

Document Number: P3287R2
Date: 2024-11-13
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	1
1.1	CHANGES FROM REVISION 0	1
1.2	CHANGES FROM REVISION 1	1
2	STRAW POLLS	1
2.1	LEWG — ST. LOUIS 2024	1
3	INTRODUCTION & MOTIVATION	2
3.1	MOTIVATION FOR A <code>STD::SIMD</code> NAMESPACE	2
3.2	<code>SIMD-GENERIC</code> PROGRAMMING	2

4	EXPLORATION	3
4.1	STATUS QUO (LATEST REVISION OF SIMD PAPERS)	4
4.2	EXPLORATIONS IN PREVIOUS REVISION(S) OF THIS PAPER	5
4.3	ALTERNATIVE 5: PLACE EVERYTHING INTO A SINGLE NAMESPACE	5
4.4	ALTERNATIVE 6: PLACE EVERYTHING BUT OBVIOUS OVERLOADS INTO A SINGLE NAMESPACE	9
4.5	ALTERNATIVE 7: PLACE SIMD INTO A SINGLE NAMESPACE WITH A DIFFERENT NAMESPACE FOR SIMD-GENERIC INTERFACES	12
4.6	ALTERNATIVE 8: EVERYTHING IN A SINGLE NAMESPACE WITH USING-DECLARATIONS IN STD	16
5	NAMING DISCUSSION OF NAMESPACE AND SIMD / SIMD_MASK	19
5.1	NAMESPACE NAMES	19
5.2	CLASS TEMPLATE NAMES	20
5.3	ON RENAMING <code>STD::SIMD::SIMD</code> TO <code>STD::SIMD::VEC</code>	20
6	RECOMMENDATION: TWO EXAMPLES AFTER RENAMING	22
7	PROPOSED POLLS	22
8	WORDING	23
A	ACKNOWLEDGMENTS	24

1

CHANGELOG

(placeholder)

1.1

CHANGES FROM REVISION 0

Previous revision: P3287R0

- Add new “Alternative 8” with using-declarations in `std` (Section 4.6).
- Rename `copy_from` to `load_from` as proposed in P3299.
- Move discussion on naming into its own section (5).
- Add more motivation for a sub-namespace (Section 3.1).

1.2

CHANGES FROM REVISION 1

Previous revision: P3287R1

- Suggest and discuss `std::datapar::simd`, i.e. rename the namespace.
- Note that `std::simd` could be an alias to `std::datapar::simd`.

2

STRAW POLLS

2.1

LEWG — ST. LOUIS 2024

Poll: We should introduce a `std::simd` namespace in P1928 (realizing that this would mean we potentially have a type whose fully-qualified name is `std::simd::simd` and would involve removing “`simd`” from `basic_simd_mask`) and remove the `simd_` prefix from utilities moved into that namespace which currently have that prefix.

SF	F	N	A	SA
9	6	3	1	1

Poll: Operations on `std::simd::simd` should be available in the namespace `std::simd`.

SF	F	N	A	SA
12	7	1	1	0

Poll: Element-wise overloads of existing functions in `std` which operate on `std::simd::simd` should also be available in the namespace `std`.

SF	F	N	A	SA
4	7	5	1	1

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

MOTIVATION FOR A `STD::SIMD` NAMESPACE

A `std::simd` namespace encapsulates everything related to data-parallel types, creating an easy-to-explore, isolated space for these functionalities. It avoids the need for (inconsistent) `simd_` prefixes by grouping related functions into their own namespace. It increases flexibility for future extensions to be organized within the `std::simd` namespace. Users can easily alias the namespace (e.g. `namespace simd = std::simd`), reducing verbosity in the code but maintaining a clean and logical structure.

The `std::simd` namespace approach might be cleaner in the long term. It avoids overloads like `std::reduce` and `std::all_of`, reducing the likelihood of confusion. While a `simd::simd` duplication in naming would be a bit awkward, it is a one-time issue that users can learn to live with. And we should consider renaming the alias template to `std::simd::vec` (`std::simd::basic_vec` for the class template).

