

Document number: P1033R1
Date: 2018-06-25
Audience: LEWG
Authors: Casey Carter
Christopher Di Bella
Reply to: christopher@codeplay.com

Rangify the uninitialised memory algorithms!

1 General

[intro]

1.1 Change Log

[intro.change.log]

1.1.1 Minor changes

[intro.change.log.minor]

The following changes are editorial in nature, or were added to the discussion in LEWG.

- Minor typographical changes (e.g. `s/typename/class/g`).
- Moved *Annex B Reference implementation* to 1.4.
- All `NoThrow` iterators delimited by `Sentinels` are now delimited by *no-throw-sentinels*.

1.1.2 Major changes

[intro.change.log.major]

The following changes affect the proposal's direction, as requested by LEWG.

- Re-specified all concepts as exposition-only.
- Removed *Annex A Compatibility features*, which deprecated the three-legged memory specialisations.
- Changed the `uninitialized_copy_n` and `uninitialized_move_n` so that they take a full output range, rather than half an output range. This was not strictly directed by LEWG, but it is in line with the direction to remove the three-legged algorithms.
- Added editor notes to preserve the requirements from *Annex A* that affect the memory specialisations in namespace `std`.

1.2 Scope

[intro.scope]

- ¹ This proposal describes functions compatible with the Ranges TS. These extensions change and add to the existing memory specialisation library facilities found in C++17.

1.3 Motivation

[intro.motivation]

- ¹ *N3351 A Concept Design for the STL* (Stroustrup and Sutton, 2012), also known as the ‘Palo Alto Report’, serves as the basis for the Ranges TS. Both the Palo Alto Report and the Ranges TS focus on the algorithms located in `<algorithm>`, and are not extended to revise the algorithms found in `<memory>` and `<numeric>`. P1033 seeks to include those algorithms found in `<memory>` alongside the algorithms revised by the Ranges TS.

1.4 Implementation compliance

[intro.compliance]

- ¹ Similarly to the Ranges TS, conformance requirements are the same as those described in 1.4 in the C++ Standard. [*Note*: Conformance is defined in terms of the behaviour of programs. — *end note*]
- ² A reference implementation can be found in the master branch of Casey Carter’s Ranges TS reference implementation. For a closer inspection, please visit [the reference implementation](#).

1.5 Namespaces, headers, and modifications to standard classes

[intro.namespace]

- ¹ All components described in this document are declared in an unspecified namespace.

[Editor’s note: The following text is taken from the C++ Standard and edited to reflect the fact that much of this document is suggesting parallel constrained facilities that are specified as diffs against the existing unconstrained facilities in namespace `std`.]

- ² The International Standard, ISO/IEC 14882, together with ISO/IEC 19217:2015 (the Concepts TS) and ISO/IEC TS 21425:2017 (the Ranges TS), provide important context and specification for this paper. This document is written as a set of changes against the C++ Working Paper, N4741. Sections are copied verbatim and modified so as to define similar but different components in namespace `std`. Effort was made to keep chapter and section numbers the same as in N4741 for the sake of easy cross-referencing with the understanding that section numbers will change in the final draft of ISO/IEC 14882 that this proposal is integrated with.
- ³ References to other entities described in this document are assumed to be qualified with `::std::ranges::`, and references to entities described in the International Standard are assumed to be qualified with `::std::`.
- ⁴ P1033 uses diff formatting to show how the proposed algorithms differ from the existing algorithms in the International Standard. [Turquoise formatting](#) indicates text added to `::std::ranges`, and [red formatting](#) indicates where the proposed algorithms diverge from their International Standard forebears. Except where expressly noted, no changes are proposed to the existing algorithms.
- ⁵ This document specifically describes changes to `<memory>`. As such, new content can be found in the `<ranges/memory>` header.

