:* Equivalent to:

```
for (; n > 0; ++result, (void) ++first, --n)
  ::new (voidify(*result))
    typename iterator_traits<ForwardIterator>::value_type(std::move(*first));
return {first,result};
```

```
namespace ranges {
  template<input_iterator I, no-throw-forward-iterator O, no-throw-sentinel<O> S>
      requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
    uninitialized_move_n_result<I, O>
      uninitialized_move_n(I ifirst, iter_difference_t<I> n, O ofirst, S olast);
}
```

8 *Preconditions:* `[ofirst, olast)` shall not overlap with `[ifirst, n)`.

9 *Effects:* Equivalent to:

```
auto t = uninitialized_move(counted_iterator(ifirst, n),
                            default_sentinel, ofirst, olast);
return {t.in.base(), t.out};
```

10 [*Note*: If an exception is thrown, some objects in the range `[first, n)` are left in a valid but unspecified state. — *end note*]

20.10.11.7 `uninitialized_fill` [uninitialized.fill]

```
template<class ForwardIterator, class T>
  void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x);
```

1 *Effects:* Equivalent to:

```
for (; first != last; ++first)
  ::new (voidify(*first))
    typename iterator_traits<ForwardIterator>::value_type(x);
```

```
namespace ranges {
  template<no-throw-forward-iterator I, no-throw-sentinel<I> S, class T>
      requires constructible_from<iter_value_t<I>, const T&>
    I uninitialized_fill(I first, S last, const T& x);
  template<no-throw-forward-range R, class T>
      requires constructible_from<range_value_t<R>, const T&>
    safe_iterator_t<R> uninitialized_fill(R&& r, const T& x);
}
```

Effects: Equivalent to:

```
    for (; first != last; ++first) {
      ::new (voidify(*first)) remove_reference_t<iter_reference_t<I>>(x);
    }
    return first;
```

```
template<class ForwardIterator, class Size, class T>
  ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
```

2 *Effects:* Equivalent to:

```
    for (; n--; ++first)
      ::new (voidify(*first))
        typename iterator_traits<ForwardIterator>::value_type(x);
    return first;
```

```
namespace ranges {
  template<no-throw-forward-iterator I, class T>
      requires constructible_from<iter_value_t<I>, const T&>
    I uninitialized_fill_n(I first, iter_difference_t<I> n, const T& x);
}
```

3 *Effects:* Equivalent to:

```
    return uninitialized_fill(counted_iterator(first, n), default_sentinel, x).base();
```

20.10.11.8 `construct_at` [specialized.construct]

```
template<class T, class... Args>
  constexpr T* construct_at(T* location, Args&&... args);

namespace ranges {
  template<class T, class... Args>
    constexpr T* construct_at(T* location, Args&&... args);
}
```

1 *Constraints:* The expression `::new (declval<void*>()) T(declval<Args>()...)` is well-formed when treated as an unevaluated operand.

2 *Effects:* Equivalent to:

```
    return ::new (voidify(*location)) T(std::forward<Args>(args)...);
```

20.10.11.9 `destroy` [specialized.destroy]

```
template<class T>
  constexpr void destroy_at(T* location);
namespace ranges {
  template<destructible T>
    constexpr void destroy_at(T* location) noexcept;
}
```

1 *Effects:*

(1.1) — If `T` is an array type, equivalent to `destroy(begin(*location), end(*location))`.

(1.2) — Otherwise, equivalent to `location->~T()`.

```
template<class ForwardIterator>
  constexpr void destroy(ForwardIterator first, ForwardIterator last);
```

2 *Effects:* Equivalent to:

```
    for (; first != last; ++first)
      destroy_at(addressof(*first));
```

```
namespace ranges {
  template<no-throw-input-iterator I, no-throw-sentinel<I> S>
      requires destructible<iter_value_t<I>>
    constexpr I destroy(I first, S last) noexcept;
```

```
  template<no-throw-input-range R>
      requires destructible<range_value_t<R>>
    constexpr safe_iterator_t<R> destroy(R&& r) noexcept;
}
```

3 *Effects:* Equivalent to:

```
  for (; first != last; ++first)
    destroy_at(addressof(*first));
  return first;
```

```
template<class ForwardIterator, class Size>
  constexpr ForwardIterator destroy_n(ForwardIterator first, Size n);
```

4 *Effects:* Equivalent to:

```
  for (; n > 0; (void)++first, --n)
    destroy_at(addressof(*first));
  return first;
```

```
namespace ranges {
  template<no-throw-input-iterator I>
      requires destructible<iter_value_t<I>>
    constexpr I destroy_n(I first, iter_difference_t<I> n) noexcept;
}
```

5 *Effects:* Equivalent to:

```
  return destroy(counted_iterator(first, n), default_sentinel).base();
```

20.10.12 C library memory allocation [c.malloc]

1 [*Note*: The header `<cstdlib>` (**??**) declares the functions described in this subclause. — *end note*]

```
void* aligned_alloc(size_t alignment, size_t size);
void* calloc(size_t nmemb, size_t size);
void* malloc(size_t size);
void* realloc(void* ptr, size_t size);
```

2 *Effects:* These functions have the semantics specified in the C standard library.

3 *Remarks:* These functions do not attempt to allocate storage by calling `::operator new()` (**??**).

4 Storage allocated directly with these functions is implicitly declared reachable (see **??**) on allocation, ceases to be declared reachable on deallocation, and need not cease to be declared reachable as the result of an `undeclare_reachable()` call. [*Note*: This allows existing C libraries to remain unaffected by restrictions on pointers that are not safely derived, at the expense of providing far fewer garbage collection and leak detection options for `malloc()`-allocated objects. It also allows `malloc()` to be implemented with a separate allocation arena, bypassing the normal `declare_reachable()` implementation. The above functions should never intentionally be used as a replacement for `declare_reachable()`, and newly written code is strongly encouraged to treat memory allocated with these functions as though it were allocated with `operator new`. — *end note*]

```
void free(void* ptr);
```

5 *Effects:* This function has the semantics specified in the C standard library.

6 *Remarks:* This function does not attempt to deallocate storage by calling `::operator delete()`.

See also: ISO C 7.22.3

20.11 Smart pointers [smartptr]

20.11.1 Class template `unique_ptr` [unique.ptr]

1 A *unique pointer* is an object that owns another object and manages that other object through a pointer. More precisely, a unique pointer is an object u that stores a pointer to a second object p and will dispose of p when u is itself destroyed (e.g., when leaving block scope (**??**)). In this context, u is said to *own* `p`.

2 The mechanism by which u disposes of p is known as p's associated *deleter*, a function object whose correct invocation results in p's appropriate disposition (typically its deletion).

3 Let the notation *u.p* denote the pointer stored by *u*, and let *u.d* denote the associated deleter. Upon request, *u* can *reset* (replace) *u.p* and *u.d* with another pointer and deleter, but properly disposes of its owned object via the associated deleter before such replacement is considered completed.

4 Each object of a type `U` instantiated from the `unique_ptr` template specified in this subclause has the strict ownership semantics, specified above, of a unique pointer. In partial satisfaction of these semantics, each such `U` is *Cpp17MoveConstructible* and *Cpp17MoveAssignable*, but is not *Cpp17CopyConstructible* nor *Cpp17CopyAssignable*. The template parameter `T` of `unique_ptr` may be an incomplete type.

5 [*Note*: The uses of `unique_ptr` include providing exception safety for dynamically allocated memory, passing ownership of dynamically allocated memory to a function, and returning dynamically allocated memory from a function. — *end note*]

20.11.1.1 Default deleters [unique.ptr.dltr]

20.11.1.1.1 In general [unique.ptr.dltr.general]

1 The class template `default_delete` serves as the default deleter (destruction policy) for the class template `unique_ptr`.

2 The template parameter `T` of `default_delete` may be an incomplete type.

20.11.1.1.2 `default_delete` [unique.ptr.dltr.dflt]

```
namespace std {
  template<class T> struct default_delete {
    constexpr default_delete() noexcept = default;
    template<class U> default_delete(const default_delete<U>&) noexcept;
    void operator()(T*) const;
  };
}
```

```
template<class U> default_delete(const default_delete<U>& other) noexcept;
```

1 *Constraints:* `U*` is implicitly convertible to `T*`

2 *Effects:* Constructs a `default_delete` object from another `default_delete<U>` object.

3 *Remarks:* This constructor shall not participate in overload resolution unless `U*` is implicitly convertible to `T*`.

```
void operator()(T* ptr) const;
```

4 *Effects:* Calls `delete` on `ptr`.

5 *Remarks:* If `T` is an incomplete type, the program is ill-formed.

20.11.1.1.3 `default_delete<T[]>` [unique.ptr.dltr.dflt1]

```
namespace std {
  template<class T> struct default_delete<T[]> {
    constexpr default_delete() noexcept = default;
    template<class U> default_delete(const default_delete<U[]>&) noexcept;
    template<class U> void operator()(U* ptr) const;
  };
}
```

```
template<class U> default_delete(const default_delete<U[]>& other) noexcept;
```

1 *Constraints:* `U(*)[]` is convertible to `T(*)[]`.

2 *Effects:* Constructs a `default_delete` object from another `default_delete<U[]>` object.

3 *Remarks:* This constructor shall not participate in overload resolution unless `U(*)[]` is convertible to `T(*)[]`.

```
template<class U> void operator()(U* ptr) const;
```

4 *Constraints:* `U(*)[]` is convertible to `T(*)[]`.

5 *Effects:* Calls `delete[]` on `ptr`.

6 *Remarks:* If `U` is an incomplete type, the program is ill-formed. This function shall not participate in overload resolution unless `U(*)[]` is convertible to `T(*)[]`.

20.11.1.2 unique_ptr for single objects [unique.ptr.single]

```
namespace std {
  template<class T, class D = default_delete<T>> class unique_ptr {
  public:
    using pointer      = see below;
    using element_type = T;
    using deleter_type = D;

    // 20.11.1.2.1, constructors
    constexpr unique_ptr() noexcept;
    explicit unique_ptr(pointer p) noexcept;
    unique_ptr(pointer p, see below d1) noexcept;
    unique_ptr(pointer p, see below d2) noexcept;
    unique_ptr(unique_ptr&& u) noexcept;
    constexpr unique_ptr(nullptr_t) noexcept;
    template<class U, class E>
      unique_ptr(unique_ptr<U, E>&& u) noexcept;

    // 20.11.1.2.2, destructor
    ~unique_ptr();

    // 20.11.1.2.3, assignment
    unique_ptr& operator=(unique_ptr&& u) noexcept;
    template<class U, class E>
      unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
    unique_ptr& operator=(nullptr_t) noexcept;

    // 20.11.1.2.4, observers
    add_lvalue_reference_t<T> operator*() const;
    pointer operator->() const noexcept;
    pointer get() const noexcept;
    deleter_type& get_deleter() noexcept;
    const deleter_type& get_deleter() const noexcept;
    explicit operator bool() const noexcept;

    // 20.11.1.2.5, modifiers
    pointer release() noexcept;
    void reset(pointer p = pointer()) noexcept;
    void swap(unique_ptr& u) noexcept;

    // disable copy from lvalue
    unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;
  };
}
```

1 The default type for the template parameter `D` is `default_delete`. A client-supplied template argument `D` shall be a function object type (20.14), lvalue reference to function, or lvalue reference to function object type for which, given a value `d` of type `D` and a value `ptr` of type `unique_ptr<T, D>::pointer`, the expression `d(ptr)` is valid and has the effect of disposing of the pointer as appropriate for that deleter.

2 If the deleter's type `D` is not a reference type, `D` shall meet the *Cpp17Destructible* requirements (Table **??**).

3 If the *qualified-id* `remove_reference_t<D>::pointer` is valid and denotes a type (**??**), then `unique_ptr<T, D>::pointer` shall be a synonym for `remove_reference_t<D>::pointer`. Otherwise `unique_ptr<T, D>::pointer` shall be a synonym for `element_type*`. The type `unique_ptr<T, D>::pointer` shall meet the *Cpp17NullablePointer* requirements (Table **??**).

4 [*Example*: Given an allocator type `X` (Table **??**) and letting `A` be a synonym for `allocator_traits<X>`, the types `A::pointer`, `A::const_pointer`, `A::void_pointer`, and `A::const_void_pointer` may be used as `unique_ptr<T, D>::pointer`. — *end example*]

20.11.1.2.1 Constructors [unique.ptr.single.ctor]

```
constexpr unique_ptr() noexcept;
```

```
constexpr unique_ptr(nullptr_t) noexcept;
```

1 ~~*Requires:*~~ *Preconditions:* `D` ~~shall meet~~meets the *Cpp17DefaultConstructible* requirements (Table **??**), and that construction shall not throw an exception.

2 *Constraints:* `is_pointer_v<deleter_type>` is `false` and `is_default_constructible_v<deleter_type>` is `true`.

3 *Effects:* Constructs a `unique_ptr` object that owns nothing, value-initializing the stored pointer and the stored deleter.

4 *Postconditions:* `get() == nullptr`. `get_deleter()` returns a reference to the stored deleter.

5 *Remarks:* If `is_pointer_v<deleter_type>` is `true` or `is_default_constructible_v<deleter_type>` is `false`, this constructor shall not participate in overload resolution.

```
explicit unique_ptr(pointer p) noexcept;
```

6 ~~*Requires:*~~ *Preconditions:* `D` ~~shall meet~~meets the *Cpp17DefaultConstructible* requirements (Table **??**), and that construction ~~shall~~does not throw an exception.

7 *Constraints:* `is_pointer_v<deleter_type>` is `false` and `is_default_constructible_v<deleter_type>` is `true`.

8 *Mandates:* This constructor shall not be selected by class template argument deduction (**??**).

9 *Effects:* Constructs a `unique_ptr` which owns `p`, initializing the stored pointer with `p` and value-initializing the stored deleter.

10 *Postconditions:* `get() == p`. `get_deleter()` returns a reference to the stored deleter.

11 *Remarks:* If `is_pointer_v<deleter_type>` is `true` or `is_default_constructible_v<deleter_type>` is `false`, this constructor shall not participate in overload resolution. If class template argument deduction (**??**) would select the function template corresponding to this constructor, then the program is ill-formed.

```
unique_ptr(pointer p, const D& d) noexcept;
unique_ptr(pointer p, remove_reference_t<D>&& d) noexcept;
```

12 *Constraints:* `is_constructible_v<D, decltype(d)>` is `true`.

13 *Mandates:* For the second constructor, `D` is not a reference type. These constructors shall not be selected by class template argument deduction (**??**).

14 ~~*Requires:*~~ *Preconditions:* For the first constructor, if `D` is not a reference type, `D` ~~shall meet~~meets the *Cpp17CopyConstructible* requirements and such construction shall not exit via an exception. For the second constructor, if `D` is not a reference type, `D` shall meet the *Cpp17MoveConstructible* requirements and such construction shall not exit via an exception.

15 *Effects:* Constructs a `unique_ptr` object which owns `p`, initializing the stored pointer with `p` and initializing the deleter from `std::forward<decltype(d)>(d)`.

16 *Remarks:* If `D` is a reference type, the second constructor is defined as deleted. These constructors shall not participate in overload resolution unless `is_constructible_v<D, decltype(d)>` is `true`.

17 *Postconditions:* `get() == p`. `get_deleter()` returns a reference to the stored deleter. If `D` is a reference type then `get_deleter()` returns a reference to the lvalue `d`.

18 *Remarks:* If class template argument deduction (**??**) would select a function template corresponding to either of these constructors, then the program is ill-formed.

19 [*Example*:

```
D d;
unique_ptr<int, D> p1(new int, D());         // D must be Cpp17MoveConstructible
unique_ptr<int, D> p2(new int, d);           // D must be Cpp17CopyConstructible
unique_ptr<int, D&> p3(new int, d);          // p3 holds a reference to d
unique_ptr<int, const D&> p4(new int, D()); // error: rvalue deleter object combined
                                             // with reference deleter type
```

— *end example*]

`unique_ptr(unique_ptr&& u) noexcept;`

20 *Constraints:* `is_move_constructible_v<D>` is `true`.

21 ~~*Requires:*~~ *Preconditions:* If `D` is not a reference type, `D` ~~shall meet~~meets the *Cpp17MoveConstructible* requirements (Table **??**). Construction of the deleter from an rvalue of type `D` ~~shall~~does not throw an exception.

22 *Effects:* Constructs a `unique_ptr` from `u`. If `D` is a reference type, this deleter is copy constructed from `u`'s deleter; otherwise, this deleter is move constructed from `u`'s deleter. [*Note*: The construction of the deleter can be implemented with `std::forward<D>`. — *end note*]

23 *Postconditions:* `get()` yields the value `u.get()` yielded before the construction. `u.get() == nullptr`. `get_deleter()` returns a reference to the stored deleter that was constructed from `u.get_deleter()`. If `D` is a reference type then `get_deleter()` and `u.get_deleter()` both reference the same lvalue deleter.

`template<class U, class E> unique_ptr(unique_ptr<U, E>&& u) noexcept;`

24 *Requires:* If `E` is not a reference type, construction of the deleter from an rvalue of type `E` shall be well-formed and shall not throw an exception. Otherwise, `E` is a reference type and construction of the deleter from an lvalue of type `E` shall be well-formed and shall not throw an exception.

25 ~~*Remarks:* This constructor shall not participate in overload resolution unless:~~*Constraints:*

(25.1) — `unique_ptr<U, E>::pointer` is implicitly convertible to `pointer`,

(25.2) — `U` is not an array type, and

(25.3) — either `D` is a reference type and `E` is the same type as `D`, or `D` is not a reference type and `E` is implicitly convertible to `D`.

26 *Preconditions:* If `E` is not a reference type, construction of the deleter from an rvalue of type `E` shall be well-formed and shall not throw an exception. Otherwise, `E` is a reference type and construction of the deleter from an lvalue of type `E` is be well-formed and does not throw an exception.

27 *Effects:* Constructs a `unique_ptr` from `u`. If `E` is a reference type, this deleter is copy constructed from `u`'s deleter; otherwise, this deleter is move constructed from `u`'s deleter. [*Note*: The deleter constructor can be implemented with `std::forward<E>`. — *end note*]

28 *Postconditions:* `get()` yields the value `u.get()` yielded before the construction. `u.get() == nullptr`. `get_deleter()` returns a reference to the stored deleter that was constructed from `u.get_deleter()`.

20.11.1.2.2 Destructor [unique.ptr.single.dtor]

`~unique_ptr();`

1 ~~*Requires:*~~ *Preconditions:* The expression `get_deleter()(get())` ~~shall be~~is well-formed, ~~shall have~~has well-defined behavior, and ~~shall~~does not throw exceptions. [*Note*: The use of `default_delete` requires `T` to be a complete type. — *end note*]

2 *Effects:* If `get() == nullptr` there are no effects. Otherwise `get_deleter()(get())`.

20.11.1.2.3 Assignment [unique.ptr.single.asgn]

`unique_ptr& operator=(unique_ptr&& u) noexcept;`

1 *Constraints:* `is_move_assignable_v<D>` is `true`.

2 ~~*Requires:*~~ *Preconditions:* If `D` is not a reference type, `D` ~~shall meet~~meets the *Cpp17MoveAssignable* requirements (Table **??**) and assignment of the deleter from an rvalue of type `D` ~~shall~~does not throw an exception. Otherwise, `D` is a reference type; `remove_reference_t<D>` ~~shall meet~~meets the *Cpp17CopyAssignable* requirements and assignment of the deleter from an lvalue of type `D` ~~shall~~does not throw an exception.

3 *Effects:* Calls `reset(u.release())` followed by `get_deleter() = std::forward<D>(u.get_deleter())`.

4 *Returns:* `*this`.

5 *Postconditions:* `u.get() == nullptr`.

```
template<class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
```

6 *Requires:* If `E` is not a reference type, assignment of the deleter from an rvalue of type `E` shall be well-formed and shall not throw an exception. Otherwise, `E` is a reference type and assignment of the deleter from an lvalue of type `E` shall be well-formed and shall not throw an exception.

7 ~~*Remarks:* This constructor shall not participate in overload resolution unless:~~*Constraints:*

(7.1) — `unique_ptr<U, E>::pointer` is implicitly convertible to `pointer`, and

(7.2) — `U` is not an array type, and

(7.3) — `is_assignable_v<D&, E&&>` is `true`.

8 *Preconditions:* If `E` is not a reference type, assignment of the deleter from an rvalue of type `E` shall be well-formed and shall not throw an exception. Otherwise, `E` is a reference type and assignment of the deleter from an lvalue of type `E` is be well-formed and does not throw an exception.

9 *Effects:* Calls `reset(u.release())` followed by `get_deleter() = std::forward<E>(u.get_deleter())`.

10 *Returns:* `*this`.

11 *Postconditions:* `u.get() == nullptr`.

```
unique_ptr& operator=(nullptr_t) noexcept;
```

12 *Effects:* As if by `reset()`.

13 *Postconditions:* `get() == nullptr`.

14 *Returns:* `*this`.

20.11.1.2.4 Observers [unique.ptr.single.observers]

```
add_lvalue_reference_t<T> operator*() const;
```

1 ~~*Requires:*~~ *Preconditions:* `get() != nullptr`.

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

```
pointer operator->() const noexcept;
```

3 ~~*Requires:*~~ *Preconditions:* `get() != nullptr`.

4 *Returns:* `get()`.

5 [*Note*: The use of this function typically requires that `T` be a complete type. — *end note*]

```
pointer get() const noexcept;
```

6 *Returns:* The stored pointer.

```
deleter_type& get_deleter() noexcept;
const deleter_type& get_deleter() const noexcept;
```

7 *Returns:* A reference to the stored deleter.

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

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

20.11.1.2.5 Modifiers [unique.ptr.single.modifiers]

```
pointer release() noexcept;
```

1 *Postconditions:* `get() == nullptr`.

2 *Returns:* The value `get()` had at the start of the call to `release`.

```
void reset(pointer p = pointer()) noexcept;
```

3 ~~*Requires:*~~ *Preconditions:* The expression `get_deleter()(get())` ~~shall be~~is well-formed, ~~shall have~~has well-defined behavior, and ~~shall~~does not throw exceptions.

4 *Effects:* Assigns `p` to the stored pointer, and then if and only if the old value of the stored pointer, `old_p`, was not equal to `nullptr`, calls `get_deleter()(old_p)`. [*Note*: The order of these operations is significant because the call to `get_deleter()` may destroy `*this`. — *end note*]

5 *Postconditions:* `get() == p`. [*Note*: The postcondition does not hold if the call to `get_deleter()` destroys `*this` since `this->get()` is no longer a valid expression. — *end note*]

```
void swap(unique_ptr& u) noexcept;
```

6 ~~*Requires:*~~ *Preconditions:* `get_deleter()` ~~shall be~~meets the swappable (**??**) requirements and ~~shall not~~does not throw an exception under `swap`.

7 *Effects:* Invokes `swap` on the stored pointers and on the stored deleters of `*this` and `u`.

20.11.1.3 `unique_ptr` for array objects with a runtime length [unique.ptr.runtime]

```
namespace std {
  template<class T, class D> class unique_ptr<T[], D> {
  public:
    using pointer      = see below;
    using element_type = T;
    using deleter_type = D;

    // 20.11.1.3.1, constructors
    constexpr unique_ptr() noexcept;
    template<class U> explicit unique_ptr(U p) noexcept;
    template<class U> unique_ptr(U p, see below d) noexcept;
    template<class U> unique_ptr(U p, see below d) noexcept;
    unique_ptr(unique_ptr&& u) noexcept;
    template<class U, class E>
      unique_ptr(unique_ptr<U, E>&& u) noexcept;
    constexpr unique_ptr(nullptr_t) noexcept;

    // destructor
    ~unique_ptr();

    // assignment
    unique_ptr& operator=(unique_ptr&& u) noexcept;
    template<class U, class E>
      unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
    unique_ptr& operator=(nullptr_t) noexcept;

    // 20.11.1.3.3, observers
    T& operator[](size_t i) const;
    pointer get() const noexcept;
    deleter_type& get_deleter() noexcept;
    const deleter_type& get_deleter() const noexcept;
    explicit operator bool() const noexcept;

    // 20.11.1.3.4, modifiers
    pointer release() noexcept;
    template<class U> void reset(U p) noexcept;
    void reset(nullptr_t = nullptr) noexcept;
    void swap(unique_ptr& u) noexcept;

    // disable copy from lvalue
    unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;
  };
}
```

1 A specialization for array types is provided with a slightly altered interface.

(1.1) — Conversions between different types of `unique_ptr<T[], D>` that would be disallowed for the corresponding pointer-to-array types, and conversions to or from the non-array forms of `unique_ptr`, produce an ill-formed program.

(1.2) — Pointers to types derived from `T` are rejected by the constructors, and by `reset`.

(1.3) — The observers `operator*` and `operator->` are not provided.

(1.4) — The indexing observer `operator[]` is provided.

(1.5) — The default deleter will call `delete[]`.

2 Descriptions are provided below only for members that differ from the primary template.

3 The template argument `T` shall be a complete type.

20.11.1.3.1 Constructors [unique.ptr.runtime.ctor]

```
template<class U> explicit unique_ptr(U p) noexcept;
```

1 This constructor behaves the same as the constructor in the primary template that takes a single parameter of type `pointer`~~except that it additionally shall not participate in overload resolution unless~~.

2 *Constraints:*

(2.1) — `U` is the same type as `pointer`, or

(2.2) — `pointer` is the same type as `element_type*`, `U` is a pointer type `V*`, and `V(*)[]` is convertible to `element_type(*)[]`.

```
template<class U> unique_ptr(U p, see below d) noexcept;
template<class U> unique_ptr(U p, see below d) noexcept;
```

3 These constructors behave the same as the constructors in the primary template that take a parameter of type `pointer` and a second parameter ~~except that they shall not participate in overload resolution unless either~~.

4 *Constraints:*

(4.1) — `U` is the same type as `pointer`,

(4.2) — `U` is `nullptr_t`, or

(4.3) — `pointer` is the same type as `element_type*`, `U` is a pointer type `V*`, and `V(*)[]` is convertible to `element_type(*)[]`.

```
template<class U, class E> unique_ptr(unique_ptr<U, E>&& u) noexcept;
```

5 This constructor behaves the same as in the primary template~~, except that it shall not participate in overload resolution unless all of the following conditions hold,~~.

6 *Constraints:* where `UP` is `unique_ptr<U, E>`:

(6.1) — `U` is an array type, and

(6.2) — `pointer` is the same type as `element_type*`, and

(6.3) — `UP::pointer` is the same type as `UP::element_type*`, and

(6.4) — `UP::element_type(*)[]` is convertible to `element_type(*)[]`, and

(6.5) — either `D` is a reference type and `E` is the same type as `D`, or `D` is not a reference type and `E` is implicitly convertible to `D`.

[*Note*: This replaces the ~~overload-resolution~~constraints specification of the primary template — *end note*]

20.11.1.3.2 Assignment [unique.ptr.runtime.asgn]

```
template<class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u)noexcept;
```

1 This operator behaves the same as in the primary template~~, except that it shall not participate in overload resolution unless all of the following conditions hold,~~.

2 *Constraints:* where `UP` is `unique_ptr<U, E>`:

(2.1) — `U` is an array type, and

(2.2) — `pointer` is the same type as `element_type*`, and

(2.3) — `UP::pointer` is the same type as `UP::element_type*`, and

(2.4) — `UP::element_type(*)[]` is convertible to `element_type(*)[]`, and

(2.5) — `is_assignable_v<D&, E&&>` is `true`.

[*Note*: This replaces the ~~overload-resolution~~constraints specification of the primary template — *end note*]

20.11.1.3.3 Observers [unique.ptr.runtime.observers]

```
T& operator[](size_t i) const;
```

1 ~~*Requires:*~~ *Preconditions:* `i` < the number of elements in the array to which the stored pointer points.

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

20.11.1.3.4 Modifiers [unique.ptr.runtime.modifiers]

```
void reset(nullptr_t p = nullptr) noexcept;
```

1 *Effects:* Equivalent to `reset(pointer())`.

```
template<class U> void reset(U p) noexcept;
```

2 This function behaves the same as the `reset` member of the primary template~~, except that it shall not participate in overload resolution unless either~~.

3 *Constraints:*

(3.1) — `U` is the same type as `pointer`, or

(3.2) — `pointer` is the same type as `element_type*`, `U` is a pointer type `V*`, and `V(*)[]` is convertible to `element_type(*)[]`.

20.11.1.4 Creation [unique.ptr.create]

```
template<class T, class... Args> unique_ptr<T> make_unique(Args&&... args);
```

1 ~~*Remarks:* This function shall not participate in overload resolution unless~~*Constraints:* `T` is not an array.

2 *Returns:* `unique_ptr<T>(new T(std::forward<Args>(args)...))`.

```
template<class T> unique_ptr<T> make_unique(size_t n);
```

3 ~~*Remarks:* This function shall not participate in overload resolution unless~~*Constraints:* `T` is an array of unknown bound.

4 *Returns:* `unique_ptr<T>(new remove_extent_t<T>[n]())`.

```
template<class T, class... Args> unspecified make_unique(Args&&...) = delete;
```

5 ~~*Remarks:* This function shall not participate in overload resolution unless~~*Constraints:* `T` is an array of known bound.

```
template<class T> unique_ptr<T> make_unique_default_init();
```

6 *Constraints:* `T` is not an array.

7 *Returns:* `unique_ptr<T>(new T)`.

```
template<class T> unique_ptr<T> make_unique_default_init(size_t n);
```

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

9 *Returns:* `unique_ptr<T>(new remove_extent_t<T>[n])`.

```
template<class T, class... Args> unspecified make_unique_default_init(Args&&...) = delete;
```

10 *Constraints:* `T` is an array of known bound.

20.11.1.5 Specialized algorithms [unique.ptr.special]

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

1 ~~*Remarks:* This function shall not participate in overload resolution unless~~*Constraints:* `is_swappable_v<D>` is `true`.

2 *Effects:* Calls `x.swap(y)`.

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

3 *Returns:* `x.get() == y.get()`.

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

4 ~~*Requires:*~~ *Preconditions:* Let `CT` denote

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

Then the specialization `less<CT>` ~~shall be~~is a function object type (20.14) that induces a strict weak ordering (**??**) on the pointer values.

5 *Mandates:* `unique_ptr<T1, D1>::pointer` is implicitly convertible to `CT` and `unique_ptr<T2, D2>::pointer` is implicitly convertible to `CT`.

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

7 *Remarks:* If `unique_ptr<T1, D1>::pointer` is not implicitly convertible to `CT` or `unique_ptr<T2, D2>::pointer` is not implicitly convertible to `CT`, the program is ill-formed.

```
template<class T1, class D1, class T2, class D2>
  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>
  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>
  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>
  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>
  bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;
```

12 *Returns:* `!x`.

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

13 ~~*Requires:*~~ *Preconditions:* The specialization `less<unique_ptr<T, D>::pointer>` ~~shall be~~is a function object type (20.14) that induces a strict weak ordering (**??**) 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>
  bool operator>(const unique_ptr<T, D>& x, nullptr_t);
template<class T, class D>
  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>
  bool operator<=(const unique_ptr<T, D>& x, nullptr_t);
```

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

18 *Returns:* `compare_three_way()(x.get(), nullptr)`.

20.11.1.6 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`.

4 *Remarks:* This function shall not participate in overload resolution unless `os << p.get()` is a valid expression.

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

```
namespace std {
  class bad_weak_ptr : public exception {
  public:
    // see ?? for the specification of the special member functions
    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`.

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

2 *Returns:* An implementation-defined NTBS.

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

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>;

    // 20.11.3.1, constructors
    constexpr shared_ptr() noexcept;
    constexpr shared_ptr(nullptr_t) noexcept : shared_ptr() { }
    template<class Y>
      explicit shared_ptr(Y* p);
    template<class Y, class D>
      shared_ptr(Y* p, D d);
```

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

    // 20.11.3.2, destructor
    ~shared_ptr();

    // 20.11.3.3, assignment
    shared_ptr& operator=(const shared_ptr& r) noexcept;
    template<class Y>
      shared_ptr& operator=(const shared_ptr<Y>& r) noexcept;
    shared_ptr& operator=(shared_ptr&& r) noexcept;
    template<class Y>
      shared_ptr& operator=(shared_ptr<Y>&& r) noexcept;
    template<class Y, class D>
      shared_ptr& operator=(unique_ptr<Y, D>&& r);

    // 20.11.3.4, modifiers
    void swap(shared_ptr& r) noexcept;
    void reset() noexcept;
    template<class Y>
      void reset(Y* p);
    template<class Y, class D>
      void reset(Y* p, D d);
    template<class Y, class D, class A>
      void reset(Y* p, D d, A a);

    // 20.11.3.5, observers
    element_type* get() const noexcept;
    T& operator*() const noexcept;
    T* operator->() const noexcept;
    element_type& operator[](ptrdiff_t i) const;
    long use_count() const noexcept;
    explicit operator bool() const noexcept;
    template<class U>
      bool owner_before(const shared_ptr<U>& b) const noexcept;
    template<class U>
      bool owner_before(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. The template parameter `T` of `shared_ptr` may be an incomplete type.

3 [*Example*:

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

— *end example*]

4 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.

5 For the purposes of subclause 20.11, 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[]`.

20.11.3.1 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` (20.11.6), 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 (**??**).

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

2 *Effects:* Constructs an empty `shared_ptr` object.

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

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

4 *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*`.

5 ~~*Requires:*~~ *Preconditions:* `Y` ~~shall be~~is a complete type. The expression `delete[] p`, when `T` is an array type, or `delete p`, when `T` is not an array type, ~~shall have~~has well-defined behavior, and ~~shall not~~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 could not be obtained.

9 *Remarks:* When `T` is an array type, this constructor shall not participate in overload resolution unless 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, this constructor shall not participate in overload resolution unless the expression `delete p` is well-formed and `Y*` is convertible to `T*`.

