constexpr for specialized memory algorithms

Document #:	P2283R1
Date:	2021-03-30
Project:	Programming Language C++
Audience:	Library Evolution Group
Reply-to:	Michael Schellenberger Costa <mschellenbergercosta@googlemail.com>

1 Revision History

— R1
  — Added feature test macros
  — Clarified scope and impact on core wording
  — Removed usage of `to_address`
  — Explained the need for `default_construct_at`
— R0
  — Initial draft

2 Introduction

This paper proposes adding `constexpr` support to the specialized memory algorithms. This is essentially a followup to [P0784R7] which added `constexpr` support for all necessary machinery.

3 Motivation and Scope

These algorithms have been forgotten in the final crunch to get C++20 out. To add insult to injury, they are essential to implementing `constexpr` container support, so every library has to provide its own internal helpers to do the exact same thing during constant evaluation. Just fill the void and add `constexpr` everywhere except the parallel overloads.

But what about `uninitialized_default_construct`? We cannot use `construct_at` there, because it would always *value*-initialize!

Or can we? Reading an indetermined value would be UB anyhow so just *value*-initialize away and be done with it? Lets consider the following code:

```
#include <memory>

constexpr bool something(const size_t numElements) {
    std::allocator<int> alloc;
    std::allocator_traits<std::allocator<int>> allty;
    auto data = allty.allocate(alloc, numElements);

    std::uninitialized_default_construct(data, data + numElements);
    const bool res = std::all_of(data, data + numElements,
                                 [](const int val) { return val == 0; }));

    std::destroy(data, data + numElements);
    allty.deallocate(alloc, data, numElements);
    return res;
}

static_assert(something(5));
```

If we go the route of "just use `construct_at` during constant evaluation" this code will compile fine due to *value*-initialization of the elements of `data`. However, it will also crash and burn during runtime, as there the elements of `data` will be *default*-initialized. As `int` is a trivial type their value is indeterminate.

If anywhere in the program we read that value we are in for some fun. We might even get lucky and use a compiler that zeroes out memory in DEBUG mode, so the bug would only materialize in RELEASE builds.

By taking the shortcut of `construct_at` we would change the meaning of the code depending on whether it is run during runtime or constant evaluation. Even worse, the bug will not appear during compile time, though we blessed it with a `static_assert`.

But what *is* the right thing? As always do as the `int`s do, because there is already support in the standard for the right thing:

```
constexpr bool somethingElse() {
  int meow;
  return true;
}

static_assert(somethingElse());
```

This is totally fine both during runtime and constant evaluation. What is the value of `meow`? We do not know, but as long as we do not read from it, all is fine. It is the users responsibility to give it a proper value before reading from it.

So we need to ensure that the semantics of the code do not change between runtime and constant evaluation, which gives us two possible solutions:

1. Add an overload of `construct_at` that takes a `default_construct_tag` and does the right thing. However, if we go through the trouble of adding a new name we should rather go with
2. Add a new library function `default_construct_at` that does the right thing. This is what is proposed here.

While the actual definition of `default_construct_at` is a bit on the verbose side, it extends existing library technology to avoid the schism between runtime and constant evaluation and keeps the library consistent with the language.

4 Impact on the Standard

This proposal would expand the list of allowed expressions under constant evaluation, which is something that should be considered carefully. That said, the addition to core wording is highly targeted to a specific use case, that would not be achievable otherwise.

