Document Number:	P3480R6
Date:	2025-06-18
Reply-to:	Matthias Kretz <m.kretz@gsi.de>
Audience:	LWG
Target:	C++26

STD::SIMD IS A RANGE

ABSTRACT

P1928 “std::simd — merge data-parallel types from the Parallelism TS 2” promised a paper on making `simd` a range. This paper explores the addition of iterators to `basic_simd` and `basic_simd_mask`.

CONTENTS

1 CHANGELOG

1.1 CHANGES FROM REVISION 0

Previous revision: P3480R0

- Simplify to a single iterator class template.
- Remove incorrect `operator-` overload.
- Discuss design choice of using a sentinel type for `end()`.

1.2 CHANGES FROM REVISION 1

Previous revision: P3480R1

- Add SG9 poll results.
- Use `default_sentinel_t` instead of a new sentinel type.
- Use an almost-mutable iterator type as directed by SG9 for non-const `begin()`.
- Fix `for_each` example to use `ranges` version.

1.3 CHANGES FROM REVISION 2

Previous revision: P3480R2

- Ask about tuple interface for `simd`.
- Clarify where [simd.iterator] should go.
- Provide proper wording.
- Fix `<=>` comparison with `default_sentinel_t`.
- Bump feature test macro?

1.4 CHANGES FROM REVISION 3

Previous revision: P3480R3

- Adjust names after introduction of the simd subnamespace.

1.5 changes from revision 4

Previous revision: P3480R4

- Change `iterator_category` to `input_iterator_tag` and add `iterator_concept` as `random_access_iterator_tag`.
- Change `int` to *`simd-size-type`* in constructor.
- Swap `constexpr` and `friend`.
- Drop `operator<=>` with `default_sentinel_t`.
- Drop `#if LEWG_WANTS_CONVERSION`, keeping the converting constructor.
- Drop precondition on *`offset_`* for `operator*`.

1.6 changes from revision 5

Previous revision: P3480R5

- Wording updates after first round of LWG review:
  - Add preconditions that `offset_` stays inside the range of 0 to `size()`.
  - Make internal constructor exposition-only and private.
  - Remove unncessary `std::`.
  - Take the `addressof` before initializing `data_`.
  - Remove unncessary precondition on `operator*`.
  - Express as many operations as possible in terms of `+=` and `-=`.
  - Add inline implementation for `begin` and `end` of `basic_simd` and `basic_simd_mask`.
  - Mark `begin` and `end` as well as the private iterator constructor `noexcept`.
- Wording updates after second round of LWG review:
  - More `noexcept` on difference and equality.
  - Use by-value parameters on `+` and `-`.

2 STRAW POLLS

2.1 SG9 at Wrocław 2024

Poll: We want `std::basic_simd` to be a range.

SF	F	N	A	SA
6	2	0	0	0

Poll: We want `std::basic_simd` to be a common range.

SF	F	N	A	SA
0	0	3	4	1

Poll: We want `std::basic_simd::operator[]` and `std::basic_simd::begin/end` in C++26 without mutation support, knowing that we might not be able to do it later due to ABI issues (e.g. `decltype(auto) f(std::simd<float> x) { return x[0]; }` could change return type).

SF	F	N	A	SA
6	2	0	0	0

Poll: We want `std::basic_simd::iterator` and `std::basic_simd::const_iterator` to be different types to make the transition to mutable iteration easier. This also means adding a non-const `begin()` overload that returns a different type than the const `begin()` overload but currently has the same semantics.

SF	F	N	A	SA
2	3	3	0	0

Poll: Use `std::default_sentinel_t` instead of *`simd-iterator-sentinel`*.
→ unanimous consent

Poll: Forward P3480R1 with the changes above to LEWG for inclusion in C++26.

