Reply to: Billy O'Neal
Microsoft
bion@microsoft.com

Fixing US 313

Contents

1 Intended changes [Changes]

This document contains changes intended to address the national body comment US 313 and LWG 3156. The intended changes are:

- Move addressof out of [specialized.algorithms] so that it remains in [utilities].
- Move the section [specialized.algorithms] into [algorithms]. [Editor's note: The section is depicted as moved here, but moved-and-unedited text is left in black.]
- Rename [specialized.algorithms] version of the iterator concepts to add 'nothrow', and move those sections into [algorithms.requirements].
- Remove duplicated front matter from [specialized.algorithms].
- As a drive-by, several incorrect "shall"s are replaced.

20 General utilities library [utilities]

20.1 General [utilities.general]

20.2 Utility components [utility]

20.3 Compile-time integer sequences [intseq]

20.4 Pairs [pairs]

20.5 Tuples [tuple]

20.6 Optional objects [optional]

20.7 Variants [variant]

20.8 Storage for any type [any]

20.9 Bitsets [bitset]

20.10 Memory [memory]

20.10.1 In general [memory.general]

1 This subclause 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–??)~~ (20.10.3–20.10.11 and 25.10). 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;
```

```
// ??, uses_allocator
template<class T, class Alloc>
  inline constexpr bool uses_allocator_v = uses_allocator<T, Alloc>::value;

// ??, 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, addressof
template<class T>
  constexpr T* addressof(T& r) noexcept;
template<class T>
  const T* addressof(const T&&) = delete;

// 25.10, specialized algorithms
// 25.10.0.1, 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 NoThrowForwardIterator>
  void uninitialized_default_construct(NoThrowForwardIterator first, NoThrowForwardIterator last);
```

```
template<class ExecutionPolicy, class NoThrowForwardIterator>
  void uninitialized_default_construct(ExecutionPolicy&& exec,        // see ??
                                       NoThrowForwardIterator first, NoThrowForwardIterator last);
template<class NoThrowForwardIterator, class Size>
  NoThrowForwardIterator uninitialized_default_construct_n(NoThrowForwardIterator first, Size n);
template<class ExecutionPolicy, class NoThrowForwardIterator, class Size>
  NoThrowForwardIterator uninitialized_default_construct_n(ExecutionPolicy&& exec,   // see ??
                                                           NoThrowForwardIterator 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 NoThrowForwardIterator>
  void uninitialized_value_construct(NoThrowForwardIterator first, NoThrowForwardIterator last);
template<class ExecutionPolicy, class NoThrowForwardIterator>
  void uninitialized_value_construct(ExecutionPolicy&& exec,  // see ??
                                     NoThrowForwardIterator first, NoThrowForwardIterator last);
template<class NoThrowForwardIterator, class Size>
  NoThrowForwardIterator uninitialized_value_construct_n(NoThrowForwardIterator first, Size n);
template<class ExecutionPolicy, class NoThrowForwardIterator, class Size>
  NoThrowForwardIterator uninitialized_value_construct_n(ExecutionPolicy&& exec, // see ??
                                                         NoThrowForwardIterator 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 NoThrowForwardIterator>
  NoThrowForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                            NoThrowForwardIterator result);
template<class ExecutionPolicy, class InputIterator, class NoThrowForwardIterator>
  NoThrowForwardIterator uninitialized_copy(ExecutionPolicy&& exec,   // see ??
                                            InputIterator first, InputIterator last,
                                            NoThrowForwardIterator result);
template<class InputIterator, class Size, class NoThrowForwardIterator>
  NoThrowForwardIterator uninitialized_copy_n(InputIterator first, Size n,
                                              NoThrowForwardIterator result);
template<class ExecutionPolicy, class InputIterator, class Size, class NoThrowForwardIterator>
  NoThrowForwardIterator uninitialized_copy_n(ExecutionPolicy&& exec,        // see ??
                                              InputIterator first, Size n,
                                              NoThrowForwardIterator 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 NoThrowForwardIterator>
  NoThrowForwardIterator uninitialized_move(InputIterator first, InputIterator last,
                                            NoThrowForwardIterator result);
template<class ExecutionPolicy, class InputIterator, class NoThrowForwardIterator>
  NoThrowForwardIterator uninitialized_move(ExecutionPolicy&& exec,   // see ??
                                            InputIterator first, InputIterator last,
                                            NoThrowForwardIterator result);
template<class InputIterator, class Size, class NoThrowForwardIterator>
  pair<InputIterator, NoThrowForwardIterator> uninitialized_move_n(InputIterator first, Size n,
                                                                   NoThrowForwardIterator result);
template<class ExecutionPolicy, class InputIterator, class Size, class NoThrowForwardIterator>
  pair<InputIterator, NoThrowForwardIterator>
    uninitialized_move_n(ExecutionPolicy&& exec,              // see ??
                         InputIterator first, Size n, NoThrowForwardIterator 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 NoThrowForwardIterator, class T>
  void uninitialized_fill(NoThrowForwardIterator first, NoThrowForwardIterator last, const T& x);
template<class ExecutionPolicy, class NoThrowForwardIterator, class T>
  void uninitialized_fill(ExecutionPolicy&& exec,              // see ??
                          NoThrowForwardIterator first, NoThrowForwardIterator last, const T& x);
template<class NoThrowForwardIterator, class Size, class T>
  NoThrowForwardIterator uninitialized_fill_n(NoThrowForwardIterator first, Size n, const T& x);
```