```
template<class Y, class D> shared_ptr(Y* p, D d);
template<class Y, class D, class A> shared_ptr(Y* p, D d, A a);
template<class D> shared_ptr(nullptr_t p, D d);
template<class D, class A> shared_ptr(nullptr_t p, D d, A a);
```

10 *Constraints:* When `T` is an array type, `is_move_constructible_v<D>` is `true`, the expression `d(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, `is_move_constructible_v<D>` is `true`, the expression `d(p)` is well-formed, and `Y*` is convertible to `T*`.

11 ~~*Requires:*~~ *Preconditions:* Construction of `d` and a deleter of type `D` initialized with `std::move(d)` ~~shall not~~do not throw exceptions. The expression `d(p)` ~~shall have~~has well-defined behavior and ~~shall not~~does not throw exceptions. `A` ~~shall meet~~meets the *Cpp17Allocator* requirements (Table **??**).

12 *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.

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

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

15 *Remarks:* When `T` is an array type, this constructor shall not participate in overload resolution unless `is_move_constructible_v<D>` is `true`, the expression `d(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, this constructor shall not participate in overload resolution unless `is_move_constructible_v<D>` is `true`, the expression `d(p)` is well-formed, and `Y*` is convertible to `T*`.

```
template<class Y> shared_ptr(const shared_ptr<Y>& r, element_type* p) noexcept;
template<class Y> shared_ptr(shared_ptr<Y>&& r, element_type* p) noexcept;
```

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

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

18 [*Note*: To avoid the possibility of a dangling pointer, the user of this constructor should ensure that `p` remains valid at least until the ownership group of `r` is destroyed. — *end note*]

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

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

20 ~~*Remarks:* The second constructor shall not participate in overload resolution unless~~*Constraints:* For the second constructor, `Y*` is compatible with `T*`.

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

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

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

23 ~~*Remarks:* The second constructor shall not participate in overload resolution unless~~*Constraints:* For the second constructor, `Y*` is compatible with `T*`.

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

25 *Postconditions:* `*this` shall contain the old value of `r`. `r` shall be empty. `r.get() == nullptr`.

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

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

27 *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.

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

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

30 ~~*Remarks:* This constructor shall not participate in overload resolution unless `Y*` is compatible with `T*`.~~

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

31 ~~*Remarks:* This constructor shall not participate in overload resolution unless~~ *Constraints:* `Y*` is compatible with `T*` and `unique_ptr<Y, D>::pointer` is convertible to `element_type*`.

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

20.11.3.2 Destructor [util.smartptr.shared.dest]

```
~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*: 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*]

20.11.3.3 Assignment [util.smartptr.shared.assign]

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

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

2 *Returns:* `*this`.

3 [*Note*: The use count updates caused by the temporary object construction and destruction are not observable side effects, so the implementation may 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 may be no-ops. — *end note*]

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

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

7 *Returns:* `*this`.

20.11.3.4 Modifiers [util.smartptr.shared.mod]

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

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

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

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

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

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

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

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

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

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

20.11.3.5 Observers [util.smartptr.shared.obs]

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

Returns: The stored pointer.

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

~~*Requires:*~~ *Preconditions:* `get() != 0`.

Returns: `*get()`.

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.

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

~~*Requires:*~~ *Preconditions:* `get() != 0`.

Returns: `get()`.

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.

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

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

Returns: `get()[i]`.

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.

Throws: Nothing.

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

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

Synchronization: None.

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

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

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

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

Returns: `get() != 0`.

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

Returns: An unspecified value such that

(18.1) — `x.owner_before(y)` defines a strict weak ordering as defined in **??**;

(18.2) — under the equivalence relation defined by `owner_before`, `!a.owner_before(b) && !b.owner_before(a)`, two `shared_ptr` or `weak_ptr` instances are equivalent if and only if they share ownership or are both empty.

20.11.3.6 Creation [util.smartptr.shared.create]

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

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

2 ~~*Requires:*~~ *Preconditions:* `A` ~~shall meet~~meets the *Cpp17Allocator* requirements (Table **??**).

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

5 *Postconditions:* `r.get() != 0 && r.use_count() == 1`, where `r` is the return value.

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*: 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, pv, v)` or

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

respectively, where `pv` points to storage suitable to hold an object of type `U` and `a2` of type `A2` is a rebound copy of the allocator `a` passed to `allocate_shared` such that its `value_type` is `remove_cv_t<U>`.

(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` shall initialize this (sub)object via the expression `allocator_traits<A2>::construct(a2, pv)`, where `pv` points to storage suitable to hold an object of type `U` and `a2` of type `A2` is a rebound copy of the allocator `a` passed to `allocate_shared` such that its `value_type` is `remove_cv_t<U>`.

(7.8) — When a (sub)object of non-array type `U` is initialized by `make_shared_default_init` or `allocate_shared_default_init`, 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` is to be destroyed, it is destroyed via the expression `pv->~U()` where `pv` 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, pv)` where `pv` points to that object of type `remove_cv_t<U>` and `a2` of type `A2` is a rebound copy of the allocator `a` passed to `allocate_shared` such that its `value_type` is `remove_cv_t<U>`.

[*Note*: 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>
  shared_ptr<T> make_shared(Args&&... args);                    // T is not array
template<class T, class A, class... Args>
  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(forward<Args>(args)...)`.

10 *Remarks:* ~~These overloads shall only participate in overload resolution when T is not an array type.~~ 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*:

```
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>
  make_shared(size_t N);                                        // T is U[]
template<class T, class A>
  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 *Remarks:* These overloads shall only participate in overload resolution when `T` is of the form `U[]`.

15 [*Example*:

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

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

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

18 *Remarks:* These overloads shall only participate in overload resolution when `T` is of the form `U[N]`.

19 [*Example*:

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

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

21 *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`.

22 ~~*Remarks:* These overloads shall only participate in overload resolution when `T` is of the form `U[]`.~~

23 [*Example*:

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

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

25 *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`.

26 *Remarks:* These overloads shall only participate in overload resolution when `T` is of the form `U[N]`.

27 [*Example*:

```
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>
  shared_ptr<T> make_shared_default_init();
template<class T, class A>
  shared_ptr<T> allocate_shared_default_init(const A& a);
```

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

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

30 [*Example*:

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

— *end example*]

```
template<class T>
  shared_ptr<T> make_shared_default_init(size_t N);
```

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

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

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

33 [*Example*:

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

— *end example*]

20.11.3.7 Comparison [util.smartptr.shared.cmp]

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

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

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

2 *Returns:* `!a`.

```
template<class T, class U>
  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*: Defining a comparison function allows `shared_ptr` objects to be used as keys in associative containers. — *end note*]

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

5 *Returns:* `compare_three_way()(a.get(), nullptr)`.

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

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

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

20.11.3.9 Casts [util.smartptr.shared.cast]

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

1 ~~*Requires:*~~ *Preconditions:* The expression `static_cast<T*>((U*)nullptr)` ~~shall be~~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*: The seemingly equivalent expression `shared_ptr<T>(static_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice. — *end note*]

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

4 ~~*Requires:*~~ *Preconditions:* The expression `dynamic_cast<T*>((U*)nullptr)` ~~shall be~~is well-formed. The expression `dynamic_cast<typename shared_ptr<T>::element_type*>(r.get())` ~~shall be~~is well formed and ~~shall have~~has well-defined behavior.

5 *Returns:*

(5.1) — 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.

(5.2) — Otherwise, `shared_ptr<T>()`.

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

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

7 ~~*Requires:*~~ *Preconditions:* The expression `const_cast<T*>((U*)nullptr)` ~~shall be~~is well-formed.

8 *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.

9 [*Note*: The seemingly equivalent expression `shared_ptr<T>(const_cast<T*>(r.get()))` will eventually 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;
```

10 ~~*Requires:*~~ *Preconditions:* The expression `reinterpret_cast<T*>((U*)nullptr)` ~~shall be~~is well-formed.

11 *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.

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

20.11.3.10 `get_deleter` [util.smartptr.getdeleter]

```
template<class D, class T>
  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*: 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*]

20.11.3.11 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`.

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

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>;
```

```
    // 20.11.4.1, constructors
    constexpr weak_ptr() noexcept;
    template<class Y>
      weak_ptr(const shared_ptr<Y>& r) noexcept;
    weak_ptr(const weak_ptr& r) noexcept;
    template<class Y>
      weak_ptr(const weak_ptr<Y>& r) noexcept;
    weak_ptr(weak_ptr&& r) noexcept;
    template<class Y>
      weak_ptr(weak_ptr<Y>&& r) noexcept;

    // 20.11.4.2, destructor
    ~weak_ptr();

    // 20.11.4.3, assignment
    weak_ptr& operator=(const weak_ptr& r) noexcept;
    template<class Y>
      weak_ptr& operator=(const weak_ptr<Y>& r) noexcept;
    template<class Y>
      weak_ptr& operator=(const shared_ptr<Y>& r) noexcept;
    weak_ptr& operator=(weak_ptr&& r) noexcept;
    template<class Y>
      weak_ptr& operator=(weak_ptr<Y>&& r) noexcept;

    // 20.11.4.4, modifiers
    void swap(weak_ptr& r) noexcept;
    void reset() noexcept;

    // 20.11.4.5, observers
    long use_count() const noexcept;
    bool expired() const noexcept;
    shared_ptr<T> lock() const noexcept;
    template<class U>
      bool owner_before(const shared_ptr<U>& b) const noexcept;
    template<class U>
      bool owner_before(const weak_ptr<U>& b) const noexcept;
  };

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

  // 20.11.4.6, specialized algorithms
  template<class T>
    void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;
}
```

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.

20.11.4.1 Constructors [util.smartptr.weak.const]

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

1 *Effects:* Constructs an empty `weak_ptr` object.

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

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

3 ~~*Remarks:* T~~*Constraints:* For the second and third constructors ~~shall not participate in overload resolution unless~~, `Y*` is compatible with `T*`.

4 *Effects:* If `r` is empty, constructs an empty `weak_ptr` object; 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()`.

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

6 ~~*Remarks:* T~~*Constraints:* For the second constructor ~~shall not participate in overload resolution unless,~~ `Y*` is compatible with `T*`.

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

8 *Postconditions:* `*this` shall contain the old value of `r`. `r` shall be empty. `r.use_count() == 0`.

20.11.4.2 Destructor [util.smartptr.weak.dest]

```
~weak_ptr();
```

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

20.11.4.3 Assignment [util.smartptr.weak.assign]

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

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

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

3 *Returns:* `*this`.

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

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

5 *Returns:* `*this`.

20.11.4.4 Modifiers [util.smartptr.weak.mod]

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

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

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

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

20.11.4.5 Observers [util.smartptr.weak.obs]

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

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

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

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

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

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

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

4 *Returns:* An unspecified value such that

(4.1) — `x.owner_before(y)` defines a strict weak ordering as defined in **??**;

(4.2) — under the equivalence relation defined by `owner_before`, `!a.owner_before(b) && !b.owner_before(a)`, two `shared_ptr` or `weak_ptr` instances are equivalent if and only if they share ownership or are both empty.

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

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

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

20.11.5 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>> {
    bool operator()(const shared_ptr<T>&, const shared_ptr<T>&) const noexcept;
    bool operator()(const shared_ptr<T>&, const weak_ptr<T>&) const noexcept;
    bool operator()(const weak_ptr<T>&, const shared_ptr<T>&) const noexcept;
  };

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

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

    using is_transparent = unspecified;
  };
}
```

2 `operator()(x, y)` shall return `x.owner_before(y)`. [*Note*: Note that

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

(2.2) — under the equivalence relation defined by `operator()`, `!operator()(a, b) && !operator()(b, a)`, two `shared_ptr` or `weak_ptr` instances are equivalent if and only if they share ownership or are both empty.

— *end note*]

20.11.6 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*:

```
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_before(q) && !q.owner_before(p)); // p and q share ownership
}
```

— *end example*]

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

  public:
    shared_ptr<T> shared_from_this();
    shared_ptr<T const> shared_from_this() const;
    weak_ptr<T> weak_from_this() noexcept;
    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;
enable_shared_from_this(const enable_shared_from_this<T>&) noexcept;
```

4 *Effects:* Value-initializes `weak_this`.

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

5 *Returns:* `*this`.

6 [*Note*: `weak_this` is not changed. — *end note*]

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

7 *Returns:* `shared_ptr<T>(weak_this)`.

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

8 *Returns:* `weak_this`.

20.11.7 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 (20.14.18) if and only if `hash<typename UP::pointer>` is enabled. When enabled, for an object `p` of type `UP`, `hash<UP>()(p)` shall evaluate 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)` shall evaluate to the same value as `hash<typename shared_ptr<T>::element_type*>()(p.get())`.

20.12 Memory resources [mem.res]

20.12.1 Header <memory_resource> synopsis [mem.res.syn]

```
namespace std::pmr {
  // 20.12.2, class memory_resource
  class memory_resource;

  bool operator==(const memory_resource& a, const memory_resource& b) noexcept;

  // 20.12.3, class template polymorphic_allocator
  template<class Tp> class polymorphic_allocator;
```

```
  template<class T1, class T2>
    bool operator==(const polymorphic_allocator<T1>& a,
                    const polymorphic_allocator<T2>& b) noexcept;

  // 20.12.4, global memory resources
  memory_resource* new_delete_resource() noexcept;
  memory_resource* null_memory_resource() noexcept;
  memory_resource* set_default_resource(memory_resource* r) noexcept;
  memory_resource* get_default_resource() noexcept;

  // 20.12.5, pool resource classes
  struct pool_options;
  class synchronized_pool_resource;
  class unsynchronized_pool_resource;
  class monotonic_buffer_resource;
}
```

20.12.2 Class `memory_resource` [mem.res.class]

1 The `memory_resource` class is an abstract interface to an unbounded set of classes encapsulating memory resources.

```
namespace std::pmr {
  class memory_resource {
    static constexpr size_t max_align = alignof(max_align_t);    // exposition only

  public:
    memory_resource() = default;
    memory_resource(const memory_resource&) = default;
    virtual ~memory_resource();

    memory_resource& operator=(const memory_resource&) = default;

    [[nodiscard]] void* allocate(size_t bytes, size_t alignment = max_align);
    void deallocate(void* p, size_t bytes, size_t alignment = max_align);

    bool is_equal(const memory_resource& other) const noexcept;

  private:
    virtual void* do_allocate(size_t bytes, size_t alignment) = 0;
    virtual void do_deallocate(void* p, size_t bytes, size_t alignment) = 0;

    virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;
  };
}
```

20.12.2.1 Public member functions [mem.res.public]

`~memory_resource();`

1 *Effects:* Destroys this `memory_resource`.

`[[nodiscard]] void* allocate(size_t bytes, size_t alignment = max_align);`

2 *Effects:* Equivalent to: `return do_allocate(bytes, alignment);`

`void deallocate(void* p, size_t bytes, size_t alignment = max_align);`

3 *Effects:* Equivalent to `do_deallocate(p, bytes, alignment)`.

`bool is_equal(const memory_resource& other) const noexcept;`

4 *Effects:* Equivalent to: `return do_is_equal(other);`

20.12.2.2 Private virtual member functions [mem.res.private]

`virtual void* do_allocate(size_t bytes, size_t alignment) = 0;`

1 ~~*Require:*~~ *Preconditions:* `alignment` ~~shall be~~is a power of two.

2 *Returns:* A derived class shall implement this function to return a pointer to allocated storage (??) with a size of at least `bytes`, aligned to the specified `alignment`.

3 *Throws:* A derived class implementation shall throw an appropriate exception if it is unable to allocate memory with the requested size and alignment.

```
virtual void do_deallocate(void* p, size_t bytes, size_t alignment) = 0;
```

4 ~~*Requires:*~~ *Preconditions:* `p` ~~shall have been~~was returned from a prior call to `allocate(bytes, alignment)` on a memory resource equal to `*this`, and the storage at `p` ~~shall not yet have~~has not yet been deallocated.

5 *Effects:* A derived class shall implement this function to dispose of allocated storage.

6 *Throws:* Nothing.

```
virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;
```

7 *Returns:* A derived class shall implement this function to return `true` if memory allocated from `this` can be deallocated from `other` and vice-versa, otherwise `false`. [*Note*: The most-derived type of `other` might not match the type of `this`. For a derived class `D`, an implementation of this function could immediately return `false` if `dynamic_cast<const D*>(&other) == nullptr`. — *end note*]

20.12.2.3 Equality [mem.res.eq]

```
bool operator==(const memory_resource& a, const memory_resource& b) noexcept;
```

1 *Returns:* `&a == &b || a.is_equal(b)`.

20.12.3 Class template `polymorphic_allocator` [mem.poly.allocator.class]

1 A specialization of class template `pmr::polymorphic_allocator` meets the *Cpp17Allocator* requirements (Table ??). Constructed with different memory resources, different instances of the same specialization of `pmr::polymorphic_allocator` can exhibit entirely different allocation behavior. This runtime polymorphism allows objects that use `polymorphic_allocator` to behave as if they used different allocator types at run time even though they use the same static allocator type.

2 All specializations of class template `pmr::polymorphic_allocator` meet the allocator completeness requirements (??).

```
namespace std::pmr {
  template<class Tp = byte> class polymorphic_allocator {
    memory_resource* memory_rsrc;       // exposition only

  public:
    using value_type = Tp;

    // 20.12.3.1, constructors
    polymorphic_allocator() noexcept;
    polymorphic_allocator(memory_resource* r);

    polymorphic_allocator(const polymorphic_allocator& other) = default;

    template<class U>
      polymorphic_allocator(const polymorphic_allocator<U>& other) noexcept;

    polymorphic_allocator& operator=(const polymorphic_allocator&) = delete;

    // 20.12.3.2, member functions
    [[nodiscard]] Tp* allocate(size_t n);
    void deallocate(Tp* p, size_t n);

    void* allocate_bytes(size_t nbytes, size_t alignment = alignof(max_align_t));
    void deallocate_bytes(void* p, size_t nbytes, size_t alignment = alignof(max_align_t));
    template<class T> T* allocate_object(size_t n = 1);
    template<class T> void deallocate_object(T* p, size_t n = 1);
    template<class T, class... CtorArgs> T* new_object(CtorArgs&&... ctor_args);
    template<class T> void delete_object(T* p);
```

```
    template<class T, class... Args>
      void construct(T* p, Args&&... args);

    template<class T>
      void destroy(T* p);

    polymorphic_allocator select_on_container_copy_construction() const;

    memory_resource* resource() const;
  };
}
```

20.12.3.1 Constructors [mem.poly.allocator.ctor]

```
polymorphic_allocator() noexcept;
```

1 *Effects:* Sets `memory_rsrc` to `get_default_resource()`.

```
polymorphic_allocator(memory_resource* r);
```

2 ~~*Requires:*~~ *Preconditions:* `r` is non-null.

3 *Effects:* Sets `memory_rsrc` to `r`.

4 *Throws:* Nothing.

5 [*Note*: This constructor provides an implicit conversion from `memory_resource*`. — *end note*]

```
template<class U> polymorphic_allocator(const polymorphic_allocator<U>& other) noexcept;
```

6 *Effects:* Sets `memory_rsrc` to `other.resource()`.

20.12.3.2 Member functions [mem.poly.allocator.mem]

```
[[nodiscard]] Tp* allocate(size_t n);
```

1 *Effects:* If `SIZE_MAX / sizeof(Tp) < n`, throws `length_error`. Otherwise equivalent to:

```
return static_cast<Tp*>(memory_rsrc->allocate(n * sizeof(Tp), alignof(Tp)));
```

```
void deallocate(Tp* p, size_t n);
```

2 ~~*Requires:*~~ *Preconditions:* `p` was allocated from a memory resource `x`, equal to `*memory_rsrc`, using `x.allocate(n * sizeof(Tp), alignof(Tp))`.

3 *Effects:* Equivalent to `memory_rsrc->deallocate(p, n * sizeof(Tp), alignof(Tp))`.

4 *Throws:* Nothing.

```
void* allocate_bytes(size_t nbytes, size_t alignment = alignof(max_align_t));
```

5 *Effects:* Equivalent to: `return memory_rsrc->allocate(nbytes, alignment);`

6 [*Note*: The return type is `void*` (rather than, e.g., `byte*`) to support conversion to an arbitrary pointer type `U*` by `static_cast<U*>`, thus facilitating construction of a `U` object in the allocated memory. — *end note*]

```
void deallocate_bytes(void* p, size_t nbytes, size_t alignment = alignof(max_align_t));
```

7 *Effects:* Equivalent to `memory_rsrc->deallocate(p, nbytes, alignment)`.

```
template<class T>
  T* allocate_object(size_t n = 1);
```

8 *Effects:* Allocates memory suitable for holding an array of `n` objects of type `T`, as follows:

(8.1) — if `SIZE_MAX / sizeof(T) < n`, throws `length_error`,

(8.2) — otherwise equivalent to:

```
return static_cast<T*>(allocate_bytes(n*sizeof(T), alignof(T)));
```

9 [*Note*: `T` is not deduced and must therefore be provided as a template argument. — *end note*]

```
template<class T>
  void deallocate_object(T* p, size_t n = 1);
```

10 *Effects:* Equivalent to `deallocate_bytes(p, n*sizeof(T), alignof(T))`.

```
template<class T, class CtorArgs...>
  T* new_object(CtorArgs&&... ctor_args);
```

11 *Effects:* Allocates and constructs an object of type `T`, as follows.
Equivalent to:

```
T* p = allocate_object<T>();
try {
  construct(p, std::forward<CtorArgs>(ctor_args)...);
} catch (...) {
  deallocate_object(p);
  throw;
}
return p;
```

12 [*Note*: `T` is not deduced and must therefore be provided as a template argument. — *end note*]

```
template<class T>
  void delete_object(T* p);
```

13 *Effects:* Equivalent to:

```
destroy(p);
deallocate_object(p);
```

```
template<class T, class... Args>
  void construct(T* p, Args&&... args);
```

14 *Mandates:* Uses-allocator construction of `T` with allocator `*this` (see 20.10.8.2) and constructor arguments `std::forward<Args>(args)...` is well-formed.

15 *Effects:* Construct a `T` object in the storage whose address is represented by `p` by uses-allocator construction with allocator `*this` and constructor arguments `std::forward<Args>(args)...`.

16 *Throws:* Nothing unless the constructor for `T` throws.

```
template<class T>
  void destroy(T* p);
```

17 *Effects:* As if by `p->~T()`.

```
polymorphic_allocator select_on_container_copy_construction() const;
```

18 *Returns:* `polymorphic_allocator()`.

19 [*Note*: The memory resource is not propagated. — *end note*]

```
memory_resource* resource() const;
```

20 *Returns:* `memory_rsrc`.

20.12.3.3 Equality [mem.poly.allocator.eq]

```
template<class T1, class T2>
  bool operator==(const polymorphic_allocator<T1>& a,
                  const polymorphic_allocator<T2>& b) noexcept;
```

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

20.12.4 Access to program-wide `memory_resource` objects [mem.res.global]

```
memory_resource* new_delete_resource() noexcept;
```

1 *Returns:* A pointer to a static-duration object of a type derived from `memory_resource` that can serve as a resource for allocating memory using `::operator new` and `::operator delete`. The same value is returned every time this function is called. For a return value `p` and a memory resource `r`, `p->is_equal(r)` returns `&r == p`.

```
memory_resource* null_memory_resource() noexcept;
```

2 *Returns:* A pointer to a static-duration object of a type derived from `memory_resource` for which `allocate()` always throws `bad_alloc` and for which `deallocate()` has no effect. The same value is returned every time this function is called. For a return value `p` and a memory resource `r`, `p->is_equal(r)` returns `&r == p`.

3 The *default memory resource pointer* is a pointer to a memory resource that is used by certain facilities when an explicit memory resource is not supplied through the interface. Its initial value is the return value of `new_delete_resource()`.

```
memory_resource* set_default_resource(memory_resource* r) noexcept;
```

4 *Effects:* If `r` is non-null, sets the value of the default memory resource pointer to `r`, otherwise sets the default memory resource pointer to `new_delete_resource()`.

5 *Returns:* The previous value of the default memory resource pointer.

6 *Remarks:* Calling the `set_default_resource` and `get_default_resource` functions shall not incur a data race. A call to the `set_default_resource` function shall synchronize with subsequent calls to the `set_default_resource` and `get_default_resource` functions.

```
memory_resource* get_default_resource() noexcept;
```

7 *Returns:* The current value of the default memory resource pointer.

20.12.5 Pool resource classes [mem.res.pool]

20.12.5.1 Classes `synchronized_pool_resource` and `unsynchronized_pool_resource` [mem.res.pool.overview]

1 The `synchronized_pool_resource` and `unsynchronized_pool_resource` classes (collectively called *pool resource classes*) are general-purpose memory resources having the following qualities:

(1.1) — Each resource frees its allocated memory on destruction, even if `deallocate` has not been called for some of the allocated blocks.

(1.2) — A pool resource consists of a collection of *pools*, serving requests for different block sizes. Each individual pool manages a collection of *chunks* that are in turn divided into blocks of uniform size, returned via calls to `do_allocate`. Each call to `do_allocate(size, alignment)` is dispatched to the pool serving the smallest blocks accommodating at least `size` bytes.

(1.3) — When a particular pool is exhausted, allocating a block from that pool results in the allocation of an additional chunk of memory from the *upstream allocator* (supplied at construction), thus replenishing the pool. With each successive replenishment, the chunk size obtained increases geometrically. [*Note*: By allocating memory in chunks, the pooling strategy increases the chance that consecutive allocations will be close together in memory. — *end note*]

(1.4) — Allocation requests that exceed the largest block size of any pool are fulfilled directly from the upstream allocator.

(1.5) — A `pool_options` struct may be passed to the pool resource constructors to tune the largest block size and the maximum chunk size.

2 A `synchronized_pool_resource` may be accessed from multiple threads without external synchronization and may have thread-specific pools to reduce synchronization costs. An `unsynchronized_pool_resource` class may not be accessed from multiple threads simultaneously and thus avoids the cost of synchronization entirely in single-threaded applications.

```
namespace std::pmr {
  struct pool_options {
    size_t max_blocks_per_chunk = 0;
    size_t largest_required_pool_block = 0;
  };

  class synchronized_pool_resource : public memory_resource {
  public:
    synchronized_pool_resource(const pool_options& opts, memory_resource* upstream);
```

```
    synchronized_pool_resource()
        : synchronized_pool_resource(pool_options(), get_default_resource()) {}
    explicit synchronized_pool_resource(memory_resource* upstream)
        : synchronized_pool_resource(pool_options(), upstream) {}
    explicit synchronized_pool_resource(const pool_options& opts)
        : synchronized_pool_resource(opts, get_default_resource()) {}

    synchronized_pool_resource(const synchronized_pool_resource&) = delete;
    virtual ~synchronized_pool_resource();

    synchronized_pool_resource& operator=(const synchronized_pool_resource&) = delete;

    void release();
    memory_resource* upstream_resource() const;
    pool_options options() const;

  protected:
    void* do_allocate(size_t bytes, size_t alignment) override;
    void do_deallocate(void* p, size_t bytes, size_t alignment) override;

    bool do_is_equal(const memory_resource& other) const noexcept override;
  };

  class unsynchronized_pool_resource : public memory_resource {
  public:
    unsynchronized_pool_resource(const pool_options& opts, memory_resource* upstream);

    unsynchronized_pool_resource()
        : unsynchronized_pool_resource(pool_options(), get_default_resource()) {}
    explicit unsynchronized_pool_resource(memory_resource* upstream)
        : unsynchronized_pool_resource(pool_options(), upstream) {}
    explicit unsynchronized_pool_resource(const pool_options& opts)
        : unsynchronized_pool_resource(opts, get_default_resource()) {}

    unsynchronized_pool_resource(const unsynchronized_pool_resource&) = delete;
    virtual ~unsynchronized_pool_resource();

    unsynchronized_pool_resource& operator=(const unsynchronized_pool_resource&) = delete;

    void release();
    memory_resource* upstream_resource() const;
    pool_options options() const;

  protected:
    void* do_allocate(size_t bytes, size_t alignment) override;
    void do_deallocate(void* p, size_t bytes, size_t alignment) override;

    bool do_is_equal(const memory_resource& other) const noexcept override;
  };
}
```

20.12.5.2 `pool_options` data members [mem.res.pool.options]

1 The members of `pool_options` comprise a set of constructor options for pool resources. The effect of each option on the pool resource behavior is described below:

`size_t max_blocks_per_chunk;`

2 The maximum number of blocks that will be allocated at once from the upstream memory resource (20.12.6) to replenish a pool. If the value of `max_blocks_per_chunk` is zero or is greater than an implementation-defined limit, that limit is used instead. The implementation may choose to use a smaller value than is specified in this field and may use different values for different pools.

```
size_t largest_required_pool_block;
```

3 The largest allocation size that is required to be fulfilled using the pooling mechanism. Attempts to allocate a single block larger than this threshold will be allocated directly from the upstream memory resource. If `largest_required_pool_block` is zero or is greater than an implementation-defined limit, that limit is used instead. The implementation may choose a pass-through threshold larger than specified in this field.

20.12.5.3 Constructors and destructors [mem.res.pool.ctor]

```
synchronized_pool_resource(const pool_options& opts, memory_resource* upstream);
unsynchronized_pool_resource(const pool_options& opts, memory_resource* upstream);
```

1 ~~*Requires:*~~ *Preconditions:* `upstream` is the address of a valid memory resource.

2 *Effects:* Constructs a pool resource object that will obtain memory from `upstream` whenever the pool resource is unable to satisfy a memory request from its own internal data structures. The resulting object will hold a copy of `upstream`, but will not own the resource to which `upstream` points. [*Note*: The intention is that calls to `upstream->allocate()` will be substantially fewer than calls to `this->allocate()` in most cases. — *end note*] The behavior of the pooling mechanism is tuned according to the value of the `opts` argument.

3 *Throws:* Nothing unless `upstream->allocate()` throws. It is unspecified if, or under what conditions, this constructor calls `upstream->allocate()`.

```
virtual ~synchronized_pool_resource();
virtual ~unsynchronized_pool_resource();
```

4 *Effects:* Calls `release()`.

20.12.5.4 Members [mem.res.pool.mem]

```
void release();
```

1 *Effects:* Calls `upstream_resource()->deallocate()` as necessary to release all allocated memory. [*Note*: The memory is released back to `upstream_resource()` even if `deallocate` has not been called for some of the allocated blocks. — *end note*]

```
memory_resource* upstream_resource() const;
```

2 *Returns:* The value of the `upstream` argument provided to the constructor of this object.

```
pool_options options() const;
```

3 *Returns:* The options that control the pooling behavior of this resource. The values in the returned struct may differ from those supplied to the pool resource constructor in that values of zero will be replaced with implementation-defined defaults, and sizes may be rounded to unspecified granularity.

```
void* do_allocate(size_t bytes, size_t alignment) override;
```

4 *Returns:* A pointer to allocated storage (**??**) with a size of at least `bytes`. The size and alignment of the allocated memory shall meet the requirements for a class derived from `memory_resource` (20.12.2).

5 *Effects:* If the pool selected for a block of size `bytes` is unable to satisfy the memory request from its own internal data structures, it will call `upstream_resource()->allocate()` to obtain more memory. If `bytes` is larger than that which the largest pool can handle, then memory will be allocated using `upstream_resource()->allocate()`.

6 *Throws:* Nothing unless `upstream_resource()->allocate()` throws.

```
void do_deallocate(void* p, size_t bytes, size_t alignment) override;
```

7 *Effects:* Returns the memory at `p` to the pool. It is unspecified if, or under what circumstances, this operation will result in a call to `upstream_resource()->deallocate()`.

8 *Throws:* Nothing.

```
bool do_is_equal(const memory_resource& other) const noexcept override;
```

9 *Returns:* `this == &other`.

20.12.6 Class `monotonic_buffer_resource` [mem.res.monotonic.buffer]

1 A `monotonic_buffer_resource` is a special-purpose memory resource intended for very fast memory allocations in situations where memory is used to build up a few objects and then is released all at once when the memory resource object is destroyed. It has the following qualities:

(1.1) — A call to `deallocate` has no effect, thus the amount of memory consumed increases monotonically until the resource is destroyed.

(1.2) — The program can supply an initial buffer, which the allocator uses to satisfy memory requests.

(1.3) — When the initial buffer (if any) is exhausted, it obtains additional buffers from an *upstream* memory resource supplied at construction. Each additional buffer is larger than the previous one, following a geometric progression.

(1.4) — It is intended for access from one thread of control at a time. Specifically, calls to `allocate` and `deallocate` do not synchronize with one another.

(1.5) — It frees the allocated memory on destruction, even if `deallocate` has not been called for some of the allocated blocks.

