Document Number:	P3287R0
Date:	2024-05-22
Reply-to:	Matthias Kretz <m.kretz@gsi.de>
Audience:	LEWG
Target:	C++26

Exploration of namespaces for std::simd

ABSTRACT

In recent discussions about `simd` in LEWG, notably on 2023-06-16 while discussing `permute`, `expand`, and `compress`, there was a request for a paper exploring placing all `simd` non-member functions into a sub-namespace. ...or potentially any other means of using namespaces to improve the `simd` API.
This paper explores a few ideas.

CONTENTS

1 CHANGELOG

(placeholder)

2 STRAW POLLS

(placeholder)

3 INTRODUCTION & MOTIVATION

Using the example of `std::permute(basic_simd, idx_perm)`, one of the unavoidable LEWG discussions/decisions is about whether `simd` can grab the name "permute", potentially blocking its use for other facilities in the standard library.[1] With P3067R0 ("Provide named permutation functions for `std::simd`"), the list of non-member functions to add to `std::` becomes: `permute`, `expand`, `compress`, `grow`, `stride`, `chunk`, `reverse`, `repeat_all`, `repeat_each`, `transpose`, `zip`, `unzip`, `cat`, `extract`, `rotate`, `shift_left`, `shift_right`, and `align`. All of these names would likely need a `simd` prefix if they want to go into `std::`.

And then we're adding `basic_simd` overloads for all of `<cmath>` and `<bit>`, ...

So we need to understand whether there are viable alternatives to `simd` naming. This paper tries to explore the field as far as I believe is still sensible. The goal is to come up with a consistent naming strategy for everything related to `simd`.

3.1 SIMD-GENERIC PROGRAMMING

In this paper I want to use the term *SIMD-generic* programming. Note that in the space of types, `basic_simd<T>` is a generalization of `T` or — vice-versa — `T` is the degenerate case of `basic_simd<T>`. (The same is true for `basic_simd_mask` and `bool`.) We've touched upon this when we talked about regularity and how `basic_simd<T>` is designed to retain regularity of each individual element inside the `basic_simd`, leading to something I called "data-parallel regularity" of `basic_simd<T>`, for lack of an existing term.

The `simd` design aims to allow users to replace `T` with `basic_simd<T>` in their code without requiring any further code changes. If this works (and because of branching on individual values of `T` it cannot work for all code) I call such code SIMD-generic.

The following text uses this term because the use of namespaces opens an interesting facility to opt in and out of some aspects of SIMD-generic programming.

1 Just to clarify, I agree with the concern and I feel uneasy with the need for `simd` to grab as many names as it would need to.

4 EXPLORATION

When exploring naming and namespacing, I use the following functions to showcase the effect. I then try to come up with all ways to use and abuse the facilities. In addition I mention the effect of the choice on SIMD-generic programming. To complete the picture, I added a concept that seems like something we might want to add in the future, but for which there is currently no proposal coming forward.

Note: we have to discuss the range vs. iterator argument to load/gather seperately. This paper does not explore the issue. I also removed `constexpr` and `noexcept` since they are irrelevant to the exploration at hand.

1. `basic_simd` generator

   Status quo (P1928R9):

```
std::simd<int> iota([](int i) { return i; });
```

2. `basic_simd` load from contiguous range

   Status quo (P1928R9):

```
std::vector<int> data = {...};
std::simd<int> chunk(data.begin());
```

3. `basic_simd` gather from contiguous range

   Status quo (P2664R6):

```
std::vector<int> data = {/*...*/};
std::simd<int> idxs = /*...*/;
std::simd<int> std::gather_from(data, idxs);
```

4. `basic_simd` permutations

   Status quo (P2664R6):

```
std::simd<int> v = /*...*/;
std::simd<int> v2 = std::permute(v, [](int i) { return i ^ 1; });
```

5. `basic_simd` ternary operator replacement

   Status quo (P1928R9):

```
std::simd<int> v = /*...*/;
std::simd<int> abs = std::simd_select(v >= 0, v, -v);
```

6. Math functions and algorithms

   Status quo (P1928R9):

```
std::simd<float> x = /*...*/;
std::simd<float> y = std::exp(x);
std::simd<float> z = std::min(x, y);
```

7. Mask reductions

   Status quo (P1928R9):

```
std::simd<float> x = /*...*/;
if (std::all_of(x > 0)) /*...*/
```

8. Simd concepts

   - Constrain whether a type is a `basic_simd<T>` with `std::integral<T>`.
   - Constrain whether a type is either `std::integral` or a `basic_simd<T>` with `std::integral<T>`.

4.1 Status quo (latest revision of SIMD papers)

Pros

- `std::simd` is as concise as it could possibly be.
- Fairly good support for SIMD-generic programming.