```
template<class ExecutionPolicy, class NoThrowForwardIterator, class Size, class T>
  NoThrowForwardIterator uninitialized_fill_n(ExecutionPolicy&& exec,         // see ??
                                              NoThrowForwardIterator 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);
}

// 25.10.0.8, 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);
}

// 25.10.0.9, destroy
template<class T>
  constexpr void destroy_at(T* location);
template<class NoThrowForwardIterator>
  constexpr void destroy(NoThrowForwardIterator first, NoThrowForwardIterator last);
template<class ExecutionPolicy, class NoThrowForwardIterator>
  constexpr void destroy(ExecutionPolicy&& exec,                  // see ??
                         NoThrowForwardIterator first, NoThrowForwardIterator last);
template<class NoThrowForwardIterator, class Size>
  constexpr NoThrowForwardIterator destroy_n(NoThrowForwardIterator first, Size n);
template<class ExecutionPolicy, class NoThrowForwardIterator, class Size>
  constexpr NoThrowForwardIterator destroy_n(ExecutionPolicy&& exec, // see ??
                                             NoThrowForwardIterator 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;
}

// ??, 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);

// ??, class bad_weak_ptr
class bad_weak_ptr;

// ??, class template shared_ptr
template<class T> class shared_ptr;
```

```
// ??, 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[]

// ??, 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;

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

// ??, 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;

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

  // ??, 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);

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

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

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

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

  // ??, 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]

20.10.4 Pointer conversion [pointer.conversion]

20.10.5 Pointer safety [util.dynamic.safety]

20.10.6 Pointer alignment [ptr.align]

20.10.7 Allocator argument tag [allocator.tag]

20.10.8 `uses_allocator` [allocator.uses]

20.10.9 Allocator traits [allocator.traits]

20.10.10 The default allocator [default.allocator]

20.10.11 `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.

25 Algorithms library [algorithms]

25.1 General [algorithms.general]

25.2 Algorithms requirements [algorithms.requirements]

1 All of the algorithms are separated from the particular implementations of data structures and are parameterized by iterator types. Because of this, they can work with program-defined data structures, as long as these data structures have iterator types satisfying the assumptions on the algorithms.

2 The entities defined in the `std::ranges` namespace in this Clause 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.

[*Example*:

```
void foo() {
  using namespace std::ranges;
  std::vector<int> vec{1,2,3};
  find(begin(vec), end(vec), 2);        // #1
}
```

The function call expression at `#1` invokes `std::ranges::find`, not `std::find`, despite that (a) the iterator type returned from `begin(vec)` and `end(vec)` may be associated with namespace `std` and (b) `std::find` is more specialized (??) than `std::ranges::find` since the former requires its first two parameters to have the same type. — *end example*]

3 For purposes of determining the existence of data races, algorithms shall not modify objects referenced through an iterator argument unless the specification requires such modification.

