Document Number: N4773
Date: 2018-10-08
Revises: N4755
Reply to: Jared Hoberock
jhoberock@nvidia.com

Working Draft, C++ Extensions for Parallelism Version 2

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

Contents

1 Scope **1**

2 Normative references **2**

3 Terms and definitions **3**

4 General **4**
4.1 Namespaces and headers 4
4.2 Feature-testing recommendations 4

5 Parallel exceptions **5**
5.1 Header `<experimental/exception_list>` synopsis 5

6 Execution policies **6**
6.1 Header `<experimental/execution>` synopsis 6
6.2 Unsequenced execution policy 6
6.3 Vector execution policy 6
6.4 Execution policy objects 7

7 Parallel algorithms **8**
7.1 Wavefront Application 8
7.2 Non-Numeric Parallel Algorithms 9

8 Task Block **16**
8.1 Header `<experimental/task_block> synopsis` 16
8.2 Class `task_cancelled_exception` 16
8.3 Class `task_block` 16
8.4 Function template `define_task_block` 18
8.5 Exception Handling 19

9 Data-Parallel Types **20**
9.1 General 20
9.2 Header `<experimental/simd>` synopsis 20
9.3 Class template `simd` 30
9.4 `simd` non-member operations 37
9.5 Class template `simd_mask` 42
9.6 Non-member operations 46

1 Scope [parallel.scope]

1 This Technical Specification describes requirements for implementations of an interface that computer programs written in the C++ programming language may use to invoke algorithms with parallel execution. The algorithms described by this Technical Specification are realizable across a broad class of computer architectures.

2 This Technical Specification is non-normative. Some of the functionality described by this Technical Specification may be considered for standardization in a future version of C++, but it is not currently part of any C++ standard. Some of the functionality in this Technical Specification may never be standardized, and other functionality may be standardized in a substantially changed form.

3 The goal of this Technical Specification is to build widespread existing practice for parallelism in the C++ programming language. It gives advice on extensions to those vendors who wish to provide them.

2 Normative references [parallel.references]

1 The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — ISO/IEC 14882:2017, *Programming languages — C++*

2 ISO/IEC 14882:2017 is herein called the C++ Standard. References to clauses within the C++ Standard are written as “C++17 §20”. The library described in ISO/IEC 14882:2017 clauses 20-33 is herein called the *C++ Standard Library*. The C++ Standard Library components described in ISO/IEC 14882:2017 clauses 28, 29.8 and 23.10.10 are herein called the *C++ Standard Algorithms Library*.

3 Unless otherwise specified, the whole of the C++ Standard’s Library introduction (C++17 §20) is included into this Technical Specification by reference.

3 Terms and definitions [parallel.defns]

1 For the purposes of this document, the terms, definitions, and symbols given in ISO/IEC 14882:2017 apply.

2 ISO and IEC maintain terminological databases for use in standardization at the following addresses:

(2.1) — ISO Online browsing platform: available at `https://www.iso.org/obp`

(2.2) — IEC Electropedia: available at `http://www.electropedia.org`

4 General [parallel.general]

4.1 Namespaces and headers [parallel.general.namespaces]

1 Since the extensions described in this Technical Specification are experimental and not part of the C++ Standard Library, they should not be declared directly within namespace std. Unless otherwise specified, all components described in this Technical Specification are declared in namespace std::experimental::parallelism_v2.

[*Note:* Once standardized, the components described by this Technical Specification are expected to be promoted to namespace std. — *end note*]

2 Each header described in this technical specification shall import the contents of std::experimental::parallelism_v2 into std::experimental as if by

```
namespace std::experimental {
  inline namespace parallelism_v2 {}
}
```

3 Unless otherwise specified, references to such entities described in this Technical Specification are assumed to be qualified with std::experimental::parallelism_v2, and references to entities described in the C++ Standard Library are assumed to be qualified with std::.

4 Extensions that are expected to eventually be added to an existing header <meow> are provided inside the <experimental/meow> header, which shall include the standard contents of <meow> as if by

```
#include <meow>
```

4.2 Feature-testing recommendations [parallel.general.features]

1 An implementation that provides support for this Technical Specification shall define the feature test macro(s) in Table 1.

Table 1 — Feature-test macro(s)

Title	Section	Macro Name	Value	Header
Task Block	8	__cpp_lib_experimental_parallel_task_block	201711	<experimental/exception_list> <experimental/task_block>
Vector and Wavefront Policies	6.2	__cpp_lib_experimental_execution_vector_policy	201711	<experimental/algorithm> <experimental/execution>
Template Library for Parallel For Loops	7.2.2, 7.2.3, 7.2.4	__cpp_lib_experimental_parallel_for_loop	201711	<experimental/algorithm>
Data-Parallel Vector Types	9	__cpp_lib_experimental_parallel_simd	201803	<experimental/simd>

5 Parallel exceptions [parallel.exceptions]

5.1 Header <experimental/exception_list> synopsis [parallel.exceptions.synopsis]

```
namespace std::experimental {
inline namespace parallelism_v2 {

  class exception_list : public exception {
  public:
    using iterator = unspecified;

    size_t size() const noexcept;
    iterator begin() const noexcept;
    iterator end() const noexcept;

    const char* what() const noexcept override;
  };
}
}
```

1 The class `exception_list` owns a sequence of `exception_ptr` objects.

2 `exception_list::iterator` is an iterator which meets the forward iterator requirements and has a value type of `exception_ptr`.

`size_t size() const noexcept;`

3 *Returns:* The number of `exception_ptr` objects contained within the `exception_list`.

4 *Complexity:* Constant time.

`iterator begin() const noexcept;`

5 *Returns:* An iterator referring to the first `exception_ptr` object returned within the `exception_list`.

`iterator end() const noexcept;`

6 *Returns:* An iterator that is past the end of the owned sequence.

`const char* what() const noexcept override;`

7 *Returns:* An implementation-defined NTBS.