This approach also aligns with other C++ standard library features, where sub-namespaces are used to logically group related functionality:

`std::chrono`, `std::execution`, `std::filesystem`, `std::linalg`, `std::numbers`, `std::pmr`, `std::ranges`, and `std::ranges::views/std::views`.

3.2

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

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.

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.

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 separately. 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

EXPLORATIONS IN PREVIOUS REVISION(S) OF THIS PAPER

- Alternative 1: every function is a non-member with `simd` prefix
- Alternative 2: every function is a non-member without `simd` prefix
- Alternative 3: place everything but types into a namespace
- Alternative 4: make all non-member functions hidden friends

4.3

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>
    load_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>;
}

```

(RO discussed naming here, it was moved to Section 5.)

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::load_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::load_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) {
    auto iota = smd::generate<smd::simd<int>>([](int i) { return i; });
    auto chunk = smd::load_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::load_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 `simds`. 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`

4.4 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>
    load_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::load_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::load_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 `simds`. 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*

ALTERNATIVE 7: PLACE SIMD INTO A SINGLE NAMESPACE WITH A DIFFERENT NAMESPACE FOR
4.5 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>
    load_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
    load_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 4.3. 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 previous 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 `simds`.

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 `simds`. 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 bases².

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

² this is normal in other languages, e. g. Python

4.6 ALTERNATIVE 8: EVERYTHING IN A SINGLE NAMESPACE WITH USING-DECLARATIONS IN STD

After LEWG feedback in St. Louis, I added this alternative, which is basically a combination of Alternatives 5 and 6 (Sections 4.3 and 4.4).

```

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>
    load_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>;
}

namespace std {
    using simd::exp;
    using simd::min;
}

```

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::load_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);
}

```

Again, 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::load_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) {

```

```

auto iota = smd::generate<smd::simd<int>>([](int i) { return i; });
auto chunk = smd::load_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) {
    return 2 * std::min(a, b);
}

```

This is different to Alternative 7 (Section 4.5), where simd-generic programming requires an opt-in, i.e. a code-change from scalar code. With this approach existing non-simd code (`scalar_only`) can be modified to use data-parallel types and every function that exists in `std` and can be used in simd-generic code needs no further work (`generic`).

- 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 `simds`. When searching for a function for `simd`, look in the `std::simd` namespace."
- SIMD-generic programming works.
- CONS**
- The class template name `std::simd::simd` is a bit awkward. (There are alternative names that we could adopt instead.)

MY RATING: I like it. Clear separation of `simd` and non-`simd` functions while still providing good support for SIMD-generic programming

5 NAMING DISCUSSION OF NAMESPACE AND SIMD / SIMD_MASK

RO of this paper discussed naming in “Alternative 5”. But renaming isn’t tied to one specific alternative but a general consideration if the class templates are moved into a namespace.

5.1

NAMESPACE NAMES

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`

If we were to use the `std::datapar` namespace, I’d expect users to write:

```
namespace dp = std::datapar;
```

analogous to how many already use:

```
namespace fs = std::filesystem;
```

Typically, the name of the namespace should capture the intent/name of the package. In this case the name `std::simd` appears to express “types and associated functions for SIMD (computation and data structures)”. The name `std::datapar` (a shorthand for `std::dataparallel`) appears to express “types and associated functions for expressing data-parallel computation and data structures”.

As already discussed back when `std::experimental::datapar<T, Abi>` was renamed to `std::experimental::simd<T, Abi>`, `datapar` is less misleading about the abstraction level: This package is about expressing data-parallelism, which only incidentally allows access to SIMD registers in a CPU.

Note that if the namespace and type names differ, and the type name can stand on its own, it is possible to introduce an alias in the `std` namespace³:

```
namespace std {
  namespace datapar {
    template<class T, class Abi> class basic_simd;
    template<class T, simd-size-type N> using simd = basic_simd<T, ... >;
  }
  using datapar::basic_simd;
  using datapar::simd;
}
```

This is not proposed at this point.

³ Thanks to Bryce for the suggestion!

5.2

CLASS TEMPLATE NAMES

In any case, I suggest renaming `basic_simd_mask` to `basic_mask`, and accordingly `simd_mask` to `mask`.

If we stick to `std::simd` as the namespace name and read the namespace as part of the type name (`simd::mask`) we could 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;`⁴ 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 the term we use in speech. Next best... `simd::array` is starting to grow on me. This term was never considered before, if I remember correctly. It appeals to me because I believe we should make CTAD and implicit conversions work for `simd<T, N> ↔ array<T, N>`⁵. In terms of bit-representation, they typically are the same thing. They differ in alignment⁶, function argument passing⁷, and whether you can apply operators that the value-type provides.

5.3

ON RENAMING `STD::SIMD::SIMD` TO `STD::SIMD::VEC`

Personally, I don’t think `std::simd::simd` is a big 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`.

⁴ huge foot-gun, which WG21 members will quickly recognize as such

⁵ See P3299

⁶ Note that alignment can influence `sizeof`.

⁷ E.g. the Itanium ABI passes `array<float, 4>` as two XMM registers and `simd<float, 4>` as one XMM register.

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

RECOMMENDATION: TWO EXAMPLES AFTER RENAMING

Option 1:

```
namespace dp = std::datapar;

// compute log for positive inputs
dp::simd<float> f(std::span<float> data)
{
    dp::simd<float> x = dp::load_from(data);
    dp::mask<float> positive = x > 0.f;
    dp::simd<float> l = std::log(dp::select(positive, x, 1.f));
    return dp::select(positive, l, x);
}
```

Option 2:

```
namespace simd = std::simd;

// compute log for positive inputs
simd::vec<float> f(std::span<float> data)
{
    simd::vec<float> x = simd::load_from(data);
    simd::mask<float> positive = x > 0.f;
    simd::vec<float> l = std::log(simd::select(positive, x, 1.f));
    return simd::select(positive, l, x);
}
```

Compare against the status quo:

```
// compute log for positive inputs
std::simd<float> f(std::span<float> data)
{
    std::simd<float> x = std::simd_load_from(data);
    std::simd_mask<float> positive = x > 0.f;
    std::simd<float> l = std::log(std::simd_select(positive, x, 1.f));
    return std::simd_select(positive, l, x);
}
```

7

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 wording instructions (Section 8) from P3287R2 *without* renaming `(basic_)simd` to `(basic_)`
`vec`

SF	F	N	A	SA

Poll: Adopt wording instructions (Section 8) from P3287R2 renaming `(basic_)simd` to `(basic_)`
`vec`

SF	F	N	A	SA

Poll: Adopt wording instructions (Section 8) from P3287R2 using namespace `std::datapar` instead of `std::simd` *without* renaming `(basic_)simd` to `(basic_)`
`vec`

SF	F	N	A	SA

8

WORDING

This is fairly mechanical work, adding namespaces and adjusting type and function names.

In general the instructions go like this:

1. Move every type and function in “Data-Parallel types [simd]” from namespace `std` to namespace `std::simd`.
2. Replace all occurrences of `basic_simd_mask` with `basic_mask`.
3. Replace all occurrences of `simd_mask` with `mask`.
4. (TBD) Replace all occurrences of `basic_simd` with `basic_vec`.
5. (TBD) Replace all occurrences of `simd` with `vec`.
6. Rename all functions and remaining types / class templates: `s/\<simd_//`.
7. `s/rebind_simd/rebind/`
8. `s/resize_simd/resize/`
9. Add using-declarations in namespace `std` for the following functions:

- all function defined in [simd.math], and
- `min`, `max`, `minmax`, and `clamp` in [simd.alg].

A

ACKNOWLEDGMENTS

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