```
namespace std::pmr {
  class monotonic_buffer_resource : public memory_resource {
    memory_resource* upstream_rsrc;      // exposition only
    void* current_buffer;                // exposition only
    size_t next_buffer_size;             // exposition only

  public:
    explicit monotonic_buffer_resource(memory_resource* upstream);
    monotonic_buffer_resource(size_t initial_size, memory_resource* upstream);
    monotonic_buffer_resource(void* buffer, size_t buffer_size, memory_resource* upstream);

    monotonic_buffer_resource()
      : monotonic_buffer_resource(get_default_resource()) {}
    explicit monotonic_buffer_resource(size_t initial_size)
      : monotonic_buffer_resource(initial_size, get_default_resource()) {}
    monotonic_buffer_resource(void* buffer, size_t buffer_size)
      : monotonic_buffer_resource(buffer, buffer_size, get_default_resource()) {}

    monotonic_buffer_resource(const monotonic_buffer_resource&) = delete;

    virtual ~monotonic_buffer_resource();

    monotonic_buffer_resource& operator=(const monotonic_buffer_resource&) = delete;

    void release();
    memory_resource* upstream_resource() const;

  protected:
    void* do_allocate(size_t bytes, size_t alignment) override;
    void do_deallocate(void* p, size_t bytes, size_t alignment) override;

    bool do_is_equal(const memory_resource& other) const noexcept override;
  };
}
```

20.12.6.1 Constructors and destructor [mem.res.monotonic.buffer.ctor]

```
explicit monotonic_buffer_resource(memory_resource* upstream);
monotonic_buffer_resource(size_t initial_size, memory_resource* upstream);
```

1 ~~*Requires:*~~ *Preconditions:* `upstream` ~~shall be~~is the address of a valid memory resource. `initial_size`, if specified, ~~shall be~~is greater than zero.

2 *Effects:* Sets `upstream_rsrc` to `upstream` and `current_buffer` to `nullptr`. If `initial_size` is specified, sets `next_buffer_size` to at least `initial_size`; otherwise sets `next_buffer_size` to an implementation-defined size.

```
monotonic_buffer_resource(void* buffer, size_t buffer_size, memory_resource* upstream);
```

3 ~~*Requires:*~~ *Preconditions:* `upstream` ~~shall be~~is the address of a valid memory resource. `buffer_size` ~~shall be~~is no larger than the number of bytes in `buffer`.

4 *Effects:* Sets `upstream_rsrc` to `upstream`, `current_buffer` to `buffer`, and `next_buffer_size` to `buffer_size` (but not less than 1), then increases `next_buffer_size` by an implementation-defined growth factor (which need not be integral).

```
~monotonic_buffer_resource();
```

5 *Effects:* Calls `release()`.

20.12.6.2 Members [mem.res.monotonic.buffer.mem]

```
void release();
```

1 *Effects:* Calls `upstream_rsrc->deallocate()` as necessary to release all allocated memory.

2 [*Note*: The memory is released back to `upstream_rsrc` even if some blocks that were allocated from `this` have not been deallocated from `this`. — *end note*]

```
memory_resource* upstream_resource() const;
```

3 *Returns:* The value of `upstream_rsrc`.

```
void* do_allocate(size_t bytes, size_t alignment) override;
```

4 *Returns:* A pointer to allocated storage (**??**) with a size of at least `bytes`. The size and alignment of the allocated memory shall meet the requirements for a class derived from `memory_resource` (20.12.2).

5 *Effects:* If the unused space in `current_buffer` can fit a block with the specified `bytes` and `alignment`, then allocate the return block from `current_buffer`; otherwise set `current_buffer` to `upstream_rsrc->allocate(n, m)`, where `n` is not less than `max(bytes, next_buffer_size)` and `m` is not less than `alignment`, and increase `next_buffer_size` by an implementation-defined growth factor (which need not be integral), then allocate the return block from the newly-allocated `current_buffer`.

6 *Throws:* Nothing unless `upstream_rsrc->allocate()` throws.

```
void do_deallocate(void* p, size_t bytes, size_t alignment) override;
```

7 *Effects:* None.

8 *Throws:* Nothing.

9 *Remarks:* Memory used by this resource increases monotonically until its destruction.

```
bool do_is_equal(const memory_resource& other) const noexcept override;
```

10 *Returns:* `this == &other`.

20.13 Class template `scoped_allocator_adaptor` [allocator.adaptor]

20.13.1 Header `<scoped_allocator>` synopsis [allocator.adaptor.syn]

```
namespace std {
  // class template scoped allocator adaptor
  template<class OuterAlloc, class... InnerAlloc>
    class scoped_allocator_adaptor;

  // 20.13.5, scoped allocator operators
  template<class OuterA1, class OuterA2, class... InnerAllocs>
    bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
                    const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
}
```

1 The class template `scoped_allocator_adaptor` is an allocator template that specifies an allocator resource (the outer allocator) to be used by a container (as any other allocator does) and also specifies an inner allocator resource to be passed to the constructor of every element within the container. This adaptor is instantiated with one outer and zero or more inner allocator types. If instantiated with only one allocator type, the inner allocator becomes the `scoped_allocator_adaptor` itself, thus using the same allocator resource for the container and every element within the container and, if the elements themselves are containers, each of their

elements recursively. If instantiated with more than one allocator, the first allocator is the outer allocator for use by the container, the second allocator is passed to the constructors of the container's elements, and, if the elements themselves are containers, the third allocator is passed to the elements' elements, and so on. If containers are nested to a depth greater than the number of allocators, the last allocator is used repeatedly, as in the single-allocator case, for any remaining recursions. [*Note*: The `scoped_allocator_adaptor` is derived from the outer allocator type so it can be substituted for the outer allocator type in most expressions. — *end note*]

```
namespace std {
  template<class OuterAlloc, class... InnerAllocs>
  class scoped_allocator_adaptor : public OuterAlloc {
  private:
    using OuterTraits = allocator_traits<OuterAlloc>;   // exposition only
    scoped_allocator_adaptor<InnerAllocs...> inner;     // exposition only

  public:
    using outer_allocator_type = OuterAlloc;
    using inner_allocator_type = see below;

    using value_type           = typename OuterTraits::value_type;
    using size_type            = typename OuterTraits::size_type;
    using difference_type      = typename OuterTraits::difference_type;
    using pointer              = typename OuterTraits::pointer;
    using const_pointer        = typename OuterTraits::const_pointer;
    using void_pointer         = typename OuterTraits::void_pointer;
    using const_void_pointer   = typename OuterTraits::const_void_pointer;

    using propagate_on_container_copy_assignment = see below;
    using propagate_on_container_move_assignment = see below;
    using propagate_on_container_swap            = see below;
    using is_always_equal                        = see below;

    template<class Tp> struct rebind {
      using other = scoped_allocator_adaptor<
        OuterTraits::template rebind_alloc<Tp>, InnerAllocs...>;
    };

    scoped_allocator_adaptor();
    template<class OuterA2>
      scoped_allocator_adaptor(OuterA2&& outerAlloc,
                               const InnerAllocs&... innerAllocs) noexcept;

    scoped_allocator_adaptor(const scoped_allocator_adaptor& other) noexcept;
    scoped_allocator_adaptor(scoped_allocator_adaptor&& other) noexcept;

    template<class OuterA2>
      scoped_allocator_adaptor(
        const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& other) noexcept;
    template<class OuterA2>
      scoped_allocator_adaptor(
        scoped_allocator_adaptor<OuterA2, InnerAllocs...>&& other) noexcept;

    scoped_allocator_adaptor& operator=(const scoped_allocator_adaptor&) = default;
    scoped_allocator_adaptor& operator=(scoped_allocator_adaptor&&) = default;

    ~scoped_allocator_adaptor();

    inner_allocator_type& inner_allocator() noexcept;
    const inner_allocator_type& inner_allocator() const noexcept;
    outer_allocator_type& outer_allocator() noexcept;
    const outer_allocator_type& outer_allocator() const noexcept;

    [[nodiscard]] pointer allocate(size_type n);
    [[nodiscard]] pointer allocate(size_type n, const_void_pointer hint);
```

```
    void deallocate(pointer p, size_type n);
    size_type max_size() const;

    template<class T, class... Args>
      void construct(T* p, Args&&... args);

    template<class T>
      void destroy(T* p);

    scoped_allocator_adaptor select_on_container_copy_construction() const;
  };

  template<class OuterAlloc, class... InnerAllocs>
    scoped_allocator_adaptor(OuterAlloc, InnerAllocs...)
      -> scoped_allocator_adaptor<OuterAlloc, InnerAllocs...>;
}
```

20.13.2 Member types [allocator.adaptor.types]

`using inner_allocator_type = see below;`

1 *Type:* `scoped_allocator_adaptor<OuterAlloc>` if `sizeof...(InnerAllocs)` is zero; otherwise, `scoped_allocator_adaptor<InnerAllocs...>`.

`using propagate_on_container_copy_assignment = see below;`

2 *Type:* `true_type` if `allocator_traits<A>::propagate_on_container_copy_assignment::value` is `true` for any `A` in the set of `OuterAlloc` and `InnerAllocs...`; otherwise, `false_type`.

`using propagate_on_container_move_assignment = see below;`

3 *Type:* `true_type` if `allocator_traits<A>::propagate_on_container_move_assignment::value` is `true` for any `A` in the set of `OuterAlloc` and `InnerAllocs...`; otherwise, `false_type`.

`using propagate_on_container_swap = see below;`

4 *Type:* `true_type` if `allocator_traits<A>::propagate_on_container_swap::value` is `true` for any `A` in the set of `OuterAlloc` and `InnerAllocs...`; otherwise, `false_type`.

`using is_always_equal = see below;`

5 *Type:* `true_type` if `allocator_traits<A>::is_always_equal::value` is `true` for every `A` in the set of `OuterAlloc` and `InnerAllocs...`; otherwise, `false_type`.

20.13.3 Constructors [allocator.adaptor.cnstr]

`scoped_allocator_adaptor();`

1 *Effects:* Value-initializes the `OuterAlloc` base class and the `inner` allocator object.

```
template<class OuterA2>
  scoped_allocator_adaptor(OuterA2&& outerAlloc, const InnerAllocs&... innerAllocs) noexcept;
```

2 *Effects:* Initializes the `OuterAlloc` base class with `std::forward<OuterA2>(outerAlloc)` and `inner` with `innerAllocs...` (hence recursively initializing each allocator within the adaptor with the corresponding allocator from the argument list).

3 ~~*Remarks:* This constructor shall not participate in overload resolution unless~~ *Constraints:* `is_constructible_v<OuterAlloc, OuterA2>` is `true`.

`scoped_allocator_adaptor(const scoped_allocator_adaptor& other) noexcept;`

4 *Effects:* Initializes each allocator within the adaptor with the corresponding allocator from `other`.

`scoped_allocator_adaptor(scoped_allocator_adaptor&& other) noexcept;`

5 *Effects:* Move constructs each allocator within the adaptor with the corresponding allocator from `other`.

```
template<class OuterA2>
  scoped_allocator_adaptor(
    const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& other) noexcept;
```

6 *Effects:* Initializes each allocator within the adaptor with the corresponding allocator from `other`.

7 ~~*Remarks:* This constructor shall not participate in overload resolution unless~~*Constraints:* `is_constructible_v<OuterAlloc, const OuterA2&>` is `true`.

```
template<class OuterA2>
  scoped_allocator_adaptor(scoped_allocator_adaptor<OuterA2, InnerAllocs...>&& other) noexcept;
```

8 *Effects:* Initializes each allocator within the adaptor with the corresponding allocator rvalue from `other`.

9 ~~*Remarks:* This constructor shall not participate in overload resolution unless~~*Constraints:* `is_constructible_v<OuterAlloc, OuterA2>` is `true`.

20.13.4 Members [allocator.adaptor.members]

1 In the `construct` member functions, *OUTERMOST*`(x)` is *OUTERMOST*`(x.outer_allocator())` if the expression `x.outer_allocator()` is valid (??) and `x` otherwise; *OUTERMOST_ALLOC_TRAITS*`(x)` is `allocator_traits<remove_reference_t<decltype(`*OUTERMOST*`(x))>>`. [*Note*: *OUTERMOST*`(x)` and *OUTERMOST_ALLOC_TRAITS*`(x)` are recursive operations. It is incumbent upon the definition of `outer_allocator()` to ensure that the recursion terminates. It will terminate for all instantiations of `scoped_allocator_adaptor`. — *end note*]

```
inner_allocator_type& inner_allocator() noexcept;
const inner_allocator_type& inner_allocator() const noexcept;
```

2 *Returns:* `*this` if `sizeof...(InnerAllocs)` is zero; otherwise, `inner`.

```
outer_allocator_type& outer_allocator() noexcept;
```

3 *Returns:* `static_cast<OuterAlloc&>(*this)`.

```
const outer_allocator_type& outer_allocator() const noexcept;
```

4 *Returns:* `static_cast<const OuterAlloc&>(*this)`.

```
[[nodiscard]] pointer allocate(size_type n);
```

5 *Returns:* `allocator_traits<OuterAlloc>::allocate(outer_allocator(), n)`.

```
[[nodiscard]] pointer allocate(size_type n, const_void_pointer hint);
```

6 *Returns:* `allocator_traits<OuterAlloc>::allocate(outer_allocator(), n, hint)`.

```
void deallocate(pointer p, size_type n) noexcept;
```

7 *Effects:* As if by: `allocator_traits<OuterAlloc>::deallocate(outer_allocator(), p, n);`

```
size_type max_size() const;
```

8 *Returns:* `allocator_traits<OuterAlloc>::max_size(outer_allocator())`.

```
template<class T, class... Args>
  void construct(T* p, Args&&... args);
```

9 *Effects:* Equivalent to:

```
apply([p, this](auto&&... newargs) {
        OUTERMOST_ALLOC_TRAITS(*this)::construct(
          OUTERMOST(*this), p,
          std::forward<decltype(newargs)>(newargs)...);
      },
      uses_allocator_construction_args<T>(inner_allocator(),
                                          std::forward<Args>(args)...));
```

```
template<class T>
  void destroy(T* p);
```

10 *Effects:* Calls *OUTERMOST_ALLOC_TRAITS*`(*this)::destroy(`*OUTERMOST*`(*this), p)`.

```
scoped_allocator_adaptor select_on_container_copy_construction() const;
```

11 *Returns:* A new `scoped_allocator_adaptor` object where each allocator `A` in the adaptor is initialized from the result of calling `allocator_traits<A>::select_on_container_copy_construction()` on the corresponding allocator in `*this`.

20.13.5 Operators [scoped.adaptor.operators]

```
template<class OuterA1, class OuterA2, class... InnerAllocs>
  bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
                  const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
```

1 *Returns:* If `sizeof...(InnerAllocs)` is zero,

```
a.outer_allocator() == b.outer_allocator()
```

otherwise

```
a.outer_allocator() == b.outer_allocator() && a.inner_allocator() == b.inner_allocator()
```

20.14 Function objects [function.objects]

1 A *function object type* is an object type (**??**) that can be the type of the *postfix-expression* in a function call (**??**, **??**).[221] A *function object* is an object of a function object type. In the places where one would expect to pass a pointer to a function to an algorithmic template (**??**), the interface is specified to accept a function object. This not only makes algorithmic templates work with pointers to functions, but also enables them to work with arbitrary function objects.

20.14.1 Header `<functional>` synopsis [functional.syn]

```
namespace std {
  // 20.14.4, invoke
  template<class F, class... Args>
    constexpr invoke_result_t<F, Args...> invoke(F&& f, Args&&... args)
      noexcept(is_nothrow_invocable_v<F, Args...>);

  // 20.14.5, reference_wrapper
  template<class T> class reference_wrapper;

  template<class T> constexpr reference_wrapper<T> ref(T&) noexcept;
  template<class T> constexpr reference_wrapper<const T> cref(const T&) noexcept;
  template<class T> void ref(const T&&) = delete;
  template<class T> void cref(const T&&) = delete;

  template<class T> constexpr reference_wrapper<T> ref(reference_wrapper<T>) noexcept;
  template<class T> constexpr reference_wrapper<const T> cref(reference_wrapper<T>) noexcept;

  template<class T> struct unwrap_reference;
  template<class T> using unwrap_reference_t = typename unwrap_reference<T>::type;
  template<class T> struct unwrap_ref_decay;
  template<class T> using unwrap_ref_decay_t = typename unwrap_ref_decay<T>::type;

  // 20.14.6, arithmetic operations
  template<class T = void> struct plus;
  template<class T = void> struct minus;
  template<class T = void> struct multiplies;
  template<class T = void> struct divides;
  template<class T = void> struct modulus;
  template<class T = void> struct negate;
  template<> struct plus<void>;
  template<> struct minus<void>;
  template<> struct multiplies<void>;
  template<> struct divides<void>;
  template<> struct modulus<void>;
  template<> struct negate<void>;
```

221) Such a type is a function pointer or a class type which has a member `operator()` or a class type which has a conversion to a pointer to function.

```
// 20.14.7, comparisons
template<class T = void> struct equal_to;
template<class T = void> struct not_equal_to;
template<class T = void> struct greater;
template<class T = void> struct less;
template<class T = void> struct greater_equal;
template<class T = void> struct less_equal;
template<> struct equal_to<void>;
template<> struct not_equal_to<void>;
template<> struct greater<void>;
template<> struct less<void>;
template<> struct greater_equal<void>;
template<> struct less_equal<void>;

// 20.14.9, logical operations
template<class T = void> struct logical_and;
template<class T = void> struct logical_or;
template<class T = void> struct logical_not;
template<> struct logical_and<void>;
template<> struct logical_or<void>;
template<> struct logical_not<void>;

// 20.14.10, bitwise operations
template<class T = void> struct bit_and;
template<class T = void> struct bit_or;
template<class T = void> struct bit_xor;
template<class T = void> struct bit_not;
template<> struct bit_and<void>;
template<> struct bit_or<void>;
template<> struct bit_xor<void>;
template<> struct bit_not<void>;

// 20.14.11, identity
struct identity;

// 20.14.12, function template not_fn
template<class F> constexpr unspecified not_fn(F&& f);

// 20.14.13, function template bind_front
template<class F, class... Args> constexpr unspecified bind_front(F&&, Args&&...);

// 20.14.14, bind
template<class T> struct is_bind_expression;
template<class T> struct is_placeholder;

template<class F, class... BoundArgs>
  constexpr unspecified bind(F&&, BoundArgs&&...);
template<class R, class F, class... BoundArgs>
  constexpr unspecified bind(F&&, BoundArgs&&...);

namespace placeholders {
  // M is the implementation-defined number of placeholders
  see below _1;
  see below _2;
             .
             .
             .
  see below _M;
}

// 20.14.15, member function adaptors
template<class R, class T>
  constexpr unspecified mem_fn(R T::*) noexcept;
```

```
  // 20.14.16, polymorphic function wrappers
  class bad_function_call;

  template<class> class function;        // not defined
  template<class R, class... ArgTypes> class function<R(ArgTypes...)>;

  template<class R, class... ArgTypes>
    void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&) noexcept;

  template<class R, class... ArgTypes>
    bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;

  // 20.14.17, searchers
  template<class ForwardIterator, class BinaryPredicate = equal_to<>>
    class default_searcher;

  template<class RandomAccessIterator,
           class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
           class BinaryPredicate = equal_to<>>
    class boyer_moore_searcher;

  template<class RandomAccessIterator,
           class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
           class BinaryPredicate = equal_to<>>
    class boyer_moore_horspool_searcher;

  // 20.14.18, hash function primary template
  template<class T>
    struct hash;

  // 20.14.14, function object binders
  template<class T>
    inline constexpr bool is_bind_expression_v = is_bind_expression<T>::value;
  template<class T>
    inline constexpr int is_placeholder_v = is_placeholder<T>::value;

  namespace ranges {
    // 20.14.8, concept-constrained comparisons
    struct equal_to;
    struct not_equal_to;
    struct greater;
    struct less;
    struct greater_equal;
    struct less_equal;
  }
}
```

1 [*Example*: If a C++ program wants to have a by-element addition of two vectors `a` and `b` containing `double` and put the result into `a`, it can do:

```
transform(a.begin(), a.end(), b.begin(), a.begin(), plus<double>());
```

— *end example*]

2 [*Example*: To negate every element of `a`:

```
transform(a.begin(), a.end(), a.begin(), negate<double>());
```

— *end example*]

20.14.2 Definitions [func.def]

1 The following definitions apply to this Clause:

2 A *call signature* is the name of a return type followed by a parenthesized comma-separated list of zero or more argument types.

3 A *callable type* is a function object type (20.14) or a pointer to member.

4 A *callable object* is an object of a callable type.

5 A *call wrapper type* is a type that holds a callable object and supports a call operation that forwards to that object.

6 A *call wrapper* is an object of a call wrapper type.

7 A *target object* is the callable object held by a call wrapper.

8 A call wrapper type may additionally hold a sequence of objects and references that may be passed as arguments to the target object. These entities are collectively referred to as *bound argument entities*.

9 The target object and bound argument entities of the call wrapper are collectively referred to as *state entities*.

20.14.3 Requirements [func.require]

1 Define *INVOKE*(f, t_1, t_2, ..., t_N) as follows:

(1.1) — ((t_1.*f)(t_2, ..., t_N) when f is a pointer to a member function of a class T and is_base_of_v<T, remove_reference_t<decltype(t_1)>> is true;

(1.2) — (t_1.get().*f)(t_2, ..., t_N) when f is a pointer to a member function of a class T and remove_cvref_t<decltype(t_1)> is a specialization of reference_wrapper;

(1.3) — ((*t_1).*f)(t_2, ..., t_N) when f is a pointer to a member function of a class T and t_1 does not satisfy the previous two items;

(1.4) — t_1.*f when N == 1 and f is a pointer to data member of a class T and is_base_of_v<T, remove_reference_t<decltype(t_1)>> is true;

(1.5) — t_1.get().*f when N == 1 and f is a pointer to data member of a class T and remove_cvref_t<decltype(t_1)> is a specialization of reference_wrapper;

(1.6) — (*t_1).*f when N == 1 and f is a pointer to data member of a class T and t_1 does not satisfy the previous two items;

(1.7) — f(t_1, t_2, ..., t_N) in all other cases.

2 Define *INVOKE*<R>(f, t_1, t_2, ..., t_N) as static_cast<void>(*INVOKE*(f, t_1, t_2, ..., t_N)) if R is *cv* void, otherwise *INVOKE*(f, t_1, t_2, ..., t_N) implicitly converted to R.

3 Every call wrapper (20.14.2) meets the *Cpp17MoveConstructible* and *Cpp17Destructible* requirements. An *argument forwarding call wrapper* is a call wrapper that can be called with an arbitrary argument list and delivers the arguments to the wrapped callable object as references. This forwarding step delivers rvalue arguments as rvalue references and lvalue arguments as lvalue references. [*Note*: In a typical implementation, argument forwarding call wrappers have an overloaded function call operator of the form

```
template<class... UnBoundArgs>
  constexpr R operator()(UnBoundArgs&&... unbound_args) cv-qual ;
```

— *end note*]

4 A *perfect forwarding call wrapper* is an argument forwarding call wrapper that forwards its state entities to the underlying call expression. This forwarding step delivers a state entity of type T as *cv* T& when the call is performed on an lvalue of the call wrapper type and as *cv* T&& otherwise, where *cv* represents the cv-qualifiers of the call wrapper and where *cv* shall be neither volatile nor const volatile.

5 A *call pattern* defines the semantics of invoking a perfect forwarding call wrapper. A postfix call performed on a perfect forwarding call wrapper is expression-equivalent (**??**) to an expression e determined from its call pattern cp by replacing all occurrences of the arguments of the call wrapper and its state entities with references as described in the corresponding forwarding steps.

6 A *simple call wrapper* is a perfect forwarding call wrapper that meets the *Cpp17CopyConstructible* and *Cpp17CopyAssignable* requirements and whose copy constructor, move constructor, and assignment operators are constexpr functions that do not throw exceptions.

7 The copy/move constructor of an argument forwarding call wrapper has the same apparent semantics as if memberwise copy/move of its state entities were performed (**??**). [*Note*: This implies that each of the copy/move constructors has the same exception-specification as the corresponding implicit definition and is declared as constexpr if the corresponding implicit definition would be considered to be constexpr. — *end note*]

8 Argument forwarding call wrappers returned by a given standard library function template have the same type if the types of their corresponding state entities are the same.

20.14.4 Function template `invoke` [func.invoke]

```
template<class F, class... Args>
  constexpr invoke_result_t<F, Args...> invoke(F&& f, Args&&... args)
    noexcept(is_nothrow_invocable_v<F, Args...>);
```

1 *Returns:* *INVOKE*(std::forward<F>(f), std::forward<Args>(args)...) (20.14.3).

20.14.5 Class template `reference_wrapper` [refwrap]

```
namespace std {
  template<class T> class reference_wrapper {
  public:
    // types
    using type = T;

    // construct/copy/destroy
    template<class U>
      constexpr reference_wrapper(U&&) noexcept(see below);
    constexpr reference_wrapper(const reference_wrapper& x) noexcept;

    // assignment
    constexpr reference_wrapper& operator=(const reference_wrapper& x) noexcept;

    // access
    constexpr operator T& () const noexcept;
    constexpr T& get() const noexcept;

    // invocation
    template<class... ArgTypes>
      constexpr invoke_result_t<T&, ArgTypes...> operator()(ArgTypes&&...) const;
  };
  template<class T>
    reference_wrapper(T&) -> reference_wrapper<T>;
}
```

1 `reference_wrapper<T>` is a *Cpp17CopyConstructible* and *Cpp17CopyAssignable* wrapper around a reference to an object or function of type `T`.

2 `reference_wrapper<T>` is a trivially copyable type (**??**).

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

20.14.5.1 Constructors and destructor [refwrap.const]

```
template<class U>
  constexpr reference_wrapper(U&& u) noexcept(see below);
```

Let *FUN* denote the exposition-only functions

```
void FUN(T&) noexcept;
void FUN(T&&) = delete;
```

1 *Constraints:* The expression *FUN*(declval<U>()) is well-formed and is_same_v<remove_cvref_t<U>, reference_wrapper> is false.

2 *Remarks:* Let *FUN* denote the exposition-only functions

```
void FUN(T&) noexcept;
void FUN(T&&) = delete;
```

This constructor shall not participate in overload resolution unless the expression *FUN*(declval<U>()) is well-formed and is_same_v<remove_cvref_t<U>, reference_wrapper> is false. The expression inside noexcept is equivalent to noexcept(*FUN*(declval<U>())).

3 *Effects:* Creates a variable `r` as if by `T& r = std::forward<U>(u)`, then constructs a `reference_wrapper` object that stores a reference to `r`.

4 *Remarks:* The expression inside `noexcept` is equivalent to `noexcept(`*`FUN`*`(declval<U>()))`.

```
constexpr reference_wrapper(const reference_wrapper& x) noexcept;
```

5 *Effects:* Constructs a `reference_wrapper` object that stores a reference to `x.get()`.

20.14.5.2 Assignment [refwrap.assign]

```
constexpr reference_wrapper& operator=(const reference_wrapper& x) noexcept;
```

1 *Postconditions:* `*this` stores a reference to `x.get()`.

20.14.5.3 Access [refwrap.access]

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

1 *Returns:* The stored reference.

```
constexpr T& get() const noexcept;
```

2 *Returns:* The stored reference.

20.14.5.4 Invocation [refwrap.invoke]

```
template<class... ArgTypes>
  constexpr invoke_result_t<T&, ArgTypes...>
    operator()(ArgTypes&&... args) const;
```

1 *Mandates:* `T` is a complete type.

2 *Returns:* *`INVOKE`*`(get(), std::forward<ArgTypes>(args)...)`. (20.14.3)

20.14.5.5 Helper functions [refwrap.helpers]

1 The template parameter `T` of the following `ref` and `cref` function templates may be an incomplete type.

```
template<class T> constexpr reference_wrapper<T> ref(T& t) noexcept;
```

2 *Returns:* `reference_wrapper<T>(t)`.

```
template<class T> constexpr reference_wrapper<T> ref(reference_wrapper<T> t) noexcept;
```

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

```
template<class T> constexpr reference_wrapper<const T> cref(const T& t) noexcept;
```

4 *Returns:* `reference_wrapper <const T>(t)`.

```
template<class T> constexpr reference_wrapper<const T> cref(reference_wrapper<T> t) noexcept;
```

5 *Returns:* `cref(t.get())`.

20.14.5.6 Transformation type trait `unwrap_reference` [refwrap.unwrapref]

```
template<class T>
  struct unwrap_reference;
```

1 If `T` is a specialization `reference_wrapper<X>` for some type `X`, the member typedef `type` of `unwrap_reference<T>` is `X&`, otherwise it is `T`.

```
template<class T>
  struct unwrap_ref_decay;
```

2 The member typedef `type` of `unwrap_ref_decay<T>` denotes the type `unwrap_reference_t<decay_t<T>>`.

20.14.6 Arithmetic operations [arithmetic.operations]

1 The library provides basic function object classes for all of the arithmetic operators in the language (**??**, **??**).

20.14.6.1 Class template `plus` [arithmetic.operations.plus]

```
template<class T = void> struct plus {
  constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;
```

1 *Returns:* `x + y`.

```
template<> struct plus<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) + std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) + std::forward<U>(u));
```

2 *Returns:* `std::forward<T>(t) + std::forward<U>(u)`.

20.14.6.2 Class template `minus` [arithmetic.operations.minus]

```
template<class T = void> struct minus {
  constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;
```

1 *Returns:* `x - y`.

```
template<> struct minus<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) - std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) - std::forward<U>(u));
```

2 *Returns:* `std::forward<T>(t) - std::forward<U>(u)`.

20.14.6.3 Class template `multiplies` [arithmetic.operations.multiplies]

```
template<class T = void> struct multiplies {
  constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;
```

1 *Returns:* `x * y`.

```
template<> struct multiplies<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) * std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) * std::forward<U>(u));
```

2 *Returns:* `std::forward<T>(t) * std::forward<U>(u)`.

20.14.6.4 Class template `divides` [arithmetic.operations.divides]

```
template<class T = void> struct divides {
  constexpr T operator()(const T& x, const T& y) const;
```

```
};

constexpr T operator()(const T& x, const T& y) const;
```

1 *Returns:* `x / y`.

```
template<> struct divides<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) / std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) / std::forward<U>(u));
```

2 *Returns:* `std::forward<T>(t) / std::forward<U>(u)`.

20.14.6.5 Class template `modulus` [arithmetic.operations.modulus]

```
template<class T = void> struct modulus {
  constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;
```

1 *Returns:* `x % y`.

```
template<> struct modulus<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) % std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) % std::forward<U>(u));
```

2 *Returns:* `std::forward<T>(t) % std::forward<U>(u)`.

20.14.6.6 Class template `negate` [arithmetic.operations.negate]

```
template<class T = void> struct negate {
  constexpr T operator()(const T& x) const;
};

constexpr T operator()(const T& x) const;
```

1 *Returns:* `-x`.

```
template<> struct negate<void> {
  template<class T> constexpr auto operator()(T&& t) const
    -> decltype(-std::forward<T>(t));

  using is_transparent = unspecified;
};

template<class T> constexpr auto operator()(T&& t) const
    -> decltype(-std::forward<T>(t));
```

2 *Returns:* `-std::forward<T>(t)`.

20.14.7 Comparisons [comparisons]

1 The library provides basic function object classes for all of the comparison operators in the language (**??**, **??**).

2 For templates `less`, `greater`, `less_equal`, and `greater_equal`, the specializations for any pointer type yield a result consistent with the implementation-defined strict total order over pointers (**??**). [*Note*: If `a < b` is well-defined for pointers `a` and `b` of type `P`, then `(a < b) == less<P>()(a, b)`, `(a > b) == greater<P>()(a, b)`, and so forth. — *end note*] For template specializations `less<void>`, `greater<void>`, `less_equal<void>`, and `greater_equal<void>`, if the call operator calls a built-in operator comparing

pointers, the call operator yields a result consistent with the implementation-defined strict total order over pointers.

20.14.7.1 Class template `equal_to` [comparisons.equal.to]

```
template<class T = void> struct equal_to {
  constexpr bool operator()(const T& x, const T& y) const;
};

constexpr bool operator()(const T& x, const T& y) const;
```

1 *Returns:* `x == y`.

```
template<> struct equal_to<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) == std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) == std::forward<U>(u));
```

2 *Returns:* `std::forward<T>(t) == std::forward<U>(u)`.

20.14.7.2 Class template `not_equal_to` [comparisons.not.equal.to]

```
template<class T = void> struct not_equal_to {
  constexpr bool operator()(const T& x, const T& y) const;
};

constexpr bool operator()(const T& x, const T& y) const;
```

1 *Returns:* `x != y`.

```
template<> struct not_equal_to<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) != std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) != std::forward<U>(u));
```

2 *Returns:* `std::forward<T>(t) != std::forward<U>(u)`.

20.14.7.3 Class template `greater` [comparisons.greater]

```
template<class T = void> struct greater {
  constexpr bool operator()(const T& x, const T& y) const;
};

constexpr bool operator()(const T& x, const T& y) const;
```

1 *Returns:* `x > y`.