6 Execution policies [parallel.execpol]

6.1 Header <experimental/execution> synopsis [parallel.execpol.synopsis]

```
#include <execution>

namespace std::experimental {
inline namespace parallelism_v2 {
namespace execution {
  // 6.2, Unsequenced execution policy
  class unsequenced_policy;

  // 6.3, Vector execution policy
  class vector_policy;

  // 6.4, Execution policy objects
  inline constexpr unsequenced_policy unseq{ unspecified };
  inline constexpr vector_policy vec{ unspecified };
}
}
}
```

6.2 Unsequenced execution policy [parallel.execpol.unseq]

```
class unsequenced_policy { unspecified };
```

1 The class `unsequenced_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be vectorized, e.g., executed on a single thread using instructions that operate on multiple data items.

2 The invocations of element access functions in parallel algorithms invoked with an execution policy of type `unsequenced_policy` are permitted to execute in an unordered fashion in the calling thread, unsequenced with respect to one another within the calling thread. [*Note:* This means that multiple function object invocations may be interleaved on a single thread. — *end note*]

3 [*Note:* This overrides the usual guarantee from the C++ Standard, C++17 §4.6 that function executions do not overlap with one another. — *end note*]

4 During the execution of a parallel algorithm with the `experimental::execution::unsequenced_policy` policy, if the invocation of an element access function exits via an uncaught exception, `terminate()` will be called.

6.3 Vector execution policy [parallel.execpol.vec]

```
class vector_policy { unspecified };
```

1 The class `vector_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be vectorized. Additionally, such vectorization will result in an execution that respects the sequencing constraints of wavefront application (7.1). [*Note:* The implementation thus makes stronger guarantees than for `unsequenced_policy`, for example. — *end note*]

2 The invocations of element access functions in parallel algorithms invoked with an execution policy of type `vector_policy` are permitted to execute in unordered fashion in the calling thread, unsequenced with respect

to one another within the calling thread, subject to the sequencing constraints of wavefront application (7.1) for the last argument to `for_loop`, `for_loop_n`, `for_loop_strided`, or `for_loop_strided_n`.

3 During the execution of a parallel algorithm with the `experimental::execution::vector_policy` policy, if the invocation of an element access function exits via an uncaught exception, `terminate()` will be called.

6.4 Execution policy objects [parallel.execpol.objects]

```
inline constexpr execution::unsequenced_policy unseq { unspecified };
inline constexpr execution::vector_policy vec { unspecified };
```

1 The header `<experimental/execution>` declares a global object associated with each type of execution policy defined by this Technical Specification.

7 Parallel algorithms [parallel.alg]

7.1 Wavefront Application [parallel.alg.wavefront]

1 For the purposes of this section, an *evaluation* is a value computation or side effect of an expression, or an execution of a statement. Initialization of a temporary object is considered a subexpression of the expression that necessitates the temporary object.

2 An evaluation A *contains* an evaluation B if:

(2.1) — A and B are not potentially concurrent (C++17 §4.7.1); and

(2.2) — the start of A is the start of B or the start of A is sequenced before the start of B; and

(2.3) — the completion of B is the completion of A or the completion of B is sequenced before the completion of A.

[*Note:* This includes evaluations occurring in function invocations. — *end note*]

3 An evaluation A is *ordered before* an evaluation B if A is deterministically sequenced before B. [*Note:* If A is indeterminately sequenced with respect to B or A and B are unsequenced, then A is not ordered before B and B is not ordered before A. The ordered before relationship is transitive. — *end note*]

4 For an evaluation A ordered before an evaluation B, both contained in the same invocation of an element access function, A is a *vertical antecedent* of B if:

(4.1) — there exists an evaluation S such that:

(4.1.1) — S contains A, and

(4.1.2) — S contains all evaluations C (if any) such that A is ordered before C and C is ordered before B,

(4.1.3) — but S does not contain B, and

(4.2) — control reached B from A without executing any of the following:

(4.2.1) — a `goto` statement or `asm` declaration that jumps to a statement outside of S, or

(4.2.2) — a `switch` statement executed within S that transfers control into a substatement of a nested selection or iteration statement, or

(4.2.3) — a `throw` [*Note:* Even if caught — *end note*] , or

(4.2.4) — a `longjmp`.

[*Note:* Vertical antecedent is an irreflexive, antisymmetric, nontransitive relationship between two evaluations. Informally, A is a vertical antecedent of B if A is sequenced immediately before B or A is nested zero or more levels within a statement S that immediately precedes B. — *end note*]

5 In the following, X_i and X_j refer to evaluations of the same expression or statement contained in the application of an element access function corresponding to the i^{th} and j^{th} elements of the input sequence. [*Note:* There might be several evaluations X_k, Y_k, etc. of a single expression or statement in application k, for example, if the expression or statement appears in a loop within the element access function. — *end note*]

6 *Horizontally matched* is an equivalence relationship between two evaluations of the same expression. An evaluation B_i is *horizontally matched* with an evaluation B_j if:

(6.1) — both are the first evaluations in their respective applications of the element access function, or

(6.2) — there exist horizontally matched evaluations A_i and A_j that are vertical antecedents of evaluations B_i and B_j, respectively.

[*Note: Horizontally matched* establishes a theoretical lock-step relationship between evaluations in different applications of an element access function. — *end note*]

7 Let f be a function called for each argument list in a sequence of argument lists. *Wavefront application* of f requires that evaluation A_i be sequenced before evaluation B_j if $i < j$ and:

(7.1) — A_i is sequenced before some evaluation B_i and B_i is horizontally matched with B_j, or

(7.2) — A_i is horizontally matched with some evaluation A_j and A_j is sequenced before B_j.