Cons

- We have a mix of non-member functions with and without `simd_` prefix.
- Most non-member functions would be nicer to read in code without the `simd` prefix. We introduce the prefix only because we are wary of the "name grab" in `std`. I. e. the motivation for the current naming scheme isn't the design of the `simd` API, but the freedom to evolve the standard library in the future.
- Load and gather (which are very similar in loading a SIMD "register" from a contiguous range of values) are inconsistent: One uses a constructor and member function, the other only a non-member function.
- Loads, stores, and the `simd` generator constructor cannot be used in SIMD-generic code.

4.2 Alternative 1: every function is a non-member with simd prefix

```
template<class V, class G>
  V
  simd_generate(G&& gen);

template<class V = void, class It, class... Flags>
  conditional_t<is_same_v<V, void>, simd<iter_value_t<It>>, V>
  simd_copy_from(It first, simd_flags<Flags...> f = {});

template<class Rg, std::integral Idx, class AbiIdx, class... Flags>
  simd<ranges::range_value_t<Rg>, basic_simd<Idx, AbiIdx>::size()>
  simd_gather_from(const Rg&& in, const basic_simd<Idx, AbiIdx>& indexes,
                   simd_flags<Flags...> f = {});

template<size_t SizeSelector = 0, class T, class Abi, class PermuteGenerator>
  simd<T, output-size>
  simd_permute(const basic_simd<T, Abi>& v, PermuteGenerator&& fn);

template<size_t Bytes, class Abi, class T, class U>
  auto
  simd_select(const basic_simd_mask<Bytes, Abi>& c, const T& a, const U& b)
    -> decltype(simd-select-impl(c, a, b));

template<class T, class Abi>
  basic_simd<T, Abi>
  simd_exp(const basic_simd<T, Abi>& x);

template<class T, class Abi>
  basic_simd<T, Abi>
  simd_min(const basic_simd<T, Abi>& x, const basic_simd<T, Abi>& y);

template<size_t Bs, class Abi>
  bool
  simd_all_of(const basic_simd_mask<Bs, Abi>&);

template<class T>
  concept simd_integral = /*...*/;

template<class T>
  concept simd_generic_integral = integral<T> or simd_integral<T>;
```

Usage example:

```
void f(std::simd<float> vf, const std::vector<int>& data) {
  auto iota = std::simd_generate<std::simd<int>>([](int i) { return i; });
  auto chunk = std::simd_copy_from(data.begin());
  auto chunk_swapped = std::simd_gather_from(data, iota ^ 1);
```

```
  auto chunk_swapped2 = std::simd_permute(chunk, [](int i) { return i ^ 1; });
  assert(std::simd_all_of(chunk_swapped == chunk_swapped2));

  vf = std::simd_select(vf > 1.f, 1.f, vf);
  vf = std::simd_exp(vf);
  auto lo = std::simd_min(iota, chunk);
}
```

There is little variation possible for the above code. The most important variation is using unqualified calls, relying on ADL:

```
void f(std::simd<float> vf, const std::vector<int>& data) {
  auto iota = std::simd_generate<std::simd<int>>([](int i) { return i; });
  auto chunk = std::simd_copy_from(data.begin());
  auto chunk_swapped = simd_gather_from(data, iota ^ 1);
  auto chunk_swapped2 = simd_permute(chunk, [](int i) { return i ^ 1; });
  assert(simd_all_of(chunk_swapped == chunk_swapped2));

  vf = simd_select(vf > 1.f, 1.f, vf);
  vf = simd_exp(vf);
  auto lo = simd_min(iota, chunk);
}
```

For SIMD-generic programming a trivial example looks like this:

```
template<std::integral T>
T scalar_only(T a, T b) {
  return 2 * std::min(a, b);
}

template<std::simd_integral T>
T simd_only(T a, T b) {
  return 2 * std::simd_min(a, b);
}

template<std::simd_generic_integral T>
T generic(T a, T b) {
  if constexpr (std::simd_integral<T>)
    return 2 * std::simd_min(a, b);
  else
    return 2 * std::min(a, b);
}
```

The ability to constrain a function like this actually resolves a missing feature in the TS that I hit when working on using `std::experimental::simd` in the core of the GNU Radio framework. Obviously, the TS couldn't have proposed any concepts. The ability to constrain a function with any of the three choices above had to be solved with an ad-hoc solution in GNU Radio.

However, looking at the implementation of the `generic` function above, this can't be what we want.

PROS
- Consistent.

⇒ Users don't need to remember which functions don't need a `simd` prefix.

- Consistent naming scheme for SIMD and SIMD-generic concepts.

CONS
- Verbose.

⇒ There's a lot of "simd" spelled out in the code. It is not adding information (IOW: it's noise) – at least in this code.

- SIMD-generic programming is barely possible (because it requires too many constexpr-if branches).