Table 1 — Proposal headers

<code><ranges/memory></code>

23 General utilities library

[utilities]

23.10 Memory

[memory]

23.10.2 Header `<ranges/memory>` synopsis

[memory.syn]

```

// 23.10.x special memory concepts
template <class I>
concept no-throw-input-iterator = see below;

template <class S, class I>
concept no-throw-sentinel = see below;

template <class Rng>
concept no-throw-input-range = see below;

template <class I>
concept no-throw-forward-iterator = see below;

template <class Rng>
concept no-throw-forward-range = see below;

// 23.10.10 specialized algorithms:

template <class ForwardIterator>
    void uninitialized_default_construct(ForwardIterator first, ForwardIterator last);
template <class ForwardIterator, class Size>
    ForwardIterator uninitialized_default_construct_n(ForwardIterator first, Size n);

template <no-throw-forward-iterator I, no-throw-sentinel <I> S>
    requires
    DefaultConstructible<value_type_t<I>>
    I uninitialized_default_construct(I first, S last);

template <no-throw-forward-range Rng>
    requires
    DefaultConstructible<value_type_t<iterator_t<Rng>>>
    safe_iterator_t<Rng> uninitialized_default_construct(Rng&& rng);

template <no-throw-forward-iterator I>
    requires
    DefaultConstructible<value_type_t<I>>
    I uninitialized_default_construct_n(I first, difference_type_t<I> n);

template <class ForwardIterator>
    void uninitialized_value_construct(ForwardIterator first, ForwardIterator last);
template <class ForwardIterator, class Size>
    ForwardIterator uninitialized_value_construct_n(ForwardIterator first, Size n);

template <no-throw-forward-iterator I, no-throw-sentinel <I> S>
    requires
    DefaultConstructible<value_type_t<I>>

```

```

    I uninitialized_value_construct(I first, S last);

template <no-throw-forward-range Rng>
    requires
DefaultConstructible<value_type_t<iterator_t<Rng>>>
    safe_iterator_t<Rng> uninitialized_value_construct(Rng&& rng);

template <no-throw-forward-iterator I>
    requires
DefaultConstructible<value_type_t<I>>
    I uninitialized_value_construct_n(I first, difference_type_t<I> n);

template <class InputIterator, class ForwardIterator>
    ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                      ForwardIterator result);
template <class InputIterator, class Size, class ForwardIterator>
    ForwardIterator uninitialized_copy_n(InputIterator first, Size n,
                                       ForwardIterator result);

template <InputIterator I, Sentinel<I> S1, no-throw-forward-iterator O, no-throw-sentinel <O> S2>
    requires
Constructible<value_type_t<O>, reference_t<I>>
    tagged_pair<tag::in(I), tag::out(O)>
uninitialized_copy(I ifirst, S1 ilast, O ofirst, S2 olast);

template <InputRange IRng, no-throw-forward-range ORng>
    requires
Constructible<value_type_t<iterator_t<ORng>>, reference_t<iterator_t<IRng>>>
    tagged_pair<tag::in(safe_iterator_t<IRng>), tag::out(safe_iterator_t<ORng>>>
uninitialized_copy(IRng&& irng, ORng&& orng);

template <InputIterator I, no-throw-forward-iterator O, no-throw-sentinel <O> S>
    requires
Constructible<value_type_t<O>, reference_t<I>>
    tagged_pair<tag::in(I), tag::out(O)>
uninitialized_copy_n(I ifirst, difference_type_t<I> n, O ofirst, S olast)

template <class InputIterator, class ForwardIterator>
    ForwardIterator uninitialized_move(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 <InputIterator I, Sentinel<I> S1, no-throw-forward-iterator O, no-throw-sentinel <O> S2>
    requires
Constructible<value_type_t<O>, rvalue_reference_t<I>>
    tagged_pair<tag::in(I), tag::out(O)>
    uninitialized_move(I ifirst, S1 ilast, O ofirst, S2 olast);

template <InputRange IRng, no-throw-forward-range ORng>
    requires
Constructible<value_type_t<iterator_t<ORng>>, rvalue_reference_t<iterator_t<IRng>>>
    tagged_pair<tag::in(safe_iterator_t<IRng>), tag::out(safe_iterator_t<ORng>>>
uninitialized_move(IRng&& irng, ORng&& orng);