SF	F	N	A	SA
7	1	0	0	0

2.2 LEWG telecon 2025-04-08

ACTION: Apply the fix: apply what's in `#if LEWG_WANTS_CONVERSION`

Poll: Modify "P3480R4: std::simd is a range" with the above action items and forward to LWG for C++29 (with a recommendation to make this a DR for 26)

SF	F	N	A	SA
10	8	0	0	0

Poll: Modify "P3480R4: std::simd is a range" with the above action items and forward to LWG with a recommendation to apply for C++26 (if possible).

SF	F	N	A	SA
12	5	2	0	0

3 Introduction, or why simd wasn't a range in the TS

The Parallelism TS 2 was based on C++17. Ranges were added in C++20. Before ranges, an iterator category was tied to whether `operator*` of iterators returned an lvalue reference. Since `basic_simd` and `basic_simd_mask` objects are not composed of sub-objects (in other words, a `simd<int>` contains no `int` *objects*), `operator[]` returns prvalues (or a proxy reference in the TS for the non-const case). An iterator needs to do the same and thus never could be in any other iterator category than *Cpp17InputIterator*. In reality, the iterator category always was "random access" (never contiguous; because while `basic_simd` is a contiguous range in memory it isn't one in the object model of C++). In order to not cement that mismatch, it was never proposed to make `basic_simd`/`basic_simd_mask` a range for the TS.

Now that the iterator concepts don't require an lvalue reference anymore we can easily make `basic_simd`/`basic_simd_mask` a read-only range. Iterator dereference would return a prvalue (a copy of the value stored in the `basic_simd`/`basic_simd_mask` object). In addition, the abstraction of a sentinel instead of an iterator pointing beyond the last value of the `basic_simd` seems like a useful tool for `basic_simd`.

4 Motivation

After the technical reasons for *not* adding iterators to `basic_simd`/`basic_simd_mask` are resolved, we still need to consider why `basic_simd` should be a range in the first place.

5 Integration with the standard library

We can improve integration of `basic_simd`/`basic_simd_mask` with the rest of the standard library. By making `basic_simd`/`basic_simd_mask` a range many of the existing facilities in the standard library become easily accessible. All of these facilities do work as intended — in other words: presenting `basic_simd`/`basic_simd_mask` as a range matches on the semantic level, not only syntactically.

5.1 read-only subscript should imply read-only iteration

With the latest WD we can write

```
std::datapar::simd<int> v = ...;
for (int i = 0; i < v.size(); ++i) {
  do_something(v[i]);
}
```

Why then, can we not also write

```
for (auto x : v) {
  do_something(x);
}
```

and

```
std::ranges::for_each(v.begin(), v.end(), [](auto x) {
  do_something(x);
});
```

and

```
v | std::views::filter([](auto x) { return x > 0; }) | std::ranges::to<std::vector>();
```

C++ users have learned that whenever a for loop with subscript does what they need to do, then a ranged for loop, standard algorithm, or range adaptor are valid alternatives. This expectation should not get an exception with `basic_simd` and `basic_simd_mask`.

5.2 present a range of simd as a range of simd's value-type

In some applications it is more efficient (and simpler) to work with `basic_simd` objects internally, instead of constantly doing loads and stores. Thus a fairly simple container that comes up in applications could be `std::vector<std::datapar::simd<float>>`. On I/O such an application typically cannot communicate in `basic_simd` objects anymore. Instead it needs to present a range of `floats`. Read-only iterators on `basic_simd` do not help with the input side. But for output we can easily turn the `vector<simd<float>>` into a range of `float`:

```
std::vector<std::datapar::simd<float>> data;
auto range_of_float = data | std::views::join;
```

6 Downsides of making simd a range

Really, I can't think of any downsides of making `basic_simd/basic_simd_mask` a range. In principle one could argue that `basic_simd/basic_simd_mask` is not a container [P0851R0]. Consequently, it shouldn't have a container interface and thus no iterators. But then we should probably remove the subscript operator as well.