5 Proposed Wording

5.1 Modify 7.7 [expr.const] §6 of [N4762] as follows

For the purposes of determining whether an expression E is a core constant expression, the evaluation of a call to a member function of std::allocator as defined in [allocator.members], where T is a literal type, does not disqualify E from being a core constant expression, even if the actual evaluation of such a call would otherwise fail the requirements for a core constant expression. Similarly, the evaluation of a call to std::destroy_at, std::ranges::destroy_at, std::default_construct_at, std::ranges::default_construct_at, std::construct_at, or std::ranges::construct_at does not disqualify E from being a core constant expression unless:

for a call to std::default_construct_at, std::ranges::default_construct_at, std::construct_at or std::ranges::construct_at, the first argument, of type T*, does not point to storage allocated with std::allocator or to an object whose lifetime began within the evaluation of E, or the evaluation of the underlying constructor call disqualifies E from being a core constant expression, or

5.2 Modify 20.10.2 [memory.syn] of [N4762] as follows

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

template<class ExecutionPolicy, class NoThrowForwardIterator>
  void uninitialized_default_construct(ExecutionPolicy&& exec,        // see [algorithms.parallel.overl
                                       NoThrowForwardIterator first,
                                       NoThrowForwardIterator last);
template<class NoThrowForwardIterator, class Size>
  constexpr NoThrowForwardIterator
    uninitialized_default_construct_n(NoThrowForwardIterator first, Size n);
template<class ExecutionPolicy, class NoThrowForwardIterator, class Size>
  NoThrowForwardIterator
    uninitialized_default_construct_n(ExecutionPolicy&& exec,         // see [algorithms.parallel.overl
                                      NoThrowForwardIterator first, Size n);

namespace ranges {
  template<no-throw-forward-iterator I, no-throw-sentinel-for<I> S>
    requires default_initializable<iter_value_t<I>>
      constexpr I uninitialized_default_construct(I first, S last);
  template<no-throw-forward-range R>
    requires default_initializable<range_value_t<R>>
      constexpr borrowed_iterator_t<R> uninitialized_default_construct(R&& r);

  template<no-throw-forward-iterator I>
    requires default_initializable<iter_value_t<I>>
      constexpr I uninitialized_default_construct_n(I first, iter_difference_t<I> n);
}

template<class NoThrowForwardIterator>
  constexpr void uninitialized_value_construct(NoThrowForwardIterator first,
                                               NoThrowForwardIterator last);
template<class ExecutionPolicy, class NoThrowForwardIterator>
  void uninitialized_value_construct(ExecutionPolicy&& exec,  // see [algorithms.parallel.overloads]
                                     NoThrowForwardIterator first,
                                     NoThrowForwardIterator last);
template<class NoThrowForwardIterator, class Size>
  constexpr NoThrowForwardIterator
    uninitialized_value_construct_n(NoThrowForwardIterator first, Size n);
template<class ExecutionPolicy, class NoThrowForwardIterator, class Size>
  NoThrowForwardIterator
    uninitialized_value_construct_n(ExecutionPolicy&& exec,   // see [algorithms.parallel.overloads]
                                    NoThrowForwardIterator first, Size n);
```

```
namespace ranges {
  template<no-throw-forward-iterator I, no-throw-sentinel-for<I> S>
    requires default_initializable<iter_value_t<I>>
      constexpr I uninitialized_value_construct(I first, S last);
  template<no-throw-forward-range R>
    requires default_initializable<range_value_t<R>>
      constexpr borrowed_iterator_t<R> uninitialized_value_construct(R&& r);

  template<no-throw-forward-iterator I>
    requires default_initializable<iter_value_t<I>>
      constexpr I uninitialized_value_construct_n(I first, iter_difference_t<I> n);
}

template<class InputIterator, class NoThrowForwardIterator>
  constexpr NoThrowForwardIterator
    uninitialized_copy(InputIterator first, InputIterator last,
                       NoThrowForwardIterator result);
template<class ExecutionPolicy, class InputIterator, class NoThrowForwardIterator>
  NoThrowForwardIterator uninitialized_copy(ExecutionPolicy&& exec,   // see [algorithms.parallel.overl
                                            InputIterator first, InputIterator last,
                                            NoThrowForwardIterator result);
template<class InputIterator, class Size, class NoThrowForwardIterator>
  constexpr 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 [algorithms.parallel.overl
                                              InputIterator first, Size n,
                                              NoThrowForwardIterator result);