MY RATING: unacceptable for lack of SIMD-generic programming; too verbose without opt-out of the verbosity; there must be a better alternative

4.3 Alternative 2: every function is a non-member without simd prefix

```
template<class V, class G>
  V
  generate(G&& gen);

template<class V = void, class It, class... Flags>
  conditional_t<is_same_v<V, void>, simd<iter_value_t<It>>, V>
  copy_from(It first, simd_flags<Flags...> f = {});

template<class Rg, std::integral Idx, class AbiIdx, class... Flags>
  simd<ranges::range_value_t<Rg>, basic_simd<Idx, AbiIdx>::size()>
  gather_from(const Rg&& in, const basic_simd<Idx, AbiIdx>& indexes,
                  simd_flags<Flags...> f = {});

template<size_t SizeSelector = 0, class T, class Abi, class PermuteGenerator>
  simd<T, output-size>
  permute(const basic_simd<T, Abi>& v, PermuteGenerator&& fn);

template<size_t Bytes, class Abi, class T, class U>
  auto
  select(const basic_simd_mask<Bytes, Abi>& c, const T& a, const U& b)
    -> decltype(simd-select-impl(c, a, b));

template<class T, class Abi>
  basic_simd<T, Abi>
  exp(const basic_simd<T, Abi>& x);

template<class T, class Abi>
  basic_simd<T, Abi>
  min(const basic_simd<T, Abi>& x, const basic_simd<T, Abi>& y);

template<size_t Bs, class Abi>
  bool
  all_of(const basic_simd_mask<Bs, Abi>&);

// no way around a prefix:
template<class T>
  concept simd_integral = /*...*/;

template<class T>
  concept simd_generic_integral = integral<T> or simd_integral<T>;
```

Usage example:

```
void f(std::simd<float> vf, const std::vector<int>& data) {
  auto iota = std::generate<std::simd<int>>([](int i) { return i; });
  auto chunk = std::copy_from(data.begin());
```

```
  auto chunk_swapped = std::gather_from(data, iota ^ 1);
  auto chunk_swapped2 = std::permute(chunk, [](int i) { return i ^ 1; });
  assert(std::all_of(chunk_swapped == chunk_swapped2));

  vf = std::select(vf > 1.f, 1.f, vf);
  vf = std::exp(vf);
  auto lo = std::min(iota, chunk);
}
```

There is little variation possible for the above code. The most important variation is using unqualified calls, relying on ADL:

```
void f(std::simd<float> vf, const std::vector<int>& data) {
  auto iota = std::generate<std::simd<int>>([](int i) { return i; });
  auto chunk = std::copy_from(data.begin());
  auto chunk_swapped = gather_from(data, iota ^ 1);
  auto chunk_swapped2 = permute(chunk, [](int i) { return i ^ 1; });
  assert(all_of(chunk_swapped == chunk_swapped2));

  vf = select(vf > 1.f, 1.f, vf);
  vf = exp(vf);
  auto lo = min(iota, chunk);
}
```

For SIMD-generic programming the example now looks like this:

```
template<std::integral T>
T scalar_only(T a, T b) {
  return 2 * std::min(a, b);
}

template<std::simd_integral T>
T simd_only(T a, T b) {
  return 2 * std::min(a, b);
}

template<std::simd_generic_integral T>
T generic(T a, T b) {
  return 2 * std::min(a, b);
}
```

PROS
- Consistent.

  ⇒ Simple to remember.
- SIMD-generic interfaces can easily be provided.

CONS
- Nothing in e.g. `auto x = std::copy_from(data.begin())` hints at the creation of a `basic_simd` object.

- Non-`simd` overloads for the same names become questionable as soon as the functionality isn't equivalent. (huge "name grab")
- If we ever need to disambiguate an inconsistently overloaded term, then it will need a `simd_` prefix. E. g. the `simd_integral` concept would be such a term. This could be considered less consistent than what we'd like to aim for.

MY RATING: unacceptable "name grab" and potentially confusing overloads

4.4 Alternative 3: place everything but types into a namespace

```
namespace std {
template<class T, class Abi>
  class basic_simd;
}

namespace std::Simd { // I don't even have one acceptable idea for a name

template<class V, class G>
  V
  generate(G&& gen);

template<class V = void, class It, class... Flags>
  conditional_t<is_same_v<V, void>, simd<iter_value_t<It>>, V>
  copy_from(It first, simd_flags<Flags...> f = {});

template<class Rg, std::integral Idx, class AbiIdx, class... Flags>
  simd<ranges::range_value_t<Rg>, basic_simd<Idx, AbiIdx>::size()>
  gather_from(const Rg&& in, const basic_simd<Idx, AbiIdx>& indexes,
                  simd_flags<Flags...> f = {});

template<size_t SizeSelector = 0, class T, class Abi, class PermuteGenerator>
  simd<T, output-size>
  permute(const basic_simd<T, Abi>& v, PermuteGenerator&& fn);

template<size_t Bytes, class Abi, class T, class U>
  auto
  select(const basic_simd_mask<Bytes, Abi>& c, const T& a, const U& b)
    -> decltype(simd-select-impl(c, a, b));

template<class T, class Abi>
  basic_simd<T, Abi>
  exp(const basic_simd<T, Abi>& x);

template<class T, class Abi>
  basic_simd<T, Abi>
  min(const basic_simd<T, Abi>& x, const basic_simd<T, Abi>& y);

template<size_t Bs, class Abi>
  bool
  all_of(const basic_simd_mask<Bs, Abi>&);

template<class T>
  concept integral = /*...*/;
```

```
template<class T>
  concept generic_integral = std::integral<T> or Simd::integral<T>;

}
```