[*Note: Wavefront application* guarantees that parallel applications i and j execute such that progress on application j never gets ahead of application i. — *end note*] [*Note:* The relationships between A_i and B_i and between A_j and B_j are *sequenced before*, not *vertical antecedent*. — *end note*]

7.2 Non-Numeric Parallel Algorithms [parallel.alg.ops]

7.2.1 Header `<experimental/algorithm>` synopsis [parallel.alg.ops.synopsis]

```
#include <algorithm>

namespace std::experimental {
inline namespace parallelism_v2 {
namespace execution {
  // 7.2.5, No vec
  template<class F>
    auto no_vec(F&& f) noexcept -> decltype(std::forward<F>(f)());

  // 7.2.6, Ordered update class
  template<class T>
    class ordered_update_t;

  // 7.2.7, Ordered update function template
  template<class T>
    ordered_update_t<T> ordered_update(T& ref) noexcept;
}

// Exposition only: Suppress template argument deduction.
template<class T> struct type_identity { using type = T; };
template<class T> using type_identity_t = typename type_identity<T>::type;

// 7.2.2, Reductions
template<class T, class BinaryOperation>
  unspecified reduction(T& var, const T& identity, BinaryOperation combiner);
template<class T>
  unspecified reduction_plus(T& var);
template<class T>
  unspecified reduction_multiplies(T& var);
template<class T>
  unspecified reduction_bit_and(T& var);
template<class T>
  unspecified reduction_bit_or(T& var);
template<class T>
  unspecified reduction_bit_xor(T& var);
template<class T>
  unspecified reduction_min(T& var);
```



```
template<class T>
  unspecified reduction_max(T& var);

// 7.2.3, Inductions
template<class T>
  unspecified induction(T&& var);
template<class T, class S>
  unspecified induction(T&& var, S stride);

// 7.2.4, For loop
template<class I, class... Rest>
  void for_loop(type_identity_t<I> start, I finish, Rest&&... rest);
template<class ExecutionPolicy,
         class I, class... Rest>
  void for_loop(ExecutionPolicy&& exec,
                type_identity_t<I> start, I finish, Rest&&... rest);
template<class I, class S, class... Rest>
  void for_loop_strided(type_identity_t<I> start, I finish,
                        S stride, Rest&&... rest);
template<class ExecutionPolicy,
         class I, class S, class... Rest>
  void for_loop_strided(ExecutionPolicy&& exec,
                        type_identity_t<I> start, I finish,
                        S stride, Rest&&... rest);
template<class I, class Size, class... Rest>
  void for_loop_n(I start, Size n, Rest&&... rest);
template<class ExecutionPolicy,
         class I, class Size, class... Rest>
  void for_loop_n(ExecutionPolicy&& exec,
                  I start, Size n, Rest&&... rest);
template<class I, class Size, class S, class... Rest>
  void for_loop_n_strided(I start, Size n, S stride, Rest&&... rest);
template<class ExecutionPolicy,
         class I, class Size, class S, class... Rest>
  void for_loop_n_strided(ExecutionPolicy&& exec,
                          I start, Size n, S stride, Rest&&... rest);
}
}
```

7.2.2 Reductions [parallel.alg.reductions]

1 Each of the function templates in this subclause (7.2.2) returns a *reduction object* of unspecified type having a *reduction value type* and encapsulating a *reduction identity* value for the reduction, a *combiner* function object, and a *live-out object* from which the initial value is obtained and into which the final value is stored.

2 An algorithm uses reduction objects by allocating an unspecified number of instances, known as *accumulators*, of the reduction value type. [*Note:* An implementation might, for example, allocate an accumulator for each thread in its private thread pool. — *end note*] Each accumulator is initialized with the object's reduction identity, except that the live-out object (which was initialized by the caller) comprises one of the accumulators. The algorithm passes a reference to an accumulator to each application of an element-access function, ensuring that no two concurrently executing invocations share the same accumulator. An accumulator can be shared between two applications that do not execute concurrently, but initialization is performed only once per accumulator.

3 Modifications to the accumulator by the application of element access functions accrue as partial results. At some point before the algorithm returns, the partial results are combined, two at a time, using the reduction

object's combiner operation until a single value remains, which is then assigned back to the live-out object. [*Note:* In order to produce useful results, modifications to the accumulator should be limited to commutative operations closely related to the combiner operation. For example if the combiner is `plus<T>`, incrementing the accumulator would be consistent with the combiner but doubling it or assigning to it would not. — *end note*]

```
template<class T, class BinaryOperation>
  unspecified reduction(T& var, const T& identity, BinaryOperation combiner);
```

4 *Requires:* `T` shall meet the requirements of `CopyConstructible` and `MoveAssignable`. The expression `var = combiner(var, var)` shall be well-formed.

5 *Returns:* A reduction object of unspecified type having reduction value type `T`, reduction identity `identity`, combiner function object `combiner`, and using the object referenced by `var` as its live-out object.

```
template<class T>
  unspecified reduction_plus(T& var);
template<class T>
  unspecified reduction_multiplies(T& var);
template<class T>
  unspecified reduction_bit_and(T& var);
template<class T>
  unspecified reduction_bit_or(T& var);
template<class T>
  unspecified reduction_bit_xor(T& var);
template<class T>
  unspecified reduction_min(T& var);
template<class T>
  unspecified reduction_max(T& var);
```

6 *Requires:* `T` shall meet the requirements of `CopyConstructible` and `MoveAssignable`.

7 *Returns:* A reduction object of unspecified type having reduction value type `T`, reduction identity and combiner operation as specified in Table 2 and using the object referenced by `var` as its live-out object.

Table 2 — Reduction identities and combiner operations

Function	Reduction Identity	Combiner Operation
`reduction_plus`	`T()`	`x + y`
`reduction_multiplies`	`T(1)`	`x * y`
`reduction_bit_and`	`(~T())`	`x & y`
`reduction_bit_or`	`T()`	`x | y`
`reduction_bit_xor`	`T()`	`x ^ y`
`reduction_min`	`var`	`min(x, y)`
`reduction_max`	`var`	`max(x, y)`

[*Example:* The following code updates each element of `y` and sets `s` to the sum of the squares.

```
extern int n;
extern float x[], y[], a;
float s = 0;
for_loop(execution::vec, 0, n,
  reduction(s, 0.0f, plus<>()),
  [&](int i, float& accum) {
    y[i] += a*x[i];
```



```
    accum += y[i]*y[i];
  }
);
```

— *end example*]

7.2.3 Inductions [parallel.alg.inductions]

1 Each of the function templates in this section return an *induction object* of unspecified type having an *induction value type* and encapsulating an initial value i of that type and, optionally, a *stride*.