namespace ranges {
  template<class I, class O>
    using uninitialized_copy_result = in_out_result<I, O>;
  template<input_iterator I, sentinel_for<I> S1,
           no-throw-forward-iterator O, no-throw-sentinel-for<O> S2>
    requires constructible_from<iter_value_t<O>, iter_reference_t<I>>
      constexpr 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>>
      constexpr uninitialized_copy_result<borrowed_iterator_t<IR>, borrowed_iterator_t<OR>>
        uninitialized_copy(IR&& in_range, OR&& out_range);

  template<class I, class O>
    using uninitialized_copy_n_result = in_out_result<I, O>;
  template<input_iterator I, no-throw-forward-iterator O, no-throw-sentinel-for<O> S>
    requires constructible_from<iter_value_t<O>, iter_reference_t<I>>
      constexpr 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>
  constexpr NoThrowForwardIterator
    uninitialized_move(InputIterator first, InputIterator last,
                       NoThrowForwardIterator result);
template<class ExecutionPolicy, class InputIterator, class NoThrowForwardIterator>
  NoThrowForwardIterator uninitialized_move(ExecutionPolicy&& exec,   // see [algorithms.parallel.overl
                                            InputIterator first, InputIterator last,
                                            NoThrowForwardIterator result);
template<class InputIterator, class Size, class NoThrowForwardIterator>
  constexpr 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 [algorithms.parallel.overloads]
                         InputIterator first, Size n, NoThrowForwardIterator result);

namespace ranges {
  template<class I, class O>
    using uninitialized_move_result = in_out_result<I, O>;
  template<input_iterator I, sentinel_for<I> S1,
           no-throw-forward-iterator O, no-throw-sentinel-for<O> S2>
    requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
      constexpr 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>>
      constexpr uninitialized_move_result<borrowed_iterator_t<IR>, borrowed_iterator_t<OR>>
        uninitialized_move(IR&& in_range, OR&& out_range);

  template<class I, class O>
    using uninitialized_move_n_result = in_out_result<I, O>;
  template<input_iterator I,
           no-throw-forward-iterator O, no-throw-sentinel-for<O> S>
    requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
      constexpr 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>
  constexpr void uninitialized_fill(NoThrowForwardIterator first,
                                    NoThrowForwardIterator last, const T& x);
template<class ExecutionPolicy, class NoThrowForwardIterator, class T>
  void uninitialized_fill(ExecutionPolicy&& exec,             // see [algorithms.parallel.overloads]
                          NoThrowForwardIterator first, NoThrowForwardIterator last,
                          const T& x);
template<class NoThrowForwardIterator, class Size, class T>
  constexpr 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 [algorithms.parallel.overloads]
                         NoThrowForwardIterator first, Size n, const T& x);

namespace ranges {
  template<no-throw-forward-iterator I, no-throw-sentinel-for<I> S, class T>
    requires constructible_from<iter_value_t<I>, const T&>
```

```
      constexpr 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&>
      constexpr borrowed_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&>
      constexpr I uninitialized_fill_n(I first, iter_difference_t<I> n, const T& x);
}

// [specialized.construct], 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);
}
```

```
// [specialized.default_construct], default_construct_at
template<class T>
  constexpr T* default_construct_at(T* location);