Usage example:

```
void f(std::simd<float> vf, const std::vector<int>& data) {
  auto iota = std::Simd::generate<std::simd<int>>([](int i) { return i; });
  auto chunk = std::Simd::copy_from(data.begin());
  auto chunk_swapped = std::Simd::gather_from(data, iota ^ 1);
  auto chunk_swapped2 = std::Simd::permute(chunk, [](int i) { return i ^ 1; });
  assert(std::Simd::all_of(chunk_swapped == chunk_swapped2));

  vf = std::Simd::select(vf > 1.f, 1.f, vf);
  vf = std::Simd::exp(vf);
  auto lo = std::Simd::min(iota, chunk);
}
```

There is little variation possible for the above code. ADL doesn't work, but a namespace alias becomes interesting:

```
namespace smd = std::Simd;

void f(std::simd<float> vf, const std::vector<int>& data) {
  auto iota = smd::generate<std::simd<int>>([](int i) { return i; });
  auto chunk = smd::copy_from(data.begin());
  auto chunk_swapped = smd::gather_from(data, iota ^ 1);
  auto chunk_swapped2 = smd::permute(chunk, [](int i) { return i ^ 1; });
  assert(smd::all_of(chunk_swapped == chunk_swapped2));

  vf = smd::select(vf > 1.f, 1.f, vf);
  vf = smd::exp(vf);
  auto lo = smd::min(iota, chunk);
}
```

For SIMD-generic programming the example now looks like this:

```
template<std::integral T>
T scalar_only(T a, T b) {
  return 2 * std::min(a, b);
}

template<std::simd::integral T>
T simd_only(T a, T b) {
  return 2 * std::simd::min(a, b);
}

template<std::simd::generic_integral T>
```

```
T generic(T a, T b) {
  if constexpr (std::simd::integral<T>)
    return 2 * std::simd::min(a, b);
  else
    return 2 * std::min(a, b);
}
```

PROS
- We are free to grab names out of the new namespace.
- any?

CONS
- The type and functions being in different namespaces is awkward.
- The required mismatch between the facility (“std::simd”) and the namespace is frustrating.

⇒ No possible good name for the namespace.

- SIMD-generic programming is barely possible (because it requires too many constexpr-if branches).

MY RATING: unacceptable for lack of SIMD-generic programming; ADL not working is not helping anything; there must be a better alternative

4.5 Alternative 4: make all non-member functions hidden friends

```
namespace std {
template<class T, class Abi>
  class basic_simd
  {
    /*...*/
    template<class V, class G>
      friend V
      generate(G&& gen);

    template<class V = void, class It, class... Flags>
      friend conditional_t<is_same_v<V, void>, simd<iter_value_t<It>>, V>
      copy_from(It first, simd_flags<Flags...> f = {});

    template<class Rg, class... Flags>
      friend simd<ranges::range_value_t<Rg>, size()>
      gather_from(const Rg&& in, const basic_simd& indexes, simd_flags<Flags...> f = {});

    template<size_t SizeSelector = 0, class PermuteGenerator>
      friend simd<T, output-size>
      permute(const basic_simd& v, PermuteGenerator&& fn);

    friend basic_simd
    exp(const basic_simd& x);

    friend basic_simd
    min(const basic_simd& x, const basic_simd& y);
  };

template<size_t Bytes, class Abi>
  class basic_simd_mask
  {
    /*...*/

    template<class T, class U>
      friend auto
      select(const basic_simd_mask& c, const T& a, const U& b)
        -> decltype(simd-select-impl(c, a, b));

    friend bool
    all_of(const basic_simd_mask&);
  };
}

// can't be members or friends
```

```
template<class T>
  concept simd_integral = /*...*/;

template<class T>
  concept simd_generic_integral = integral<T> or simd_integral<T>;
```

Let's skip over usage examples because:

CONS

- This doesn't even work! No way to call e.g. `generate` or `copy_from`.
- The requirement to always call unqualified is strange.
- Makes SIMD-generic programming really hard.