2 For each element in the input range, an algorithm over input sequence S computes an *induction value* from an induction variable and ordinal position p within S by the formula $i + p * stride$ if a stride was specified or $i + p$ otherwise. This induction value is passed to the element access function.

3 An induction object may refer to a *live-out* object to hold the final value of the induction sequence. When the algorithm using the induction object completes, the live-out object is assigned the value $i + n * stride$, where n is the number of elements in the input range.

```
template<class T>
  unspecified induction(T&& var);
template<class T, class S>
  unspecified induction(T&& var, S stride);
```

4 *Returns:* An induction object with induction value type `remove_cv_t<remove_reference_t<T>>`, initial value `var`, and (if specified) stride `stride`. If `T` is an lvalue reference to non-`const` type, then the object referenced by `var` becomes the live-out object for the induction object; otherwise there is no live-out object.

7.2.4 For loop [parallel.alg.forloop]

```
template<class I, class... Rest>
  void for_loop(type_identity_t<I> start, I finish, Rest&&... rest);
template<class ExecutionPolicy, class I, class... Rest>
  void for_loop(ExecutionPolicy&& exec, type_identity_t<I> start, I finish, Rest&&... rest);

template<class I, class S, class... Rest>
  void for_loop_strided(type_identity_t<I> start, I finish, S stride, Rest&&... rest);
template<class ExecutionPolicy, class I, class S, class... Rest>
  void for_loop_strided(ExecutionPolicy&& exec, type_identity_t<I> start, I finish, S stride,
                        Rest&&... rest);

template<class I, class Size, class... Rest>
  void for_loop_n(I start, Size n, Rest&&... rest);
template<class ExecutionPolicy, class I, class Size, class... Rest>
  void for_loop_n(ExecutionPolicy&& exec, I start, Size n, Rest&&... rest);

template<class I, class Size, class S, class... Rest>
  void for_loop_n_strided(I start, Size n, S stride, Rest&&... rest);
template<class ExecutionPolicy, class I, class Size, class S, class... Rest>
  void for_loop_n_strided(ExecutionPolicy&& exec, I start, Size n, S stride, Rest&&... rest);
```

1 *Requires:* For the overloads with an `ExecutionPolicy`, `I` shall be an integral type or meet the requirements of a forward iterator type; otherwise, `I` shall be an integral type or meet the requirements of an input iterator type. `Size` shall be an integral type and `n` shall be non-negative. `S` shall have integral type and `stride` shall have non-zero value. `stride` shall be negative only if `I` has integral type or meets the requirements of a bidirectional iterator. The `rest` parameter pack shall have at least one

element, comprising objects returned by invocations of `reduction` (7.2.2) and/or `induction` (7.2.3) function templates followed by exactly one invocable element-access function, f. For the overloads with an `ExecutionPolicy`, f shall meet the requirements of `CopyConstructible`; otherwise, f shall meet the requirements of `MoveConstructible`.

2 *Effects:* Applies f to each element in the *input sequence*, as described below, with additional arguments corresponding to the reductions and inductions in the `rest` parameter pack. The length of the input sequence is:

(2.1) — `n`, if specified,

(2.2) — otherwise `finish - start` if neither `n` nor `stride` is specified,

(2.3) — otherwise `1 + (finish-start-1)/stride` if `stride` is positive,

(2.4) — otherwise `1 + (start-finish-1)/-stride`.

The first element in the input sequence is `start`. Each subsequent element is generated by adding `stride` to the previous element, if `stride` is specified, otherwise by incrementing the previous element. [*Note:* As described in the C++ Standard, C++17 §28.1, arithmetic on non-random-access iterators is performed using `advance` and `distance`. — *end note*] [*Note:* The order of the elements of the input sequence is important for determining ordinal position of an application of f, even though the applications themselves may be unordered. — *end note*]

The first argument to f is an element from the input sequence.[*Note:* If `I` is an iterator type, the iterators in the input sequence are not dereferenced before being passed to f. — *end note*] For each member of the `rest` parameter pack excluding f, an additional argument is passed to each application of f as follows:

(2.5) — If the pack member is an object returned by a call to a reduction function listed in section 7.2.2, then the additional argument is a reference to an accumulator of that reduction object.

(2.6) — If the pack member is an object returned by a call to `induction`, then the additional argument is the induction value for that induction object corresponding to the position of the application of f in the input sequence.

3 *Complexity:* Applies f exactly once for each element of the input sequence.

4 *Remarks:* If f returns a result, the result is ignored.

7.2.5 No vec [parallel.alg.novec]

```
template<class F>
  auto no_vec(F&& f) noexcept -> decltype(std::forward<F>(f)());
```

1 *Effects:* Evaluates `std::forward<F>(f)()`. When invoked within an element access function in a parallel algorithm using `vector_policy`, if two calls to `no_vec` are horizontally matched within a wavefront application of an element access function over input sequence S, then the execution of `f` in the application for one element in S is sequenced before the execution of `f` in the application for a subsequent element in S; otherwise, there is no effect on sequencing.

2 *Returns:* The result of `f`.

3 *Notes:* If `f` exits via an exception, then `terminate` will be called, consistent with all other potentially-throwing operations invoked with `vector_policy` execution.