```

```

template <InputIterator I, no-throw-forward-iterator O, no-throw-sentinel <O> S>
    requires
Constructible<value_type_t<O>, rvalue_reference_t<I>>
    tagged_pair<tag::in(I), tag::out(O)>
uninitialized_move_n(I ifirst, difference_type_t<I> n, O ofirst, S olast);

template <class ForwardIterator, class T>
    void uninitialized_fill(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 <no-throw-forward-iterator I, no-throw-sentinel <I> S, class T>
    requires
Constructible<value_type_t<I>, const T&>
    I uninitialized_fill(I first, S last, const T& x);

template <no-throw-forward-range Rng, class T>
    requires
Constructible<value_type_t<iterator_t<Rng>>, const T&>
    safe_iterator_t<Rng> uninitialized_fill(Rng&& rng, const T& x);

template <no-throw-forward-iterator I, class T>
    requires
Constructible<value_type_t<I>, const T&>
    I uninitialized_fill_n(I first, const difference_type_t<I> n, const T& x);

template <class T>
    void destroy_at(T* location);
template <class ForwardIterator>
    void destroy(ForwardIterator first, ForwardIterator last);
template <class ForwardIterator, class Size>
    ForwardIterator destroy_n(ForwardIterator first, Size n);

template <Destructible T>
    void destroy_at(T* location) noexcept;

template <no-throw-input-iterator I, no-throw-sentinel <I> S>
    requires
Destructible<value_type_t<I>>
    I destroy(I first, S last) noexcept;

template <no-throw-input-range Rng>
    requires
Destructible<value_type_t<iterator_t<Rng>>
    safe_iterator_t<Rng> destroy(Rng&& rng) noexcept;

template <no-throw-input-iterator I>
    requires
Destructible<value_type_t<I>>
    I destroy_n(I first, difference_type_t<I> n) noexcept;

```

23.10.10 Specialized algorithms

[specialized.algorithms]

¹ Throughout this sub-clause, the names of template parameters are used to express type requirements.

- (1.1) — If an algorithm’s template parameter is named `InputIterator`, the template argument shall satisfy the requirements of an input iterator (??).
- (1.2) — If an algorithm’s template parameter is named `ForwardIterator`, the template argument shall satisfy the requirements of a forward iterator (??), and is required to have the property that no exceptions are thrown from increment, assignment, comparison, or indirection through valid iterators.

¹ All of the algorithms specified in (23.10.10) shall only operate on ranges of complete objects. Use of these functions on ranges of subobjects is undefined.

[Editor’s note: This paragraph applies to all memory specialisations, including those already defined in the International Standard.]

² This section defines the following concepts:

```
template <class I>
concept no-throw-input-iterator = // exposition only
    InputIterator<I> &&
    is_lvalue_reference_v<reference_t<I>> &&
    Same<remove_cv_ref_t<reference_t<I>>, value_type_t<I>>;
```

³ No exceptions are thrown from increment, copy, move, assignment, or indirection through valid iterators.

⁴ [Note: The distinction between `InputIterator` and `no-throw-input-iterator` is purely semantic. — end note]

```
template <class S, class I>
concept no-throw-sentinel = // exposition only
    Sentinel<S, I>;
```

⁵ No exceptions are thrown from comparisons between objects of type `I` and `S`.

⁶ [Note: The distinction between `Sentinel` and `no-throw-sentinel` is purely semantic. — end note]

```
template <class Rng>
concept no-throw-input-range = // exposition only
    Range<Rng> &&
    no-throw-input-iterator <iterator_t<Rng>> &&
    no-throw-sentinel <sentinel_t<Rng>, iterator_t<Rng>>;
```

⁷ No exceptions are thrown from calls to `begin` and `end` on an object of type `Rng`.