```
template<> struct greater<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) > std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) > std::forward<U>(u));
```

2 *Returns:* `std::forward<T>(t) > std::forward<U>(u)`.

20.14.7.4 Class template `less` [comparisons.less]

```
template<class T = void> struct less {
  constexpr bool operator()(const T& x, const T& y) const;
};

constexpr bool operator()(const T& x, const T& y) const;
```

1 *Returns:* `x < y`.

```
template<> struct less<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) < std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) < std::forward<U>(u));
```

2 *Returns:* `std::forward<T>(t) < std::forward<U>(u)`.

20.14.7.5 Class template `greater_equal` [comparisons.greater.equal]

```
template<class T = void> struct greater_equal {
  constexpr bool operator()(const T& x, const T& y) const;
};

constexpr bool operator()(const T& x, const T& y) const;
```

1 *Returns:* `x >= y`.

```
template<> struct greater_equal<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) >= std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) >= std::forward<U>(u));
```

2 *Returns:* `std::forward<T>(t) >= std::forward<U>(u)`.

20.14.7.6 Class template `less_equal` [comparisons.less.equal]

```
template<class T = void> struct less_equal {
  constexpr bool operator()(const T& x, const T& y) const;
};

constexpr bool operator()(const T& x, const T& y) const;
```

1 *Returns:* `x <= y`.

```
template<> struct less_equal<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) <= std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) <= std::forward<U>(u));
```

2 *Returns:* `std::forward<T>(t) <= std::forward<U>(u)`.

20.14.8 Concept-constrained comparisons [range.cmp]

1 In this subclause, *BUILTIN-PTR-CMP*(`T`, *op*, `U`) for types `T` and `U` and where *op* is an equality (**??**) or relational operator (**??**) is a boolean constant expression. *BUILTIN-PTR-CMP*(`T`, *op*, `U`) is `true` if and only if *op* in the expression `declval<T>()` *op* `declval<U>()` resolves to a built-in operator comparing pointers.

```
struct ranges::equal_to {
  template<class T, class U>
    requires equality_comparable_with<T, U> || BUILTIN-PTR-CMP(T, ==, U)
  constexpr bool operator()(T&& t, U&& u) const;

  using is_transparent = unspecified;
};
```

2 ~~*Preconditions:*~~ *Remarks:* If the expression `std::forward<T>(t) == std::forward<U>(u)` results in a call to a built-in operator `==` comparing pointers of type `P`, the conversion sequences from both `T` and `U` to `P` shall be equality-preserving (**??**).

3 *Effects:*

(3.1) — If the expression `std::forward<T>(t) == std::forward<U>(u)` results in a call to a built-in operator `==` comparing pointers: returns `false` if either (the converted value of) `t` precedes `u` or `u` precedes `t` in the implementation-defined strict total order over pointers (**??**) and otherwise `true`.

(3.2) — Otherwise, equivalent to: `return std::forward<T>(t) == std::forward<U>(u);`

```
struct ranges::not_equal_to {
  template<class T, class U>
    requires equality_comparable_with<T, U> || BUILTIN-PTR-CMP(T, ==, U)
  constexpr bool operator()(T&& t, U&& u) const;

  using is_transparent = unspecified;
};
```

4 `operator()` has effects equivalent to:

```
return !ranges::equal_to{}(std::forward<T>(t), std::forward<U>(u));
```

```
struct ranges::greater {
  template<class T, class U>
    requires totally_ordered_with<T, U> || BUILTIN-PTR-CMP(U, <, T)
  constexpr bool operator()(T&& t, U&& u) const;

  using is_transparent = unspecified;
};
```

5 `operator()` has effects equivalent to:

```
return ranges::less{}(std::forward<U>(u), std::forward<T>(t));
```

```
struct ranges::less {
  template<class T, class U>
    requires totally_ordered_with<T, U> || BUILTIN-PTR-CMP(T, <, U)
  constexpr bool operator()(T&& t, U&& u) const;

  using is_transparent = unspecified;
};
```

6 *Preconditions:* If the expression `std::forward<T>(t) < std::forward<U>(u)` results in a call to a built-in operator `<` comparing pointers of type `P`, the conversion sequences from both `T` and `U` to `P` shall be equality-preserving (**??**). For any expressions `ET` and `EU` such that `decltype((ET))` is `T` and `decltype((EU))` is `U`, exactly one of `ranges::less{}(ET, EU)`, `ranges::less{}(EU, ET)`, or `ranges::equal_to{}(ET, EU)` shall be `true`.

7 *Effects:*

(7.1) — If the expression `std::forward<T>(t) < std::forward<U>(u)` results in a call to a built-in operator `<` comparing pointers: returns `true` if (the converted value of) `t` precedes `u` in the implementation-defined strict total order over pointers (**??**) and otherwise `false`.

(7.2) — Otherwise, equivalent to: `return std::forward<T>(t) < std::forward<U>(u);`

```
struct ranges::greater_equal {
  template<class T, class U>
    requires totally_ordered_with<T, U> || BUILTIN-PTR-CMP(T, <, U)
  constexpr bool operator()(T&& t, U&& u) const;
```

```
  using is_transparent = unspecified;
};
```

8 `operator()` has effects equivalent to:

```
return !ranges::less{}(std::forward<T>(t), std::forward<U>(u));
```

```
struct ranges::less_equal {
  template<class T, class U>
    requires totally_ordered_with<T, U> || BUILTIN-PTR-CMP(U, <, T)
  constexpr bool operator()(T&& t, U&& u) const;

  using is_transparent = unspecified;
};
```

9 `operator()` has effects equivalent to:

```
return !ranges::less{}(std::forward<U>(u), std::forward<T>(t));
```

20.14.9 Logical operations [logical.operations]

1 The library provides basic function object classes for all of the logical operators in the language (??, ??, ??).

20.14.9.1 Class template `logical_and` [logical.operations.and]

```
template<class T = void> struct logical_and {
  constexpr bool operator()(const T& x, const T& y) const;
};

constexpr bool operator()(const T& x, const T& y) const;
```

1 *Returns:* `x && y`.

```
template<> struct logical_and<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) && std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) && std::forward<U>(u));
```

2 *Returns:* `std::forward<T>(t) && std::forward<U>(u)`.

20.14.9.2 Class template `logical_or` [logical.operations.or]

```
template<class T = void> struct logical_or {
  constexpr bool operator()(const T& x, const T& y) const;
};

constexpr bool operator()(const T& x, const T& y) const;
```

1 *Returns:* `x || y`.

```
template<> struct logical_or<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) || std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) || std::forward<U>(u));
```

2 *Returns:* `std::forward<T>(t) || std::forward<U>(u)`.

20.14.9.3 Class template `logical_not` [logical.operations.not]

```
template<class T = void> struct logical_not {
  constexpr bool operator()(const T& x) const;
};

constexpr bool operator()(const T& x) const;
```

1 *Returns:* `!x`.

```
template<> struct logical_not<void> {
  template<class T> constexpr auto operator()(T&& t) const
    -> decltype(!std::forward<T>(t));

  using is_transparent = unspecified;
};

template<class T> constexpr auto operator()(T&& t) const
    -> decltype(!std::forward<T>(t));
```

2 *Returns:* `!std::forward<T>(t)`.

20.14.10 Bitwise operations [bitwise.operations]

1 The library provides basic function object classes for all of the bitwise operators in the language (**??**, **??**, **??**, **??**).

20.14.10.1 Class template `bit_and` [bitwise.operations.and]

```
template<class T = void> struct bit_and {
  constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;
```

1 *Returns:* `x & y`.

```
template<> struct bit_and<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) & std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) & std::forward<U>(u));
```

2 *Returns:* `std::forward<T>(t) & std::forward<U>(u)`.

20.14.10.2 Class template `bit_or` [bitwise.operations.or]

```
template<class T = void> struct bit_or {
  constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;
```

1 *Returns:* `x | y`.

```
template<> struct bit_or<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) | std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) | std::forward<U>(u));
```

2 *Returns:* `std::forward<T>(t) | std::forward<U>(u)`.

20.14.10.3 Class template `bit_xor` [bitwise.operations.xor]

```
template<class T = void> struct bit_xor {
  constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;
```

1 *Returns:* `x ^ y`.

```
template<> struct bit_xor<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) ^ std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) ^ std::forward<U>(u));
```

2 *Returns:* `std::forward<T>(t) ^ std::forward<U>(u)`.

20.14.10.4 Class template `bit_not` [bitwise.operations.not]

```
template<class T = void> struct bit_not {
  constexpr T operator()(const T& x) const;
};

constexpr T operator()(const T& x) const;
```

1 *Returns:* `~x`.

```
template<> struct bit_not<void> {
  template<class T> constexpr auto operator()(T&& t) const
    -> decltype(~std::forward<T>(t));

  using is_transparent = unspecified;
};

template<class T> constexpr auto operator()(T&&) const
    -> decltype(~std::forward<T>(t));
```

2 *Returns:* `~std::forward<T>(t)`.

20.14.11 Class `identity` [func.identity]

```
struct identity {
  template<class T>
    constexpr T&& operator()(T&& t) const noexcept;

  using is_transparent = unspecified;
};

template<class T>
  constexpr T&& operator()(T&& t) const noexcept;
```

1 *Effects:* Equivalent to: `return std::forward<T>(t);`

20.14.12 Function template `not_fn` [func.not.fn]

```
template<class F> constexpr unspecified not_fn(F&& f);
```

1 In the text that follows:

(1.1) — `g` is a value of the result of a `not_fn` invocation,

(1.2) — `FD` is the type `decay_t<F>`,

(1.3) — `fd` is the target object of `g` (20.14.2) of type `FD`, direct-non-list-initialized with `std::forward<F>(f)`,

(1.4) — `call_args` is an argument pack used in a function call expression (**??**) of `g`.

2 *Mandates:* `is_constructible_v<FD, F> && is_move_constructible_v<FD>` is true.

3 *Preconditions:* `FD` meets the *Cpp17MoveConstructible* requirements.

4 *Returns:* A perfect forwarding call wrapper `g` with call pattern `!invoke(fd, call_args...)`.

5 *Throws:* Any exception thrown by the initialization of `fd`.

20.14.13 Function template `bind_front` [func.bind.front]

```
template<class F, class... Args>
  constexpr unspecified bind_front(F&& f, Args&&... args);
```

1 In the text that follows:

(1.1) — `g` is a value of the result of a `bind_front` invocation,

(1.2) — `FD` is the type `decay_t<F>`,

(1.3) — `fd` is the target object of `g` (20.14.2) of type `FD`, direct-non-list-initialized with `std::forward<F>(f)`,

(1.4) — `BoundArgs` is a pack that denotes `decay_t<Args>...`,

(1.5) — `bound_args` is a pack of bound argument entities of `g` (20.14.2) of types `BoundArgs...`, direct-non-list-initialized with `std::forward<Args>(args)...`, respectively, and

(1.6) — `call_args` is an argument pack used in a function call expression (??) of `g`.

2 *Mandates:*

```
is_constructible_v<FD, F> &&
is_move_constructible_v<FD> &&
(is_constructible_v<BoundArgs, Args> && ...) &&
(is_move_constructible_v<BoundArgs> && ...)
```

is true.

3 *Preconditions:* `FD` meets the *Cpp17MoveConstructible* requirements. For each $\texttt{T}_i$ in `BoundArgs`, if $\texttt{T}_i$ is an object type, $\texttt{T}_i$ meets the *Cpp17MoveConstructible* requirements.

4 *Returns:* A perfect forwarding call wrapper `g` with call pattern `invoke(fd, bound_args..., call_args...)`.

5 *Throws:* Any exception thrown by the initialization of the state entities of `g` (20.14.2).

20.14.14 Function object binders [func.bind]

1 This subclause describes a uniform mechanism for binding arguments of callable objects.

20.14.14.1 Class template `is_bind_expression` [func.bind.isbind]

```
namespace std {
  template<class T> struct is_bind_expression;  // see below
}
```

1 The class template `is_bind_expression` can be used to detect function objects generated by `bind`. The function template `bind` uses `is_bind_expression` to detect subexpressions.

2 Specializations of the `is_bind_expression` template shall meet the *Cpp17UnaryTypeTrait* requirements (20.15.1). The implementation provides a definition that has a base characteristic of `true_type` if `T` is a type returned from `bind`, otherwise it has a base characteristic of `false_type`. A program may specialize this template for a program-defined type `T` to have a base characteristic of `true_type` to indicate that `T` should be treated as a subexpression in a `bind` call.

20.14.14.2 Class template `is_placeholder` [func.bind.isplace]

```
namespace std {
  template<class T> struct is_placeholder;      // see below
}
```

1 The class template `is_placeholder` can be used to detect the standard placeholders `_1`, `_2`, and so on. The function template `bind` uses `is_placeholder` to detect placeholders.

2 Specializations of the `is_placeholder` template shall meet the *Cpp17UnaryTypeTrait* requirements (20.15.1). The implementation provides a definition that has the base characteristic of `integral_constant<int, J>` if

`T` is the type of `std::placeholders::_J`, otherwise it has a base characteristic of `integral_constant<int, 0>`. A program may specialize this template for a program-defined type `T` to have a base characteristic of `integral_constant<int, N>` with `N > 0` to indicate that `T` should be treated as a placeholder type.

20.14.14.3 Function template `bind` [func.bind.bind]

1 In the text that follows:

(1.1) — `g` is a value of the result of a `bind` invocation,

(1.2) — `FD` is the type `decay_t<F>`,

(1.3) — `fd` is an lvalue that is a target object of `g` (20.14.2) of type `FD` direct-non-list-initialized with `std::forward<F>(f)`,

(1.4) — $\texttt{T}_i$ is the i^{th} type in the template parameter pack `BoundArgs`,

(1.5) — $\texttt{TD}_i$ is the type `decay_t<`$\texttt{T}_i$`>`,

(1.6) — $\texttt{t}_i$ is the i^{th} argument in the function parameter pack `bound_args`,

(1.7) — $\texttt{td}_i$ is a bound argument entity of `g` (20.14.2) of type $\texttt{TD}_i$ direct-non-list-initialized with `std::forward<`$\texttt{T}_i$`>(`$\texttt{t}_i$`)`,

(1.8) — $\texttt{U}_j$ is the j^{th} deduced type of the `UnBoundArgs&&...` parameter of the argument forwarding call wrapper, and

(1.9) — $\texttt{u}_j$ is the j^{th} argument associated with $\texttt{U}_j$.

```
template<class F, class... BoundArgs>
  constexpr unspecified bind(F&& f, BoundArgs&&... bound_args);
template<class R, class F, class... BoundArgs>
  constexpr unspecified bind(F&& f, BoundArgs&&... bound_args);
```

2 *Mandates:* `is_constructible_v<FD, F>` is `true`. For each $\texttt{T}_i$ in `BoundArgs`, `is_constructible_v<`$\texttt{TD}_i$`, `$\texttt{T}_i$`>` is `true`.

3 *Preconditions:* `FD` and each $\texttt{TD}_i$ meet the *Cpp17MoveConstructible* and *Cpp17Destructible* requirements. *INVOKE*`(fd, `$\texttt{w}_1$`, `$\texttt{w}_2$`, ..., `$\texttt{w}_N$`)` (20.14.3) is a valid expression for some values $\texttt{w}_1$, $\texttt{w}_2$, ..., $\texttt{w}_N$, where N has the value `sizeof...(bound_args)`.

4 *Returns:* An argument forwarding call wrapper `g` (20.14.3). A program that attempts to invoke a volatile-qualified `g` is ill-formed. When `g` is not volatile-qualified, invocation of `g(`$\texttt{u}_1$`, `$\texttt{u}_2$`, ..., `$\texttt{u}_M$`)` is expression-equivalent (**??**) to

```
INVOKE(static_cast<Vfd>(vfd),
       static_cast<V1>(v1), static_cast<V2>(v2), ..., static_cast<VN>(vN))
```

for the first overload, and

```
INVOKE<R>(static_cast<Vfd>(vfd),
          static_cast<V1>(v1), static_cast<V2>(v2), ..., static_cast<VN>(vN))
```

for the second overload, where the values and types of the target argument $\texttt{v}_{\texttt{fd}}$ and of the bound arguments $\texttt{v}_1$, $\texttt{v}_2$, ..., $\texttt{v}_N$ are determined as specified below.

5 *Throws:* Any exception thrown by the initialization of the state entities of `g`.

6 [*Note*: If all of `FD` and $\texttt{TD}_i$ meet the requirements of *Cpp17CopyConstructible*, then the return type meets the requirements of *Cpp17CopyConstructible*. — *end note*]

7 The values of the *bound arguments* $\texttt{v}_1$, $\texttt{v}_2$, ..., $\texttt{v}_N$ and their corresponding types $\texttt{V}_1$, $\texttt{V}_2$, ..., $\texttt{V}_N$ depend on the types $\texttt{TD}_i$ derived from the call to `bind` and the cv-qualifiers *cv* of the call wrapper `g` as follows:

(7.1) — if $\texttt{TD}_i$ is `reference_wrapper<T>`, the argument is $\texttt{td}_i$`.get()` and its type $\texttt{V}_i$ is `T&`;

(7.2) — if the value of `is_bind_expression_v<`$\texttt{TD}_i$`>` is `true`, the argument is

```
static_cast<cv TDi&>(tdi)(std::forward<Uj>(uj)...)
```

and its type $\texttt{V}_i$ is `invoke_result_t<`*cv* $\texttt{TD}_i$`&, `$\texttt{U}_j$`...>&&`;

(7.3) — if the value `j` of `is_placeholder_v<`$\texttt{TD}_i$`>` is not zero, the argument is `std::forward<`$\texttt{U}_j$`>(`$\texttt{u}_j$`)` and its type $\texttt{V}_i$ is $\texttt{U}_j$`&&`;

(7.4) — otherwise, the value is $\texttt{td}_i$ and its type $\texttt{V}_i$ is *cv* $\texttt{TD}_i$`&`.

8 The value of the target argument v_{fd} is `fd` and its corresponding type V_{fd} is *cv* `FD&`.

20.14.14.4 Placeholders [func.bind.place]

```
namespace std::placeholders {
  // M is the implementation-defined number of placeholders
  see below _1;
  see below _2;
              .
              .
              .
  see below _M;
}
```

1 All placeholder types meet the *Cpp17DefaultConstructible* and *Cpp17CopyConstructible* requirements, and their default constructors and copy/move constructors are constexpr functions that do not throw exceptions. It is implementation-defined whether placeholder types meet the *Cpp17CopyAssignable* requirements, but if so, their copy assignment operators are constexpr functions that do not throw exceptions.

2 Placeholders should be defined as:

```
inline constexpr unspecified _1{};
```

If they are not, they shall be declared as:

```
extern unspecified _1;
```

20.14.15 Function template `mem_fn` [func.memfn]

```
template<class R, class T> constexpr unspecified mem_fn(R T::* pm) noexcept;
```

1 *Returns:* A simple call wrapper (20.14.2) `fn` with call pattern `invoke(pmd, call_args...)`, where `pmd` is the target object of `fn` of type `R T::*` direct-non-list-initialized with `pm`, and `call_args` is an argument pack used in a function call expression (**??**) of `pm`.

20.14.16 Polymorphic function wrappers [func.wrap]

1 This subclause describes a polymorphic wrapper class that encapsulates arbitrary callable objects.

20.14.16.1 Class `bad_function_call` [func.wrap.badcall]

1 An exception of type `bad_function_call` is thrown by `function::operator()` (20.14.16.2.4) when the function wrapper object has no target.

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

2 *Returns:* An implementation-defined NTBS.

20.14.16.2 Class template `function` [func.wrap.func]

```
namespace std {
  template<class> class function;        // not defined

  template<class R, class... ArgTypes>
  class function<R(ArgTypes...)> {
  public:
    using result_type = R;

    // 20.14.16.2.1, construct/copy/destroy
    function() noexcept;
    function(nullptr_t) noexcept;
    function(const function&);
    function(function&&) noexcept;
    template<class F> function(F);
```

```
    function& operator=(const function&);
    function& operator=(function&&);
    function& operator=(nullptr_t) noexcept;
    template<class F> function& operator=(F&&);
    template<class F> function& operator=(reference_wrapper<F>) noexcept;

    ~function();

    // 20.14.16.2.2, function modifiers
    void swap(function&) noexcept;

    // 20.14.16.2.3, function capacity
    explicit operator bool() const noexcept;

    // 20.14.16.2.4, function invocation
    R operator()(ArgTypes...) const;

    // 20.14.16.2.5, function target access
    const type_info& target_type() const noexcept;
    template<class T>       T* target() noexcept;
    template<class T> const T* target() const noexcept;
  };

  template<class R, class... ArgTypes>
    function(R(*)(ArgTypes...)) -> function<R(ArgTypes...)>;

  template<class F> function(F) -> function<see below>;

  // 20.14.16.2.6, null pointer comparison functions
  template<class R, class... ArgTypes>
    bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;

  // 20.14.16.2.7, specialized algorithms
  template<class R, class... ArgTypes>
    void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&) noexcept;
}
```

1 The `function` class template provides polymorphic wrappers that generalize the notion of a function pointer. Wrappers can store, copy, and call arbitrary callable objects (20.14.2), given a call signature (20.14.2), allowing functions to be first-class objects.

2 A callable type (20.14.2) `F` is *Lvalue-Callable* for argument types `ArgTypes` and return type `R` if the expression *INVOKE*`<R>(declval<F&>(), declval<ArgTypes>()...)`, considered as an unevaluated operand (**??**), is well-formed (20.14.3).

3 The `function` class template is a call wrapper (20.14.2) whose call signature (20.14.2) is `R(ArgTypes...)`.

4 [*Note*: The types deduced by the deduction guides for `function` may change in future versions of this International Standard. — *end note*]

20.14.16.2.1 Constructors and destructor [func.wrap.func.con]

```
function() noexcept;
```

1 *Postconditions:* `!*this`.

```
function(nullptr_t) noexcept;
```

2 *Postconditions:* `!*this`.

```
function(const function& f);
```

3 *Postconditions:* `!*this` if `!f`; otherwise, `*this` targets a copy of `f.target()`.

4 *Throws:* Nothing if `f`'s target is a specialization of `reference_wrapper` or a function pointer. Otherwise, may throw `bad_alloc` or any exception thrown by the copy constructor of the stored callable object. [*Note*: Implementations should avoid the use of dynamically allocated memory for small callable objects,

for example, where `f`'s target is an object holding only a pointer or reference to an object and a member function pointer. — *end note*]

```
function(function&& f) noexcept;
```

Postconditions: If `!f`, `*this` has no target; otherwise, the target of `*this` is equivalent to the target of `f` before the construction, and `f` is in a valid state with an unspecified value.

[*Note*: Implementations should avoid the use of dynamically allocated memory for small callable objects, for example, where `f`'s target is an object holding only a pointer or reference to an object and a member function pointer. — *end note*]

```
template<class F> function(F f);
```

~~*Requires:*~~ *Preconditions:* `F` ~~shall be~~meets the *Cpp17CopyConstructible* requirements.

~~*Remarks:* This constructor template shall not participate in overload resolution unless~~*Constraints:* `F` is Lvalue-Callable (20.14.16.2) for argument types `ArgTypes...` and return type `R`.

Postconditions: `!*this` if any of the following hold:

(9.1) — `f` is a null function pointer value.

(9.2) — `f` is a null member pointer value.

(9.3) — `F` is an instance of the `function` class template, and `!f`.

Otherwise, `*this` targets a copy of `f` initialized with `std::move(f)`. [*Note*: Implementations should avoid the use of dynamically allocated memory for small callable objects, for example, where `f` is an object holding only a pointer or reference to an object and a member function pointer. — *end note*]

Throws: Nothing if `f` is a specialization of `reference_wrapper` or a function pointer. Otherwise, may throw `bad_alloc` or any exception thrown by `F`'s copy or move constructor.

```
template<class F> function(F) -> function<see below>;
```

Constraints: `&F::operator()` is well-formed when treated as an unevaluated operand.

Remarks: ~~This deduction guide participates in overload resolution only if `&F::operator()` is well-formed when treated as an unevaluated operand. In that case, if~~If `decltype(&F::operator())` is of the form `R(G::*)(A...)` *cv* `&`$_{opt}$ `noexcept`$_{opt}$ for a class type `G`, then the deduced type is `function<R(A...)>`.

[*Example*:

```
void f() {
  int i{5};
  function g = [&](double) { return i; };        // deduces function<int(double)>
}
```

— *end example*]

```
function& operator=(const function& f);
```

Effects: As if by `function(f).swap(*this);`

Returns: `*this`.

```
function& operator=(function&& f);
```

Effects: Replaces the target of `*this` with the target of `f`.

Returns: `*this`.

```
function& operator=(nullptr_t) noexcept;
```

Effects: If `*this != nullptr`, destroys the target of `this`.

Postconditions: `!(*this)`.

Returns: `*this`.

```
template<class F> function& operator=(F&& f);
```

Effects: As if by: `function(std::forward<F>(f)).swap(*this);`

Returns: `*this`.

24 *Remarks:* ~~*Remarks:* This assignment operator shall not participate in overload resolution unless~~ *Constraints:* `decay_t<F>` is Lvalue-Callable (20.14.16.2) for argument types `ArgTypes...` and return type `R`.

```
template<class F> function& operator=(reference_wrapper<F> f) noexcept;
```

25 *Effects:* As if by: `function(f).swap(*this);`

26 *Returns:* `*this`.

```
~function();
```

27 *Effects:* If `*this != nullptr`, destroys the target of `this`.

20.14.16.2.2 Modifiers [func.wrap.func.mod]

```
void swap(function& other) noexcept;
```

1 *Effects:* Interchanges the targets of `*this` and `other`.

20.14.16.2.3 Capacity [func.wrap.func.cap]

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

1 *Returns:* `true` if `*this` has a target, otherwise `false`.

20.14.16.2.4 Invocation [func.wrap.func.inv]

```
R operator()(ArgTypes... args) const;
```

1 *Returns:* *INVOKE*`<R>(f, std::forward<ArgTypes>(args)...)` (20.14.3), where `f` is the target object (20.14.2) of `*this`.

2 *Throws:* `bad_function_call` if `!*this`; otherwise, any exception thrown by the wrapped callable object.

20.14.16.2.5 Target access [func.wrap.func.targ]

```
const type_info& target_type() const noexcept;
```

1 *Returns:* If `*this` has a target of type `T`, `typeid(T)`; otherwise, `typeid(void)`.

```
template<class T>       T* target() noexcept;
template<class T> const T* target() const noexcept;
```

2 *Returns:* If `target_type() == typeid(T)` a pointer to the stored function target; otherwise a null pointer.

20.14.16.2.6 Null pointer comparison functions [func.wrap.func.nullptr]

```
template<class R, class... ArgTypes>
  bool operator==(const function<R(ArgTypes...)>& f, nullptr_t) noexcept;
```

1 *Returns:* `!f`.

20.14.16.2.7 Specialized algorithms [func.wrap.func.alg]

```
template<class R, class... ArgTypes>
  void swap(function<R(ArgTypes...)>& f1, function<R(ArgTypes...)>& f2) noexcept;
```

1 *Effects:* As if by: `f1.swap(f2);`

20.14.17 Searchers [func.search]

1 This subclause provides function object types (20.14) for operations that search for a sequence `[pat_first, pat_last)` in another sequence `[first, last)` that is provided to the object's function call operator. The first sequence (the pattern to be searched for) is provided to the object's constructor, and the second (the sequence to be searched) is provided to the function call operator.

2 Each specialization of a class template specified in this subclause 20.14.17 shall meet the *Cpp17CopyConstructible* and *Cpp17CopyAssignable* requirements. Template parameters named

(2.1) — `ForwardIterator`,

(2.2) — `ForwardIterator1`,

(2.3) — `ForwardIterator2`,

(2.4) — `RandomAccessIterator`,

(2.5) — `RandomAccessIterator1`,

(2.6) — `RandomAccessIterator2`, and

(2.7) — `BinaryPredicate`

of templates specified in this subclause 20.14.17 shall meet the same requirements and semantics as specified in **??**. Template parameters named `Hash` shall meet the *Cpp17Hash* requirements (Table **??**).

3 The Boyer-Moore searcher implements the Boyer-Moore search algorithm. The Boyer-Moore-Horspool searcher implements the Boyer-Moore-Horspool search algorithm. In general, the Boyer-Moore searcher will use more memory and give better runtime performance than Boyer-Moore-Horspool.

20.14.17.1 Class template `default_searcher` [func.search.default]

```
template<class ForwardIterator1, class BinaryPredicate = equal_to<>>
  class default_searcher {
  public:
    constexpr default_searcher(ForwardIterator1 pat_first, ForwardIterator1 pat_last,
                               BinaryPredicate pred = BinaryPredicate());

    template<class ForwardIterator2>
      constexpr pair<ForwardIterator2, ForwardIterator2>
        operator()(ForwardIterator2 first, ForwardIterator2 last) const;

  private:
    ForwardIterator1 pat_first_;        // exposition only
    ForwardIterator1 pat_last_;         // exposition only
    BinaryPredicate pred_;              // exposition only
  };
```

```
constexpr default_searcher(ForwardIterator pat_first, ForwardIterator pat_last,
                           BinaryPredicate pred = BinaryPredicate());
```

1 *Effects:* Constructs a `default_searcher` object, initializing `pat_first_` with `pat_first`, `pat_last_` with `pat_last`, and `pred_` with `pred`.

2 *Throws:* Any exception thrown by the copy constructor of `BinaryPredicate` or `ForwardIterator1`.

```
template<class ForwardIterator2>
  constexpr pair<ForwardIterator2, ForwardIterator2>
    operator()(ForwardIterator2 first, ForwardIterator2 last) const;
```

3 *Effects:* Returns a pair of iterators `i` and `j` such that

(3.1) — `i == search(first, last, pat_first_, pat_last_, pred_)`, and

(3.2) — if `i == last`, then `j == last`, otherwise `j == next(i, distance(pat_first_, pat_last_))`.

20.14.17.2 Class template `boyer_moore_searcher` [func.search.bm]

```
template<class RandomAccessIterator1,
         class Hash = hash<typename iterator_traits<RandomAccessIterator1>::value_type>,
         class BinaryPredicate = equal_to<>>
  class boyer_moore_searcher {
  public:
    boyer_moore_searcher(RandomAccessIterator1 pat_first,
                         RandomAccessIterator1 pat_last,
                         Hash hf = Hash(),
                         BinaryPredicate pred = BinaryPredicate());

    template<class RandomAccessIterator2>
      pair<RandomAccessIterator2, RandomAccessIterator2>
        operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;

  private:
    RandomAccessIterator1 pat_first_;   // exposition only
```

```
    RandomAccessIterator1 pat_last_;     // exposition only
    Hash hash_;                          // exposition only
    BinaryPredicate pred_;               // exposition only
  };

boyer_moore_searcher(RandomAccessIterator1 pat_first,
                     RandomAccessIterator1 pat_last,
                     Hash hf = Hash(),
                     BinaryPredicate pred = BinaryPredicate());
```

1 ~~*Requires:*~~ *Preconditions:* The value type of `RandomAccessIterator1` ~~shall meet~~meets the *Cpp17DefaultConstructible* requirements, the *Cpp17CopyConstructible* requirements, and the *Cpp17CopyAssignable* requirements.

2 ~~*Requires:*~~ *Preconditions:* For any two values `A` and `B` of the type `iterator_traits<RandomAccessIterator1>::value_type`, if `pred(A, B) == true`, then `hf(A) == hf(B)` ~~shall be~~is `true`.

3 *Effects:* Initializes `pat_first_` with `pat_first`, `pat_last_` with `pat_last`, `hash_` with `hf`, and `pred_` with `pred`.

4 *Throws:* Any exception thrown by the copy constructor of `RandomAccessIterator1`, or by the default constructor, copy constructor, or the copy assignment operator of the value type of `RandomAccessIterator1`, or the copy constructor or `operator()` of `BinaryPredicate` or `Hash`. May throw `bad_alloc` if additional memory needed for internal data structures cannot be allocated.

