Introduce `default_construct_at`

Document #:	DXXXXR0
Date:	2021-11-21
Project:	Programming Language C++
Audience:	EWG

Reply-to:	Michael Schellenberger Costa <mschellenbergercosta@googlemail.com>

1 Revision History

- R0
  - Initial draft

2 Introduction

This paper proposes adding a new utility function `default_construct_at` to the standard library that is usable inside a core constant expression. This is essentially a followup to [P2283R1] which added `constexpr` support for most of the specialized memory algorithms but factored out this core wording change.

3 Motivation and Scope

Currently it is impossible to *default*-initialize a variable at given memory location inside a core constant expression. The issue is that we cannot use placement new as that is forbidden during constant evaluation and we cannot use `construct_at` as this always *value*-initializes.

However, there is no reason why *default*-initialization should be forbidden inside a core constant expression. In fact it is currently possible to simply write `int meow;` and leave the variable `meow` uninitialized. As long as `meow` is not read before it is assigned a value all is fine.

Finally, with the machinery in place we can add `constexpr` to the remaining specialized memory algorithms.

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 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;
```

5.5 Feature test macro

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

6 Implementation Experience

— This already works under clang without any issue (https://godbolt.org/z/oGY19vPKc);
— To the authors knowledge MSVC simply replaces calls to `construct_at` with “fairy magic”, so it is rather straithforward to implement.
— Need to further investigate GCCs behavior

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

[P2283R1] Michael Schellenberger Costa. 2021-04-19. constexpr for specialized memory algorithms. https://wg21.link/p2283r1