⁸ [Note: The distinction between `InputRange` and `no-throw-input-range` is purely semantic. — end note]

```
template <class I>
concept no-throw-forward-iterator = // exposition only
    no-throw-input-iterator <I> &&
    no-throw-sentinel <I, I> &&
    ForwardIterator<I>;
```

⁹ [Note: The distinction between `ForwardIterator` and `no-throw-forward-iterator` is purely semantic. — end note]

```
template <class Rng>
concept no-throw-forward-range = // exposition only
    no-throw-forward-iterator <iterator_t<Rng>> &&
    no-throw-input-range <Rng> &&
    ForwardRange<Rng>;
```

10 *[Note: The distinction between ForwardRange and `no-throw-forward-range` is purely semantic. — end note]*

Unless otherwise specified, if an exception is thrown in the following algorithms there are no effects.

23.10.10.1 uninitialized_default_construct [uninitialized.construct.default]

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

template <no-throw-forward-iterator I, no-throw-sentinel <I> S>
    requires
    DefaultConstructible<value_type_t<I>>
    I uninitialized_default_construct(I first, S last);
```

1 *Effects:* Equivalent to:

```
for (; first != last; ++first)
    ::new (const_cast<void*>(static_cast<const volatile void*>(addressof(*first))))
        typename iterator_traits<ForwardIterator>::value_typeremove_reference_t<reference_t<I>>;
return first;
```

[Editor's note: `const_cast<void*>` is necessary to ensure that `::operator new<void*>` ('True Placement New') is called. The decision to cast `const`-ness away after calling `addressof` is an alternative to preventing users from being unable to pass ranges that are non-`const`.

When `addressof(*i)` returns a `const T*`, this will not convert to `void*`, and so no suitable overload of the True Placement New is found.

It is noted that `const`-qualified means 'do not modify', and that the `const_cast` ignores this (and is thus lying). However, these algorithms are also claiming that they iterate over objects of type `T`: nary a `T` is in the range. We present this as 'the objects in this range should be `const`', rather than 'the memory here is `const`']

```
template <no-throw-forward-range Rng>
    requires
    DefaultConstructible<value_type_t<iterator_t<Rng>>>
    safe_iterator_t<Rng> uninitialized_default_construct(Rng&& rng);
```

2 *Effects:* Equivalent to:

```
return uninitialized_default_construct(begin(rng), end(rng));
```

```
template <class ForwardIterator, class Size>
    ForwardIterator uninitialized_default_construct_n(ForwardIterator first, Size n);
```

2 *Effects:* Equivalent to:

```
for (; n>0; (void)++first, --n)
    ::new (static_cast<void*>(addressof(*first)))
        typename iterator_traits<ForwardIterator>::value_type;
return first;
```

```
template <no-throw-forward-iterator I>
    requires
    DefaultConstructible<value_type_t<I>>
    I uninitialized_default_construct_n(I first, difference_type_t<I> n);
```


3 *Effects:* Equivalent to:

```
return uninitialized_default_construct(make_counted_iterator(first, n),
    default_sentinel{}).base();
```

23.10.10.2 uninitialized_value_construct

[uninitialized.construct.value]

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

template <no-throw-forward-iterator I, no-throw-sentinel <I> S>
    requires
    DefaultConstructible<value_type_t<I>>
    I uninitialized_value_construct(I first, S last);
```

1 *Effects:* Equivalent to:

```
for (; first != last; ++first)
    ::new (const_cast<void*>(static_cast<const volatile void*>(addressof(*first))))
        typename_iterator_traits<ForwardIterator>::value_type_remove_reference_t<reference_t<I>>();
return first;

template <no-throw-forward-range Rng>
    requires
    DefaultConstructible<value_type_t<iterator_t<Rng>>>
    safe_iterator_t<Rng> uninitialized_value_construct(Rng&& rng);
```