MY RATING: Garbage

4.6 Alternative 5: place everything into a single namespace

```
namespace std::simd {

template<class T, class Abi = /*...*/>
  class basic_simd;

template<class T, simd-size-type N = /*...*/>
  using simd = basic_simd<T, deduce-t<T, N>>;

template<class V, class G>
  V
  generate(G&& gen);

template<class V = void, class It, class... Flags>
  conditional_t<is_same_v<V, void>, simd<iter_value_t<It>>, V>
  copy_from(It first, simd_flags<Flags...> f = {});

template<class Rg, std::integral Idx, class AbiIdx, class... Flags>
  simd<ranges::range_value_t<Rg>, basic_simd<Idx, AbiIdx>::size()>
  gather_from(const Rg&& in, const basic_simd<Idx, AbiIdx>& indexes,
                  simd_flags<Flags...> f = {});

template<size_t SizeSelector = 0, class T, class Abi, class PermuteGenerator>
  simd<T, output-size>
  permute(const basic_simd<T, Abi>& v, PermuteGenerator&& fn);

template<size_t Bytes, class Abi, class T, class U>
  auto
  select(const basic_simd_mask<Bytes, Abi>& c, const T& a, const U& b)
    -> decltype(simd-select-impl(c, a, b));

template<class T, class Abi>
  basic_simd<T, Abi>
  exp(const basic_simd<T, Abi>& x);

template<class T, class Abi>
  basic_simd<T, Abi>
  min(const basic_simd<T, Abi>& x, const basic_simd<T, Abi>& y);

template<size_t Bs, class Abi>
  bool
  all_of(const basic_simd_mask<Bs, Abi>&);

template<class T>
  concept integral = /*...*/;
```

```
template<class T>
  concept generic_integral = std::integral<T> or std::simd::integral<T>;

}
```

Conceivable *variations* for the `std::simd` namespace are

- `std::datapar` (The `basic_simd` and `basic_simd_mask` types are in the "Data-parallel types" section in the IS.)
- `std::dp` (data-parallel)
- `std::dpt` (data-parallel types)
- `std::unseq`

Personally, I don't believe any of these are an improvement.

However, I would suggest renaming `std::simd::basic_simd_mask` to `std::simd::basic_mask`, and accordingly `simd::simd_mask` to `simd::mask`.

Consequently, if we're reading the namespace as part of the type name (`simd::mask`) we should consider renaming `simd::simd` to:

`simd::vector` We often speak about "SIMD vectors"; so in principle this a good name. However, I fear that using the heavily overloaded term "vector" has too much potential for confusion. Especially the use of `using namespace std; using namespace std::simd;`[2] or even just `using namespace std::simd` by itself would lead to a lot of confusion.

`simd::vec` This name tries to avoid the confusion by spelling "vector" as an abbreviation (and thus avoid the "hold on, why does it say `vector` here?" moments when reviewing code)

`simd::value` Note the naming precedent in `valarray`, which is called "value array".

`simd::values`

`simd::array` The static extent matches `std::array`; it's a `std::array` with SIMD operations; also, I believe conversions between `simd` and `std::array` of equal extent should be implicit…

From all of these, I'd prefer if we could use `simd::vector<T>` — and in the library where this work originates it was called `Vc::Vector<T>` — but I fear this will lead to confusion and just isn't worth the trouble. It seems however that `simd::vec<T>` could resolve that issue and still be fairly close to

2 huge foot-gun, which WG21 members will quickly recognize as such

the term we use in speech. Next best... `simd::array` is starting to grow on me. This term was never considered before (IIRC[3]). It appeals to me be because I believe we should make CTAD and implicit conversions work for `simd<T, N>` ↔ `array<T, N>`. In terms of bit-representation, they typically are the same thing. They differ in alignment[4], function argument passing[5], and whether you can apply operators that the value-type provides.

For now, I don't want to propose a name change. But please give me feedback if you think I should / should not (propose a name change).

Usage example:

```
void f(std::simd::simd<float> vf, const std::vector<int>& data) {
  auto iota = std::simd::generate<std::simd::simd<int>>([](int i) { return i; });
  auto chunk = std::simd::copy_from(data.begin());
  auto chunk_swapped = std::simd::gather_from(data, iota ^ 1);
  auto chunk_swapped2 = std::simd::permute(chunk, [](int i) { return i ^ 1; });
  assert(std::simd::all_of(chunk_swapped == chunk_swapped2));

  vf = std::simd::select(vf > 1.f, 1.f, vf);
  vf = std::simd::exp(vf);
  auto lo = std::simd::min(iota, chunk);
}
```

This is fairly verbose, so a user might decide to rather rely on ADL:

```
void f(std::simd::simd<float> vf, const std::vector<int>& data) {
  auto iota = std::simd::generate<std::simd::simd<int>>([](int i) { return i; });
  auto chunk = std::simd::copy_from(data.begin());
  auto chunk_swapped = gather_from(data, iota ^ 1);
  auto chunk_swapped2 = permute(chunk, [](int i) { return i ^ 1; });
  assert(all_of(chunk_swapped == chunk_swapped2));

  vf = select(vf > 1.f, 1.f, vf);
  vf = exp(vf);
  auto lo = min(iota, chunk);
}
```