[*Example:*

```
extern float y[];
extern int* p;
for_loop(vec, 0, n, [&](int i) {
  y[i] += y[i+1];
  if (y[i] < 0) {
```



```
        no_vec([]{
          *p++ = i;
        });
      }
    });
```

The updates `*p++ = i` will occur in the same order as if the policy were `seq`. — *end example*]

7.2.6 Ordered update class [parallel.alg.ordupdate.class]

```
template<class T>
class ordered_update_t {
  T& ref_;          // exposition only
public:
  ordered_update_t(T& loc) noexcept
    : ref_(loc) {}
  ordered_update_t(const ordered_update_t&) = delete;
  ordered_update_t& operator=(const ordered_update_t&) = delete;

  template <class U>
    auto operator=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ = std::move(rhs); }); }
  template <class U>
    auto operator+=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ += std::move(rhs); }); }
  template <class U>
    auto operator-=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ -= std::move(rhs); }); }
  template <class U>
    auto operator*=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ *= std::move(rhs); }); }
  template <class U>
    auto operator/=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ /= std::move(rhs); }); }
  template <class U>
    auto operator%=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ %= std::move(rhs); }); }
  template <class U>
    auto operator>>=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ >>= std::move(rhs); }); }
  template <class U>
    auto operator<<=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ <<= std::move(rhs); }); }
  template <class U>
    auto operator&=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ &= std::move(rhs); }); }
  template <class U>
    auto operator^=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ ^= std::move(rhs); }); }
  template <class U>
    auto operator|=(U rhs) const noexcept
      { return no_vec([&]{ return ref_ |= std::move(rhs); }); }

  auto operator++() const noexcept
    { return no_vec([&]{ return ++ref_; }); }
  auto operator++(int) const noexcept
```



```
    { return no_vec([&]{ return ref_++; }); }
  auto operator--() const noexcept
    { return no_vec([&]{ return --ref_; }); }
  auto operator--(int) const noexcept
    { return no_vec([&]{ return ref_--; }); }
};
```

1 An object of type `ordered_update_t<T>` is a proxy for an object of type `T` intended to be used within a parallel application of an element access function using a policy object of type `vector_policy`. Simple increments, assignments, and compound assignments to the object are forwarded to the proxied object, but are sequenced as though executed within a `no_vec` invocation. [*Note:* The return-value deduction of the forwarded operations results in these operations returning by value, not reference. This formulation prevents accidental collisions on accesses to the return value. — *end note*]

7.2.7 Ordered update function template [parallel.alg.ordupdate.func]

```
template<T>
  ordered_update_t<T> ordered_update(T& loc) noexcept;
```

1 *Returns:* `{ loc }`.

8 Task Block [parallel.taskblock]

8.1 Header <experimental/task_block> synopsis [parallel.taskblock.synopsis]

```
namespace std::experimental {
inline namespace parallelism_v2 {
  class task_cancelled_exception;

  class task_block;

  template<class F>
    void define_task_block(F&& f);

  template<class f>
    void define_task_block_restore_thread(F&& f);
}
}
```

8.2 Class task_cancelled_exception [parallel.taskblock.task__cancelled__exception]

```
namespace std::experimental {
inline namespace parallelism_v2 {

  class task_cancelled_exception : public exception
  {
  public:
    task_cancelled_exception() noexcept;
    virtual const char* what() const noexcept override;
  };
}
}
```

1 The class `task_cancelled_exception` defines the type of objects thrown by `task_block::run` or `task_block::wait` if they detect than an exception is pending within the current parallel block. See 8.5, below.

```
virtual const char* what() const noexcept;
```

2 *Returns:* An implementation-defined NTBS.

8.3 Class task_block [parallel.taskblock.class]

```
namespace std::experimental {
inline namespace parallelism_v2 {

  class task_block
  {
  private:
    ~task_block();

  public:
    task_block(const task_block&) = delete;
    task_block& operator=(const task_block&) = delete;
    void operator&() const = delete;
```



```
    template<class F>
      void run(F&& f);

    void wait();
  };
}
}
```

1 The class `task_block` defines an interface for forking and joining parallel tasks. The `define_task_block` and `define_task_block_restore_thread` function templates create an object of type `task_block` and pass a reference to that object to a user-provided function object.

2 An object of class `task_block` cannot be constructed, destroyed, copied, or moved except by the implementation of the task block library. Taking the address of a `task_block` object via `operator&` is ill-formed. Obtaining its address by any other means (including `addressof`) results in a pointer with an unspecified value; dereferencing such a pointer results in undefined behavior.

3 A `task_block` is *active* if it was created by the *nearest enclosing task block*, where *task block* refers to an invocation of `define_task_block` or `define_task_block_restore_thread` and *nearest enclosing* means the most recent invocation that has not yet completed. Code designated for execution in another thread by means other than the facilities in this section (e.g., using `thread` or `async`) are not enclosed in the task block and a `task_block` passed to (or captured by) such code is not active within that code. Performing any operation on a `task_block` that is not active results in undefined behavior.

4 When the argument to `task_block::run` is called, no `task_block` is active, not even the `task_block` on which `run` was called. (The function object should not, therefore, capture a `task_block` from the surrounding block.)

[*Example:*

```
define_task_block([&](auto& tb) {
  tb.run([&]{
    tb.run([] { f(); });               // Error: tb is not active within run
    define_task_block([&](auto& tb2) { // Define new task block
      tb2.run(f);
      ...
    });
  });
  ...
});
```

— end example]

[*Note:* Implementations are encouraged to diagnose the above error at translation time. *— end note*]

8.3.1 `task_block` member function template `run` [parallel.taskblock.class.run]

`template<class F> void run(F&& f);`

1 *Requires:* `F` shall be `MoveConstructible`. `DECAY_COPY(std::forward<F>(f))()` shall be a valid expression.

2 *Requires:* `*this` shall be the active `task_block`.

3 *Effects:* Evaluates `DECAY_COPY(std::forward<F>(f))()`, where `DECAY_COPY(std::forward<F>(f))` is evaluated synchronously within the current thread. The call to the resulting copy of the function object is permitted to run on an unspecified thread created by the implementation in an unordered fashion relative to the sequence of operations following the call to `run(f)` (the continuation), or

indeterminately sequenced within the same thread as the continuation. The call to `run` synchronizes with the call to the function object. The completion of the call to the function object synchronizes with the next invocation of wait on the same `task_block` or completion of the nearest enclosing task block (i.e., the `define_task_block` or `define_task_block_restore_thread` that created this `task_block`).

4 *Throws:* `task_cancelled_exception`, as described in 8.5.