4 Throughout this Clause, where the template parameters are not constrained, the names of template parameters are used to express type requirements.

(4.1) — If an algorithm's template parameter is named `InputIterator`, `InputIterator1`, or `InputIterator2`, the template argument shall meet the *Cpp17InputIterator* requirements (??).

(4.2) — If an algorithm's template parameter is named `OutputIterator`, `OutputIterator1`, or `OutputIterator2`, the template argument shall meet the *Cpp17OutputIterator* requirements (??).

(4.3) — If an algorithm's template parameter is named `ForwardIterator`, `ForwardIterator1`, or `ForwardIterator2`, the template argument shall meet the *Cpp17ForwardIterator* requirements (??).

(4.4) — If an algorithm's template parameter is named `NoThrowForwardIterator`, 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.

(4.5) — If an algorithm's template parameter is named `BidirectionalIterator`, `BidirectionalIterator1`, or `BidirectionalIterator2`, the template argument shall meet the *Cpp17BidirectionalIterator* requirements (??).

(4.6) — If an algorithm's template parameter is named `RandomAccessIterator`, `RandomAccessIterator1`, or `RandomAccessIterator2`, the template argument shall meet the *Cpp17RandomAccessIterator* requirements (??).

5 If an algorithm's *Effects:* element specifies that a value pointed to by any iterator passed as an argument is modified, then that algorithm has an additional type requirement: The type of that argument shall meet the requirements of a mutable iterator (??). [*Note*: This requirement does not affect arguments that are named `OutputIterator`, `OutputIterator1`, or `OutputIterator2`, because output iterators must always be mutable, nor does it affect arguments that are constrained, for which mutability requirements are expressed explicitly. — *end note*]

6 Both in-place and copying versions are provided for certain algorithms.[1] When such a version is provided for

1) The decision whether to include a copying version was usually based on complexity considerations. When the cost of doing the operation dominates the cost of copy, the copying version is not included. For example, `sort_copy` is not included because the cost of sorting is much more significant, and users might as well do `copy` followed by `sort`.

algorithm it is called *algorithm*_`copy`. Algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`).

7 When not otherwise constrained, the `Predicate` parameter is used whenever an algorithm expects a function object (20.14) that, when applied to the result of dereferencing the corresponding iterator, returns a value testable as `true`. In other words, if an algorithm takes `Predicate pred` as its argument and `first` as its iterator argument with value type `T`, it should work correctly in the construct `pred(*first)` contextually converted to `bool` (??). The function object `pred` shall not apply any non-constant function through the dereferenced iterator. Given a glvalue `u` of type (possibly `const`) `T` that designates the same object as `*first`, `pred(u)` shall be a valid expression that is equal to `pred(*first)`.

8 When not otherwise constrained, the `BinaryPredicate` parameter is used whenever an algorithm expects a function object that when applied to the result of dereferencing two corresponding iterators or to dereferencing an iterator and type `T` when `T` is part of the signature returns a value testable as `true`. In other words, if an algorithm takes `BinaryPredicate binary_pred` as its argument and `first1` and `first2` as its iterator arguments with respective value types `T1` and `T2`, it should work correctly in the construct `binary_pred(*first1, *first2)` contextually converted to `bool` (??). Unless otherwise specified, `BinaryPredicate` always takes the first iterator's `value_type` as its first argument, that is, in those cases when `T value` is part of the signature, it should work correctly in the construct `binary_pred(*first1, value)` contextually converted to `bool` (??). `binary_pred` shall not apply any non-constant function through the dereferenced iterators. Given a glvalue `u` of type (possibly `const`) `T1` that designates the same object as `*first1`, and a glvalue `v` of type (possibly `const`) `T2` that designates the same object as `*first2`, `binary_pred(u, *first2)`, `binary_pred(*first1, v)`, and `binary_pred(u, v)` shall each be a valid expression that is equal to `binary_pred(*first1, *first2)`, and `binary_pred(u, value)` shall be a valid expression that is equal to `binary_pred(*first1, value)`.

9 The parameters `UnaryOperation`, `BinaryOperation`, `BinaryOperation1`, and `BinaryOperation2` are used whenever an algorithm expects a function object (20.14).