But as we can see, ADL only works for some of the functions. If the function requires a template argument or none of the arguments are a `basic_simd`/ `basic_simd_mask`, then the call still must be qualified. Consequently, if a user wants to reduce the character overhead, a namespace alias might be better suited:

```
namespace smd = std::simd;

void f(smd::simd<float> vf, const std::vector<int>& data) {
```

3 if I remember correctly

4 Note that alignment can influence `sizeof`.

5 E. g. the Itanium ABI passes `array<float, 4>` as two `XMM` registers and `simd<float, 4>` as one `XMM` register.

```
  auto iota = smd::generate<smd::simd<int>>([](int i) { return i; });
  auto chunk = smd::copy_from(data.begin());
  auto chunk_swapped = smd::gather_from(data, iota ^ 1);
  auto chunk_swapped2 = smd::permute(chunk, [](int i) { return i ^ 1; });
  assert(smd::all_of(chunk_swapped == chunk_swapped2));

  vf = smd::select(vf > 1.f, 1.f, vf);
  vf = smd::exp(vf);
  auto lo = smd::min(iota, chunk);
}
```

The SIMD-generic programming example from previous sections now looks like this:

```
template<std::integral T>
T scalar_only(T a, T b) {
  return 2 * std::min(a, b);
}

template<std::simd::integral T>
T simd_only(T a, T b) {
  return 2 * std::simd::min(a, b);
}

template<std::simd::generic_integral T>
T generic(T a, T b) {
  if constexpr (std::simd::integral<T>)
    return 2 * std::simd::min(a, b);
  else
    return 2 * std::min(a, b);
}
```

Another user might be looking for a way to qualify e.g. `<cmath>` functions such that they work both with `T` and `basic_simd<T>`. To that end one needs to basically inline `std::simd` into `std` and thus write:

```
namespace xstd {
  using namespace std;
  using namespace std::simd;
}

void f(xstd::simd<float> vf, const xstd::vector<int>& data) {
  auto iota = xstd::generate<xstd::simd<int>>([](int i) { return i; });
  auto chunk = xstd::copy_from(data.begin());
  auto chunk_swapped = xstd::gather_from(data, iota ^ 1);
  auto chunk_swapped2 = xstd::permute(chunk, [](int i) { return i ^ 1; });
  assert(xstd::all_of(chunk_swapped == chunk_swapped2));

  vf = xstd::select(vf > 1.f, 1.f, vf);
```

```
  vf = xstd::exp(vf);
  auto lo = xstd::min(iota, chunk);
}
```

I need to be convinced that the latter pattern isn't a liability, and therefore I wouldn't allow this to go through code review without raising a red flag.

PROS
- We are free to grab names out of the new namespace.
- ADL still works.
- Consistent.

⇒ Users only need to learn: "If it's in the `std::simd` namespace then it works for `simd`s. When searching for a function for `simd`, look in the `std::simd` namespace."

CONS
- SIMD-generic programming just got harder.
- The class template name `std::simd::simd` is a bit awkward. (There are alternative names that we could adopt instead.)

MY RATING: unacceptable for lack of SIMD-generic programming; interesting if we get rid of the out-of-the-box requirement for constexpr-if

4.7 Alternative 6: Place everything but obvious overloads into a single namespace

The preceding alternative probably went too far with moving <cmath> overloads and algorithms like min, clamp, etc. into the std::simd namespace. So let's keep all functions that are a clear overload (f(simd<T>)) from an existing function (f(T)) directly in the std namespace. This is the "namespace equivalent" to the status-quo approach of whether a simd_ prefix is needed or not.

```
namespace std::simd {

template<class T, class Abi = /*...*/>
  class basic_simd;

template<class T, simd-size-type N = /*...*/>
  using simd = basic_simd<T, deduce-t<T, N>>;

template<class V, class G>
  V
  generate(G&& gen);

template<class V = void, class It, class... Flags>
  conditional_t<is_same_v<V, void>, simd<iter_value_t<It>>, V>
  copy_from(It first, simd_flags<Flags...> f = {});

template<class Rg, std::integral Idx, class AbiIdx, class... Flags>
  simd<ranges::range_value_t<Rg>, basic_simd<Idx, AbiIdx>::size()>
  gather_from(const Rg&& in, const basic_simd<Idx, AbiIdx>& indexes,
                  simd_flags<Flags...> f = {});

template<size_t SizeSelector = 0, class T, class Abi, class PermuteGenerator>
  simd<T, output-size>
  permute(const basic_simd<T, Abi>& v, PermuteGenerator&& fn);

template<size_t Bytes, class Abi, class T, class U>
  auto
  select(const basic_simd_mask<Bytes, Abi>& c, const T& a, const U& b)
    -> decltype(simd-select-impl(c, a, b));

template<size_t Bs, class Abi>
  bool
  all_of(const basic_simd_mask<Bs, Abi>&);

template<class T>
  concept integral = /*...*/;

template<class T>
  concept generic_integral = std::integral<T> or std::simd::integral<T>;
```

```
}

namespace std {

template<class T, class Abi>
  simd::basic_simd<T, Abi>
  exp(const simd::basic_simd<T, Abi>& x);

template<class T, class Abi>
  simd::basic_simd<T, Abi>
  min(const simd::basic_simd<T, Abi>& x, const simd::basic_simd<T, Abi>& y);

}
```

