Document Number:	P3480R0
Date:	2024-10-16
Reply-to:	Matthias Kretz <m.kretz@gsi.de>
Audience:	SG9, LEWG
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

(placeholder)

2 STRAW POLLS

(placeholder)

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 P1928R12 we can write

```
std::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::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::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::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 WORDING

This is just a sketch derived from my implementation of `basic_simd` and `basic_simd_mask` iterators.

Add the following:

[simd.iterators]

```
namespace std
{
  class simd-iterator-sentinel {};

  template <typename T, typename Abi>
    class simd-iterator
    {
      const basic_simd<T, Abi>* data_ = nullptr; // exposition only
      int offset_ = 0;                           // exposition only

    public:
      using value_type = T;
      using iterator_category = std::random_access_iterator_tag;
      using difference_type = int;

      constexpr simd-iterator() = default;

      constexpr
      simd-iterator(const basic_simd<T, Abi>& d, int x)
      : data_(&d), offset_(x)
      {}

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

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

      constexpr value_type
      operator*() const
      { return (*data_)[offset_]; }

      constexpr simd-iterator&
      operator++()
      {
        ++offset_;
        return *this;
      }

      constexpr simd-iterator
```

```
operator++(int)
{
  simd-iterator r = *this;
  ++offset_;
  return r;
}

constexpr simd-iterator&
operator--()
{
  --offset_;
  return *this;
}

constexpr simd-iterator
operator--(int)
{
  simd-iterator r = *this;
  --offset_;
  return r;
}

constexpr difference_type
operator-(simd-iterator rhs) const
{ return offset_ - rhs.offset_; }

constexpr friend difference_type
operator-(simd-iterator it, simd-iterator-sentinel)
{ return it.offset_ - difference_type(_Vp::size.value); }

constexpr friend difference_type
operator-(simd-iterator-sentinel, simd-iterator it)
{ return difference_type(_Vp::size.value) - it.offset_; }

constexpr friend simd-iterator
operator+(difference_type x, const simd-iterator& it)
{ return simd-iterator(*it.data_, it.offset_ + x); }

constexpr friend simd-iterator
operator+(const simd-iterator& it, difference_type x)
{ return simd-iterator(*it.data_, it.offset_ + x); }

constexpr friend simd-iterator
operator-(difference_type x, const simd-iterator& it)
{ return simd-iterator(*it.data_, it.offset_ - x); }
```

```
    constexpr friend simd-iterator
    operator-(const simd-iterator& it, difference_type x)
    { return simd-iterator(*it.data_, it.offset_ - x); }

    constexpr simd-iterator&
    operator+=(difference_type x)
    {
      offset_ += x;
      return *this;
    }

    constexpr simd-iterator&
    operator-=(difference_type x)
    {
      offset_ -= x;
      return *this;
    }

    constexpr value_type
    operator[](difference_type i) const
    { return (*data_)[offset_ + i]; }

    constexpr friend auto operator<=>(simd-iterator a, simd-iterator b)
    { return a.offset_ <=> b.offset_; }

    constexpr friend bool operator==(simd-iterator a, simd-iterator b) = default;

    constexpr friend bool operator==(simd-iterator a, simd-iterator-sentinel)
    { return a.offset_ == difference_type(_Vp::size.value); }
  };

template <size_t Bs, typename Abi>
  class mask-iterator
  {
    const basic_mask<Bs, Abi>* data_ = nullptr; // exposition only
    int offset_ = 0;                            // exposition only

  public:
    using value_type = bool;
    using iterator_category = std::random_access_iterator_tag;
    using difference_type = int;

    constexpr mask-iterator() = default;

    constexpr
    mask-iterator(const basic_mask<Bs, Abi>& d, int x)
```

```
: data_(&d), offset_(x)
{}

constexpr
mask-iterator(const mask-iterator &) = default;

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

constexpr value_type
operator*() const
{ return (*data_)[offset_]; }

constexpr mask-iterator&
operator++()
{
  ++offset_;
  return *this;
}

constexpr mask-iterator
operator++(int)
{
  mask-iterator r = *this;
  ++offset_;
  return r;
}

constexpr mask-iterator&
operator--()
{
  --offset_;
  return *this;
}

constexpr mask-iterator
operator--(int)
{
  mask-iterator r = *this;
  --offset_;
  return r;
}

constexpr difference_type
operator-(mask-iterator rhs) const
{ return offset_ - rhs.offset_; }
```

```
    constexpr friend mask-iterator
    operator+(difference_type x, const mask-iterator& it)
    { return mask-iterator(*it.data_, it.offset_ + x); }

    constexpr friend mask-iterator
    operator+(const mask-iterator& it, difference_type x)
    { return mask-iterator(*it.data_, it.offset_ + x); }

    constexpr friend mask-iterator
    operator-(difference_type x, const mask-iterator& it)
    { return mask-iterator(*it.data_, it.offset_ - x); }

    constexpr friend mask-iterator
    operator-(const mask-iterator& it, difference_type x)
    { return mask-iterator(*it.data_, it.offset_ - x); }

    constexpr mask-iterator&
    operator+=(difference_type x)
    {
      offset_ += x;
      return *this;
    }

    constexpr mask-iterator&
    operator-=(difference_type x)
    {
      offset_ -= x;
      return *this;
    }

    constexpr value_type
    operator[](difference_type i) const
    { return (*data_)[offset_ + i]; }

    constexpr friend auto operator<=>(mask-iterator a, mask-iterator b)
    { return a.offset_ <=> b.offset_; }

    constexpr friend bool operator==(mask-iterator a, mask-iterator b) = default;

    constexpr friend bool operator==(mask-iterator a, simd-iterator-sentinel)
    { return a.offset_ == difference_type(_Vp::size.value); }
  };
}
```

[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<T, Abi>;
  using const_iterator = iterator;

  constexpr const_iterator begin() const;
  constexpr const_iterator cbegin() const;
  constexpr simd-iterator-sentinel end() const;
  constexpr simd-iterator-sentinel cend() const;
```

[simd.mask.overview]

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

  constexpr const_iterator begin() const;
  constexpr const_iterator cbegin() const;
  constexpr simd-iterator-sentinel end() const;
  constexpr simd-iterator-sentinel cend() const;
```

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.