10 [*Note*: Unless otherwise specified, algorithms that take function objects as arguments are permitted to copy those function objects freely. Programmers for whom object identity is important should consider using a wrapper class that points to a noncopied implementation object such as `reference_wrapper<T>` (??), or some equivalent solution. — *end note*]

11 When the description of an algorithm gives an expression such as `*first == value` for a condition, the expression shall evaluate to either `true` or `false` in boolean contexts.

12 In the description of the algorithms, 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 those of

```
auto tmp = a;
for (; n < 0; ++n) --tmp;
for (; n > 0; --n) ++tmp;
return tmp;
```

Similarly, operator `-` is used for some combinations of iterators and sentinel types for which it does not have to be defined. If `[a, b)` denotes a range, the semantics of `b - a` in these cases are the same as those of

```
iter_difference_t<remove_reference_t<decltype(a)>> n = 0;
for (auto tmp = a; tmp != b; ++tmp) ++n;
return n;
```

and if `[b, a)` denotes a range, the same as those of

```
iter_difference_t<remove_reference_t<decltype(b)>> n = 0;
for (auto tmp = b; tmp != a; ++tmp) --n;
return n;
```

13 In the description of algorithm return values, a sentinel value `s` denoting the end of a range `[i, s)` is sometimes returned where an iterator is expected. In these cases, the semantics are as if the sentinel is converted into an iterator using `ranges::next(i, s)`.

14 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 in namespace `ranges` that takes separate iterator and sentinel arguments.

15 The number and order of deducible template parameters for algorithm declarations are unspecified, except where explicitly stated otherwise. [*Note*: Consequently, the algorithms may not be called with explicitly-specified template argument lists. — *end note*]

16 The class templates binary_transform_result, for_each_result, minmax_result, mismatch_result, next_permutation_result, copy_result, and partition_copy_result have the template parameters, data members, and special members specified below. They have no base classes or members other than those specified.

25.3 Parallel algorithms [algorithms.parallel]

25.4 Header <algorithm> synopsis [algorithm.syn]

25.5 Non-modifying sequence operations [alg.nonmodifying]

25.6 Mutating sequence operations [alg.modifying.operations]

25.7 Sorting and related operations [alg.sorting]

25.8 Header <numeric> synopsis [numeric.ops.overview]

25.9 Generalized numeric operations [numeric.ops]

25.10 Specialized <memory> algorithms [specialized.algorithms]

[Editor's note: The section heading is renamed from "Specialized algorithms"]

1 The contents specified in this subclause 25.10 are declared in the header <memory> (20.10.2).

2 Throughout this subclause, the names of template parameters are used to express type requirements for those algorithms defined directly in namespace std.

(2.1) — If an algorithm's template parameter is named InputIterator, the template argument shall meet the *Cpp17InputIterator* requirements (**??**).

(2.2) — 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.

3 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.

4 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.

5 The following additional requirements apply for those algorithms defined in namespace std::ranges:

(5.1) — 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.

(5.2) — 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.

(5.3) — 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*]

6 [*Note*: When invoked on ranges of potentially overlapping subobjects (**??**), the algorithms specified in this subclause 25.10 result in undefined behavior. — *end note*]

7 Some algorithms ~~defined~~specified 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)));
  }
```

25.10.0.1 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>>;
```

25.10.0.2 addressof [specialized.addressof_MOVED]

[Editor's note: This section is left in [utilities].]

```
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.

25.10.0.3 uninitialized_default_construct [uninitialized.construct.default]

```
template<class NoThrowForwardIterator>
  void uninitialized_default_construct(NoThrowForwardIterator first, NoThrowForwardIterator last);
```

1 *Effects:* Equivalent to:

```
for (; first != last; ++first)
  ::new (voidify(*first))
    typename iterator_traits<NoThrowForwardIterator>::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 NoThrowForwardIterator, class Size>
  NoThrowForwardIterator uninitialized_default_construct_n(NoThrowForwardIterator first, Size n);
```

3 *Effects:* Equivalent to:

```
for (; n > 0; (void)++first, --n)
  ::new (voidify(*first))
    typename iterator_traits<NoThrowForwardIterator>::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();
```

25.10.0.4 `uninitialized_value_construct` [uninitialized.construct.value]