namespace ranges {
  template<class T>
    constexpr T* default_construct_at(T* location);
}
```

5.3 Modify 25.11.3 [uninitialized.construct.default] of [N4762] as follows

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

  Effects: Equivalent to:
    for (; first != last; ++first)
-     ::new (voidify(*first)) typename iterator_traits<NoThrowForwardIterator>::value_type;
+     default_construct_at(addressof(*first));

  namespace ranges {
    template<no-throw-forward-iterator I, no-throw-sentinel-for<I> S>
      requires default_initializable<iter_value_t<I>>
-     I uninitialized_default_construct(I first, S last);
+     constexpr I uninitialized_default_construct(I first, S last);
    template<no-throw-forward-range R>
      requires default_initializable<range_value_t<R>>
-     borrowed_iterator_t<R> uninitialized_default_construct(R&& r);
+     constexpr borrowed_iterator_t<R> uninitialized_default_construct(R&& r);
  }

  Effects: Equivalent to:
    for (; first != last; ++first)
```

```
-     ::new (voidify(*first)) remove_reference_t<iter_reference_t<I>>;
+     default_construct_at(addressof(*first));
    return first;

  template<class NoThrowForwardIterator, class Size>
-   NoThrowForwardIterator uninitialized_default_construct_n(NoThrowForwardIterator first, Size n);
+   constexpr NoThrowForwardIterator
+     uninitialized_default_construct_n(NoThrowForwardIterator first, Size n);

  Effects: Equivalent to:
    for (; n > 0; (void)++first, --n)
-     ::new (voidify(*first)) typename iterator_traits<NoThrowForwardIterator>::value_type;
+     default_construct_at(addressof(*first));
    return first;

  namespace ranges {
    template<no-throw-forward-iterator I>
      requires default_initializable<iter_value_t<I>>
-     I uninitialized_default_construct_n(I first, iter_difference_t<I> n);
+     constexpr I uninitialized_default_construct_n(I first, iter_difference_t<I> n);
  }

  Effects: Equivalent to:
    return uninitialized_default_construct(counted_iterator(first, n),
                                           default_sentinel).base();
```

5.4 Modify 25.11.4 [uninitialized.construct.value] of [N4762] as follows

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

  Effects: Equivalent to:
    for (; first != last; ++first)
-     ::new (voidify(*first)) typename iterator_traits<NoThrowForwardIterator>::value_type();
+     construct_at(addressof(*first));

  namespace ranges {
    template<no-throw-forward-iterator I, no-throw-sentinel-for<I> S>
      requires value_initializable<iter_value_t<I>>
-     I uninitialized_value_construct(I first, S last);
+     constexpr I uninitialized_value_construct(I first, S last);
    template<no-throw-forward-range R>
      requires value_initializable<range_value_t<R>>
-     borrowed_iterator_t<R> uninitialized_value_construct(R&& r);
+     constexpr borrowed_iterator_t<R> uninitialized_value_construct(R&& r);
  }

  Effects: Equivalent to:
    for (; first != last; ++first)
-     ::new (voidify(*first)) remove_reference_t<iter_reference_t<I>>();
+     construct_at(addressof(*first));
```

```
    return first;

  template<class NoThrowForwardIterator, class Size>
-   NoThrowForwardIterator uninitialized_value_construct_n(NoThrowForwardIterator first, Size n);
+   constexpr NoThrowForwardIterator
+     uninitialized_value_construct_n(NoThrowForwardIterator first, Size n);

  Effects: Equivalent to:
    for (; n > 0; (void)++first, --n)
-     ::new (voidify(*first)) typename iterator_traits<NoThrowForwardIterator>::value_type();
+     construct_at(addressof(*first));
    return first;

  namespace ranges {
    template<no-throw-forward-iterator I>
      requires value_initializable<iter_value_t<I>>
-     I uninitialized_value_construct_n(I first, iter_difference_t<I> n);
+     constexpr I uninitialized_value_construct_n(I first, iter_difference_t<I> n);
  }