```
template<class RandomAccessIterator2>
  pair<RandomAccessIterator2, RandomAccessIterator2>
    operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;
```

5 ~~*Requires:*~~ *Mandates:* `RandomAccessIterator1` and `RandomAccessIterator2` ~~shall~~ have the same value type.

6 *Effects:* Finds a subsequence of equal values in a sequence.

7 *Returns:* A pair of iterators `i` and `j` such that

(7.1) — `i` is the first iterator in the range `[first, last - (pat_last_ - pat_first_))` such that for every non-negative integer `n` less than `pat_last_ - pat_first_` the following condition holds: `pred(*(i + n), *(pat_first_ + n)) != false`, and

(7.2) — `j == next(i, distance(pat_first_, pat_last_))`.

Returns `make_pair(first, first)` if `[pat_first_, pat_last_)` is empty, otherwise returns `make_pair(last, last)` if no such iterator is found.

8 *Complexity:* At most `(last - first) * (pat_last_ - pat_first_)` applications of the predicate.

20.14.17.3 Class template `boyer_moore_horspool_searcher` [func.search.bmh]

```
template<class RandomAccessIterator1,
         class Hash = hash<typename iterator_traits<RandomAccessIterator1>::value_type>,
         class BinaryPredicate = equal_to<>>
  class boyer_moore_horspool_searcher {
  public:
    boyer_moore_horspool_searcher(RandomAccessIterator1 pat_first,
                                  RandomAccessIterator1 pat_last,
                                  Hash hf = Hash(),
                                  BinaryPredicate pred = BinaryPredicate());

    template<class RandomAccessIterator2>
      pair<RandomAccessIterator2, RandomAccessIterator2>
        operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;

  private:
    RandomAccessIterator1 pat_first_;    // exposition only
    RandomAccessIterator1 pat_last_;     // exposition only
    Hash hash_;                          // exposition only
    BinaryPredicate pred_;               // exposition only
  };
```

```
boyer_moore_horspool_searcher(RandomAccessIterator1 pat_first,
                              RandomAccessIterator1 pat_last,
                              Hash hf = Hash(),
                              BinaryPredicate pred = BinaryPredicate());
```

1 ~~*Requires:*~~ *Preconditions:* The value type of `RandomAccessIterator1` ~~shall meet~~meets the *Cpp17DefaultConstructible*, *Cpp17CopyConstructible*, and *Cpp17CopyAssignable* requirements.

2 ~~*Requires:*~~ *Preconditions:* For any two values `A` and `B` of the type `iterator_traits<RandomAccessIterator1>::value_type`, if `pred(A, B) == true`, then `hf(A) == hf(B)` ~~shall be~~is `true`.

3 *Effects:* Initializes `pat_first_` with `pat_first`, `pat_last_` with `pat_last`, `hash_` with `hf`, and `pred_` with `pred`.

4 *Throws:* Any exception thrown by the copy constructor of `RandomAccessIterator1`, or by the default constructor, copy constructor, or the copy assignment operator of the value type of `RandomAccessIterator1` or the copy constructor or `operator()` of `BinaryPredicate` or `Hash`. May throw `bad_alloc` if additional memory needed for internal data structures cannot be allocated.

```
template<class RandomAccessIterator2>
  pair<RandomAccessIterator2, RandomAccessIterator2>
    operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;
```

5 ~~*Requires:*~~ *Mandates:* `RandomAccessIterator1` and `RandomAccessIterator2` ~~shall~~ have the same value type.

6 *Effects:* Finds a subsequence of equal values in a sequence.

7 *Returns:* A pair of iterators `i` and `j` such that

(7.1) — `i` is the first iterator `i` in the range `[first, last - (pat_last_ - pat_first_))` such that for every non-negative integer `n` less than `pat_last_ - pat_first_` the following condition holds: `pred(*(i + n), *(pat_first_ + n)) != false`, and

(7.2) — `j == next(i, distance(pat_first_, pat_last_))`.

Returns `make_pair(first, first)` if `[pat_first_, pat_last_)` is empty, otherwise returns `make_pair(last, last)` if no such iterator is found.

8 *Complexity:* At most `(last - first) * (pat_last_ - pat_first_)` applications of the predicate.

20.14.18 Class template `hash` [unord.hash]

1 The unordered associative containers defined in **??** use specializations of the class template `hash` (20.14.1) as the default hash function.

2 Each specialization of `hash` is either enabled or disabled, as described below. [*Note*: Enabled specializations meet the *Cpp17Hash* requirements, and disabled specializations do not. — *end note*] Each header that declares the template `hash` provides enabled specializations of `hash` for `nullptr_t` and all cv-unqualified arithmetic, enumeration, and pointer types. For any type `Key` for which neither the library nor the user provides an explicit or partial specialization of the class template `hash`, `hash<Key>` is disabled.

3 If the library provides an explicit or partial specialization of `hash<Key>`, that specialization is enabled except as noted otherwise, and its member functions are `noexcept` except as noted otherwise.

4 If `H` is a disabled specialization of `hash`, these values are `false`: `is_default_constructible_v<H>`, `is_copy_constructible_v<H>`, `is_move_constructible_v<H>`, `is_copy_assignable_v<H>`, and `is_move_assignable_v<H>`. Disabled specializations of `hash` are not function object types (20.14). [*Note*: This means that the specialization of `hash` exists, but any attempts to use it as a *Cpp17Hash* will be ill-formed. — *end note*]

5 An enabled specialization `hash<Key>` will:

(5.1) — meet the *Cpp17Hash* requirements (Table **??**), with `Key` as the function call argument type, the *Cpp17DefaultConstructible* requirements (Table **??**), the *Cpp17CopyAssignable* requirements (Table **??**),

(5.2) — be swappable (**??**) for lvalues,

(5.3) — meet the requirement that if `k1 == k2` is `true`, `h(k1) == h(k2)` is also `true`, where `h` is an object of type `hash<Key>` and `k1` and `k2` are objects of type `Key`;

(5.4) — meet the requirement that the expression `h(k)`, where `h` is an object of type `hash<Key>` and `k` is an object of type `Key`, shall not throw an exception unless `hash<Key>` is a program-defined specialization that depends on at least one program-defined type.

20.15 Metaprogramming and type traits [meta]

1 This subclause describes components used by C++ programs, particularly in templates, to support the widest possible range of types, optimise template code usage, detect type related user errors, and perform type inference and transformation at compile time. It includes type classification traits, type property inspection traits, and type transformations. The type classification traits describe a complete taxonomy of all possible C++ types, and state where in that taxonomy a given type belongs. The type property inspection traits allow important characteristics of types or of combinations of types to be inspected. The type transformations allow certain properties of types to be manipulated.

2 All functions specified in this subclause are signal-safe (**??**).

20.15.1 Requirements [meta.rqmts]

1 A *Cpp17UnaryTypeTrait* describes a property of a type. It shall be a class template that takes one template type argument and, optionally, additional arguments that help define the property being described. It shall be *Cpp17DefaultConstructible*, *Cpp17CopyConstructible*, and publicly and unambiguously derived, directly or indirectly, from its *base characteristic*, which is a specialization of the template `integral_constant` (20.15.3), with the arguments to the template `integral_constant` determined by the requirements for the particular property being described. The member names of the base characteristic shall not be hidden and shall be unambiguously available in the *Cpp17UnaryTypeTrait*.

2 A *Cpp17BinaryTypeTrait* describes a relationship between two types. It shall be a class template that takes two template type arguments and, optionally, additional arguments that help define the relationship being described. It shall be *Cpp17DefaultConstructible*, *Cpp17CopyConstructible*, and publicly and unambiguously derived, directly or indirectly, from its *base characteristic*, which is a specialization of the template `integral_constant` (20.15.3), with the arguments to the template `integral_constant` determined by the requirements for the particular relationship being described. The member names of the base characteristic shall not be hidden and shall be unambiguously available in the *Cpp17BinaryTypeTrait*.

3 A *Cpp17TransformationTrait* modifies a property of a type. It shall be a class template that takes one template type argument and, optionally, additional arguments that help define the modification. It shall define a publicly accessible nested type named `type`, which shall be a synonym for the modified type.

4 Unless otherwise specified, the behavior of a program that adds specializations for any of the templates specified in this subclause 20.15 is undefined.

5 Unless otherwise specified, an incomplete type may be used to instantiate a template specified in this subclause. The behavior of a program is undefined if:

(5.1) — an instantiation of a template specified in subclause 20.15 directly or indirectly depends on an incompletely-defined object type `T`, and

(5.2) — that instantiation could yield a different result were `T` hypothetically completed.

20.15.2 Header `<type_traits>` synopsis [meta.type.synop]

```
namespace std {
  // 20.15.3, helper class
  template<class T, T v> struct integral_constant;

  template<bool B>
    using bool_constant = integral_constant<bool, B>;
  using true_type  = bool_constant<true>;
  using false_type = bool_constant<false>;

  // 20.15.4.1, primary type categories
  template<class T> struct is_void;
  template<class T> struct is_null_pointer;
  template<class T> struct is_integral;
  template<class T> struct is_floating_point;
  template<class T> struct is_array;
  template<class T> struct is_pointer;
```

```
template<class T> struct is_lvalue_reference;
template<class T> struct is_rvalue_reference;
template<class T> struct is_member_object_pointer;
template<class T> struct is_member_function_pointer;
template<class T> struct is_enum;
template<class T> struct is_union;
template<class T> struct is_class;
template<class T> struct is_function;

// 20.15.4.2, composite type categories
template<class T> struct is_reference;
template<class T> struct is_arithmetic;
template<class T> struct is_fundamental;
template<class T> struct is_object;
template<class T> struct is_scalar;
template<class T> struct is_compound;
template<class T> struct is_member_pointer;

// 20.15.4.3, type properties
template<class T> struct is_const;
template<class T> struct is_volatile;
template<class T> struct is_trivial;
template<class T> struct is_trivially_copyable;
template<class T> struct is_standard_layout;
template<class T> struct is_empty;
template<class T> struct is_polymorphic;
template<class T> struct is_abstract;
template<class T> struct is_final;
template<class T> struct is_aggregate;

template<class T> struct is_signed;
template<class T> struct is_unsigned;
template<class T> struct is_bounded_array;
template<class T> struct is_unbounded_array;

template<class T, class... Args> struct is_constructible;
template<class T> struct is_default_constructible;
template<class T> struct is_copy_constructible;
template<class T> struct is_move_constructible;

template<class T, class U> struct is_assignable;
template<class T> struct is_copy_assignable;
template<class T> struct is_move_assignable;

template<class T, class U> struct is_swappable_with;
template<class T> struct is_swappable;

template<class T> struct is_destructible;

template<class T, class... Args> struct is_trivially_constructible;
template<class T> struct is_trivially_default_constructible;
template<class T> struct is_trivially_copy_constructible;
template<class T> struct is_trivially_move_constructible;

template<class T, class U> struct is_trivially_assignable;
template<class T> struct is_trivially_copy_assignable;
template<class T> struct is_trivially_move_assignable;
template<class T> struct is_trivially_destructible;

template<class T, class... Args> struct is_nothrow_constructible;
template<class T> struct is_nothrow_default_constructible;
template<class T> struct is_nothrow_copy_constructible;
template<class T> struct is_nothrow_move_constructible;
```

```
template<class T, class U> struct is_nothrow_assignable;
template<class T> struct is_nothrow_copy_assignable;
template<class T> struct is_nothrow_move_assignable;

template<class T, class U> struct is_nothrow_swappable_with;
template<class T> struct is_nothrow_swappable;

template<class T> struct is_nothrow_destructible;

template<class T> struct has_virtual_destructor;

template<class T> struct has_unique_object_representations;

template<class T> struct has_strong_structural_equality;

// 20.15.5, type property queries
template<class T> struct alignment_of;
template<class T> struct rank;
template<class T, unsigned I = 0> struct extent;

// 20.15.6, type relations
template<class T, class U> struct is_same;
template<class Base, class Derived> struct is_base_of;
template<class From, class To> struct is_convertible;
template<class From, class To> struct is_nothrow_convertible;
template<class T, class U> struct is_layout_compatible;
template<class Base, class Derived> struct is_pointer_interconvertible_base_of;

template<class Fn, class... ArgTypes> struct is_invocable;
template<class R, class Fn, class... ArgTypes> struct is_invocable_r;

template<class Fn, class... ArgTypes> struct is_nothrow_invocable;
template<class R, class Fn, class... ArgTypes> struct is_nothrow_invocable_r;

// 20.15.7.1, const-volatile modifications
template<class T> struct remove_const;
template<class T> struct remove_volatile;
template<class T> struct remove_cv;
template<class T> struct add_const;
template<class T> struct add_volatile;
template<class T> struct add_cv;

template<class T>
  using remove_const_t    = typename remove_const<T>::type;
template<class T>
  using remove_volatile_t = typename remove_volatile<T>::type;
template<class T>
  using remove_cv_t       = typename remove_cv<T>::type;
template<class T>
  using add_const_t       = typename add_const<T>::type;
template<class T>
  using add_volatile_t    = typename add_volatile<T>::type;
template<class T>
  using add_cv_t          = typename add_cv<T>::type;

// 20.15.7.2, reference modifications
template<class T> struct remove_reference;
template<class T> struct add_lvalue_reference;
template<class T> struct add_rvalue_reference;

template<class T>
  using remove_reference_t     = typename remove_reference<T>::type;
template<class T>
  using add_lvalue_reference_t = typename add_lvalue_reference<T>::type;
```

```
template<class T>
  using add_rvalue_reference_t = typename add_rvalue_reference<T>::type;

// 20.15.7.3, sign modifications
template<class T> struct make_signed;
template<class T> struct make_unsigned;

template<class T>
  using make_signed_t   = typename make_signed<T>::type;
template<class T>
  using make_unsigned_t = typename make_unsigned<T>::type;

// 20.15.7.4, array modifications
template<class T> struct remove_extent;
template<class T> struct remove_all_extents;

template<class T>
  using remove_extent_t      = typename remove_extent<T>::type;
template<class T>
  using remove_all_extents_t = typename remove_all_extents<T>::type;

// 20.15.7.5, pointer modifications
template<class T> struct remove_pointer;
template<class T> struct add_pointer;

template<class T>
  using remove_pointer_t = typename remove_pointer<T>::type;
template<class T>
  using add_pointer_t    = typename add_pointer<T>::type;

// 20.15.7.6, other transformations
template<class T> struct type_identity;
template<size_t Len, size_t Align = default-alignment> // see 20.15.7.6
  struct aligned_storage;
template<size_t Len, class... Types> struct aligned_union;
template<class T> struct remove_cvref;
template<class T> struct decay;
template<bool, class T = void> struct enable_if;
template<bool, class T, class F> struct conditional;
template<class... T> struct common_type;
template<class T, class U, template<class> class TQual, template<class> class UQual>
  struct basic_common_reference { };
template<class... T> struct common_reference;
template<class T> struct underlying_type;
template<class Fn, class... ArgTypes> struct invoke_result;

template<class T>
  using type_identity_t    = typename type_identity<T>::type;
template<size_t Len, size_t Align = default-alignment> // see 20.15.7.6
  using aligned_storage_t  = typename aligned_storage<Len, Align>::type;
template<size_t Len, class... Types>
  using aligned_union_t    = typename aligned_union<Len, Types...>::type;
template<class T>
  using remove_cvref_t     = typename remove_cvref<T>::type;
template<class T>
  using decay_t            = typename decay<T>::type;
template<bool b, class T = void>
  using enable_if_t        = typename enable_if<b, T>::type;
template<bool b, class T, class F>
  using conditional_t      = typename conditional<b, T, F>::type;
template<class... T>
  using common_type_t      = typename common_type<T...>::type;
template<class... T>
  using common_reference_t = typename common_reference<T...>::type;
```

```
template<class T>
  using underlying_type_t  = typename underlying_type<T>::type;
template<class Fn, class... ArgTypes>
  using invoke_result_t    = typename invoke_result<Fn, ArgTypes...>::type;
template<class...>
  using void_t             = void;

// 20.15.8, logical operator traits
template<class... B> struct conjunction;
template<class... B> struct disjunction;
template<class B> struct negation;

// 20.15.4.1, primary type categories
template<class T>
  inline constexpr bool is_void_v = is_void<T>::value;
template<class T>
  inline constexpr bool is_null_pointer_v = is_null_pointer<T>::value;
template<class T>
  inline constexpr bool is_integral_v = is_integral<T>::value;
template<class T>
  inline constexpr bool is_floating_point_v = is_floating_point<T>::value;
template<class T>
  inline constexpr bool is_array_v = is_array<T>::value;
template<class T>
  inline constexpr bool is_pointer_v = is_pointer<T>::value;
template<class T>
  inline constexpr bool is_lvalue_reference_v = is_lvalue_reference<T>::value;
template<class T>
  inline constexpr bool is_rvalue_reference_v = is_rvalue_reference<T>::value;
template<class T>
  inline constexpr bool is_member_object_pointer_v = is_member_object_pointer<T>::value;
template<class T>
  inline constexpr bool is_member_function_pointer_v = is_member_function_pointer<T>::value;
template<class T>
  inline constexpr bool is_enum_v = is_enum<T>::value;
template<class T>
  inline constexpr bool is_union_v = is_union<T>::value;
template<class T>
  inline constexpr bool is_class_v = is_class<T>::value;
template<class T>
  inline constexpr bool is_function_v = is_function<T>::value;

// 20.15.4.2, composite type categories
template<class T>
  inline constexpr bool is_reference_v = is_reference<T>::value;
template<class T>
  inline constexpr bool is_arithmetic_v = is_arithmetic<T>::value;
template<class T>
  inline constexpr bool is_fundamental_v = is_fundamental<T>::value;
template<class T>
  inline constexpr bool is_object_v = is_object<T>::value;
template<class T>
  inline constexpr bool is_scalar_v = is_scalar<T>::value;
template<class T>
  inline constexpr bool is_compound_v = is_compound<T>::value;
template<class T>
  inline constexpr bool is_member_pointer_v = is_member_pointer<T>::value;

// 20.15.4.3, type properties
template<class T>
  inline constexpr bool is_const_v = is_const<T>::value;
template<class T>
  inline constexpr bool is_volatile_v = is_volatile<T>::value;
```

```
template<class T>
  inline constexpr bool is_trivial_v = is_trivial<T>::value;
template<class T>
  inline constexpr bool is_trivially_copyable_v = is_trivially_copyable<T>::value;
template<class T>
  inline constexpr bool is_standard_layout_v = is_standard_layout<T>::value;
template<class T>
  inline constexpr bool is_empty_v = is_empty<T>::value;
template<class T>
  inline constexpr bool is_polymorphic_v = is_polymorphic<T>::value;
template<class T>
  inline constexpr bool is_abstract_v = is_abstract<T>::value;
template<class T>
  inline constexpr bool is_final_v = is_final<T>::value;
template<class T>
  inline constexpr bool is_aggregate_v = is_aggregate<T>::value;
template<class T>
  inline constexpr bool is_signed_v = is_signed<T>::value;
template<class T>
  inline constexpr bool is_unsigned_v = is_unsigned<T>::value;
template<class T>
  inline constexpr bool is_bounded_array_v = is_bounded_array<T>::value;
template<class T>
  inline constexpr bool is_unbounded_array_v = is_unbounded_array<T>::value;
template<class T, class... Args>
  inline constexpr bool is_constructible_v = is_constructible<T, Args...>::value;
template<class T>
  inline constexpr bool is_default_constructible_v = is_default_constructible<T>::value;
template<class T>
  inline constexpr bool is_copy_constructible_v = is_copy_constructible<T>::value;
template<class T>
  inline constexpr bool is_move_constructible_v = is_move_constructible<T>::value;
template<class T, class U>
  inline constexpr bool is_assignable_v = is_assignable<T, U>::value;
template<class T>
  inline constexpr bool is_copy_assignable_v = is_copy_assignable<T>::value;
template<class T>
  inline constexpr bool is_move_assignable_v = is_move_assignable<T>::value;
template<class T, class U>
  inline constexpr bool is_swappable_with_v = is_swappable_with<T, U>::value;
template<class T>
  inline constexpr bool is_swappable_v = is_swappable<T>::value;
template<class T>
  inline constexpr bool is_destructible_v = is_destructible<T>::value;
template<class T, class... Args>
  inline constexpr bool is_trivially_constructible_v
    = is_trivially_constructible<T, Args...>::value;
template<class T>
  inline constexpr bool is_trivially_default_constructible_v
    = is_trivially_default_constructible<T>::value;
template<class T>
  inline constexpr bool is_trivially_copy_constructible_v
    = is_trivially_copy_constructible<T>::value;
template<class T>
  inline constexpr bool is_trivially_move_constructible_v
    = is_trivially_move_constructible<T>::value;
template<class T, class U>
  inline constexpr bool is_trivially_assignable_v = is_trivially_assignable<T, U>::value;
template<class T>
  inline constexpr bool is_trivially_copy_assignable_v
    = is_trivially_copy_assignable<T>::value;
template<class T>
  inline constexpr bool is_trivially_move_assignable_v
    = is_trivially_move_assignable<T>::value;
```

```
template<class T>
  inline constexpr bool is_trivially_destructible_v = is_trivially_destructible<T>::value;
template<class T, class... Args>
  inline constexpr bool is_nothrow_constructible_v
    = is_nothrow_constructible<T, Args...>::value;
template<class T>
  inline constexpr bool is_nothrow_default_constructible_v
    = is_nothrow_default_constructible<T>::value;
template<class T>
  inline constexpr bool is_nothrow_copy_constructible_v
  = is_nothrow_copy_constructible<T>::value;
template<class T>
  inline constexpr bool is_nothrow_move_constructible_v
    = is_nothrow_move_constructible<T>::value;
template<class T, class U>
  inline constexpr bool is_nothrow_assignable_v = is_nothrow_assignable<T, U>::value;
template<class T>
  inline constexpr bool is_nothrow_copy_assignable_v = is_nothrow_copy_assignable<T>::value;
template<class T>
  inline constexpr bool is_nothrow_move_assignable_v = is_nothrow_move_assignable<T>::value;
template<class T, class U>
  inline constexpr bool is_nothrow_swappable_with_v = is_nothrow_swappable_with<T, U>::value;
template<class T>
  inline constexpr bool is_nothrow_swappable_v = is_nothrow_swappable<T>::value;
template<class T>
  inline constexpr bool is_nothrow_destructible_v = is_nothrow_destructible<T>::value;
template<class T>
  inline constexpr bool has_virtual_destructor_v = has_virtual_destructor<T>::value;
template<class T>
  inline constexpr bool has_unique_object_representations_v
    = has_unique_object_representations<T>::value;

template<class T>
  inline constexpr bool has_strong_structural_equality_v
    =  has_strong_structural_equality<T>::value;

// 20.15.5, type property queries
template<class T>
  inline constexpr size_t alignment_of_v = alignment_of<T>::value;
template<class T>
  inline constexpr size_t rank_v = rank<T>::value;
template<class T, unsigned I = 0>
  inline constexpr size_t extent_v = extent<T, I>::value;

// 20.15.6, type relations
template<class T, class U>
  inline constexpr bool is_same_v = is_same<T, U>::value;
template<class Base, class Derived>
  inline constexpr bool is_base_of_v = is_base_of<Base, Derived>::value;
template<class From, class To>
  inline constexpr bool is_convertible_v = is_convertible<From, To>::value;
template<class From, class To>
  inline constexpr bool is_nothrow_convertible_v = is_nothrow_convertible<From, To>::value;
template<class T, class U>
  inline constexpr bool is_layout_compatible_v = is_layout_compatible<T, U>::value;
template<class Base, class Derived>
  inline constexpr bool is_pointer_interconvertible_base_of_v
    = is_pointer_interconvertible_base_of<Base, Derived>::value;
template<class Fn, class... ArgTypes>
  inline constexpr bool is_invocable_v = is_invocable<Fn, ArgTypes...>::value;
template<class R, class Fn, class... ArgTypes>
  inline constexpr bool is_invocable_r_v = is_invocable_r<R, Fn, ArgTypes...>::value;
template<class Fn, class... ArgTypes>
  inline constexpr bool is_nothrow_invocable_v = is_nothrow_invocable<Fn, ArgTypes...>::value;
```

```
  template<class R, class Fn, class... ArgTypes>
    inline constexpr bool is_nothrow_invocable_r_v
      = is_nothrow_invocable_r<R, Fn, ArgTypes...>::value;

  // 20.15.8, logical operator traits
  template<class... B>
    inline constexpr bool conjunction_v = conjunction<B...>::value;
  template<class... B>
    inline constexpr bool disjunction_v = disjunction<B...>::value;
  template<class B>
    inline constexpr bool negation_v = negation<B>::value;

  // 20.15.9, member relationships
  template<class S, class M>
    constexpr bool is_pointer_interconvertible_with_class(M S::*m) noexcept;
  template<class S1, class S2, class M1, class M2>
    constexpr bool is_corresponding_member(M1 S1::*m1, M2 S2::*m2) noexcept;

  // 20.15.10, constant evaluation context
  constexpr bool is_constant_evaluated() noexcept;
}
```

20.15.3 Helper classes [meta.help]

```
namespace std {
  template<class T, T v> struct integral_constant {
    static constexpr T value = v;

    using value_type = T;
    using type = integral_constant<T, v>;

    constexpr operator value_type() const noexcept { return value; }
    constexpr value_type operator()() const noexcept { return value; }
  };
}
```

1 The class template `integral_constant`, alias template `bool_constant`, and its associated *typedef-name*s `true_type` and `false_type` are used as base classes to define the interface for various type traits.

20.15.4 Unary type traits [meta.unary]

1 This subclause contains templates that may be used to query the properties of a type at compile time.

2 Each of these templates shall be a *Cpp17UnaryTypeTrait* (20.15.1) with a base characteristic of `true_type` if the corresponding condition is `true`, otherwise `false_type`.

20.15.4.1 Primary type categories [meta.unary.cat]

1 The primary type categories correspond to the descriptions given in subclause **??** of the C++ standard.

2 For any given type `T`, the result of applying one of these templates to `T` and to *cv* `T` shall yield the same result.

3 [*Note*: For any given type `T`, exactly one of the primary type categories has a `value` member that evaluates to `true`. — *end note*]

Table 45: Primary type category predicates [tab:meta.unary.cat]

Template	Condition	Comments
`template<class T>` `struct is_void;`	`T` is `void`	
`template<class T>` `struct is_null_pointer;`	`T` is `nullptr_t` (**??**)	
`template<class T>` `struct is_integral;`	`T` is an integral type (**??**)	
`template<class T>` `struct is_floating_point;`	`T` is a floating-point type (**??**)	

Table 45: Primary type category predicates (continued)

Template	Condition	Comments
`template<class T> struct is_array;`	`T` is an array type (**??**) of known or unknown extent	Class template `array` (**??**) is not an array type.
`template<class T> struct is_pointer;`	`T` is a pointer type (**??**)	Includes pointers to functions but not pointers to non-static members.
`template<class T> struct is_lvalue_reference;`	`T` is an lvalue reference type (**??**)	
`template<class T> struct is_rvalue_reference;`	`T` is an rvalue reference type (**??**)	
`template<class T> struct is_member_object_pointer;`	`T` is a pointer to data member	
`template<class T> struct is_member_function_pointer;`	`T` is a pointer to member function	
`template<class T> struct is_enum;`	`T` is an enumeration type (**??**)	
`template<class T> struct is_union;`	`T` is a union type (**??**)	
`template<class T> struct is_class;`	`T` is a non-union class type (**??**)	
`template<class T> struct is_function;`	`T` is a function type (**??**)	

20.15.4.2 Composite type traits [meta.unary.comp]

1 These templates provide convenient compositions of the primary type categories, corresponding to the descriptions given in subclause **??**.

2 For any given type `T`, the result of applying one of these templates to `T` and to *cv* `T` shall yield the same result.

Table 46: Composite type category predicates [tab:meta.unary.comp]

Template	Condition	Comments
`template<class T> struct is_reference;`	`T` is an lvalue reference or an rvalue reference	
`template<class T> struct is_arithmetic;`	`T` is an arithmetic type (**??**)	
`template<class T> struct is_fundamental;`	`T` is a fundamental type (**??**)	
`template<class T> struct is_object;`	`T` is an object type (**??**)	
`template<class T> struct is_scalar;`	`T` is a scalar type (**??**)	
`template<class T> struct is_compound;`	`T` is a compound type (**??**)	
`template<class T> struct is_member_pointer;`	`T` is a pointer-to-member type (**??**)	

20.15.4.3 Type properties [meta.unary.prop]

1 These templates provide access to some of the more important properties of types.

2 It is unspecified whether the library defines any full or partial specializations of any of these templates.

3 For all of the class templates `X` declared in this subclause, instantiating that template with a template-argument that is a class template specialization may result in the implicit instantiation of the template argument if and only if the semantics of `X` require that the argument is a complete type.

4 For the purpose of defining the templates in this subclause, a function call expression `declval<T>()` for any type `T` is considered to be a trivial (**??**, **??**) function call that is not an odr-use (**??**) of `declval` in the context of the corresponding definition notwithstanding the restrictions of 20.2.6.

Table 47: Type property predicates [tab:meta.unary.prop]

Template	Condition	Preconditions
`template<class T> struct is_const;`	`T` is const-qualified (**??**)	
`template<class T> struct is_volatile;`	`T` is volatile-qualified (**??**)	
`template<class T> struct is_trivial;`	`T` is a trivial type (**??**)	`remove_all_extents_t<T>` shall be a complete type or *cv* `void`.
`template<class T> struct is_trivially_copyable;`	`T` is a trivially copyable type (**??**)	`remove_all_extents_t<T>` shall be a complete type or *cv* `void`.
`template<class T> struct is_standard_layout;`	`T` is a standard-layout type (**??**)	`remove_all_extents_t<T>` shall be a complete type or *cv* `void`.
`template<class T> struct is_empty;`	`T` is a class type, but not a union type, with no non-static data members other than subobjects of zero size, no virtual member functions, no virtual base classes, and no base class `B` for which `is_empty_v<B>` is `false`.	If `T` is a non-union class type, `T` shall be a complete type.
`template<class T> struct is_polymorphic;`	`T` is a polymorphic class (**??**)	If `T` is a non-union class type, `T` shall be a complete type.
`template<class T> struct is_abstract;`	`T` is an abstract class (**??**)	If `T` is a non-union class type, `T` shall be a complete type.
`template<class T> struct is_final;`	`T` is a class type marked with the *class-virt-specifier* `final` (**??**). [*Note*: A union is a class type that can be marked with `final`. — *end note*]	If `T` is a class type, `T` shall be a complete type.
`template<class T> struct is_aggregate;`	`T` is an aggregate type (**??**)	`remove_all_extents_t<T>` shall be a complete type or *cv* `void`.
`template<class T> struct is_signed;`	If `is_arithmetic_v<T>` is `true`, the same result as `T(-1) < T(0)`; otherwise, `false`	
`template<class T> struct is_unsigned;`	If `is_arithmetic_v<T>` is `true`, the same result as `T(0) < T(-1)`; otherwise, `false`	
`template<class T> struct is_bounded_array;`	`T` is an array type of known bound (**??**)	
`template<class T> struct is_unbounded_array;`	`T` is an array type of unknown bound (**??**)	

Table 47: Type property predicates (continued)

Template	Condition	Preconditions
`template<class T, class... Args> struct is_constructible;`	For a function type `T` or for a *cv* `void` type `T`, `is_constructible_v<T, Args...>` is `false`, otherwise *see below*	`T` and all types in the template parameter pack `Args` shall be complete types, *cv* `void`, or arrays of unknown bound.
`template<class T> struct is_default_constructible;`	`is_constructible_v<T>` is `true`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T> struct is_copy_constructible;`	For a referenceable type `T` (**??**), the same result as `is_constructible_v<T, const T&>`, otherwise `false`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T> struct is_move_constructible;`	For a referenceable type `T`, the same result as `is_constructible_v<T, T&&>`, otherwise `false`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T, class U> struct is_assignable;`	The expression `declval<T>() = declval<U>()` is well-formed when treated as an unevaluated operand (**??**). Access checking is performed as if in a context unrelated to `T` and `U`. Only the validity of the immediate context of the assignment expression is considered. [*Note*: The compilation of the expression can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. — *end note*]	`T` and `U` shall be complete types, *cv* `void`, or arrays of unknown bound.
`template<class T> struct is_copy_assignable;`	For a referenceable type `T`, the same result as `is_assignable_v<T&, const T&>`, otherwise `false`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T> struct is_move_assignable;`	For a referenceable type `T`, the same result as `is_assignable_v<T&, T&&>`, otherwise `false`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.

Table 47: Type property predicates (continued)

Template	Condition	Preconditions
`template<class T, class U> struct is_swappable_with;`	The expressions `swap(declval<T>(), declval<U>())` and `swap(declval<U>(), declval<T>())` are each well-formed when treated as an unevaluated operand (**??**) in an overload-resolution context for swappable values (**??**). Access checking is performed as if in a context unrelated to `T` and `U`. Only the validity of the immediate context of the `swap` expressions is considered. [*Note*: The compilation of the expressions can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. — *end note*]	`T` and `U` shall be complete types, *cv* `void`, or arrays of unknown bound.
`template<class T> struct is_swappable;`	For a referenceable type `T`, the same result as `is_swappable_with_v<T&, T&>`, otherwise `false`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T> struct is_destructible;`	Either `T` is a reference type, or `T` is a complete object type for which the expression `declval<U&>().~U()` is well-formed when treated as an unevaluated operand (**??**), where `U` is `remove_all_extents_t<T>`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T, class... Args> struct is_trivially_constructible;`	`is_constructible_v<T, Args...>` is `true` and the variable definition for `is_constructible`, as defined below, is known to call no operation that is not trivial (**??**, **??**).	`T` and all types in the template parameter pack `Args` shall be complete types, *cv* `void`, or arrays of unknown bound.

Table 47: Type property predicates (continued)

Template	Condition	Preconditions
`template<class T> struct is_trivially_default_constructible;`	`is_trivially_constructible_v<T>` is `true`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T> struct is_trivially_copy_constructible;`	For a referenceable type `T`, the same result as `is_trivially_constructible_v<T, const T&>`, otherwise `false`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T> struct is_trivially_move_constructible;`	For a referenceable type `T`, the same result as `is_trivially_constructible_v<T, T&&>`, otherwise `false`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T, class U> struct is_trivially_assignable;`	`is_assignable_v<T, U>` is `true` and the assignment, as defined by `is_assignable`, is known to call no operation that is not trivial (**??**, **??**).	`T` and `U` shall be complete types, *cv* `void`, or arrays of unknown bound.
`template<class T> struct is_trivially_copy_assignable;`	For a referenceable type `T`, the same result as `is_trivially_assignable_v<T&, const T&>`, otherwise `false`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T> struct is_trivially_move_assignable;`	For a referenceable type `T`, the same result as `is_trivially_assignable_v<T&, T&&>`, otherwise `false`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T> struct is_trivially_destructible;`	`is_destructible_v<T>` is `true` and `remove_all_extents_t<T>` is either a non-class type or a class type with a trivial destructor.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T, class... Args> struct is_nothrow_constructible;`	`is_constructible_v<T, Args...>` is `true` and the variable definition for `is_constructible`, as defined below, is known not to throw any exceptions (**??**).	`T` and all types in the template parameter pack `Args` shall be complete types, *cv* `void`, or arrays of unknown bound.
`template<class T> struct is_nothrow_default_constructible;`	`is_nothrow_constructible_v<T>` is `true`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T> struct is_nothrow_copy_constructible;`	For a referenceable type `T`, the same result as `is_nothrow_constructible_v<T, const T&>`, otherwise `false`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.

Table 47: Type property predicates (continued)

Template	Condition	Preconditions

`template<class T>`
`struct`
`is_nothrow_move_constructible;`
	For a referenceable type `T`, the same result as `is_nothrow_-constructible_v<T, T&&>`, otherwise `false`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.

`template<class T, class U>` `struct is_nothrow_assignable;`	`is_assignable_v<T, U>` is `true` and the assignment is known not to throw any exceptions (**??**).	`T` and `U` shall be complete types, *cv* `void`, or arrays of unknown bound.
`template<class T>` `struct is_nothrow_copy_assignable;`	For a referenceable type `T`, the same result as `is_nothrow_-assignable_v<T&, const T&>`, otherwise `false`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T>` `struct is_nothrow_move_assignable;`	For a referenceable type `T`, the same result as `is_nothrow_-assignable_v<T&, T&&>`, otherwise `false`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T, class U>` `struct is_nothrow_swappable_with;`	`is_swappable_with_v<T, U>` is `true` and each `swap` expression of the definition of `is_swappable_with<T, U>` is known not to throw any exceptions (**??**).	`T` and `U` shall be complete types, *cv* `void`, or arrays of unknown bound.
`template<class T>` `struct is_nothrow_swappable;`	For a referenceable type `T`, the same result as `is_nothrow_swappable_-with_v<T&, T&>`, otherwise `false`.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T>` `struct is_nothrow_destructible;`	`is_destructible_v<T>` is `true` and the indicated destructor is known not to throw any exceptions (**??**).	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.
`template<class T>` `struct has_virtual_destructor;`	`T` has a virtual destructor (**??**)	If `T` is a non-union class type, `T` shall be a complete type.

`template<class T>`
`struct`
`has_unique_object_representations;`
	For an array type `T`, the same result as `has_unique_object_-representations_-v<remove_all_extents_-t<T>>`, otherwise *see below*.	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.

`template<class T>`
`struct`
`has_strong_structural_equality;`
	The type `T` has strong structural equality (**??**).	`T` shall be a complete type, *cv* `void`, or an array of unknown bound.

5 [*Example*:

```
is_const_v<const volatile int>      // true
is_const_v<const int*>              // false
is_const_v<const int&>              // false
is_const_v<int[3]>                  // false
is_const_v<const int[3]>            // true
```

— *end example*]

6 [*Example*:

```
remove_const_t<const volatile int>  // volatile int
remove_const_t<const int* const>    // const int*
remove_const_t<const int&>          // const int&
remove_const_t<const int[3]>        // int[3]
```

— *end example*]

7 [*Example*:

```
// Given:
struct P final { };
union U1 { };
union U2 final { };

// the following assertions hold:
static_assert(!is_final_v<int>);
static_assert(is_final_v<P>);
static_assert(!is_final_v<U1>);
static_assert(is_final_v<U2>);
```

— *end example*]

8 The predicate condition for a template specialization `is_constructible<T, Args...>` shall be satisfied if and only if the following variable definition would be well-formed for some invented variable `t`:

```
T t(declval<Args>()...);
```

[*Note*: These tokens are never interpreted as a function declaration. — *end note*] Access checking is performed as if in a context unrelated to `T` and any of the `Args`. Only the validity of the immediate context of the variable initialization is considered. [*Note*: The evaluation of the initialization can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. — *end note*]

9 The predicate condition for a template specialization `has_unique_object_representations<T>` shall be satisfied if and only if:

(9.1) — `T` is trivially copyable, and

(9.2) — any two objects of type `T` with the same value have the same object representation, where two objects of array or non-union class type are considered to have the same value if their respective sequences of direct subobjects have the same values, and two objects of union type are considered to have the same value if they have the same active member and the corresponding members have the same value.

The set of scalar types for which this condition holds is implementation-defined. [*Note*: If a type has padding bits, the condition does not hold; otherwise, the condition holds true for integral types. — *end note*]

20.15.5 Type property queries [meta.unary.prop.query]

1 This subclause contains templates that may be used to query properties of types at compile time.

Table 48: Type property queries [tab:meta.unary.prop.query]

Template	Value
`template<class T>` `struct alignment_of;`	`alignof(T)`. ~~*Requires:*~~ *Mandates:* `alignof(T)` ~~shall be~~is a valid expression (**??**)
`template<class T>` `struct rank;`	If `T` names an array type, an integer value representing the number of dimensions of `T`; otherwise, 0.

`template<class T,`
`unsigned I = 0>`
`struct extent;`
	If `T` is not an array type, or if it has rank less than or equal to `I`, or if `I` is 0 and `T` has type "array of unknown bound of `U`", then 0; otherwise, the bound (**??**) of the `I`$^{\text{th}}$ dimension of `T`, where indexing of `I` is zero-based

2 Each of these templates shall be a *Cpp17UnaryTypeTrait* (20.15.1) with a base characteristic of `integral_constant<size_t, Value>`.

3 [*Example*:

```
// the following assertions hold:
assert(rank_v<int> == 0);
assert(rank_v<int[2]> == 1);
assert(rank_v<int[][4]> == 2);
```

— *end example*]

4 [*Example*:

```
// the following assertions hold:
assert(extent_v<int> == 0);
assert(extent_v<int[2]> == 2);
assert(extent_v<int[2][4]> == 2);
assert(extent_v<int[][4]> == 0);
assert((extent_v<int, 1>) == 0);
assert((extent_v<int[2], 1>) == 0);
assert((extent_v<int[2][4], 1>) == 4);
assert((extent_v<int[][4], 1>) == 4);
```

— *end example*]

20.15.6 Relationships between types [meta.rel]

1 This subclause contains templates that may be used to query relationships between types at compile time.

2 Each of these templates shall be a *Cpp17BinaryTypeTrait* (20.15.1) with a base characteristic of `true_type` if the corresponding condition is true, otherwise `false_type`.

Table 49: Type relationship predicates [tab:meta.rel]

Template	Condition	Comments
`template<class T, class U> struct is_same;`	`T` and `U` name the same type with the same cv-qualifications	
`template<class Base, class Derived> struct is_base_of;`	`Base` is a base class of `Derived` (**??**) without regard to cv-qualifiers or `Base` and `Derived` are not unions and name the same class type without regard to cv-qualifiers	If `Base` and `Derived` are non-union class types and are not possibly cv-qualified versions of the same type, `Derived` shall be a complete type. [*Note*: Base classes that are private, protected, or ambiguous are, nonetheless, base classes. — *end note*]
`template<class From, class To> struct is_convertible;`	*see below*	`From` and `To` shall be complete types, *cv* `void`, or arrays of unknown bound.
`template<class From, class To> struct is_nothrow_convertible;`	`is_convertible_v<From, To>` is `true` and the conversion, as defined by `is_convertible`, is known not to throw any exceptions (**??**)	`From` and `To` shall be complete types, *cv* `void`, or arrays of unknown bound.
`template<class T, class U> struct is_layout_compatible;`	`T` and `U` are layout-compatible (**??**)	`T` and `U` shall be complete types, *cv* `void`, or arrays of unknown bound.

Table 49: Type relationship predicates (continued)

Template	Condition	Comments
`template<class Base, class Derived> struct is_pointer_interconvertible_base_of;`	`Derived` is unambiguously derived from `Base` without regard to cv-qualifiers, and each object of type `Derived` is pointer-interconvertible (??) with its `Base` subobject, or `Base` and `Derived` are not unions and name the same class type without regard to cv-qualifiers.	If `Base` and `Derived` are non-union class types and are not (possibly cv-qualified versions of) the same type, `Derived` shall be a complete type.
`template<class Fn, class... ArgTypes> struct is_invocable;`	The expression *INVOKE*`(declval<Fn>(), declval<ArgTypes>()...)` is well-formed when treated as an unevaluated operand	`Fn` and all types in the template parameter pack `ArgTypes` shall be complete types, *cv* `void`, or arrays of unknown bound.
`template<class R, class Fn, class... ArgTypes> struct is_invocable_r;`	The expression *INVOKE*`<R>(declval<Fn>(), declval<ArgTypes>()...)` is well-formed when treated as an unevaluated operand	`Fn`, `R`, and all types in the template parameter pack `ArgTypes` shall be complete types, *cv* `void`, or arrays of unknown bound.
`template<class Fn, class... ArgTypes> struct is_nothrow_invocable;`	`is_invocable_v<Fn, ArgTypes...>` is `true` and the expression *INVOKE*`(declval<Fn>(), declval<ArgTypes>()...)` is known not to throw any exceptions (??)	`Fn` and all types in the template parameter pack `ArgTypes` shall be complete types, *cv* `void`, or arrays of unknown bound.
`template<class R, class Fn, class... ArgTypes> struct is_nothrow_invocable_r;`	`is_invocable_r_v<R, Fn, ArgTypes...>` is `true` and the expression *INVOKE*`<R>(declval<Fn>(), declval<ArgTypes>()...)` is known not to throw any exceptions (??)	`Fn`, `R`, and all types in the template parameter pack `ArgTypes` shall be complete types, *cv* `void`, or arrays of unknown bound.

3 For the purpose of defining the templates in this subclause, a function call expression `declval<T>()` for any type `T` is considered to be a trivial (??, ??) function call that is not an odr-use (??) of `declval` in the context of the corresponding definition notwithstanding the restrictions of 20.2.6.

4 [*Example*:

```
struct B {};
struct B1 : B {};
struct B2 : B {};
struct D : private B1, private B2 {};

is_base_of_v<B, D>              // true
is_base_of_v<const B, D>        // true
is_base_of_v<B, const D>        // true
is_base_of_v<B, const B>        // true
is_base_of_v<D, B>              // false
is_base_of_v<B&, D&>            // false
is_base_of_v<B[3], D[3]>        // false
is_base_of_v<int, int>          // false
```

— *end example*]

5 The predicate condition for a template specialization `is_convertible<From, To>` shall be satisfied if and only if the return expression in the following code would be well-formed, including any implicit conversions to the return type of the function:

```
To test() {
  return declval<From>();
}
```

[*Note*: This requirement gives well-defined results for reference types, void types, array types, and function types. — *end note*] Access checking is performed in a context unrelated to `To` and `From`. Only the validity of the immediate context of the *expression* of the `return` statement (**??**) (including initialization of the returned object or reference) is considered. [*Note*: The initialization can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. — *end note*]

20.15.7 Transformations between types [meta.trans]

1 This subclause contains templates that may be used to transform one type to another following some predefined rule.

2 Each of the templates in this subclause shall be a *Cpp17TransformationTrait* (20.15.1).

20.15.7.1 Const-volatile modifications [meta.trans.cv]

Table 50: Const-volatile modifications [tab:meta.trans.cv]

Template	Comments
`template<class T>` `struct remove_const;`	The member typedef `type` names the same type as `T` except that any top-level const-qualifier has been removed. [*Example*: `remove_const_t<const volatile int>` evaluates to `volatile int`, whereas `remove_const_t<const int*>` evaluates to `const int*`. — *end example*]
`template<class T>` `struct remove_volatile;`	The member typedef `type` names the same type as `T` except that any top-level volatile-qualifier has been removed. [*Example*: `remove_volatile_t<const volatile int>` evaluates to `const int`, whereas `remove_volatile_t<volatile int*>` evaluates to `volatile int*`. — *end example*]
`template<class T>` `struct remove_cv;`	The member typedef `type` shall be the same as `T` except that any top-level cv-qualifier has been removed. [*Example*: `remove_cv_t<const volatile int>` evaluates to `int`, whereas `remove_cv_t<const volatile int*>` evaluates to `const volatile int*`. — *end example*]
`template<class T>` `struct add_const;`	If `T` is a reference, function, or top-level const-qualified type, then `type` names the same type as `T`, otherwise `T const`.
`template<class T>` `struct add_volatile;`	If `T` is a reference, function, or top-level volatile-qualified type, then `type` names the same type as `T`, otherwise `T volatile`.
`template<class T>` `struct add_cv;`	The member typedef `type` names the same type as `add_const_t<add_volatile_t<T>>`.

20.15.7.2 Reference modifications [meta.trans.ref]

Table 51: Reference modifications [tab:meta.trans.ref]

Template	Comments
`template<class T>` `struct remove_reference;`	If `T` has type "reference to `T1`" then the member typedef `type` names `T1`; otherwise, `type` names `T`.

`template<class T>`
`struct`
`add_lvalue_reference;`
	If `T` names a referenceable type (**??**) then the member typedef `type` names `T&`; otherwise, `type` names `T`. [*Note*: This rule reflects the semantics of reference collapsing (**??**). — *end note*]

Table 51: Reference modifications (continued)

Template	Comments

`template<class T>`
`struct`
`add_rvalue_reference;`
	If `T` names a referenceable type then the member typedef `type` names `T&&`; otherwise, `type` names `T`. [*Note*: This rule reflects the semantics of reference collapsing (**??**). For example, when a type `T` names a type `T1&`, the type `add_rvalue_reference_t<T>` is not an rvalue reference. — *end note*]

20.15.7.3 Sign modifications [meta.trans.sign]

Table 52: Sign modifications [tab:meta.trans.sign]

Template	Comments
`template<class T>` `struct make_signed;`	If `T` names a (possibly cv-qualified) signed integer type (**??**) then the member typedef `type` names the type `T`; otherwise, if `T` names a (possibly cv-qualified) unsigned integer type then `type` names the corresponding signed integer type, with the same cv-qualifiers as `T`; otherwise, `type` names the signed integer type with smallest rank (**??**) for which `sizeof(T) == sizeof(type)`, with the same cv-qualifiers as `T`. ~~*Requires:*~~ *Mandates:* `T` ~~shall be~~is a (possibly cv-qualified) integral type or enumeration but not a `bool` type.
`template<class T>` `struct make_unsigned;`	If `T` names a (possibly cv-qualified) unsigned integer type (**??**) then the member typedef `type` names the type `T`; otherwise, if `T` names a (possibly cv-qualified) signed integer type then `type` names the corresponding unsigned integer type, with the same cv-qualifiers as `T`; otherwise, `type` names the unsigned integer type with smallest rank (**??**) for which `sizeof(T) == sizeof(type)`, with the same cv-qualifiers as `T`. ~~*Requires:*~~ *Mandates:* `T` ~~shall be~~is a (possibly cv-qualified) integral type or enumeration but not a `bool` type.

20.15.7.4 Array modifications [meta.trans.arr]

Table 53: Array modifications [tab:meta.trans.arr]

Template	Comments
`template<class T>` `struct remove_extent;`	If `T` names a type "array of `U`", the member typedef `type` shall be `U`, otherwise `T`. [*Note*: For multidimensional arrays, only the first array dimension is removed. For a type "array of `const U`", the resulting type is `const U`. — *end note*]
`template<class T>` `struct remove_all_extents;`	If `T` is "multi-dimensional array of `U`", the resulting member typedef `type` is `U`, otherwise `T`.

1 [*Example*:

```
// the following assertions hold:
assert((is_same_v<remove_extent_t<int>, int>));
assert((is_same_v<remove_extent_t<int[2]>, int>));
assert((is_same_v<remove_extent_t<int[2][3]>, int[3]>));
assert((is_same_v<remove_extent_t<int[][3]>, int[3]>));
```

— *end example*]

2 [*Example*:

```
// the following assertions hold:
assert((is_same_v<remove_all_extents_t<int>, int>));
assert((is_same_v<remove_all_extents_t<int[2]>, int>));
assert((is_same_v<remove_all_extents_t<int[2][3]>, int>));
```

```
assert((is_same_v<remove_all_extents_t<int[][3]>, int>));
```

— *end example*]

20.15.7.5 Pointer modifications [meta.trans.ptr]

Table 54: Pointer modifications [tab:meta.trans.ptr]

Template	Comments
`template<class T> struct remove_pointer;`	If `T` has type "(possibly cv-qualified) pointer to `T1`" then the member typedef `type` names `T1`; otherwise, it names `T`.
`template<class T> struct add_pointer;`	If `T` names a referenceable type (??) or a *cv* `void` type then the member typedef `type` names the same type as `remove_reference_t<T>*`; otherwise, `type` names `T`.

20.15.7.6 Other transformations [meta.trans.other]

Table 55: Other transformations [tab:meta.trans.other]

Template	Comments
`template<class T> struct type_identity;`	The member typedef `type` names the type `T`.
`template<size_t Len, size_t Align = default-alignment> struct aligned_storage;`	The value of *default-alignment* shall be the most stringent alignment requirement for any C++ object type whose size is no greater than `Len` (??). The member typedef `type` shall be a trivial standard-layout type suitable for use as uninitialized storage for any object whose size is at most `Len` and whose alignment is a divisor of `Align`. ~~*Requires:*~~ *Preconditions:* `Len` ~~shall not be~~is not zero. `Align` ~~shall be~~is equal to `alignof(T)` for some type `T` or to *default-alignment*.
`template<size_t Len, class... Types> struct aligned_union;`	The member typedef `type` shall be a trivial standard-layout type suitable for use as uninitialized storage for any object whose type is listed in `Types`; its size shall be at least `Len`. The static member `alignment_value` shall be an integral constant of type `size_t` whose value is the strictest alignment of all types listed in `Types`. ~~*Requires:*~~ *Mandates:* At least one type is provided. Each type in the template parameter pack `Types` ~~shall be~~is a complete object type.
`template<class T> struct remove_cvref;`	The member typedef `type` names the same type as `remove_cv_t<remove_reference_t<T>>`.
`template<class T> struct decay;`	Let `U` be `remove_reference_t<T>`. If `is_array_v<U>` is `true`, the member typedef `type` shall equal `remove_extent_t<U>*`. If `is_function_v<U>` is `true`, the member typedef `type` shall equal `add_pointer_t<U>`. Otherwise the member typedef `type` equals `remove_cv_t<U>`. [*Note*: This behavior is similar to the lvalue-to-rvalue (??), array-to-pointer (??), and function-to-pointer (??) conversions applied when an lvalue is used as an rvalue, but also strips cv-qualifiers from class types in order to more closely model by-value argument passing. — *end note*]
`template<bool B, class T = void> struct enable_if;`	If `B` is `true`, the member typedef `type` shall equal `T`; otherwise, there shall be no member `type`.
`template<bool B, class T, class F> struct conditional;`	If `B` is `true`, the member typedef `type` shall equal `T`. If `B` is `false`, the member typedef `type` shall equal `F`.
`template<class... T> struct common_type;`	Unless this trait is specialized (as specified in Note B, below), the member `type` shall be defined or omitted as specified in Note A, below. If it is omitted, there shall be no member `type`. Each type in the template parameter pack `T` shall be complete, *cv* `void`, or an array of unknown bound.

Table 55: Other transformations (continued)

Template	Comments
`template<class, class, template<class> class, template<class> class> struct basic_common_reference;`	Unless this trait is specialized (as specified in Note D, below), there shall be no member `type`.
`template<class... T> struct common_reference;`	The member *typedef-name* `type` is defined or omitted as specified in Note C, below. Each type in the parameter pack `T` shall be complete or *cv* `void`.
`template<class T> struct underlying_type;`	If `T` is an enumeration type, the member typedef `type` names the underlying type of `T` (??); otherwise, there is no member `type`. *Mandates:* `T` is not an incomplete enumeration type.
`template<class Fn, class... ArgTypes> struct invoke_result;`	If the expression *INVOKE*`(declval<Fn>(), declval<ArgTypes>()...)` is well-formed when treated as an unevaluated operand (??), the member typedef `type` names the type `decltype(`*INVOKE*`(declval<Fn>(), declval<ArgTypes>()...))`; otherwise, there shall be no member `type`. Access checking is performed as if in a context unrelated to `Fn` and `ArgTypes`. Only the validity of the immediate context of the expression is considered. [*Note*: The compilation of the expression can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. — *end note*] ~~*Requires:*~~ *Preconditions:* `Fn` and all types in the template parameter pack `ArgTypes` ~~shall be~~are complete types, *cv* `void`, or arrays of unknown bound.

1 [*Note*: A typical implementation would define `aligned_storage` as:

```
template<size_t Len, size_t Alignment>
struct aligned_storage {
  typedef struct {
    alignas(Alignment) unsigned char __data[Len];
  } type;
};
```

— *end note*]

2 Let:

(2.1) — *CREF*`(A)` be `add_lvalue_reference_t<const remove_reference_t<A>>`,

(2.2) — *XREF*`(A)` denote a unary alias template `T` such that `T<U>` denotes the same type as `U` with the addition of `A`'s cv and reference qualifiers, for a non-reference cv-unqualified type `U`,

(2.3) — *COPYCV*`(FROM, TO)` be an alias for type `TO` with the addition of `FROM`'s top-level cv-qualifiers, [*Example*: *COPYCV*`(const int, volatile short)` is an alias for `const volatile short`. — *end example*]

(2.4) — *COND-RES*`(X, Y)` be `decltype(false ? declval<X(&)()>()() : declval<Y(&)()>()())`.

Given types `A` and `B`, let `X` be `remove_reference_t<A>`, let `Y` be `remove_reference_t<B>`, and let *COMMON-REF*`(A, B)` be:

(2.5) — If `A` and `B` are both lvalue reference types, *COMMON-REF*`(A, B)` is *COND-RES*`(`*COPYCV*`(X, Y) &, `*COPYCV*`(Y, X) &)` if that type exists and is a reference type.

(2.6) — Otherwise, let `C` be `remove_reference_t<`*COMMON-REF*`(X&, Y&)>&&`. If `A` and `B` are both rvalue reference types, `C` is well-formed, and `is_convertible_v<A, C> && is_convertible_v<B, C>` is `true`, then *COMMON-REF*`(A, B)` is `C`.

(2.7) — Otherwise, let `D` be *COMMON-REF*`(const X&, Y&)`. If `A` is an rvalue reference and `B` is an lvalue reference and `D` is well-formed and `is_convertible_v<A, D>` is `true`, then *COMMON-REF*`(A, B)` is `D`.

(2.8) — Otherwise, if `A` is an lvalue reference and `B` is an rvalue reference, then *`COMMON-REF`*`(A, B)` is *`COMMON-REF`*`(B, A)`.

(2.9) — Otherwise, *`COMMON-REF`*`(A, B)` is ill-formed.

If any of the types computed above is ill-formed, then *`COMMON-REF`*`(A, B)` is ill-formed.

3 Note A: For the `common_type` trait applied to a template parameter pack `T` of types, the member `type` shall be either defined or not present as follows:

(3.1) — If `sizeof...(T)` is zero, there shall be no member `type`.

(3.2) — If `sizeof...(T)` is one, let `T0` denote the sole type constituting the pack `T`. The member *typedef-name* `type` shall denote the same type, if any, as `common_type_t<T0, T0>`; otherwise there shall be no member `type`.

(3.3) — If `sizeof...(T)` is two, let the first and second types constituting `T` be denoted by `T1` and `T2`, respectively, and let `D1` and `D2` denote the same types as `decay_t<T1>` and `decay_t<T2>`, respectively.

(3.3.1) — If `is_same_v<T1, D1>` is `false` or `is_same_v<T2, D2>` is `false`, let `C` denote the same type, if any, as `common_type_t<D1, D2>`.

(3.3.2) — [*Note*: None of the following will apply if there is a specialization `common_type<D1, D2>`. — *end note*]

(3.3.3) — Otherwise, if both `D1` and `D2` denote comparison category types (**??**), let `C` denote the common comparison type (**??**) of `D1` and `D2`.

(3.3.4) — Otherwise, if

```
decay_t<decltype(false ? declval<D1>() : declval<D2>())>
```

denotes a valid type, let `C` denote that type.

(3.3.5) — Otherwise, if *`COND-RES`*`(`*`CREF`*`(D1), `*`CREF`*`(D2))` denotes a type, let `C` denote the type `decay_t<`*`COND-RES`*`(`*`CREF`*`(D1), `*`CREF`*`(D2))>`.

In either case, the member *typedef-name* `type` shall denote the same type, if any, as `C`. Otherwise, there shall be no member `type`.

(3.4) — If `sizeof...(T)` is greater than two, let `T1`, `T2`, and `R`, respectively, denote the first, second, and (pack of) remaining types constituting `T`. Let `C` denote the same type, if any, as `common_type_t<T1, T2>`. If there is such a type `C`, the member *typedef-name* `type` shall denote the same type, if any, as `common_type_t<C, R...>`. Otherwise, there shall be no member `type`.

4 Note B: Notwithstanding the provisions of 20.15.2, and pursuant to **??**, a program may specialize `common_type<T1, T2>` for types `T1` and `T2` such that `is_same_v<T1, decay_t<T1>>` and `is_same_v<T2, decay_t<T2>>` are each `true`. [*Note*: Such specializations are needed when only explicit conversions are desired between the template arguments. — *end note*] Such a specialization need not have a member named `type`, but if it does, that member shall be a *typedef-name* for an accessible and unambiguous cv-unqualified non-reference type `C` to which each of the types `T1` and `T2` is explicitly convertible. Moreover, `common_type_t<T1, T2>` shall denote the same type, if any, as does `common_type_t<T2, T1>`. No diagnostic is required for a violation of this Note's rules.

5 Note C: For the `common_reference` trait applied to a parameter pack `T` of types, the member `type` shall be either defined or not present as follows:

(5.1) — If `sizeof...(T)` is zero, there shall be no member `type`.

(5.2) — Otherwise, if `sizeof...(T)` is one, let `T0` denote the sole type in the pack `T`. The member typedef `type` shall denote the same type as `T0`.

(5.3) — Otherwise, if `sizeof...(T)` is two, let `T1` and `T2` denote the two types in the pack `T`. Then

(5.3.1) — If `T1` and `T2` are reference types and *`COMMON-REF`*`(T1, T2)` is well-formed, then the member typedef `type` denotes that type.

(5.3.2) — Otherwise, if `basic_common_reference<remove_cvref_t<T1>, remove_cvref_t<T2>, `*`XREF`*`(T1), `*`XREF`*`(T2)>::type` is well-formed, then the member typedef `type` denotes that type.

(5.3.3) — Otherwise, if *`COND-RES`*`(T1, T2)` is well-formed, then the member typedef `type` denotes that type.

(5.3.4) — Otherwise, if `common_type_t<T1, T2>` is well-formed, then the member typedef `type` denotes that type.

(5.3.5) — Otherwise, there shall be no member `type`.

(5.4) — Otherwise, if `sizeof...(T)` is greater than two, let `T1`, `T2`, and `Rest`, respectively, denote the first, second, and (pack of) remaining types comprising `T`. Let `C` be the type `common_reference_t<T1, T2>`. Then:

(5.4.1) — If there is such a type `C`, the member typedef `type` shall denote the same type, if any, as `common_reference_t<C, Rest...>`.

(5.4.2) — Otherwise, there shall be no member `type`.

6 Note D: Notwithstanding the provisions of 20.15.2, and pursuant to **??**, a program may partially specialize `basic_common_reference<T, U, TQual, UQual>` for types `T` and `U` such that `is_same_v<T, decay_t<T>>` and `is_same_v<U, decay_t<U>>` are each `true`. [*Note*: Such specializations can be used to influence the result of `common_reference`, and are needed when only explicit conversions are desired between the template arguments. — *end note*] Such a specialization need not have a member named `type`, but if it does, that member shall be a *typedef-name* for an accessible and unambiguous type `C` to which each of the types `TQual<T>` and `UQual<U>` is convertible. Moreover, `basic_common_reference<T, U, TQual, UQual>::type` shall denote the same type, if any, as does `basic_common_reference<U, T, UQual, TQual>::type`. No diagnostic is required for a violation of these rules.

7 [*Example*: Given these definitions:

```
using PF1 = bool  (&)();
using PF2 = short (*)(long);

struct S {
  operator PF2() const;
  double operator()(char, int&);
  void fn(long) const;
  char data;
};

using PMF = void (S::*)(long) const;
using PMD = char  S::*;
```

the following assertions will hold:

```
static_assert(is_same_v<invoke_result_t<S, int>, short>);
static_assert(is_same_v<invoke_result_t<S&, unsigned char, int&>, double>);
static_assert(is_same_v<invoke_result_t<PF1>, bool>);
static_assert(is_same_v<invoke_result_t<PMF, unique_ptr<S>, int>, void>);
static_assert(is_same_v<invoke_result_t<PMD, S>, char&&>);
static_assert(is_same_v<invoke_result_t<PMD, const S*>, const char&>);
```

— *end example*]

20.15.8 Logical operator traits [meta.logical]

1 This subclause describes type traits for applying logical operators to other type traits.

```
template<class... B> struct conjunction : see below { };
```

2 The class template `conjunction` forms the logical conjunction of its template type arguments.

3 For a specialization `conjunction<`B_1`, ..., `B_N`>`, if there is a template type argument B_i for which `bool(`B_i`::value)` is `false`, then instantiating `conjunction<`B_1`, ..., `B_N`>::value` does not require the instantiation of B_j`::value` for $j > i$. [*Note*: This is analogous to the short-circuiting behavior of the built-in operator `&&`. — *end note*]

4 Every template type argument for which B_i`::value` is instantiated shall be usable as a base class and shall have a member `value` which is convertible to `bool`, is not hidden, and is unambiguously available in the type.

5 The specialization `conjunction<`B_1`, ..., `B_N`>` has a public and unambiguous base that is either

(5.1) — the first type B_i in the list `true_type, `B_1`, ..., `B_N for which `bool(`B_i`::value)` is `false`, or

(5.2) — if there is no such B_i, the last type in the list.

[*Note*: This means a specialization of `conjunction` does not necessarily inherit from either `true_type` or `false_type`. — *end note*]

6 The member names of the base class, other than `conjunction` and `operator=`, shall not be hidden and shall be unambiguously available in `conjunction`.

```
template<class... B> struct disjunction : see below { };
```

7 The class template `disjunction` forms the logical disjunction of its template type arguments.

8 For a specialization `disjunction<`B_1`, ..., `B_N`>`, if there is a template type argument B_i for which `bool(`B_i`::value)` is `true`, then instantiating `disjunction<`B_1`, ..., `B_N`>::value` does not require the instantiation of B_j`::value` for $j > i$. [*Note*: This is analogous to the short-circuiting behavior of the built-in operator `||`. — *end note*]

9 Every template type argument for which B_i`::value` is instantiated shall be usable as a base class and shall have a member `value` which is convertible to `bool`, is not hidden, and is unambiguously available in the type.

10 The specialization `disjunction<`B_1`, ..., `B_N`>` has a public and unambiguous base that is either

(10.1) — the first type B_i in the list `false_type, `B_1`, ..., `B_N for which `bool(`B_i`::value)` is `true`, or

(10.2) — if there is no such B_i, the last type in the list.

[*Note*: This means a specialization of `disjunction` does not necessarily inherit from either `true_type` or `false_type`. — *end note*]

11 The member names of the base class, other than `disjunction` and `operator=`, shall not be hidden and shall be unambiguously available in `disjunction`.

```
template<class B> struct negation : see below { };
```

12 The class template `negation` forms the logical negation of its template type argument. The type `negation<B>` is a *Cpp17UnaryTypeTrait* with a base characteristic of `bool_constant<!bool(B::value)>`.

20.15.9 Member relationships [meta.member]

```
template<class S, class M>
  constexpr bool is_pointer_interconvertible_with_class(M S::*m) noexcept;
```

1 *Mandates:* `S` is a complete type.

2 *Returns:* `true` if and only if `S` is a standard-layout type, `M` is an object type, `m` is not null, and each object `s` of type `S` is pointer-interconvertible (**??**) with its subobject `s.*m`.