```
template<class NoThrowForwardIterator>
  void uninitialized_value_construct(NoThrowForwardIterator first, NoThrowForwardIterator last);
```

1 *Effects:* Equivalent to:

```
for (; first != last; ++first)
  ::new (voidify(*first))
    typename iterator_traits<NoThrowForwardIterator>::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 NoThrowForwardIterator, class Size>
  NoThrowForwardIterator uninitialized_value_construct_n(NoThrowForwardIterator first, Size n);
```

3 *Effects:* Equivalent to:

```
for (; n > 0; (void)++first, --n)
  ::new (voidify(*first))
    typename iterator_traits<NoThrowForwardIterator>::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();
```

25.10.0.5 `uninitialized_copy` [uninitialized.copy]

```
template<class InputIterator, class NoThrowForwardIterator>
  NoThrowForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                            NoThrowForwardIterator result);
```

1 *Expects:* `[result, (last - first))` ~~shall~~does not overlap with `[first, last)`.

2 *Effects:* Equivalent to:

```
for (; first != last; ++result, (void) ++first)
  ::new (voidify(*result))
    typename iterator_traits<NoThrowForwardIterator>::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 *Expects:* `[ofirst, olast)` ~~shall~~does 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 NoThrowForwardIterator>
  NoThrowForwardIterator uninitialized_copy_n(InputIterator first, Size n, NoThrowForwardIterator result);
```

6 *Expects:* `[result, n)` ~~shall~~does not overlap with `[first, n)`.

7 *Effects:* Equivalent to:

```
for ( ; n > 0; ++result, (void) ++first, --n) {
  ::new (voidify(*result))
    typename iterator_traits<NoThrowForwardIterator>::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 *Expects:* `[ofirst, olast)` ~~shall~~does 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};
```

25.10.0.6 uninitialized_move [uninitialized.move]

```
template<class InputIterator, class NoThrowForwardIterator>
  NoThrowForwardIterator uninitialized_move(InputIterator first, InputIterator last,
                                            NoThrowForwardIterator result);
```

1 *Expects:* [result, (last - first)) ~~shall~~does not overlap with [first, last).

2 *Effects:* Equivalent to:

```
    for (; first != last; (void)++result, ++first)
      ::new (voidify(*result))
        typename iterator_traits<NoThrowForwardIterator>::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 *Expects:* [ofirst, olast) ~~shall~~does 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 NoThrowForwardIterator>
  pair<InputIterator, NoThrowForwardIterator>
    uninitialized_move_n(InputIterator first, Size n, NoThrowForwardIterator result);
```

6 *Expects:* [result, n) ~~shall~~does not overlap with [first, n).

7 *Effects:* Equivalent to:

```
    for (; n > 0; ++result, (void) ++first, --n)
      ::new (voidify(*result))
        typename iterator_traits<NoThrowForwardIterator>::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 *Expects:* [ofirst, olast) ~~shall~~does 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*]

25.10.0.7 `uninitialized_fill` [uninitialized.fill]

```
template<class NoThrowForwardIterator, class T>
  void uninitialized_fill(NoThrowForwardIterator first, NoThrowForwardIterator last, const T& x);
```

1 *Effects:* Equivalent to:

```
for (; first != last; ++first)
  ::new (voidify(*first))
    typename iterator_traits<NoThrowForwardIterator>::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 NoThrowForwardIterator, class Size, class T>
  NoThrowForwardIterator uninitialized_fill_n(NoThrowForwardIterator first, Size n, const T& x);
```

2 *Effects:* Equivalent to:

```
for (; n--; ++first)
  ::new (voidify(*first))
    typename iterator_traits<NoThrowForwardIterator>::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();
```

25.10.0.8 `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)...);
```

25.10.0.9 `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 NoThrowForwardIterator>
  constexpr void destroy(NoThrowForwardIterator first, NoThrowForwardIterator 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 NoThrowForwardIterator, class Size>
  constexpr NoThrowForwardIterator destroy_n(NoThrowForwardIterator 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();
```

25.11 C library algorithms [alg.c.library]