5 *Remarks:* The `run` function may return on a thread other than the one on which it was called; in such cases, completion of the call to `run` synchronizes with the continuation. [*Note:* The return from `run` is ordered similarly to an ordinary function call in a single thread. — *end note*]

6 *Remarks:* The invocation of the user-supplied function object `f` may be immediate or may be delayed until compute resources are available. `run` might or might not return before the invocation of `f` completes.

8.3.2 `task_block` member function `wait` [parallel.taskblock.class.wait]

```
void wait();
```

1 *Preconditions:* `*this` shall be the active `task_block`.

2 *Effects:* Blocks until the tasks spawned using this `task_block` have completed.

3 *Throws:* `task_cancelled_exception`, as described in 8.5.

4 *Postconditions:* All tasks spawned by the nearest enclosing task block have completed.

5 *Remarks:* The `wait` function may return on a thread other than the one on which it was called; in such cases, completion of the call to `wait` synchronizes with subsequent operations. [*Note:* The return from `wait` is ordered similarly to an ordinary function call in a single thread. — *end note*]

[*Example:*

```
define_task_block([&](auto& tb) {
  tb.run([&]{ process(a, w, x); });     // Process a[w] through a[x]
  if (y < x) tb.wait();                 // Wait if overlap between [w,x) and [y,z)
  process(a, y, z);                     // Process a[y] through a[z]
});
```

— *end example*]

8.4 Function template `define_task_block` [parallel.taskblock.define_task_block]

```
template<class F> void define_task_block(F&& f);
template<class F> void define_task_block_restore_thread(F&& f);
```

1 *Requires:* Given an lvalue `tb` of type `task_block`, the expression `f(tb)` shall be well-formed.

2 *Effects:* Constructs a `task_block tb` and calls `f(tb)`.

3 *Throws:* `exception_list`, as specified in 8.5.

4 *Postconditions:* All tasks spawned from `f` have finished execution.

5 *Remarks:* The `define_task_block` function may return on a thread other than the one on which it was called unless there are no task blocks active on entry to `define_task_block` (see 8.3), in which case the function returns on the original thread. When `define_task_block` returns on a different thread, it synchronizes with operations following the call. [*Note:* The return from `define_task_block` is ordered similarly to an ordinary function call in a single thread. — *end note*] The `define_task_block_restore_thread` function always returns on the same thread as the one on which it was called.

6 *Notes:* It is expected (but not mandated) that `f` will (directly or indirectly) call `tb.run(`*`function-object`*`)`.

8.5 Exception Handling [parallel.taskblock.exceptions]

1 Every task_block has an associated exception list. When the task block starts, its associated exception list is empty.

2 When an exception is thrown from the user-provided function object passed to define_task_block or define_task_block_restore_thread, it is added to the exception list for that task block. Similarly, when an exception is thrown from the user-provided function object passed into task_block::run, the exception object is added to the exception list associated with the nearest enclosing task block. In both cases, an implementation may discard any pending tasks that have not yet been invoked. Tasks that are already in progress are not interrupted except at a call to task_block::run or task_block::wait as described below.

3 If the implementation is able to detect that an exception has been thrown by another task within the same nearest enclosing task block, then task_block::run or task_block::wait may throw task_canceled_exception; these instances of task_canceled_exception are not added to the exception list of the corresponding task block.

4 When a task block finishes with a non-empty exception list, the exceptions are aggregated into an exception_list object, which is then thrown from the task block.

5 The order of the exceptions in the exception_list object is unspecified.

9 Data-Parallel Types [parallel.simd]

9.1 General [parallel.simd.general]

1 The data-parallel library consists of data-parallel types and operations on these types. A data-parallel type consists of elements of an underlying arithmetic type, called the *element type*. The number of elements is a constant for each data-parallel type and called the *width* of that type.

2 Throughout this Clause, the term *data-parallel type* refers to all *supported* (9.3.1) specializations of the `simd` and `simd_mask` class templates. A *data-parallel object* is an object of *data-parallel type*.

3 An *element-wise operation* applies a specified operation to the elements of one or more data-parallel objects. Each such application is unsequenced with respect to the others. A *unary element-wise operation* is an element-wise operation that applies a unary operation to each element of a data-parallel object. A *binary element-wise operation* is an element-wise operation that applies a binary operation to corresponding elements of two data-parallel objects.

4 Throughout this Clause, the set of *vectorizable types* for a data-parallel type comprises all cv-unqualified arithmetic types other than `bool`.

5 [*Note:* The intent is to support acceleration through data-parallel execution resources, such as SIMD registers and instructions or execution units driven by a common instruction decoder. If such execution resources are unavailable, the interfaces support a transparent fallback to sequential execution. — *end note*]