```
template<class S1, class S2, class M1, class M2>
  constexpr bool is_corresponding_member(M1 S1::*m1, M2 S2::*m2) noexcept;
```

3 *Mandates:* `S1` and `S2` are complete types.

4 *Returns:* `true` if and only if `S1` and `S2` are standard-layout types, `M1` and `M2` are object types, `m1` and `m2` are not null, and `m1` and `m2` point to corresponding members of the common initial sequence (**??**) of `S1` and `S2`.

5 [*Note*: The type of a pointer-to-member expression `&C::b` is not always a pointer to member of `C`, leading to potentially surprising results when using these functions in conjunction with inheritance. [*Example*:

```
struct A { int a; };                    // a standard-layout class
struct B { int b; };                    // a standard-layout class
struct C: public A, public B { };       // not a standard-layout class

static_assert( is_pointer_interconvertible_with_class( &C::b ) );
  // Succeeds because, despite its appearance, &C::b has type
  // "pointer to member of B of type int".
static_assert( is_pointer_interconvertible_with_class<C>( &C::b ) );
  // Forces the use of class C, and fails.

static_assert( is_corresponding_member( &C::a, &C::b ) );
```

```
  // Succeeds because, despite its appearance, &C::a and &C::b have types
  // "pointer to member of A of type int" and
  // "pointer to member of B of type int", respectively.
static_assert( is_corresponding_member<C, C>( &C::a, &C::b ) );
  // Forces the use of class C, and fails.
```

— *end example*] — *end note*]

20.15.10 Constant evaluation context [meta.const.eval]

```
constexpr bool is_constant_evaluated() noexcept;
```

1 *Returns:* `true` if and only if evaluation of the call occurs within the evaluation of an expression or conversion that is manifestly constant-evaluated (**??**).

2 [*Example*:

```
constexpr void f(unsigned char *p, int n) {
  if (std::is_constant_evaluated()) {          // should not be a constexpr if statement
    for (int k = 0; k<n; ++k) p[k] = 0;
  } else {
    memset(p, 0, n);                           // not a core constant expression
  }
}
```

— *end example*]

20.16 Compile-time rational arithmetic [ratio]

20.16.1 In general [ratio.general]

1 Subclause 20.16 describes the ratio library. It provides a class template `ratio` which exactly represents any finite rational number with a numerator and denominator representable by compile-time constants of type `intmax_t`.

2 Throughout subclause 20.16, the names of template parameters are used to express type requirements. If a template parameter is named `R1` or `R2`, and the template argument is not a specialization of the `ratio` template, the program is ill-formed.

20.16.2 Header <ratio> synopsis [ratio.syn]

```
namespace std {
  // 20.16.3, class template ratio
  template<intmax_t N, intmax_t D = 1> class ratio;

  // 20.16.4, ratio arithmetic
  template<class R1, class R2> using ratio_add = see below;
  template<class R1, class R2> using ratio_subtract = see below;
  template<class R1, class R2> using ratio_multiply = see below;
  template<class R1, class R2> using ratio_divide = see below;

  // 20.16.5, ratio comparison
  template<class R1, class R2> struct ratio_equal;
  template<class R1, class R2> struct ratio_not_equal;
  template<class R1, class R2> struct ratio_less;
  template<class R1, class R2> struct ratio_less_equal;
  template<class R1, class R2> struct ratio_greater;
  template<class R1, class R2> struct ratio_greater_equal;

  template<class R1, class R2>
    inline constexpr bool ratio_equal_v = ratio_equal<R1, R2>::value;
  template<class R1, class R2>
    inline constexpr bool ratio_not_equal_v = ratio_not_equal<R1, R2>::value;
  template<class R1, class R2>
    inline constexpr bool ratio_less_v = ratio_less<R1, R2>::value;
  template<class R1, class R2>
    inline constexpr bool ratio_less_equal_v = ratio_less_equal<R1, R2>::value;
  template<class R1, class R2>
    inline constexpr bool ratio_greater_v = ratio_greater<R1, R2>::value;
```

```
  template<class R1, class R2>
    inline constexpr bool ratio_greater_equal_v = ratio_greater_equal<R1, R2>::value;

  // 20.16.6, convenience SI typedefs
  using yocto = ratio<1, 1'000'000'000'000'000'000'000'000>;  // see below
  using zepto = ratio<1,     1'000'000'000'000'000'000'000>;  // see below
  using atto  = ratio<1,         1'000'000'000'000'000'000>;
  using femto = ratio<1,             1'000'000'000'000'000>;
  using pico  = ratio<1,                 1'000'000'000'000>;
  using nano  = ratio<1,                     1'000'000'000>;
  using micro = ratio<1,                         1'000'000>;
  using milli = ratio<1,                             1'000>;
  using centi = ratio<1,                               100>;
  using deci  = ratio<1,                                10>;
  using deca  = ratio<                                  10, 1>;
  using hecto = ratio<                                 100, 1>;
  using kilo  = ratio<                               1'000, 1>;
  using mega  = ratio<                           1'000'000, 1>;
  using giga  = ratio<                       1'000'000'000, 1>;
  using tera  = ratio<                   1'000'000'000'000, 1>;
  using peta  = ratio<               1'000'000'000'000'000, 1>;
  using exa   = ratio<           1'000'000'000'000'000'000, 1>;
  using zetta = ratio<       1'000'000'000'000'000'000'000, 1>;  // see below
  using yotta = ratio<1'000'000'000'000'000'000'000'000, 1>;  // see below
}
```

20.16.3 Class template `ratio` [ratio.ratio]

```
namespace std {
  template<intmax_t N, intmax_t D = 1> class ratio {
  public:
    static constexpr intmax_t num;
    static constexpr intmax_t den;
    using type = ratio<num, den>;
  };
}
```

1 If the template argument `D` is zero or the absolute values of either of the template arguments `N` and `D` is not representable by type `intmax_t`, the program is ill-formed. [*Note*: These rules ensure that infinite ratios are avoided and that for any negative input, there exists a representable value of its absolute value which is positive. This excludes the most negative value. — *end note*]

2 The static data members `num` and `den` shall have the following values, where `gcd` represents the greatest common divisor of the absolute values of `N` and `D`:

(2.1) — `num` shall have the value `sign(N) * sign(D) * abs(N) / gcd`.

(2.2) — `den` shall have the value `abs(D) / gcd`.

20.16.4 Arithmetic on `ratios` [ratio.arithmetic]

1 Each of the alias templates `ratio_add`, `ratio_subtract`, `ratio_multiply`, and `ratio_divide` denotes the result of an arithmetic computation on two `ratios` `R1` and `R2`. With `X` and `Y` computed (in the absence of arithmetic overflow) as specified by Table 56, each alias denotes a `ratio<U, V>` such that `U` is the same as `ratio<X, Y>::num` and `V` is the same as `ratio<X, Y>::den`.

2 If it is not possible to represent `U` or `V` with `intmax_t`, the program is ill-formed. Otherwise, an implementation should yield correct values of `U` and `V`. If it is not possible to represent `X` or `Y` with `intmax_t`, the program is ill-formed unless the implementation yields correct values of `U` and `V`.

3 [*Example*:

```
static_assert(ratio_add<ratio<1, 3>, ratio<1, 6>>::num == 1, "1/3+1/6 == 1/2");
static_assert(ratio_add<ratio<1, 3>, ratio<1, 6>>::den == 2, "1/3+1/6 == 1/2");
static_assert(ratio_multiply<ratio<1, 3>, ratio<3, 2>>::num == 1, "1/3*3/2 == 1/2");
static_assert(ratio_multiply<ratio<1, 3>, ratio<3, 2>>::den == 2, "1/3*3/2 == 1/2");
```

Table 56: Expressions used to perform ratio arithmetic [tab:ratio.arithmetic]

Type	Value of X	Value of Y
ratio_add<R1, R2>	R1::num * R2::den + R2::num * R1::den	R1::den * R2::den
ratio_subtract<R1, R2>	R1::num * R2::den - R2::num * R1::den	R1::den * R2::den
ratio_multiply<R1, R2>	R1::num * R2::num	R1::den * R2::den
ratio_divide<R1, R2>	R1::num * R2::den	R1::den * R2::num

```
// The following cases may cause the program to be ill-formed under some implementations
static_assert(ratio_add<ratio<1, INT_MAX>, ratio<1, INT_MAX>>::num == 2,
  "1/MAX+1/MAX == 2/MAX");
static_assert(ratio_add<ratio<1, INT_MAX>, ratio<1, INT_MAX>>::den == INT_MAX,
  "1/MAX+1/MAX == 2/MAX");
static_assert(ratio_multiply<ratio<1, INT_MAX>, ratio<INT_MAX, 2>>::num == 1,
  "1/MAX * MAX/2 == 1/2");
static_assert(ratio_multiply<ratio<1, INT_MAX>, ratio<INT_MAX, 2>>::den == 2,
  "1/MAX * MAX/2 == 1/2");
```

— *end example*]

20.16.5 Comparison of ratios [ratio.comparison]

```
template<class R1, class R2>
  struct ratio_equal : bool_constant<R1::num == R2::num && R1::den == R2::den> { };

template<class R1, class R2>
  struct ratio_not_equal : bool_constant<!ratio_equal_v<R1, R2>> { };

template<class R1, class R2>
  struct ratio_less : bool_constant<see below> { };
```

1 If R1::num × R2::den is less than R2::num × R1::den, ratio_less<R1, R2> shall be derived from bool_constant<true>; otherwise it shall be derived from bool_constant<false>. Implementations may use other algorithms to compute this relationship to avoid overflow. If overflow occurs, the program is ill-formed.

```
template<class R1, class R2>
  struct ratio_less_equal : bool_constant<!ratio_less_v<R2, R1>> { };

template<class R1, class R2>
  struct ratio_greater : bool_constant<ratio_less_v<R2, R1>> { };

template<class R1, class R2>
  struct ratio_greater_equal : bool_constant<!ratio_less_v<R1, R2>> { };
```

20.16.6 SI types for ratio [ratio.si]

1 For each of the *typedef-name*s yocto, zepto, zetta, and yotta, if both of the constants used in its specification are representable by intmax_t, the typedef shall be defined; if either of the constants is not representable by intmax_t, the typedef shall not be defined.

20.17 Class type_index [type.index]

20.17.1 Header <typeindex> synopsis [type.index.synopsis]

```
namespace std {
  class type_index;
  template<class T> struct hash;
  template<> struct hash<type_index>;
}
```

20.17.2 `type_index` overview [type.index.overview]

```
namespace std {
  class type_index {
  public:
    type_index(const type_info& rhs) noexcept;
    bool operator==(const type_index& rhs) const noexcept;
    bool operator< (const type_index& rhs) const noexcept;
    bool operator> (const type_index& rhs) const noexcept;
    bool operator<=(const type_index& rhs) const noexcept;
    bool operator>=(const type_index& rhs) const noexcept;
    strong_ordering operator<=>(const type_index& rhs) const noexcept;
    size_t hash_code() const noexcept;
    const char* name() const noexcept;

  private:
    const type_info* target;     // exposition only
    // Note that the use of a pointer here, rather than a reference,
    // means that the default copy/move constructor and assignment
    // operators will be provided and work as expected.
  };
}
```

1 The class `type_index` provides a simple wrapper for `type_info` which can be used as an index type in associative containers (**??**) and in unordered associative containers (**??**).

20.17.3 `type_index` members [type.index.members]

`type_index(const type_info& rhs) noexcept;`

1 *Effects:* Constructs a `type_index` object, the equivalent of `target = &rhs`.

`bool operator==(const type_index& rhs) const noexcept;`

2 *Returns:* `*target == *rhs.target`.

`bool operator<(const type_index& rhs) const noexcept;`

3 *Returns:* `target->before(*rhs.target)`.

`bool operator>(const type_index& rhs) const noexcept;`

4 *Returns:* `rhs.target->before(*target)`.

`bool operator<=(const type_index& rhs) const noexcept;`

5 *Returns:* `!rhs.target->before(*target)`.

`bool operator>=(const type_index& rhs) const noexcept;`

6 *Returns:* `!target->before(*rhs.target)`.

`strong_ordering operator<=>(const type_index& rhs) const noexcept;`

7 *Effects:* Equivalent to:

```
if (*target == *rhs.target) return strong_ordering::equal;
if (target->before(*rhs.target)) return strong_ordering::less;
return strong_ordering::greater;
```

`size_t hash_code() const noexcept;`

8 *Returns:* `target->hash_code()`.

`const char* name() const noexcept;`

9 *Returns:* `target->name()`.

20.17.4 Hash support [type.index.hash]

```
template<> struct hash<type_index>;
```

1 For an object `index` of type `type_index`, `hash<type_index>()(index)` shall evaluate to the same result as `index.hash_code()`.

20.18 Execution policies [execpol]

20.18.1 In general [execpol.general]

1 Subclause 20.18 describes classes that are *execution policy* types. An object of an execution policy type indicates the kinds of parallelism allowed in the execution of an algorithm and expresses the consequent requirements on the element access functions. [*Example*:

```
using namespace std;
vector<int> v = /* ... */;

// standard sequential sort
sort(v.begin(), v.end());

// explicitly sequential sort
sort(execution::seq, v.begin(), v.end());

// permitting parallel execution
sort(execution::par, v.begin(), v.end());

// permitting vectorization as well
sort(execution::par_unseq, v.begin(), v.end());
```

— *end example*] [*Note*: Because different parallel architectures may require idiosyncratic parameters for efficient execution, implementations may provide additional execution policies to those described in this standard as extensions. — *end note*]

20.18.2 Header <execution> synopsis [execution.syn]

```
namespace std {
  // 20.18.3, execution policy type trait
  template<class T> struct is_execution_policy;
  template<class T> inline constexpr bool is_execution_policy_v = is_execution_policy<T>::value;
}

namespace std::execution {
  // 20.18.4, sequenced execution policy
  class sequenced_policy;

  // 20.18.5, parallel execution policy
  class parallel_policy;

  // 20.18.6, parallel and unsequenced execution policy
  class parallel_unsequenced_policy;

  // 20.18.7, unsequenced execution policy
  class unsequenced_policy;

  // 20.18.8, execution policy objects
  inline constexpr sequenced_policy           seq{ unspecified };
  inline constexpr parallel_policy            par{ unspecified };
  inline constexpr parallel_unsequenced_policy par_unseq{ unspecified };
  inline constexpr unsequenced_policy         unseq{ unspecified };
}
```

20.18.3 Execution policy type trait [execpol.type]

```
template<class T> struct is_execution_policy { see below };
```

1 `is_execution_policy` can be used to detect execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.

2 `is_execution_policy<T>` shall be a *Cpp17UnaryTypeTrait* with a base characteristic of `true_type` if `T` is the type of a standard or implementation-defined execution policy, otherwise `false_type`.

[*Note*: This provision reserves the privilege of creating non-standard execution policies to the library implementation. — *end note*]

3 The behavior of a program that adds specializations for `is_execution_policy` is undefined.

20.18.4 Sequenced execution policy [execpol.seq]

```
class execution::sequenced_policy { unspecified };
```

1 The class `execution::sequenced_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and require that a parallel algorithm's execution may not be parallelized.

2 During the execution of a parallel algorithm with the `execution::sequenced_policy` policy, if the invocation of an element access function exits via an uncaught exception, `terminate()` shall be called.

20.18.5 Parallel execution policy [execpol.par]

```
class execution::parallel_policy { unspecified };
```

1 The class `execution::parallel_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized.

2 During the execution of a parallel algorithm with the `execution::parallel_policy` policy, if the invocation of an element access function exits via an uncaught exception, `terminate()` shall be called.

20.18.6 Parallel and unsequenced execution policy [execpol.parunseq]

```
class execution::parallel_unsequenced_policy { unspecified };
```

1 The class `execution::parallel_unsequenced_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized and vectorized.

2 During the execution of a parallel algorithm with the `execution::parallel_unsequenced_policy` policy, if the invocation of an element access function exits via an uncaught exception, `terminate()` shall be called.

20.18.7 Unsequenced execution policy [execpol.unseq]

```
class execution::unsequenced_policy { unspecified };
```

1 The class `unsequenced_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be vectorized, e.g., executed on a single thread using instructions that operate on multiple data items.

2 During the execution of a parallel algorithm with the `execution::unsequenced_policy` policy, if the invocation of an element access function exits via an uncaught exception, `terminate()` shall be called.

20.18.8 Execution policy objects [execpol.objects]

```
inline constexpr execution::sequenced_policy            execution::seq{ unspecified };
inline constexpr execution::parallel_policy             execution::par{ unspecified };
inline constexpr execution::parallel_unsequenced_policy execution::par_unseq{ unspecified };
inline constexpr execution::unsequenced_policy          execution::unseq{ unspecified };
```

1 The header `<execution>` declares global objects associated with each type of execution policy.

20.19 Primitive numeric conversions [charconv]

20.19.1 Header <charconv> synopsis [charconv.syn]

```
namespace std {
  // floating-point format for primitive numerical conversion
  enum class chars_format {
    scientific = unspecified,
    fixed = unspecified,
```

```
    hex = unspecified,
    general = fixed | scientific
  };

  // 20.19.2, primitive numerical output conversion
  struct to_chars_result {
    char* ptr;
    errc ec;
    friend bool operator==(const to_chars_result&, const to_chars_result&) = default;
  };

  to_chars_result to_chars(char* first, char* last, see below value, int base = 10);
  to_chars_result to_chars(char* first, char* last, bool value, int base = 10) = delete;

  to_chars_result to_chars(char* first, char* last, float value);
  to_chars_result to_chars(char* first, char* last, double value);
  to_chars_result to_chars(char* first, char* last, long double value);

  to_chars_result to_chars(char* first, char* last, float value, chars_format fmt);
  to_chars_result to_chars(char* first, char* last, double value, chars_format fmt);
  to_chars_result to_chars(char* first, char* last, long double value, chars_format fmt);

  to_chars_result to_chars(char* first, char* last, float value,
                           chars_format fmt, int precision);
  to_chars_result to_chars(char* first, char* last, double value,
                           chars_format fmt, int precision);
  to_chars_result to_chars(char* first, char* last, long double value,
                           chars_format fmt, int precision);

  // 20.19.3, primitive numerical input conversion
  struct from_chars_result {
    const char* ptr;
    errc ec;
    friend bool operator==(const from_chars_result&, const from_chars_result&) = default;
  };

  from_chars_result from_chars(const char* first, const char* last,
                               see below& value, int base = 10);

  from_chars_result from_chars(const char* first, const char* last, float& value,
                               chars_format fmt = chars_format::general);
  from_chars_result from_chars(const char* first, const char* last, double& value,
                               chars_format fmt = chars_format::general);
  from_chars_result from_chars(const char* first, const char* last, long double& value,
                               chars_format fmt = chars_format::general);
}
```

1 The type `chars_format` is a bitmask type (**??**) with elements `scientific`, `fixed`, and `hex`.

20.19.2 Primitive numeric output conversion [charconv.to.chars]

1 All functions named `to_chars` convert `value` into a character string by successively filling the range `[first, last)`, where `[first, last)` is required to be a valid range. If the member `ec` of the return value is such that the value is equal to the value of a value-initialized `errc`, the conversion was successful and the member `ptr` is the one-past-the-end pointer of the characters written. Otherwise, the member `ec` has the value `errc::value_too_large`, the member `ptr` has the value `last`, and the contents of the range `[first, last)` are unspecified.

2 The functions that take a floating-point `value` but not a `precision` parameter ensure that the string representation consists of the smallest number of characters such that there is at least one digit before the radix point (if present) and parsing the representation using the corresponding `from_chars` function recovers `value` exactly. [*Note*: This guarantee applies only if `to_chars` and `from_chars` are executed on the same implementation. — *end note*] If there are several such representations, the representation with the smallest

difference from the floating-point argument value is chosen, resolving any remaining ties using rounding according to `round_to_nearest` (**??**).

3 The functions taking a `chars_format` parameter determine the conversion specifier for `printf` as follows: The conversion specifier is `f` if `fmt` is `chars_format::fixed`, `e` if `fmt` is `chars_format::scientific`, `a` (without leading `"0x"` in the result) if `fmt` is `chars_format::hex`, and `g` if `fmt` is `chars_format::general`.

```
to_chars_result to_chars(char* first, char* last, see below value, int base = 10);
```

4 ~~*Requires:*~~ *Preconditions:* `base` has a value between 2 and 36 (inclusive).

5 *Effects:* The value of `value` is converted to a string of digits in the given base (with no redundant leading zeroes). Digits in the range 10..35 (inclusive) are represented as lowercase characters `a`..`z`. If `value` is less than zero, the representation starts with `'-'`.

6 *Throws:* Nothing.

7 *Remarks:* The implementation shall provide overloads for all signed and unsigned integer types and `char` as the type of the parameter `value`.

```
to_chars_result to_chars(char* first, char* last, float value);
to_chars_result to_chars(char* first, char* last, double value);
to_chars_result to_chars(char* first, char* last, long double value);
```

8 *Effects:* `value` is converted to a string in the style of `printf` in the `"C"` locale. The conversion specifier is `f` or `e`, chosen according to the requirement for a shortest representation (see above); a tie is resolved in favor of `f`.

9 *Throws:* Nothing.

```
to_chars_result to_chars(char* first, char* last, float value, chars_format fmt);
to_chars_result to_chars(char* first, char* last, double value, chars_format fmt);
to_chars_result to_chars(char* first, char* last, long double value, chars_format fmt);
```

10 ~~*Requires:*~~ *Preconditions:* `fmt` has the value of one of the enumerators of `chars_format`.

11 *Effects:* `value` is converted to a string in the style of `printf` in the `"C"` locale.

12 *Throws:* Nothing.

```
to_chars_result to_chars(char* first, char* last, float value,
                         chars_format fmt, int precision);
to_chars_result to_chars(char* first, char* last, double value,
                         chars_format fmt, int precision);
to_chars_result to_chars(char* first, char* last, long double value,
                         chars_format fmt, int precision);
```

13 ~~*Requires:*~~ *Preconditions:* `fmt` has the value of one of the enumerators of `chars_format`.

14 *Effects:* `value` is converted to a string in the style of `printf` in the `"C"` locale with the given precision.

15 *Throws:* Nothing.

See also: ISO C 7.21.6.1

20.19.3 Primitive numeric input conversion [charconv.from.chars]

1 All functions named `from_chars` analyze the string `[first, last)` for a pattern, where `[first, last)` is required to be a valid range. If no characters match the pattern, `value` is unmodified, the member `ptr` of the return value is `first` and the member `ec` is equal to `errc::invalid_argument`. [*Note*: If the pattern allows for an optional sign, but the string has no digit characters following the sign, no characters match the pattern. — *end note*] Otherwise, the characters matching the pattern are interpreted as a representation of a value of the type of `value`. The member `ptr` of the return value points to the first character not matching the pattern, or has the value `last` if all characters match. If the parsed value is not in the range representable by the type of `value`, `value` is unmodified and the member `ec` of the return value is equal to `errc::result_out_of_range`. Otherwise, `value` is set to the parsed value, after rounding according to `round_to_nearest` (**??**), and the member `ec` is value-initialized.

```
from_chars_result from_chars(const char* first, const char* last,
                             see below& value, int base = 10);
```

2 ~~*Requires:*~~ *Preconditions:* `base` has a value between 2 and 36 (inclusive).

3 *Effects:* The pattern is the expected form of the subject sequence in the `"C"` locale for the given nonzero base, as described for `strtol`, except that no `"0x"` or `"0X"` prefix shall appear if the value of `base` is 16, and except that `'-'` is the only sign that may appear, and only if `value` has a signed type.

4 *Throws:* Nothing.

5 *Remarks:* The implementation shall provide overloads for all signed and unsigned integer types and `char` as the referenced type of the parameter `value`.

```
from_chars_result from_chars(const char* first, const char* last, float& value,
                             chars_format fmt = chars_format::general);
from_chars_result from_chars(const char* first, const char* last, double& value,
                             chars_format fmt = chars_format::general);
from_chars_result from_chars(const char* first, const char* last, long double& value,
                             chars_format fmt = chars_format::general);
```

6 ~~*Requires:*~~ *Preconditions:* `fmt` has the value of one of the enumerators of `chars_format`.

7 *Effects:* The pattern is the expected form of the subject sequence in the `"C"` locale, as described for `strtod`, except that

(7.1) — the sign `'+'` may only appear in the exponent part;

(7.2) — if `fmt` has `chars_format::scientific` set but not `chars_format::fixed`, the otherwise optional exponent part shall appear;

(7.3) — if `fmt` has `chars_format::fixed` set but not `chars_format::scientific`, the optional exponent part shall not appear; and

(7.4) — if `fmt` is `chars_format::hex`, the prefix `"0x"` or `"0X"` is assumed. [*Example*: The string `0x123` is parsed to have the value `0` with remaining characters `x123`. — *end example*]

In any case, the resulting `value` is one of at most two floating-point values closest to the value of the string matching the pattern.

8 *Throws:* Nothing.

See also: ISO C 7.22.1.3, 7.22.1.4

20.20 Formatting [format]

20.20.1 Header <format> synopsis [format.syn]

```
namespace std {
  // 20.20.3, formatting functions
  template<class... Args>
    string format(string_view fmt, const Args&... args);
  template<class... Args>
    wstring format(wstring_view fmt, const Args&... args);
  template<class... Args>
    string format(const locale& loc, string_view fmt, const Args&... args);
  template<class... Args>
    wstring format(const locale& loc, wstring_view fmt, const Args&... args);

  string vformat(string_view fmt, format_args args);
  wstring vformat(wstring_view fmt, wformat_args args);
  string vformat(const locale& loc, string_view fmt, format_args args);
  wstring vformat(const locale& loc, wstring_view fmt, wformat_args args);

  template<class Out, class... Args>
    Out format_to(Out out, string_view fmt, const Args&... args);
  template<class Out, class... Args>
    Out format_to(Out out, wstring_view fmt, const Args&... args);
  template<class Out, class... Args>
    Out format_to(Out out, const locale& loc, string_view fmt, const Args&... args);
  template<class Out, class... Args>
    Out format_to(Out out, const locale& loc, wstring_view fmt, const Args&... args);

  template<class Out>
    Out vformat_to(Out out, string_view fmt, format_args_t<Out, char> args);
```

```
template<class Out>
  Out vformat_to(Out out, wstring_view fmt, format_args_t<Out, wchar_t> args);
template<class Out>
  Out vformat_to(Out out, const locale& loc, string_view fmt,
                 format_args_t<Out, char> args);
template<class Out>
  Out vformat_to(Out out, const locale& loc, wstring_view fmt,
                 format_args_t<Out, wchar_t> args);

template<class Out> struct format_to_n_result {
  Out out;
  iter_difference_t<Out> size;
};
template<class Out, class... Args>
  format_to_n_result<Out> format_to_n(Out out, iter_difference_t<Out> n,
                                      string_view fmt, const Args&... args);
template<class Out, class... Args>
  format_to_n_result<Out> format_to_n(Out out, iter_difference_t<Out> n,
                                      wstring_view fmt, const Args&... args);
template<class Out, class... Args>
  format_to_n_result<Out> format_to_n(Out out, iter_difference_t<Out> n,
                                      const locale& loc, string_view fmt,
                                      const Args&... args);
template<class Out, class... Args>
  format_to_n_result<Out> format_to_n(Out out, iter_difference_t<Out> n,
                                      const locale& loc, wstring_view fmt,
                                      const Args&... args);

template<class... Args>
  size_t formatted_size(string_view fmt, const Args&... args);
template<class... Args>
  size_t formatted_size(wstring_view fmt, const Args&... args);
template<class... Args>
  size_t formatted_size(const locale& loc, string_view fmt, const Args&... args);
template<class... Args>
  size_t formatted_size(const locale& loc, wstring_view fmt, const Args&... args);

// 20.20.4, formatter
template<class T, class charT = char> struct formatter;

// 20.20.4.3, class template basic_format_parse_context
template<class charT> class basic_format_parse_context;
using format_parse_context = basic_format_parse_context<char>;
using wformat_parse_context = basic_format_parse_context<wchar_t>;

template<class Out, class charT> class basic_format_context;
using format_context = basic_format_context<unspecified, char>;
using wformat_context = basic_format_context<unspecified, wchar_t>;

// 20.20.5, arguments
// 20.20.5.1, class template basic_format_arg
template<class Context> class basic_format_arg;

template<class Visitor, class Context>
  see below visit_format_arg(Visitor&& vis, basic_format_arg<Context> arg);

// 20.20.5.2, class template format-arg-store
template<class Context, class... Args> struct format-arg-store;       // exposition only

template<class Context = format_context, class... Args>
  format-arg-store<Context, Args...>
    make_format_args(const Args&... args);
```

```
  template<class... Args>
    format-arg-store<wformat_context, Args...>
      make_wformat_args(const Args&... args);

  // 20.20.5.3, class template basic_format_args
  template<class Context> class basic_format_args;
  using format_args = basic_format_args<format_context>;
  using wformat_args = basic_format_args<wformat_context>;

  template<class Out, class charT>
    using format_args_t = basic_format_args<basic_format_context<Out, charT>>;

  // 20.20.6, class format_error
  class format_error;
}
```

20.20.2 Format string [format.string]

20.20.2.1 In general [format.string.general]

1 A *format string* is a (possibly empty) sequence of *replacement fields*, *escape sequences*, and characters other than `{` and `}`. Let `charT` be the character type of the format string. Each character that is not part of a replacement field or an escape sequence is copied unchanged to the output. An escape sequence is one of `{{` or `}}`. It is replaced with `{` or `}`, respectively, in the output. The syntax of replacement fields is as follows:

replacement-field:
    `{` *arg-id*$_{opt}$ *format-specifier*$_{opt}$ `}`

arg-id:
    `0`
    positive-integer

positive-integer:
    nonzero-digit
    positive-integer digit

nonnegative-integer:
    digit
    nonnegative-integer digit

nonzero-digit: one of
    `1 2 3 4 5 6 7 8 9`

digit: one of
    `0 1 2 3 4 5 6 7 8 9`

format-specifier:
    `:` *format-spec*

format-spec:
    as specified by the `formatter` specialization for the argument type

2 The *arg-id* field specifies the index of the argument in `args` whose value is to be formatted and inserted into the output instead of the replacement field. The optional *format-specifier* field explicitly specifies a format for the replacement value.

3 [*Example*:

```
string s = format("{0}-{{", 8);          // value of s is "8-{"
```

— *end example*]

4 If all *arg-id*s in a format string are omitted (including those in the *format-spec*, as interpreted by the corresponding `formatter` specialization), argument indices 0, 1, 2, ... will automatically be used in that order. If some *arg-id*s are omitted and some are present, the string is not a format string. [*Note*: A format string cannot contain a mixture of automatic and manual indexing. — *end note*] [*Example*:

```
string s0 = format("{} to {}",   "a", "b"); // OK, automatic indexing
string s1 = format("{1} to {0}", "a", "b"); // OK, manual indexing
string s2 = format("{0} to {}",  "a", "b"); // not a format string (mixing automatic and manual indexing),
                                            // throws format_error
string s3 = format("{} to {1}",  "a", "b"); // not a format string (mixing automatic and manual indexing),
```

```
                                        // throws format_error
```

— *end example*]

5 The *format-spec* field contains *format specifications* that define how the value should be presented. Each type can define its own interpretation of the *format-spec* field. [*Example*:

(5.1) — For arithmetic, pointer, and string types the *format-spec* is interpreted as a *std-format-spec* as described in (20.20.2.2).

(5.2) — For chrono types the *format-spec* is interpreted as a *chrono-format-spec* as described in (**??**).

(5.3) — For user-defined `formatter` specializations, the behavior of the `parse` member function determines how the *format-spec* is interpreted.

— *end example*]

20.20.2.2 Standard format specifiers [format.string.std]

1 Each `formatter` specializations described in 20.20.4.2 for fundamental and string types interprets *format-spec* as a *std-format-spec*. [*Note*: The format specification can be used to specify such details as field width, alignment, padding, and decimal precision. Some of the formatting options are only supported for arithmetic types. — *end note*] The syntax of format specifications is as follows:

std-format-spec:
fill-and-align$_{opt}$ *sign*$_{opt}$ `#`$_{opt}$ `0`$_{opt}$ *width*$_{opt}$ *precision*$_{opt}$ `L`$_{opt}$ *type*$_{opt}$

fill-and-align:
fill$_{opt}$ *align*

fill:
any character other than `{` or `}`

align: one of
`<` `>` `^`

sign: one of
`+` `-` space

width:
positive-integer
`{` *arg-id* `}`

precision:
`.` *nonnegative-integer*
`.` `{` *arg-id* `}`

type: one of
`a` `A` `b` `B` `c` `d` `e` `E` `f` `F` `g` `G` `o` `p` `s` `x` `X`

2 [*Note*: The *fill* character can be any character other than `{` or `}`. The presence of a fill character is signaled by the character following it, which must be one of the alignment options. If the second character of *std-format-spec* is not a valid alignment option, then it is assumed that both the fill character and the alignment option are absent. — *end note*]

3 The meaning of the various alignment options is as specified in Table 57. [*Example*:

```
char c = 120;
string s0 = format("{:6}", 42);          // value of s0 is "    42"
string s1 = format("{:6}", 'x');         // value of s1 is "x     "
string s2 = format("{:*<6}", 'x');       // value of s2 is "x*****"
string s3 = format("{:*>6}", 'x');       // value of s3 is "*****x"
string s4 = format("{:*^6}", 'x');       // value of s4 is "**x***"
string s5 = format("{:6d}", c);          // value of s5 is "   120"
string s6 = format("{:6}", true);        // value of s6 is "true  "
```

— *end example*] [*Note*: Unless a minimum field width is defined, the field width is determined by the size of the content and the alignment option has no effect. — *end note*]

4 The *sign* option is only valid for arithmetic types other than `charT` and `bool` or when an integer presentation type is specified. The meaning of the various options is as specified in Table 58.

5 The *sign* option applies to floating-point infinity and NaN. [*Example*:

```
double inf = numeric_limits<double>::infinity();
```

Table 57: Meaning of *align* options [tab:format.align]

Option	Meaning
`<`	Forces the field to be left-aligned within the available space. This is the default for non-arithmetic types, `charT`, and `bool`, unless an integer presentation type is specified.
`>`	Forces the field to be right-aligned within the available space. This is the default for arithmetic types other than `charT` and `bool` or when an integer presentation type is specified.
`^`	Forces the field to be centered within the available space by inserting $\lfloor \frac{n}{2} \rfloor$ characters before and $\lceil \frac{n}{2} \rceil$ characters after the value, where n is the total number of fill characters to insert.

Table 58: Meaning of *sign* options [tab:format.sign]

Option	Meaning
`+`	Indicates that a sign should be used for both non-negative and negative numbers.
`-`	Indicates that a sign should be used only for negative numbers (this is the default behavior).
space	Indicates that a leading space should be used for non-negative numbers, and a minus sign for negative numbers.