  Effects: Equivalent to:
    return uninitialized_value_construct(counted_iterator(first, n),
                                         value_sentinel).base();
```

5.5 Modify 25.11.5 [uninitialized.copy] of [N4762] as follows

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

  Preconditions:
    result + [0, (last - first)) does not overlap with [first, last).

  Effects: Equivalent to:
    for (; first != last; ++result, (void) ++first)
-     ::new (voidify(*result))
-       typename iterator_traits<NoThrowForwardIterator>::value_type(*first);
+     construct_at(addressof(*result), *first);

  Returns: result.

  namespace ranges {
    template<input_iterator I, sentinel_for<I> S1,
             no-throw-forward-iterator O, no-throw-sentinel-for<O> S2>
      requires constructible_from<iter_value_t<O>, iter_reference_t<I>>
-     uninitialized_copy_result<I, O>
+     constexpr 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<borrowed_iterator_t<IR>, borrowed_iterator_t<OR>>
+     constexpr uninitialized_copy_result<borrowed_iterator_t<IR>, borrowed_iterator_t<OR>>
        uninitialized_copy(IR&& in_range, OR&& out_range);
  }

  Preconditions:
    [ofirst, olast) does not overlap with [ifirst, ilast).

  Effects: Equivalent to:
    for (; ifirst != ilast && ofirst != olast; ++ofirst, (void)++ifirst)
-     ::new (voidify(*ofirst)) remove_reference_t<iter_reference_t<O>>(*ifirst);
+     construct_at(addressof(*ofirst), *ifirst);
    return {std::move(ifirst), ofirst};

  template<class InputIterator, class Size, class NoThrowForwardIterator>
-   NoThrowForwardIterator uninitialized_copy_n(InputIterator first, Size n,
-                                               NoThrowForwardIterator result);
+   constexpr NoThrowForwardIterator
+     uninitialized_copy_n(InputIterator first, Size n, NoThrowForwardIterator result);

  Preconditions:
    result + [0, n) does not overlap with first + [0, n).

  Effects: Equivalent to:
    for ( ; n > 0; ++result, (void) ++first, --n)
-     ::new (voidify(*result))
-       typename iterator_traits<NoThrowForwardIterator>::value_type(*first);
+     construct_at(addressof(*result), *first);

  Returns: result.

  namespace ranges {
    template<input_iterator I, no-throw-forward-iterator O, no-throw-sentinel-for<O> S>
      requires constructible_from<iter_value_t<O>, iter_reference_t<I>>
-     uninitialized_copy_n_result<I, O>
+     constexpr uninitialized_copy_n_result<I, O>
        uninitialized_copy_n(I ifirst, iter_difference_t<I> n, O ofirst, S olast);
  }

  Preconditions:
    [ofirst, olast) does not overlap with ifirst + [0, n).

  Effects: Equivalent to:
    auto t = uninitialized_copy(counted_iterator(ifirst, n),
                                default_sentinel, ofirst, olast);
    return {std::move(t.in).base(), t.out};
```

5.6 Modify 25.11.6 [uninitialized.move] of [N4762] as follows

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

  Preconditions:
    result + [0, (last - first)) does not overlap with [first, last).

  Effects: Equivalent to:
    for (; first != last; ++result, (void) ++first)
-     ::new (voidify(*result))
-       typename iterator_traits<NoThrowForwardIterator>::value_type(std::move(*first));
+     construct_at(addressof(*result), std::move(*first));

  Returns: result.

  namespace ranges {
    template<input_iterator I, sentinel_for<I> S1,
            no-throw-forward-iterator O, no-throw-sentinel-for<O> S2>
      requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
-     uninitialized_move_result<I, O>
+     constexpr 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<borrowed_iterator_t<IR>, borrowed_iterator_t<OR>>
+     constexpr uninitialized_move_result<borrowed_iterator_t<IR>, borrowed_iterator_t<OR>>
        uninitialized_move(IR&& in_range, OR&& out_range);
  }

  Preconditions:
    [ofirst, olast) does not overlap with [ifirst, ilast).

  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);
+     construct_at(addressof(*ofirst), ranges::iter_move(ifirst);
    return {std::move(ifirst), ofirst};

  [Note 1: If an exception is thrown, some objects in the range [first, last) are left in a valid, but un

  template<class InputIterator, class Size, class NoThrowForwardIterator>
-   NoThrowForwardIterator uninitialized_move_n(InputIterator first, Size n,
-                                               NoThrowForwardIterator result);
+   constexpr NoThrowForwardIterator
+     uninitialized_move_n(InputIterator first, Size n, NoThrowForwardIterator result);

  Preconditions:
    result + [0, n) does not overlap with first + [0, n).