9.2 Header `<experimental/simd>` synopsis [parallel.simd.synopsis]

```
namespace std::experimental {
inline namespace parallelism_v2 {
  namespace simd_abi {
    using scalar = see below;
    template<int N> using fixed_size = see below;
    template<class T> inline constexpr int max_fixed_size = implementation-defined;
    template<class T> using compatible = implementation-defined;
    template<class T> using native = implementation-defined;

    template<class T, size_t N, class... Abis> struct deduce { using type = see below; };
    template<class T, size_t N, class... Abis> using deduce_t =
      typename deduce<T, N, Abis...>::type;
  }

  struct element_aligned_tag {};
  struct vector_aligned_tag {};
  template<size_t> struct overaligned_tag {};
  inline constexpr element_aligned_tag element_aligned{};
  inline constexpr vector_aligned_tag vector_aligned{};
  template<size_t N> inline constexpr overaligned_tag<N> overaligned{};

  // 9.2.2, simd type traits
  template<class T> struct is_abi_tag;
  template<class T> inline constexpr bool is_abi_tag_v = is_abi_tag<T>::value;

  template<class T> struct is_simd;
  template<class T> inline constexpr bool is_simd_v = is_simd<T>::value;
```



```
template<class T> struct is_simd_mask;
template<class T> inline constexpr bool is_simd_mask_v = is_simd_mask<T>::value;

template<class T> struct is_simd_flag_type;
template<class T> inline constexpr bool is_simd_flag_type_v =
  is_simd_flag_type<T>::value;

template<class T, class Abi = simd_abi::compatible<T>> struct simd_size;
template<class T, class Abi = simd_abi::compatible<T>>
  inline constexpr size_t simd_size_v = simd_size<T,Abi>::value;

template<class T, class U = typename T::value_type> struct memory_alignment;
template<class T, class U = typename T::value_type>
  inline constexpr size_t memory_alignment_v = memory_alignment<T,U>::value;

template<class T, class V> struct rebind_simd { using type = see below; };
template<class T, class V> using rebind_simd_t = typename rebind_simd<T, V>::type;
template<int N, class V> struct resize_simd { using type = see below; };
template<int N, class V> using resize_simd_t = typename resize_simd<N, V>::type;

// 9.3, Class template simd
template<class T, class Abi = simd_abi::compatible<T>> class simd;
template<class T> using native_simd = simd<T, simd_abi::native<T>>;
template<class T, int N> using fixed_size_simd = simd<T, simd_abi::fixed_size<N>>;

// 9.5, Class template simd_mask
template<class T, class Abi = simd_abi::compatible<T>> class simd_mask;
template<class T> using native_simd_mask = simd_mask<T, simd_abi::native<T>>;
template<class T, int N> using fixed_size_simd_mask =
  simd_mask<T, simd_abi::fixed_size<N>>;

// 9.4.5, Casts
template<class T, class U, class Abi> see below simd_cast(const simd<U, Abi>&) noexcept;
template<class T, class U, class Abi> see below static_simd_cast(const simd<U, Abi>&) noexcept;

template<class T, class Abi>
  fixed_size_simd<T, simd_size_v<T, Abi>>
    to_fixed_size(const simd<T, Abi>&) noexcept;
template<class T, class Abi>
  fixed_size_simd_mask<T, simd_size_v<T, Abi>>
    to_fixed_size(const simd_mask<T, Abi>&) noexcept;
template<class T, int N>
  native_simd<T> to_native(const fixed_size_simd<T, N>&) noexcept;
template<class T, int N>
  native_simd_mask<T> to_native(const fixed_size_simd_mask<T, N>&) noexcept;
template<class T, int N>
  simd<T> to_compatible(const fixed_size_simd<T, N>&) noexcept;
template<class T, int N>
  simd_mask<T> to_compatible(const fixed_size_simd_mask<T, N>&) noexcept;

template<size_t... Sizes, class T, class Abi>
  tuple<simd<T, simd_abi::deduce_t<T, Sizes>>...>
    split(const simd<T, Abi>&) noexcept;
template<size_t... Sizes, class T, class Abi>
  tuple<simd_mask<T, simd_mask_abi::deduce_t<T, Sizes>>...>
```



```
    split(const simd_mask<T, Abi>&) noexcept;
template<class V, class Abi>
  array<V, simd_size_v<typename V::value_type, Abi> / V::size()>
    split(const simd<typename V::value_type, Abi>&) noexcept;
template<class V, class Abi>
  array<V, simd_size_v<typename V::simd_type::value_type, Abi> / V::size()>
    split(const simd_mask<typename V::simd_type::value_type, Abi>&) noexcept;

template<size_t N, class T, class A>
  array<resize_simd<simd_size_v<T, A> / N, simd<T, A>>, N>
    split_by(const simd<T, A>& x) noexcept;
template<size_t N, class T, class A>
  array<resize_simd<simd_size_v<T, A> / N, simd_mask<T, A>>, N>
    split_by(const simd_mask<T, A>& x) noexcept;

template<class T, class... Abis>
  simd<T, simd_abi::deduce_t<T, (simd_size_v<T, Abis> + ...)>>
    concat(const simd<T, Abis>&...) noexcept;
template<class T, class... Abis>
  simd_mask<T, simd_abi::deduce_t<T, (simd_size_v<T, Abis> + ...)>>
    concat(const simd_mask<T, Abis>&...) noexcept;

template<class T, class Abi, size_t N>
  resize_simd<simd_size_v<T, Abi> * N, simd<T, Abi>>
    concat(const array<simd<T, Abi>, N>& arr) noexcept;
template<class T, class Abi, size_t N>
  resize_simd<simd_size_v<T, Abi> * N, simd_mask<T, Abi>>
    concat(const array<simd_mask<T, Abi>, N>& arr) noexcept;

// 9.6.4, Reductions
template<class T, class Abi> bool all_of(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> bool any_of(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> bool none_of(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> bool some_of(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> int popcount(const simd_mask<T, Abi>&) noexcept;
template<class T, class Abi> int find_first_set(const simd_mask<T, Abi>&);
template<class T, class Abi> int find_last_set(const simd_mask<T, Abi>&);

bool all_of(T) noexcept;
bool any_of(T) noexcept;
bool none_of(T) noexcept;
bool some_of(T) noexcept;
int popcount(T) noexcept;
int find_first_set(T);
int find_last_set(T);

// 9.2.3, Where expression class templates
template<class M, class T> class const_where_expression;
template<class M, class T> class where_expression;

// 9.6.5, Where functions
template<class T, class Abi>
  where_expression<simd_mask<T, Abi>, simd<T, Abi>>
    where(const typename simd<T, Abi>::mask_type&, simd<T, Abi>&) noexcept;
```



```
template<class T, class Abi>
  const_where_expression<simd_mask<T, Abi>, simd<T, Abi>>
    where(const typename simd<T, Abi>::mask_type&, const simd<T, Abi>&) noexcept;

template<class T, class Abi>
  where_expression<simd_mask<T, Abi>, simd_mask<T, Abi>>
    where(const type_identity_t<simd_mask<T, Abit>>&, simd_mask<T, Abi>&) noexcept;

template<class T, class Abi>
  const_where_expression<simd_mask<T, Abi>, simd_mask<T, Abi>>
    where(const type_identity_t<simd_mask<T, Abit>>&, const simd_mask<T, Abi>&) noexcept;

template<class T>
  where_expression<bool, T>
    where(see below k, T& d) noexcept;

template<class T>
  const_where_expression<bool, T>
    where(see below k, const T& d) noexcept;

// 9.4.4, Reductions
template<class T, class Abi, class BinaryOperation = plus<>>
  T reduce(const simd<T, Abi>&,
           BinaryOperation = {});

template<class M, class V, class BinaryOperation>
  typename V::value_type reduce(const const_where_expression<M, V>& x,
                                typename V::value_type identity_element,
                                BinaryOperation binary_op);
template<class M, class V>
  typename V::value_type reduce(const const_where_expression<M, V>& x,
                                plus<> binary_op = {}) noexcept;
template<class M, class V>
  typename V::value_type reduce(const const_where_expression<M, V>& x,
                                multiplies<> binary_op) noexcept;
template<class M, class V>
  typename V::value_type reduce(const const_where_expression<M, V>& x,
                                bit_and<> binary_op) noexcept;
template<class M, class V>
  typename V::value_type reduce(const const_where_expression<M, V>& x,
                                bit_or<> binary_op) noexcept;
template<class M, class V>
  typename V::value_type reduce(const const_where_expression<M, V>& x,
                                bit_xor<> binary_op) noexcept;

template<class T, class Abi>
  T hmin(const simd<T, abi>&) noexcept;
template<class M, class V>
  typename V::value_type hmin(const const_where_expression<M, V>&) noexcept;
template<class T, class Abi>
  T hmax(const simd<T, abi>&) noexcept;
template<class M, class V>
  typename V::value_type hmax(const const_where_expression<M, V>&) noexcept;

// 9.4.6, Algorithms
```



```
  template<class T, class Abi>
    simd<T, Abi>
      min(const simd<T, Abi>& a, const simd<T, Abi>& b) noexcept;
  template<class T, class Abi>
    simd<T, Abi>
      max(const simd<T, Abi>& a, const simd<T, Abi>& b) noexcept;
  template<class T, class Abi>
    pair<simd<T, Abi>, simd<T, Abi>>
      minmax(const simd<T, Abi>& a, const simd<T, Abi>& b) noexcept;
  template<class T, class Abi>
    simd<T, Abi>
      clamp(const simd<T, Abi>& v,
            const simd<T, Abi>& lo,
            const simd<T, Abi>& hi);
}
}
```