7 Design choice: sentinel

The `basic_simd` iterator type must have a reference/pointer to the `basic_simd` object it is iterating together with an offset, where into the `basic_simd` it is pointing. Because of these two members (and their type), the iterator already knows the complete bounds of the range it is pointing into. Consequently, a single `basic_simd` iterator can always determine whether it points at the beginning or end of the range, it doesn't need to compare against another offset. A sentinel type allows asking that question via `operator==`. Thus, instead of comparing two runtime offset members on `operator==`, a compare against a sentinel is implemented as a compare against a compile-time constant. This makes it easier for the compiler to optimize and reduces the size of the `end()` sentinel to a single byte (empty type).

8 Open question: add tuple interface

`std::array` implements the tuple interface. Should `std::simd` also implement `tuple_size`, `tuple_element`, and `get`?

9 Wording

9.1 Feature test macro

In [version.syn] bump the `__cpp_lib_simd` version.

9.2 Add [simd.iterator]

Add a new subclause before §29.10.6 [simd.class]:

[simd]

(9.2.1) **29.10.6 Class *simd-iterator* [simd.iterator]**

```
namespace std::datapar {
  template <class V>
  class simd-iterator {          // exposition only
    V* data_ = nullptr;          // exposition only
    simd-size-type offset_ = 0; // exposition only

    constexpr simd-iterator(V& d, simd-size-type off) noexcept; // exposition only

  public:
    using value_type = typename V::value_type;
    using iterator_category = input_iterator_tag;
    using iterator_concept = random_access_iterator_tag;
    using difference_type = simd-size-type;
```

```
    constexpr simd-iterator() = default;

    constexpr simd-iterator(const simd-iterator&) = default;
    constexpr simd-iterator& operator=(const simd-iterator&) = default;

    constexpr simd-iterator(const simd-iterator<remove_const_t<V>>&) requires is_const_v<V>;

    constexpr value_type operator*() const;

    constexpr simd-iterator& operator++();
    constexpr simd-iterator operator++(int);
    constexpr simd-iterator& operator--();
    constexpr simd-iterator operator--(int);

    constexpr simd-iterator& operator+=(difference_type n);
    constexpr simd-iterator& operator-=(difference_type n);

    constexpr value_type operator[](difference_type n) const;

    friend constexpr bool operator==(simd-iterator a, simd-iterator b) = default;
    friend constexpr bool operator==(simd-iterator a, default_sentinel_t) noexcept;
    friend constexpr auto operator<=>(simd-iterator a, simd-iterator b);

    friend constexpr simd-iterator operator+(simd-iterator i, difference_type n);
    friend constexpr simd-iterator operator+(difference_type n, simd-iterator i);
    friend constexpr simd-iterator operator-(simd-iterator i, difference_type n);

    friend constexpr difference_type operator-(simd-iterator a, simd-iterator b);
    friend constexpr difference_type operator-(simd-iterator i, default_sentinel_t) noexcept;
    friend constexpr difference_type operator-(default_sentinel_t, simd-iterator i) noexcept;
  };
}
```

`constexpr simd-iterator(V& d, simd-size-type off) noexcept;`

1 *Effects*: Initializes *data_* with `addressof(d)` and *offset_* with `off`.

`constexpr simd-iterator(const simd-iterator<remove_const_t<V>>& i) requires is_const_v<V>;`

2 *Effects*: Initializes *data_* with `i.`*data_* and *offset_* with `i.`*offset_*.

`constexpr value_type operator*() const;`

3 *Effects*: Equivalent to: `return (*`*data_*`)[`*offset_*`];`

```
constexpr simd-iterator& operator++();
```

4 *Effects*: Equivalent to: `return *this += 1;`

```
constexpr simd-iterator operator++(int);
```

5 *Effects*: Equivalent to:

```
simd-iterator tmp = *this;
*this += 1;
return tmp;
```

```
constexpr simd-iterator& operator--();
```

6 *Effects*: Equivalent to: `return *this -= 1;`

```
constexpr simd-iterator operator--(int);
```

7 *Effects*: Equivalent to:

```
simd-iterator tmp = *this;
*this -= 1;
return tmp;
```

```
constexpr simd-iterator& operator+=(difference_type n);
```

8 *Preconditions*: `offset_ + n` is in the range $[0, \texttt{V::size()}]$.