2 *Effects:* Equivalent to:

```
return uninitialized_value_construct(begin(rng), end(rng));

template <class ForwardIterator, class Size>
    ForwardIterator uninitialized_value_construct_n(ForwardIterator first, Size n);
```

2 *Effects:* Equivalent to:

```
for (; n>0; (void)++first, --n)
    ::new (static_cast<void*>(addressof(*first)))
        typename_iterator_traits<ForwardIterator>::value_type();
return first;

template <no-throw-forward-iterator I>
    requires
    DefaultConstructible<value_type_t<I>>
    I uninitialized_value_construct_n(I first, difference_type_t<I> n);
```

3 *Effects:* Equivalent to:

```
return uninitialized_value_construct(make_counted_iterator(first, n),
    default_sentinel{}).base();
```

23.10.10.3 uninitialized_copy

[uninitialized.copy]

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

1 *Effects:* As if by:

```

    for (; first != last; ++result, (void)++first)
        ::new (static_cast<void*>(addressof(*result)))
            typename iterator_traits<ForwardIterator>::value_type;

```

2 *Returns:* result

```

template <InputIterator I, Sentinel<I> S1, no-throw-forward-iterator O, no-throw-sentinel <O> S2>
    requires
    Constructible<value_type_t<O>, reference_t<I>>
    tagged_pair<tag::in(I), tag::out(O)>
    uninitialized_copy(I ifirst, S1 ilast, O ofirst, S2 olast);

```

1 *Effects:* Equivalent to:

```

    for (; ifirst != ilast && ofirst != olast; ++ofirst, (void)++ifirst) {
        ::new (const_cast<void*>(static_cast<const volatile void*>(addressof(*ofirst))))
            remove_reference_t<reference_t<O>>(*ifirst);
    }
    return {ifirst, ofirst};

```

2 *Requires:* [ofirst,olast) shall not overlap with [ifirst,ilast).

[Editor's note: This paragraph also applies to the three-legged `std::uninitialized_copy`.]

```

template <InputRange IRng, no-throw-forward-range ORng>
    requires
    Constructible<value_type_t<iterator_t<ORng>>, reference_t<iterator_t<IRng>>>
    tagged_pair<tag::in(safe_iterator_t<IRng>), tag::out(safe_iterator_t<ORng>0)>
    uninitialized_copy(IRng&& irng, ORng&& orng);

```

3 *Effects:* Equivalent to:

```

    return uninitialized_copy(begin(irng), end(irng), begin(orng), end(orng));

```

4 *Requires:* orng shall not overlap with irng.

```

template <class InputIterator, class Size, class ForwardIterator>
    ForwardIterator uninitialized_copy_n(InputIterator first, Size n,
                                       ForwardIterator result);

```

3 *Effects:* As if by:

```

    for (; n > 0; ++result, (void) ++first, --n) {
        ::new (static_cast<void*>(addressof(*result)))
            typename iterator_traits<ForwardIterator>::value_type(*first);
    }

```

4 *Returns:* result

```

template <InputIterator I, no-throw-forward-iterator O, no-throw-sentinel <O> S>
    requires
    Constructible<value_type_t<O>, reference_t<I>>
    tagged_pair<tag::in(I), tag::out(O)>
    uninitialized_copy_n(I ifirst, difference_type_t<I> n, O ofirst, S olast);

```

5 *Effects:* Equivalent to:

```

    auto t = uninitialized_copy(make_counted_iterator(ifirst, n),
                               default_sentinel{}, ofirst, olast).base();
    return {t.in().base(), t.out()};

```

⁶ *Requires:* [ofirst,olast) shall not overlap with [ifirst,next(ifirst, n)).

[Editor's note: This paragraph also applies to `std::uninitialized_copy_n`.]