  Effects: Equivalent to:
    for ( ; n > 0; ++result, (void) ++first, --n)
-     ::new (voidify(*result))
-       typename iterator_traits<NoThrowForwardIterator>::value_type(std::move(*first));
+     construct_at(addressof(*result), std::move(*first));
```

```
  Returns: result.

  namespace ranges {
    template<input_iterator I, no-throw-forward-iterator O, no-throw-sentinel-for<O> S>
      requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
-     uninitialized_move_n_result<I, O>
+     constexpr uninitialized_move_n_result<I, O>
        uninitialized_move_n(I ifirst, iter_difference_t<I> n, O ofirst, S olast);
  }

  Preconditions:
    [ofirst, olast) does not overlap with ifirst + [0, n).

  Effects: Equivalent to:
    auto t = uninitialized_move(counted_iterator(ifirst, n),
                                default_sentinel, ofirst, olast);
    return {std::move(t.in).base(), t.out};

  [Note 2: If an exception is thrown, some objects in the range first + [0, n) are left in a valid but un
```

5.7 Modify 25.11.7 [uninitialized.fill] of [N4762] as follows

```
  template<class NoThrowForwardIterator, class T>
-   void uninitialized_fill(NoThrowForwardIterator first, NoThrowForwardIterator last, const T& x);
+   constexpr void uninitialized_fill(NoThrowForwardIterator first,
+                                     NoThrowForwardIterator last, const T& x);
  Effects: Equivalent to:
    for (; first != last; ++first)
-     ::new (voidify(*first))
-       typename iterator_traits<NoThrowForwardIterator>::value_type(x);
+     construct_at(addressof(*first), x);

  namespace ranges {
    template<no-throw-forward-iterator I, no-throw-sentinel-for<I> S, class T>
      requires constructible_from<iter_value_t<I>, const T&>
-     I uninitialized_fill(I first, S last, const T& x);
+     constexpr 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&>
-     borrowed_iterator_t<R> uninitialized_fill(R&& r, const T& x);
+     constexpr borrowed_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);
+     construct_at(addressof(*first), x);
    return first;

  template<class NoThrowForwardIterator, class Size, class T>
-   NoThrowForwardIterator uninitialized_fill_n(NoThrowForwardIterator first, Size n, const T& x);
+   constexpr NoThrowForwardIterator uninitialized_fill_n(NoThrowForwardIterator first,
+                                                         Size n, const T& x);
```

```
  Effects: Equivalent to:
    for (; n--; ++first)
-     ::new (voidify(*first))
-       typename iterator_traits<NoThrowForwardIterator>::value_type(x);
+     construct_at(addressof(*first), 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);
+     constexpr I uninitialized_fill_n(I first, iter_difference_t<I> n, const T& x);
  }

  Effects: Equivalent to:
    return uninitialized_fill(counted_iterator(first, n), default_sentinel, x).base();
```

5.8 Add 25.11.8 [special.default_construct] to [N4762]

```
  template<class T>
    constexpr T* default_construct_at(T* location);

  namespace ranges {
    template<class T>
      constexpr T* default_construct_at(T* location);
  }

  Constraints:
    The expression ::new (declval<void*>()) T is well-formed when treated as an unevaluated operand.

  Effects: Equivalent to:
    return ::new (voidify(*location)) T;
```

5.9 Feature test macro

Increase the value of `__cpp_lib_raw_memory_algorithms` to the date of adoption.

6 Implementation Experience

— `Microsoft STL` This has been partially implemented for support of constexpr vector in MSVC STL.

7 Acknowledgements

Big thanks go to JeanHeyd Meneide for proof reading and discussions.

8 References

[N4762] Richard Smith. 2018-07-07. Working Draft, Standard for Programming Language C++.
https://wg21.link/n4762

[P0784R7] 2019. More constexpr containers.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0784r7.html