9 *Effects*: Equivalent to:

```
offset_ += n;
return *this;
```

```
constexpr simd-iterator& operator-=(difference_type n);
```

10 *Preconditions*: `offset_ - n` is in the range $[0, \texttt{V::size()}]$.

11 *Effects*: Equivalent to:

```
offset_ -= n;
return *this;
```

```
constexpr value_type operator[](difference_type n) const;
```

12 *Effects*: Equivalent to: `return (*data_)[offset_ + n];`

```
friend constexpr bool operator==(simd-iterator i, default_sentinel_t) noexcept;
```

13 *Effects*: Equivalent to: `return i.offset_ == V::size();`

```
friend constexpr auto operator<=>(simd-iterator a, simd-iterator b);
```

14 *Preconditions*: `a.data_ == b.data_` is `true`.

15 *Effects*: Equivalent to: `return a.offset_ <=> b.offset_;`

```
friend constexpr simd-iterator operator+(simd-iterator i, difference_type n);
friend constexpr simd-iterator operator+(difference_type n, simd-iterator i);
```

16 *Effects*: Equivalent to: `return i += n;`

```
friend constexpr simd-iterator operator-(simd-iterator i, difference_type n);
```

17 *Effects*: Equivalent to: `return i -= n;`

```
friend constexpr difference_type operator-(simd-iterator a, simd-iterator b);
```

18 *Preconditions*: `a.data_ == b.data_` is `true`.

19 *Effects*: Equivalent to: `return a.offset_ - b.offset_;`

```
friend constexpr difference_type operator-(simd-iterator i, default_sentinel_t) noexcept;
```

20 *Effects*: Equivalent to: `return i.offset_ - V::size();`

```
friend constexpr difference_type operator-(default_sentinel_t, simd-iterator i) noexcept;
```

21 *Effects*: Equivalent to: `return V::size() - i.offset_;`

9.3 modify [simd.overview]

[simd.overview]

```
template<class T, class Abi> class basic_simd {
public:
  using value_type = T;
  using mask_type = basic_simd_mask<sizeof(T), Abi>;
  using abi_type = Abi;
  using iterator = simd-iterator<basic_simd>;
  using const_iterator = simd-iterator<const basic_simd>;

  constexpr iterator begin() noexcept { return {*this, 0}; }
  constexpr const_iterator begin() const noexcept { return {*this, 0}; }
  constexpr const_iterator cbegin() const noexcept { return {*this, 0}; }
  constexpr default_sentinel_t end() const noexcept { return {}; }
  constexpr default_sentinel_t cend() const noexcept { return {}; }
```

9.4 modify [simd.mask.overview]

[simd.mask.overview]

```
template<size_t Bytes, class Abi> class basic_simd_mask {
public:
  using value_type = bool;
  using abi_type = Abi;
  using iterator = simd-iterator<basic_simd_mask>;
  using const_iterator = simd-iterator<const basic_simd_mask>;

  constexpr iterator begin() noexcept { return {*this, 0}; }
  constexpr const_iterator begin() const noexcept { return {*this, 0}; }
  constexpr const_iterator cbegin() const noexcept { return {*this, 0}; }
  constexpr default_sentinel_t end() const noexcept { return {}; }
  constexpr default_sentinel_t cend() const noexcept { return {}; }
```

A Bibliography

[P0851R0] Matthias Kretz. *P0851R0: simd<T> is neither a product type nor a container type*. ISO/IEC C++ Standards Committee Paper. 2017. URL: https://wg21.link/p0851r0.