23.10.10.4 uninitialized_move

[uninitialized.move]

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

¹ *Effects:* Equivalent to:

```
for (; first != last; (void)++result, ++first)
    ::new (static_cast<void*>(addressof(*result)))
        typename iterator_traits<ForwardIterator>::value_type(iter_move(first));
return result;
```

```
template <InputIterator I, Sentinel<I> S1, no-throw-forward-iterator O, no-throw-sentinel <O> S2>
    requires
    Constructible<value_type_t<O>, rvalue_reference_t<I>>
    tagged_pair<tag::in(I), tag::out(O)>
    uninitialized_move(I ifirst, S1 ilast, O ofirst, S2 olast);
```

¹ *Effects:* Equivalent to:

```
for (; ifirst != ilast && ofirst != olast; ++ofirst, (void)++ifirst) {
    ::new (const_cast<void*>(static_cast<const volatile void*>(addressof(*ofirst))))
        remove_reference_t<reference_t<O>>(iter_move(ifirst));
}
return {ifirst, ofirst};
```

² *Remarks:* If an exception is thrown, some objects in the range [first,last) are left in a valid, but unspecified state.

³ *Requires:* [ofirst,olast) shall not overlap with [ifirst,ilast).

[Editor's note: This paragraph also applies to the three-legged `std::uninitialized_move`.]

```
template <InputRange IRng, no-throw-forward-range ORng>
    requires
    Constructible<value_type_t<iterator_t<ORng>>, rvalue_reference_t<iterator_t<IRng>>>
    tagged_pair<tag::in(safe_iterator_t<IRng>), tag::out(safe_iterator_t<ORng>>)
    uninitialized_move(IRng&& irng, ORng&& orng);
```

⁴ *Effects:* Equivalent to:

```
return uninitialized_move(begin(irng), end(irng), begin(orng), end(orng));
```

⁵ *Remarks:* If an exception is thrown, some objects in the range [begin(rng),end(rng)) are left in a valid, but unspecified state.

⁶ *Requires:* orng shall not overlap with irng.

```
template <class InputIterator, class Size, class ForwardIterator>
    pair<InputIterator, ForwardIterator>
    uninitialized_move_n(InputIterator first, Size n, ForwardIterator result);
```

³ *Effects:* Equivalent to:

```
for (; n > 0; ++result, (void) ++first, --n)
    ::new (static_cast<void*>(addressof(*result)))
        typename iterator_traits<ForwardIterator>::value_type(iter_move(first));
return {first,result};
```

```
template <InputIterator I, no-throw-forward-iterator O, no-throw-sentinel <O> S>
requires
    Constructible<value_type_t<O>, rvalue_reference_t<I>>
tagged_pair<tag::in(I), tag::out(O)>
    uninitialized_move_n(I ifirst, difference_type_t<I> n, O ofirst, S olast);
```

⁴ *Effects:* Equivalent to:

```
auto t = uninitialized_move(make_counted_iterator(ifirst, n),
                           default_sentinel{}, ofirst, olast).base();
return {t.in().base(), t.out()};
```

⁴ *Remarks:* If an exception is thrown, some objects in the range [ifirst,next(ifirst, n)) are left in a valid, but unspecified state.

⁶ *Requires:* [ofirst,olast) shall not overlap with [ifirst,next(ifirst, n)).

[Editor's note: This paragraph also applies to std::uninitialized_move_n.]