Usage example:

```
void f(std::simd::simd<float> vf, const std::vector<int>& data) {
  auto iota = std::simd::generate<std::simd::simd<int>>([](int i) { return i; });
  auto chunk = std::simd::copy_from(data.begin());
  auto chunk_swapped = std::simd::gather_from(data, iota ^ 1);
  auto chunk_swapped2 = std::simd::permute(chunk, [](int i) { return i ^ 1; });
  assert(std::simd::all_of(chunk_swapped == chunk_swapped2));

  vf = std::simd::select(vf > 1.f, 1.f, vf);
  vf = std::exp(vf);
  auto lo = std::min(iota, chunk);
}
```

When relying on ADL, nothing changes compared to the example in the preceding section. However, if we now create a namespace alias and call everything fully qualified, the necessary qualifications could be considered slightly incoherent:

```
namespace smd = std::simd;

void f(smd::simd<float> vf, const std::vector<int>& data) {
  auto iota = smd::generate<smd::simd<int>>([](int i) { return i; });
  auto chunk = smd::copy_from(data.begin());
  auto chunk_swapped = smd::gather_from(data, iota ^ 1);
  auto chunk_swapped2 = smd::permute(chunk, [](int i) { return i ^ 1; });
  assert(smd::all_of(chunk_swapped == chunk_swapped2));

  vf = smd::select(vf > 1.f, 1.f, vf);
  vf = std::exp(vf);
  auto lo = std::min(iota, chunk);
}
```

At this point all functions already work for SIMD-generic code (or can be made to work with suitable overloads in the `std::simd` namespace). If LEWG were to adopt this naming style, then we need to decide on a per function basis, whether the function is "SIMD-only" or whether an overload for the value-type is useful on its own. For the latter, the function goes into `std` otherwise it needs to go into `std::simd`.

The SIMD-generic programming example from previous sections now looks like this:

```
template<std::integral T>
T scalar_only(T a, T b) {
  return 2 * std::min(a, b);
}

template<std::simd::integral T>
T simd_only(T a, T b) {
  return 2 * std::min(a, b);
}

template<std::simd::generic_integral T>
T generic(T a, T b) {
  return 2 * std::min(a, b);
}
```

PROS

- We are free to grab names out of the new namespace.
- ADL works.
- Fairly consistent.

⇒ Users need to learn: "If it's in the `std::simd` namespace then it works for `simd`s. When searching for a function for `simd`, if the same function exists / could exist for scalars look for it in `std`, otherwise look in the `std::simd` namespace."

- SIMD-generic programming is straightforward to provide and use.

CONS

- The class template name `std::simd::simd` is a bit awkward.
- We have a mix of non-member functions in `std` and `std::simd`.

MY RATING: acceptable; but not much different from the status quo — not convinced this is actually *better*

4.8 Alternative 7: place simd into a single namespace with a different namespace for simd-generic interfaces

```
namespace std::simd {

template<class T, class Abi = /*...*/>
  class basic_simd;

template<class T, simd-size-type N = /*...*/>
  using simd = basic_simd<T, deduce-t<T, N>>;

template<class V, class G>
  V
  generate(G&& gen);

template<class V = void, class It, class... Flags>
  conditional_t<is_same_v<V, void>, simd<iter_value_t<It>>, V>
  copy_from(It first, simd_flags<Flags...> f = {});

template<class Rg, std::integral Idx, class AbiIdx, class... Flags>
  simd<ranges::range_value_t<Rg>, basic_simd<Idx, AbiIdx>::size()>
  gather_from(const Rg&& in, const basic_simd<Idx, AbiIdx>& indexes,
                  simd_flags<Flags...> f = {});

template<size_t SizeSelector = 0, class T, class Abi, class PermuteGenerator>
  simd<T, output-size>
  permute(const basic_simd<T, Abi>& v, PermuteGenerator&& fn);

template<size_t Bytes, class Abi, class T, class U>
  auto
  select(const basic_simd_mask<Bytes, Abi>& c, const T& a, const U& b)
    -> decltype(simd-select-impl(c, a, b));

template<class T, class Abi>
  basic_simd<T, Abi>
  exp(const basic_simd<T, Abi>& x);

template<class T, class Abi>
  basic_simd<T, Abi>
  min(const basic_simd<T, Abi>& x, const basic_simd<T, Abi>& y);

template<size_t Bs, class Abi>
  bool
  all_of(const basic_simd_mask<Bs, Abi>&);

template<class T>
  concept integral = /*...*/;
```

```
} // std::simd

namespace std::simd_generic {

namespace scalar {

template<vectorizable T, class G>
  T
  generate(G&& gen);

template<vectorizable T, class It, class... Flags>
  T
  copy_from(It first, simd_flags<Flags...> f = {});

template<class Rg, std::integral Idx, class... Flags>
  ranges::range_value_t<Rg>
  gather_from(const Rg&& in, Idx index, simd_flags<Flags...> f = {});

template<class T, class U>
  auto
  select(bool c, const T& a, const U& b)
    -> decltype(simd-select-impl(c, a, b));

using std::exp;

using std::min;

bool
all_of(same_as<bool>);

} // (std::simd_generic::)scalar

using namespace std::simd;

using namespace std::simd_generic::scalar;

template<class T>
  concept integral = std::integral<T> or std::simd::integral<T>;

} // std::simd_generic
```