```
double nan = numeric_limits<double>::quiet_NaN();
string s0 = format("{0:},{0:+},{0:-},{0: }", 1);        // value of s0 is "1,+1,1, 1"
string s1 = format("{0:},{0:+},{0:-},{0: }", -1);       // value of s1 is "-1,-1,-1,-1"
string s2 = format("{0:},{0:+},{0:-},{0: }", inf);      // value of s2 is "inf,+inf,inf, inf"
string s3 = format("{0:},{0:+},{0:-},{0: }", nan);      // value of s3 is "nan,+nan,nan, nan"
```

— *end example*]

6 The `#` option causes the *alternate form* to be used for the conversion. This option is only valid for arithmetic types other than `charT` and `bool` or when an integer presentation type is specified. For integral types, the alternate form adds the base prefix (if any) specified in Table 60 to the output value. For floating-point types, the alternate form causes the result of the conversion to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it. In addition, for `g` and `G` conversions, trailing zeros are not removed from the result.

7 The *positive-integer* in *width* is a decimal integer defining the minimum field width. If *width* is not specified, there is no minimum field width, and the field width is determined based on the content of the field.

8 A zero (`0`) character preceding the *width* field pads the field with leading zeros (following any indication of sign or base) to the field width, except when applied to an infinity or NaN. This option is only valid for arithmetic types other than `charT` and `bool` or when an integer presentation type is specified. If the `0` character and an *align* option both appear, the `0` character is ignored. [*Example*:

```
char c = 120;
string s1 = format("{:+06d}", c);       // value of s1 is "+00120"
string s2 = format("{:#06x}", 0xa);     // value of s2 is "0x000a"
string s3 = format("{:<06}", -42);      // value of s3 is "-42    " (0 is ignored because of < alignment)
```

— *end example*]

9 The *nonnegative-integer* in *precision* is a decimal integer defining the precision or maximum field size. It can only be used with floating-point and string types. For floating-point types this field specifies the formatting precision. For string types it specifies how many characters will be used from the string.

10 When the `L` option is used, the form used for the conversion is called the *locale-specific form*. The `L` option is only valid for arithmetic types, and its effect depends upon the type.

(10.1) — For integral types, the locale-specific form causes the context's locale to be used to insert the appropriate digit group separator characters.

(10.2) — For floating-point types, the locale-specific form causes the context's locale to be used to insert the appropriate digit group and radix separator characters.

(10.3) — For the textual representation of `bool`, the locale-specific form causes the context's locale to be used to insert the appropriate string as if obtained with `numpunct::truename` or `numpunct::falsename`.

11 The *type* determines how the data should be presented.

12 The available string presentation types are specified in Table 59.

Table 59: Meaning of *type* options for strings [tab:format.type.string]

Type	Meaning
none, `s`	Copies the string to the output.

13 The meaning of some non-string presentation types is defined in terms of a call to `to_chars`. In such cases, let `[first, last)` be a range large enough to hold the `to_chars` output and `value` be the formatting argument value. Formatting is done as if by calling `to_chars` as specified and copying the output through the output iterator of the format context. [*Note*: Additional padding and adjustments are performed prior to copying the output through the output iterator as specified by the format specifiers. — *end note*]

14 The available integer presentation types for integral types other than `bool` and `charT` are specified in Table 60. [*Example*:

```
string s0 = format("{}", 42);                    // value of s0 is "42"
string s1 = format("{0:b} {0:d} {0:o} {0:x}", 42);  // value of s1 is "101010 42 52 2a"
string s2 = format("{0:#x} {0:#X}", 42);         // value of s2 is "0x2a 0X2A"
string s3 = format("{:L}", 1234);                // value of s3 might be "1,234"
                                                 // (depending on the locale)
```

— *end example*]

Table 60: Meaning of *type* options for integer types [tab:format.type.int]

Type	Meaning
`b`	`to_chars(first, last, value, 2)`; the base prefix is `0b`.
`B`	The same as `b`, except that the base prefix is `0B`.
`c`	Copies the character `static_cast<charT>(value)` to the output. Throws `format_error` if `value` is not in the range of representable values for `charT`.
`d`	`to_chars(first, last, value)`.
`o`	`to_chars(first, last, value, 8)`; the base prefix is `0` if `value` is nonzero and is empty otherwise.
`x`	`to_chars(first, last, value, 16)`; the base prefix is `0x`.
`X`	The same as `x`, except that it uses uppercase letters for digits above 9 and the base prefix is `0X`.
none	The same as `d`. [*Note*: If the formatting argument type is `charT` or `bool`, the default is instead `c` or `s`, respectively. — *end note*]

15 The available `charT` presentation types are specified in Table 61.

Table 61: Meaning of *type* options for `charT` [tab:format.type.char]

Type	Meaning
none, `c`	Copies the character to the output.
`b`, `B`, `d`, `o`, `x`, `X`	As specified in Table 60.

16 The available `bool` presentation types are specified in Table 62.

17 The available floating-point presentation types and their meanings for values other than infinity and NaN are specified in Table 63. For lower-case presentation types, infinity and NaN are formatted as `inf` and `nan`, respectively. For upper-case presentation types, infinity and NaN are formatted as `INF` and `NAN`, respectively. [*Note*: In either case, a sign is included if indicated by the *sign* option. — *end note*]

18 The available pointer presentation types and their mapping to `to_chars` are specified in Table 64. [*Note*: Pointer presentation types also apply to `nullptr_t`. — *end note*]

Table 62: Meaning of *type* options for `bool` [tab:format.type.bool]

Type	Meaning
none, `s`	Copies textual representation, either `true` or `false`, to the output.
`b`, `B`, `c`, `d`, `o`, `x`, `X`	As specified in Table 60 for the value `static_cast<unsigned char>(value)`.

Table 63: Meaning of *type* options for floating-point types [tab:format.type.float]

Type	Meaning

`a`	If *precision* is specified, equivalent to
	`to_chars(first, last, value, chars_format::hex, precision)`
	where `precision` is the specified formatting precision; equivalent to
	`to_chars(first, last, value, chars_format::hex)`
	otherwise.

`A`	The same as `a`, except that it uses uppercase letters for digits above 9 and `P` to indicate the exponent.
`e`	Equivalent to `to_chars(first, last, value, chars_format::scientific, precision)` where `precision` is the specified formatting precision, or `6` if *precision* is not specified.
`E`	The same as `e`, except that it uses `E` to indicate exponent.
`f`, `F`	Equivalent to `to_chars(first, last, value, chars_format::fixed, precision)` where `precision` is the specified formatting precision, or `6` if *precision* is not specified.
`g`	Equivalent to `to_chars(first, last, value, chars_format::general, precision)` where `precision` is the specified formatting precision, or `6` if *precision* is not specified.
`G`	The same as `g`, except that it uses `E` to indicate exponent.

none	If *precision* is specified, equivalent to
	`to_chars(first, last, value, chars_format::general, precision)`
	where `precision` is the specified formatting precision; equivalent to
	`to_chars(first, last, value)`
	otherwise.

Table 64: Meaning of *type* options for pointer types [tab:format.type.ptr]

Type	Meaning
none, `p`	If `uintptr_t` is defined, `to_chars(first, last, reinterpret_cast<uintptr_t>(value), 16)` with the prefix `0x` added to the output; otherwise, implementation-defined.

20.20.3 Formatting functions [format.functions]

1 In the description of the functions, operator `+` is used for some of the iterator categories for which it does not have to be defined. In these cases the semantics of `a + n` are the same as in **??**.

```
template<class... Args>
  string format(string_view fmt, const Args&... args);
```

2 *Effects:* Equivalent to:

```
return vformat(fmt, make_format_args(args...));
```

```
template<class... Args>
  wstring format(wstring_view fmt, const Args&... args);
```

3 *Effects:* Equivalent to:

```
return vformat(fmt, make_wformat_args(args...));
```

```
template<class... Args>
  string format(const locale& loc, string_view fmt, const Args&... args);
```

4 *Effects:* Equivalent to:

```
return vformat(loc, fmt, make_format_args(args...));
```

```
template<class... Args>
  wstring format(const locale& loc, wstring_view fmt, const Args&... args);
```

5 *Effects:* Equivalent to:

```
return vformat(loc, fmt, make_wformat_args(args...));
```

```
string vformat(string_view fmt, format_args args);
wstring vformat(wstring_view fmt, wformat_args args);
string vformat(const locale& loc, string_view fmt, format_args args);
wstring vformat(const locale& loc, wstring_view fmt, wformat_args args);
```

6 *Returns:* A string object holding the character representation of formatting arguments provided by `args` formatted according to specifications given in `fmt`. If present, `loc` is used for locale-specific formatting.

7 *Throws:* `format_error` if `fmt` is not a format string.

```
template<class Out, class... Args>
  Out format_to(Out out, string_view fmt, const Args&... args);
template<class Out, class... Args>
  Out format_to(Out out, wstring_view fmt, const Args&... args);
```

8 *Effects:* Equivalent to:

```
using context = basic_format_context<Out, decltype(fmt)::value_type>;
return vformat_to(out, fmt, {make_format_args<context>(args...)});
```

```
template<class Out, class... Args>
  Out format_to(Out out, const locale& loc, string_view fmt, const Args&... args);
template<class Out, class... Args>
  Out format_to(Out out, const locale& loc, wstring_view fmt, const Args&... args);
```

9 *Effects:* Equivalent to:

```
using context = basic_format_context<Out, decltype(fmt)::value_type>;
return vformat_to(out, loc, fmt, {make_format_args<context>(args...)});
```

```
template<class Out>
  Out vformat_to(Out out, string_view fmt, format_args_t<Out, char> args);
template<class Out>
  Out vformat_to(Out out, wstring_view fmt, format_args_t<Out, wchar_t> args);
template<class Out>
  Out vformat_to(Out out, const locale& loc, string_view fmt,
                 format_args_t<Out, char> args);
template<class Out>
  Out vformat_to(Out out, const locale& loc, wstring_view fmt,
                 format_args_t<Out, wchar_t> args);
```

10 Let `charT` be `decltype(fmt)::value_type`.

11 *Constraints:* `Out` satisfies `output_iterator<const charT&>`.

12 *Preconditions:* `Out` models `output_iterator<const charT&>`.

13 *Effects:* Places the character representation of formatting the arguments provided by `args`, formatted according to the specifications given in `fmt`, into the range `[out, out + N)`, where `N` is `formatted_size(fmt, args...)` for the functions without a `loc` parameter and `formatted_size(loc, fmt, args...)` for the functions with a `loc` parameter. If present, `loc` is used for locale-specific formatting.

14 *Returns:* `out + N`.

15 *Throws:* `format_error` if `fmt` is not a format string.

```
template<class Out, class... Args>
  format_to_n_result<Out> format_to_n(Out out, iter_difference_t<Out> n,
                                      string_view fmt, const Args&... args);
```

```
template<class Out, class... Args>
  format_to_n_result<Out> format_to_n(Out out, iter_difference_t<Out> n,
                                      wstring_view fmt, const Args&... args);
template<class Out, class... Args>
  format_to_n_result<Out> format_to_n(Out out, iter_difference_t<Out> n,
                                      const locale& loc, string_view fmt,
                                      const Args&... args);
template<class Out, class... Args>
  format_to_n_result<Out> format_to_n(Out out, iter_difference_t<Out> n,
                                      const locale& loc, wstring_view fmt,
                                      const Args&... args);
```

16 Let

(16.1) — `charT` be `decltype(fmt)::value_type`,

(16.2) — `N` be `formatted_size(fmt, args...)` for the functions without a `loc` parameter and `formatted_size(loc, fmt, args...)` for the functions with a `loc` parameter, and

(16.3) — `M` be `clamp(n, 0, N)`.

17 *Constraints:* `Out` satisfies `output_iterator<const charT&>`.

18 *Preconditions:* `Out` models `output_iterator<const charT&>`, and `formatter<`T_i`, charT>` meets the *Formatter* requirements (20.20.4.1) for each T_i in `Args`.

19 *Effects:* Places the first `M` characters of the character representation of formatting the arguments provided by `args`, formatted according to the specifications given in `fmt`, into the range `[out, out + M)`. If present, `loc` is used for locale-specific formatting.

20 *Returns:* `{out + M, N}`.

21 *Throws:* `format_error` if `fmt` is not a format string.

```
template<class... Args>
  size_t formatted_size(string_view fmt, const Args&... args);
template<class... Args>
  size_t formatted_size(wstring_view fmt, const Args&... args);
template<class... Args>
  size_t formatted_size(const locale& loc, string_view fmt, const Args&... args);
template<class... Args>
  size_t formatted_size(const locale& loc, wstring_view fmt, const Args&... args);
```

22 Let `charT` be `decltype(fmt)::value_type`.

23 *Preconditions:* `formatter<`T_i`, charT>` meets the *Formatter* requirements (20.20.4.1) for each T_i in `Args`.

24 *Returns:* The number of characters in the character representation of formatting arguments `args` formatted according to specifications given in `fmt`. If present, `loc` is used for locale-specific formatting.

25 *Throws:* `format_error` if `fmt` is not a format string.

20.20.4 Formatter [format.formatter]

20.20.4.1 Formatter requirements [formatter.requirements]

1 A type `F` meets the *Formatter* requirements if:

(1.1) — it meets the

(1.1.1) — *Cpp17DefaultConstructible* (Table **??**),

(1.1.2) — *Cpp17CopyConstructible* (Table **??**),

(1.1.3) — *Cpp17CopyAssignable* (Table **??**), and

(1.1.4) — *Cpp17Destructible* (Table **??**)

requirements,

(1.2) — it is swappable (**??**) for lvalues, and

(1.3) — the expressions shown in Table 65 are valid and have the indicated semantics.

2 Given character type `charT`, output iterator type `Out`, and formatting argument type `T`, in Table 65:

(2.1) — `f` is a value of type `F`,

(2.2) — `u` is an lvalue of type `T`,

(2.3) — `t` is a value of a type convertible to (possibly const) `T`,

(2.4) — `PC` is `basic_format_parse_context<charT>`,

(2.5) — `FC` is `basic_format_context<Out, charT>`,

(2.6) — `pc` is an lvalue of type `PC`, and

(2.7) — `fc` is an lvalue of type `FC`.

`pc.begin()` points to the beginning of the *format-spec* (20.20.2) of the replacement field being formatted in the format string. If *format-spec* is empty then either `pc.begin() == pc.end()` or `*pc.begin() == '}'`.

Table 65: *Formatter* requirements [tab:formatter]

Expression	Return type	Requirement
`f.parse(pc)`	`PC::iterator`	Parses *format-spec* (20.20.2) for type `T` in the range `[pc.begin(), pc.end())` until the first unmatched character. Throws `format_error` unless the whole range is parsed or the unmatched character is `}`. [*Note*: This allows formatters to emit meaningful error messages. — *end note*] Stores the parsed format specifiers in `*this` and returns an iterator past the end of the parsed range.
`f.format(t, fc)`	`FC::iterator`	Formats `t` according to the specifiers stored in `*this`, writes the output to `fc.out()` and returns an iterator past the end of the output range. The output shall only depend on `t`, `fc.locale()`, and the range `[pc.begin(), pc.end())` from the last call to `f.parse(pc)`.
`f.format(u, fc)`	`FC::iterator`	As above, but does not modify `u`.

20.20.4.2 Formatter specializations [format.formatter.spec]

1 The functions defined in 20.20.3 use specializations of the class template `formatter` to format individual arguments.

2 Let `charT` be either `char` or `wchar_t`. Each specialization of `formatter` is either enabled or disabled, as described below. [*Note*: Enabled specializations meet the *Formatter* requirements, and disabled specializations do not. — *end note*] Each header that declares the template `formatter` provides the following enabled specializations:

(2.1) — The specializations

```
template<> struct formatter<char, char>;
template<> struct formatter<char, wchar_t>;
template<> struct formatter<wchar_t, wchar_t>;
```

(2.2) — For each `charT`, the string type specializations

```
template<> struct formatter<charT*, charT>;
template<> struct formatter<const charT*, charT>;
template<size_t N> struct formatter<const charT[N], charT>;
template<class traits, class Allocator>
  struct formatter<basic_string<charT, traits, Allocator>, charT>;
template<class traits>
  struct formatter<basic_string_view<charT, traits>, charT>;
```

(2.3) — For each `charT`, for each cv-unqualified arithmetic type `ArithmeticT` other than `char`, `wchar_t`, `char8_t`, `char16_t`, or `char32_t`, a specialization

```
template<> struct formatter<ArithmeticT, charT>;
```

(2.4) — For each `charT`, the pointer type specializations

```
template<> struct formatter<nullptr_t, charT>;
template<> struct formatter<void*, charT>;
template<> struct formatter<const void*, charT>;
```

The `parse` member functions of these formatters interpret the format specification as a *std-format-spec* as described in 20.20.2.2. [*Note*: Specializations such as `formatter<wchar_t, char>` and `formatter<const char*, wchar_t>` that would require implicit multibyte / wide string or character conversion are disabled. — *end note*]

3 For any types `T` and `charT` for which neither the library nor the user provides an explicit or partial specialization of the class template `formatter`, `formatter<T, charT>` is disabled.

4 If the library provides an explicit or partial specialization of `formatter<T, charT>`, that specialization is enabled except as noted otherwise.

5 If `F` is a disabled specialization of `formatter`, these values are `false`:

(5.1) — `is_default_constructible_v<F>`,

(5.2) — `is_copy_constructible_v<F>`,

(5.3) — `is_move_constructible_v<F>`,

(5.4) — `is_copy_assignable_v<F>`, and

(5.5) — `is_move_assignable_v<F>`.

6 An enabled specialization `formatter<T, charT>` meets the *Formatter* requirements (20.20.4.1). [*Example*:

```
#include <format>

enum color { red, green, blue };
const char* color_names[] = { "red", "green", "blue" };

template<> struct std::formatter<color> : std::formatter<const char*> {
  auto format(color c, format_context& ctx) {
    return formatter<const char*>::format(color_names[c], ctx);
  }
};

struct err {};

std::string s0 = std::format("{}", 42);          // OK, library-provided formatter
std::string s1 = std::format("{}", L"foo");      // ill-formed: disabled formatter
std::string s2 = std::format("{}", red);         // OK, user-provided formatter
std::string s3 = std::format("{}", err{});       // ill-formed: disabled formatter
```

— *end example*]

20.20.4.3 Class template `basic_format_parse_context` [format.parse.ctx]

```
namespace std {
  template<class charT>
  class basic_format_parse_context {
  public:
    using char_type = charT;
    using const_iterator = typename basic_string_view<charT>::const_iterator;
    using iterator = const_iterator;

  private:
    iterator begin_;                                    // exposition only
    iterator end_;                                      // exposition only
    enum indexing { unknown, manual, automatic };       // exposition only
    indexing indexing_;                                 // exposition only
    size_t next_arg_id_;                                // exposition only
    size_t num_args_;                                   // exposition only
```

```
  public:
    constexpr explicit basic_format_parse_context(basic_string_view<charT> fmt,
                                                  size_t num_args = 0) noexcept;
    basic_format_parse_context(const basic_format_parse_context&) = delete;
    basic_format_parse_context& operator=(const basic_format_parse_context&) = delete;

    constexpr const_iterator begin() const noexcept;
    constexpr const_iterator end() const noexcept;
    constexpr void advance_to(const_iterator it);

    constexpr size_t next_arg_id();
    constexpr void check_arg_id(size_t id);
  };
}
```

1 An instance of `basic_format_parse_context` holds the format string parsing state consisting of the format string range being parsed and the argument counter for automatic indexing.

```
constexpr explicit basic_format_parse_context(basic_string_view<charT> fmt,
                                              size_t num_args = 0) noexcept;
```

2 *Effects:* Initializes `begin_` with `fmt.begin()`, `end_` with `fmt.end()`, `indexing_` with `unknown`, `next_arg_id_` with `0`, and `num_args_` with `num_args`.

```
constexpr const_iterator begin() const noexcept;
```

3 *Returns:* `begin_`.

```
constexpr const_iterator end() const noexcept;
```

4 *Returns:* `end_`.

```
constexpr void advance_to(const_iterator it);
```

5 *Preconditions:* `end()` is reachable from `it`.

6 *Effects:* Equivalent to: `begin_ = it;`

```
constexpr size_t next_arg_id();
```

7 *Effects:* If `indexing_ != manual`, equivalent to:

```
if (indexing_ == unknown)
  indexing_ = automatic;
return next_arg_id_++;
```

8 *Throws:* `format_error` if `indexing_ == manual` which indicates mixing of automatic and manual argument indexing.

```
constexpr void check_arg_id(size_t id);
```

9 *Effects:* If `indexing_ != automatic`, equivalent to:

```
if (indexing_ == unknown)
  indexing_ = manual;
```

10 *Throws:* `format_error` if `indexing_ == automatic` which indicates mixing of automatic and manual argument indexing.

11 *Remarks:* Call expressions where `id >= num_args_` are not core constant expressions (**??**).

20.20.4.4 Class template `basic_format_context` [format.context]

```
namespace std {
  template<class Out, class charT>
  class basic_format_context {
    basic_format_args<basic_format_context> args_;       // exposition only
    Out out_;                                            // exposition only

  public:
    using iterator = Out;
    using char_type = charT;
    template<class T> using formatter_type = formatter<T, charT>;
```

```
    basic_format_arg<basic_format_context> arg(size_t id) const;
    std::locale locale();

    iterator out();
    void advance_to(iterator it);
  };
}
```

1 An instance of `basic_format_context` holds formatting state consisting of the formatting arguments and the output iterator.

2 `Out` shall model `output_iterator<const charT&>`.

3 `format_context` is an alias for a specialization of `basic_format_context` with an output iterator that appends to `string`, such as `back_insert_iterator<string>`. Similarly, `wformat_context` is an alias for a specialization of `basic_format_context` with an output iterator that appends to `wstring`.

4 [*Note*: For a given type `charT`, implementations are encouraged to provide a single instantiation of `basic_format_context` for appending to `basic_string<charT>`, `vector<charT>`, or any other container with contiguous storage by wrapping those in temporary objects with a uniform interface (such as a `span<charT>`) and polymorphic reallocation. — *end note*]

```
basic_format_arg<basic_format_context> arg(size_t id) const;
```

5 *Returns:* `args_.get(id)`.

```
std::locale locale();
```

6 *Returns:* The locale passed to the formatting function if the latter takes one, and `std::locale()` otherwise.

```
iterator out();
```

7 *Returns:* `out_`.

```
void advance_to(iterator it);
```

8 *Effects:* Equivalent to: `out_ = it;`

[*Example*:

```
struct S { int value; };

template<> struct std::formatter<S> {
  size_t width_arg_id = 0;

  // Parses a width argument id in the format { digit }.
  constexpr auto parse(format_parse_context& ctx) {
    auto iter = ctx.begin();
    auto get_char = [&]() { return iter != ctx.end() ? *iter : 0; };
    if (get_char() != '{')
      return iter;
    ++iter;
    char c = get_char();
    if (!isdigit(c) || (++iter, get_char()) != '}')
      throw format_error("invalid format");
    width_arg_id = c - '0';
    ctx.check_arg_id(width_arg_id);
    return ++iter;
  }

  // Formats an S with width given by the argument width_arg_id.
  auto format(S s, format_context& ctx) {
    int width = visit_format_arg([](auto value) -> int {
      if constexpr (!is_integral_v<decltype(value)>)
        throw format_error("width is not integral");
      else if (value < 0 || value > numeric_limits<int>::max())
        throw format_error("invalid width");
```

```
      else
        return value;
      }, ctx.arg(width_arg_id));
    return format_to(ctx.out(), "{0:x<{1}}", s.value, width);
  }
};

std::string s = std::format("{0:{1}}", S{42}, 10);  // value of s is "xxxxxxxx42"
```

— *end example*]

20.20.5 Arguments [format.arguments]

20.20.5.1 Class template `basic_format_arg` [format.arg]

```
namespace std {
  template<class Context>
  class basic_format_arg {
  public:
    class handle;

  private:
    using char_type = typename Context::char_type;                          // exposition only

    variant<monostate, bool, char_type,
            int, unsigned int, long long int, unsigned long long int,
            float, double, long double,
            const char_type*, basic_string_view<char_type>,
            const void*, handle> value;                                     // exposition only

    template<class T> explicit basic_format_arg(const T& v) noexcept;       // exposition only
    explicit basic_format_arg(float n) noexcept;                            // exposition only
    explicit basic_format_arg(double n) noexcept;                           // exposition only
    explicit basic_format_arg(long double n) noexcept;                      // exposition only
    explicit basic_format_arg(const char_type* s);                          // exposition only

    template<class traits>
      explicit basic_format_arg(
        basic_string_view<char_type, traits> s) noexcept;                   // exposition only

    template<class traits, class Allocator>
      explicit basic_format_arg(
        const basic_string<char_type, traits, Allocator>& s) noexcept;      // exposition only

    explicit basic_format_arg(nullptr_t) noexcept;                          // exposition only

    template<class T>
      explicit basic_format_arg(const T* p) noexcept;                       // exposition only

    template<class Visitor, class Ctx>
      friend auto visit_format_arg(Visitor&& vis,
                                   basic_format_arg<Ctx> arg);              // exposition only

    template<class Ctx, class... Args>
      friend format-arg-store<Ctx, Args...>
        make_format_args(const Args&... args);                              // exposition only

  public:
    basic_format_arg() noexcept;

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

1 An instance of `basic_format_arg` provides access to a formatting argument for user-defined formatters.

2 The behavior of a program that adds specializations of `basic_format_arg` is undefined.

```
basic_format_arg() noexcept;
```

3 *Postconditions:* `!(*this)`.

```
template<class T> explicit basic_format_arg(const T& v) noexcept;
```

4 *Constraints:* The template specialization

```
typename Context::template formatter_type<T>
```

meets the *Formatter* requirements (20.20.4.1). The extent to which an implementation determines that the specialization meets the *Formatter* requirements is unspecified, except that as a minimum the expression

```
typename Context::template formatter_type<T>()
  .format(declval<const T&>(), declval<Context&>())
```

shall be well-formed when treated as an unevaluated operand.

5 *Effects:*

(5.1) — if `T` is `bool` or `char_type`, initializes `value` with `v`; otherwise,

(5.2) — if `T` is `char` and `char_type` is `wchar_t`, initializes `value` with `static_cast<wchar_t>(v)`; otherwise,

(5.3) — if `T` is a signed integer type (**??**) and `sizeof(T) <= sizeof(int)`, initializes `value` with `static_cast<int>(v)`; otherwise,

(5.4) — if `T` is an unsigned integer type and `sizeof(T) <= sizeof(unsigned int)`, initializes `value` with `static_cast<unsigned int>(v)`; otherwise,

(5.5) — if `T` is a signed integer type and `sizeof(T) <= sizeof(long long int)`, initializes `value` with `static_cast<long long int>(v)`; otherwise,

(5.6) — if `T` is an unsigned integer type and `sizeof(T) <= sizeof(unsigned long long int)`, initializes `value` with `static_cast<unsigned long long int>(v)`; otherwise,

(5.7) — initializes `value` with `handle(v)`.

```
explicit basic_format_arg(float n) noexcept;
explicit basic_format_arg(double n) noexcept;
explicit basic_format_arg(long double n) noexcept;
```

6 *Effects:* Initializes `value` with `n`.

```
explicit basic_format_arg(const char_type* s);
```

7 *Preconditions:* `s` points to a NTCTS (**??**).

8 *Effects:* Initializes `value` with `s`.

```
template<class traits>
  explicit basic_format_arg(basic_string_view<char_type, traits> s) noexcept;
```

9 *Effects:* Initializes `value` with `s`.

```
template<class traits, class Allocator>
  explicit basic_format_arg(
    const basic_string<char_type, traits, Allocator>& s) noexcept;
```

10 *Effects:* Initializes `value` with `basic_string_view<char_type>(s.data(), s.size())`.

```
explicit basic_format_arg(nullptr_t) noexcept;
```

11 *Effects:* Initializes `value` with `static_cast<const void*>(nullptr)`.

```
template<class T> explicit basic_format_arg(const T* p) noexcept;
```

12 *Constraints:* `is_void_v<T>` is `true`.

13 *Effects:* Initializes `value` with `p`.

14 [*Note*: Constructing `basic_format_arg` from a pointer to a member is ill-formed unless the user provides an enabled specialization of `formatter` for that pointer to member type. — *end note*]

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

15 *Returns:* `!holds_alternative<monostate>(value)`.

16 The class `handle` allows formatting an object of a user-defined type.

```
namespace std {
  template<class Context>
  class basic_format_arg<Context>::handle {
    const void* ptr_;                                          // exposition only
    void (*format_)(basic_format_parse_context<char_type>&,
                    Context&, const void*);                    // exposition only

    template<class T> explicit handle(const T& val) noexcept;  // exposition only

    friend class basic_format_arg<Context>;                    // exposition only

  public:
    void format(basic_format_parse_context<char_type>&, Context& ctx) const;
  };
}
```

```
template<class T> explicit handle(const T& val) noexcept;
```

17 *Effects:* Initializes `ptr_` with `addressof(val)` and `format_` with

```
[](basic_format_parse_context<char_type>& parse_ctx,
   Context& format_ctx, const void* ptr) {
  typename Context::template formatter_type<T> f;
  parse_ctx.advance_to(f.parse(parse_ctx));
  format_ctx.advance_to(f.format(*static_cast<const T*>(ptr), format_ctx));
}
```

```
void format(basic_format_parse_context<char_type>& parse_ctx, Context& format_ctx) const;
```

18 *Effects:* Equivalent to: `format_(parse_ctx, format_ctx, ptr_);`

```
template<class Visitor, class Context>
  see below visit_format_arg(Visitor&& vis, basic_format_arg<Context> arg);
```

19 *Effects:* Equivalent to: `return visit(forward<Visitor>(vis), arg.value);`

20.20.5.2 Class template *format-arg-store* [format.arg.store]

```
namespace std {
  template<class Context, class... Args>
  struct format-arg-store {      // exposition only
    array<basic_format_arg<Context>, sizeof...(Args)> args;
  };
}
```

1 An instance of *format-arg-store* stores formatting arguments.

```
template<class Context = format_context, class... Args>
  format-arg-store<Context, Args...> make_format_args(const Args&... args);
```

2 *Preconditions:* The type `typename Context::template formatter_type<`T_i`>` meets the *Formatter* requirements (20.20.4.1) for each T_i in `Args`.

3 *Returns:* `{basic_format_arg<Context>(args)...}`.

```
template<class... Args>
  format-arg-store<wformat_context, Args...> make_wformat_args(const Args&... args);
```

4 *Effects:* Equivalent to: `return make_format_args<wformat_context>(args...);`

20.20.5.3 Class template basic_format_args [format.args]

```
namespace std {
  template<class Context>
  class basic_format_args {
    size_t size_;                              // exposition only
```

```
    const basic_format_arg<Context>* data_;      // exposition only

  public:
    basic_format_args() noexcept;

    template<class... Args>
      basic_format_args(const format-arg-store<Context, Args...>& store) noexcept;

    basic_format_arg<Context> get(size_t i) const noexcept;
  };
}
```

1 An instance of `basic_format_args` provides access to formatting arguments.

```
basic_format_args() noexcept;
```

2 *Effects:* Initializes `size_` with `0`.

```
template<class... Args>
  basic_format_args(const format-arg-store<Context, Args...>& store) noexcept;
```

3 *Effects:* Initializes `size_` with `sizeof...(Args)` and `data_` with `store.args.data()`.

```
basic_format_arg<Context> get(size_t i) const noexcept;
```

4 *Returns:* `i < size_ ? data_[i] : basic_format_arg<Context>()`.

[*Note*: Implementations are encouraged to optimize the representation of `basic_format_args` for small number of formatting arguments by storing indices of type alternatives separately from values and packing the former. — *end note*]

20.20.6 Class `format_error` [format.error]

```
namespace std {
  class format_error : public runtime_error {
  public:
    explicit format_error(const string& what_arg);
    explicit format_error(const char* what_arg);
  };
}
```

1 The class `format_error` defines the type of objects thrown as exceptions to report errors from the formatting library.

```
format_error(const string& what_arg);
```

2 *Postconditions:* `strcmp(what(), what_arg.c_str()) == 0`.

```
format_error(const char* what_arg);
```

3 *Postconditions:* `strcmp(what(), what_arg) == 0`.