23.10.10.5 uninitialized_fill

[uninitialized.fill]

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

```
template <no-throw-forward-iterator I, no-throw-sentinel <I> S, class T>
requires
    Constructible<value_type_t<I>, const T&>
I uninitialized_fill(I first, S last, const T& x);
```

¹ *Effects:* Equivalent to:

```
for (; first != last; ++first) {
    ::new (const_cast<void*>(static_cast<const volatile void*>(addressof(*first))))
        typename iterator_traits<ForwardIterator>::value_typeremove_reference_t<reference<I>>(x);
}
return first;
```

```
template <no-throw-forward-range Rng, class T>
requires
    Constructible<value_type_t<iterator_t<Rng>>, const T&>
safe_iterator_t<Rng> uninitialized_fill(Rng&& rng, const T& x)
```

² *Effects:* Equivalent to:

```
return uninitialized_fill(begin(rng), end(rng), x);
```

```
template <class ForwardIterator, class Size, class T>
ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
```

² *Effects:* As if by:

```
for (; n--; ++first)
    ::new (static_cast<void*>(addressof(*first)))
        typename iterator_traits<ForwardIterator>::value_type(x);
return first;
```

P1033R1 Rangify the uninitialised memory algorithms!

```
template <no-throw-forward-iterator I, class T>
requires
    Constructible<value_type_t<I>, const T&>
I uninitialized_fill_n(I first, const difference_type_t<I> n, const T& x);
```

⁴ *Effects*: Equivalent to:

```
return uninitialized_fill(make_counted_iterator(first, n), default_sentinel{}, x).base();
```

23.10.10.6 destroy

[specialized.destroy]

```
template <class T>
void destroy_at(T* location);

template <Destructible T>
void destroy_at(T* location) noexcept;
```

¹ *Effects*: Equivalent to:

```
location->~T();

template <class ForwardIterator>
void destroy(ForwardIterator first, ForwardIterator last);

template <no-throw-input-iterator I, no-throw-sentinel <I> S>
requires
    Destructible<value_type_t<I>>
I destroy(I first, S last) noexcept;
```

² *Effects*: Equivalent to:

```
for (; first != last; ++first)
    destroy_at(addressof(*first));
return first;
```

[Editor's note: The International Standard requires `destroy` be a `ForwardIterator` to ensure that the iterator's `reference` type is a reference type. This has been relaxed in P1033, as *no-throw-input-iterator* checks that `reference_t<I>` reference type.

The choice to weaken the iterator requirement from the International Standard is because the algorithm is a single-pass algorithm; thus, semantically, works on input ranges.]

```
template <no-throw-input-range Rng>
requires
    Destructible<value_type_t<iterator_t<Rng>>>
safe_iterator_t<Rng> destroy(Rng&& rng) noexcept;
```

³ *Effects*: Equivalent to:

```
return destroy(begin(rng), end(rng));
```

```
template <class ForwardIterator, class Size>
ForwardIterator destroy_n(ForwardIterator first, Size n);
```

³ *Effects*: Equivalent to:

```
for (; n > 0; (void)++first, --n)
    destroy_at(addressof(*first));
return first;
```

P1033R1 Rangify the uninitialised memory algorithms!

```
template <no-throw-input-iterator I>  
requires  
    Destructible<value_type_t<I>  
I destroy_n(I first, difference_type_t<I> n) noexcept;
```

⁴ *Effects:* Equivalent to:

```
return destroy(make_counted_iterator(first, n), default_sentinel{}).base();
```

Annex A (informative)

Acknowledgements [acknowledgements]

N4671 Working Draft, C++ Extensions for Ranges provided the template for the layout of P1033.

Eric Niebler and Tim Song both provided invaluable feedback during the review process of this paper.

Bibliography

- [1] Casey Carter and Eric Niebler. CMCSTL2: An implementation of C++ extensions for ranges. <https://github.com/caseycarter/cmctl2/>. Accessed: 2016-10-15.
- [2] Eric Niebler, Casey Carter, and Christopher Di Bella. <https://github.com/ericniebler/stl2/pull/224>. Accessed: 2016-10-18.
- [3] Eric Niebler, Casey Carter, and Christopher Di Bella. Should there be a range equivalent of `std::uninitialized_x`? <https://github.com/ericniebler/stl2/issues/223>. Accessed: 2016-10-15.