The usage example looks exactly like in Section 4.6. There is also no difference with regard to ADL and using a namespace alias.

However, the situation for SIMD-generic programming is rather different. At this point a user can be very clear about "scalar-only" (`std`), "simd-only" (`std::simd`), and SIMD-generic (`std::simd_generic`) code. Thus, our recurring example becomes:

```
template<std::simd::integral T>
template<std::integral T>
T scalar_only(T a, T b) {
  return 2 * std::min(a, b);
}

T simd_only(T a, T b) {
  return 2 * std::simd::min(a, b);
}

template<std::simd_generic::integral T>
T fun(T a, T b) {
  return 2 * std::simd_generic::min(a, b);
}
```

Now the namespace of the `integral` concept matches the namespace of the functions that we need to use. There's a clear mechanism from opting into SIMD-generic overloads — or avoiding them when they are not required. All the prevision definitions of SIMD-integral and SIMD-generic-integral concepts didn't have this clear association with a set of function overloads.

The ability to choose between `std::simd` and `std::simd_generic` also provides another level of clarity in stating intent: Do you expect your code to be called only with `basic_simd<T>` or also with `T`?

Note that, as declared above, also `<cmath>` overloads are in different namespaces. Thus, instead of writing `using std::exp`, I can now write `using std::simd_generic::exp` and all unqualified `exp` calls are overloaded for scalars and `simd`s.

I expect that many users might be interested in shortening `std::simd` and even more `std::simd_generic`. If that's the case, we're going to see many namespace aliases for the two namespaces.

PROS

- We are free to grab names out of the new namespace.
- ADL still works.
- Consistent.

⇒ Users only need to learn: "If it's in the `std::simd` namespace then it works for `simd`s. When searching for a function for `simd`, look in the `std::simd` namespace. When it needs to work generically for `simd` and scalars, just switch to `std::simd_generic`."

- Opt-in SIMD-generic programming that is fairly "safe" with regard to accidentally calling the wrong overload.

cons
- The class template name `std::simd::simd` still is a bit awkward. (standard SIMD vector / `std::simd::vec`?)
- `std::simd_generic` is too long and will be abbreviated with different namespace aliases in different code bases[6].

my rating: sold; feels good after implementing it; happy about the clear separation of scalar / SIMD / SIMD-generic; happy about concise code through namespace aliases

4.8.1 on renaming std::simd::simd to std::simd::vec

Personally, I don't think `std::simd::simd` is a problem. Especially, considering that users might introduce a namespace alias or even — heaven forbid — import the whole `std::simd` (or `std::simd_generic`) namespace into their local scope. If `vec` needs to stand on its own without the `simd::` part of the name, I fear we might lose clarity compared to `simd`.

I believe the situation is different for `std::simd::simd_mask`, which, in my opinion, can live without the `simd_` part in its name. Thus, even after a `using namespace std::simd;` the alias template name `mask` is expressive enough. (Because `mask` only appears in proximity to `simd` — if it appears in code at all.)

6 this is normal in other languages, e. g. Python

5 PROPOSED POLLS

Any vote would be against the status quo, which so far can be summarized as:

- types and functions go directly into `std`
- when naming a function for `simd`,
  - if the same function exists / could exist for scalars or a range: no `simd_` prefix,
  - otherwise the function name needs a `simd_` prefix
- traits and types need a `simd` in their name

Poll: Adopt Alternative 7 from P3287R0 while renaming (`basic_`)`simd_mask` to (`basic_`)`mask` (without making a decision on non-member load, store, and generate)

SF	F	N	A	SA

Poll: Adopt Alternative 7 from P3287R0 while renaming (`basic_`)`simd_mask` to (`basic_`)`mask` and (`basic_`)`simd` to (`basic_`)`vec` (without making a decision on non-member load, store, and generate)

SF	F	N	A	SA

6 WORDING

TBD

A ACKNOWLEDGMENTS

Daniel Towner and Ruslan Arutyunyan contributed to this paper via discussions / reviews.