1 The header `<experimental/simd>` defines class templates, tag types, trait types, and function templates for element-wise operations on data-parallel objects.

9.2.1 `simd` ABI tags [parallel.simd.abi]

```
namespace simd_abi {
  using scalar = see below;
  template<int N> using fixed_size = see below;
  template<class T> inline constexpr int max_fixed_size = implementation-defined;
  template<class T> using compatible = implementation-defined;
  template<class T> using native = implementation-defined;
}
```

1 An *ABI tag* is a type in the `std::experimental::parallelism_v2::simd_abi` namespace that indicates a choice of size and binary representation for objects of data-parallel type. [*Note:* The intent is for the size and binary representation to depend on the target architecture. — *end note*] The ABI tag, together with a given element type implies a number of elements. ABI tag types are used as the second template argument to `simd` and `simd_mask`.

2 [*Note:* The ABI tag is orthogonal to selecting the machine instruction set. The selected machine instruction set limits the usable ABI tag types, though (see 9.3.1). The ABI tags enable users to safely pass objects of data-parallel type between translation unit boundaries (e.g. function calls or I/O). — *end note*]

3 `scalar` is an alias for an unspecified ABI tag that is different from `fixed_size<1>`. Use of the `scalar` tag type requires data-parallel types to store a single element (i.e., `simd<T, simd_abi::scalar>::size()` returns 1).

4 The value of `max_fixed_size<T>` is at least 32.

5 `fixed_size<N>` is an alias for an unspecified ABI tag. `fixed_size` does not introduce a non-deduced context. Use of the `simd_abi::fixed_size<N>` tag type requires data-parallel types to store `N` elements (i.e. `simd<T, simd_abi::fixed_size<N>>::size()` is `N`). `simd<T, fixed_size<N>>` and `simd_mask<T, fixed_size<N>>` with `N > 0` and `N <= max_fixed_size<T>` shall be supported. Additionally, for every supported `simd<T, Abi>` (see 9.3.1), where `Abi` is an ABI tag that is not a specialization of `simd_abi::fixed_size`, `N == simd<T, Abi>::size()` shall be supported.

6 [*Note:* It is unspecified whether `simd<T, fixed_size<T, fixed_size<N>>` with `N > max_fixed_size<T>` is supported. The value of `max_fixed_size<T>` can depend on compiler flags and can change between different compiler versions. — *end note*]

7 [*Note:* An implementation can forego ABI compatibility between differently compiled translation units

for `simd` and `simd_mask` specializations using the same `simd_abi::fixed_size<N>` tag. Otherwise, the efficiency of `simd<T, Abi>` is likely to be better than for `simd<T, fixed_size<simd_size_v<T, Abi>>>` (with `Abi` not a specialization of `simd_abi::fixed_size`). — *end note*]

8 An implementation may define additional *extended ABI tag* types in the `std::experimental::parallelism_v2::simd_abi` namespace, to support other forms of data-parallel computation.

9 `compatible<T>` is an implementation-defined alias for an ABI tag. [*Note:* The intent is to use the ABI tag producing the most efficient data-parallel execution for the element type `T` that ensures ABI compatibility between translation units on the target architecture. — *end note*] [*Example:* Consider a target architecture supporting the extended ABI tags `__simd128` and `__simd256`, where the `__simd256` type requires an optional ISA extension on said architecture. Also, the target architecture does not support `long double` with either ABI tag. The implementation therefore defines `compatible<T>` as an alias for:

(9.1) — `scalar` if `T` is the same type as `long double`, and

(9.2) — `__simd128` otherwise.

— *end example*]

10 `native<T>` is an implementation-defined alias for an ABI tag. [*Note:* The intent is to use the ABI tag producing the most efficient data-parallel execution for the element type `T` that is supported on the currently targeted system. For target architectures without ISA extensions, the `native<T>` and `compatible<T>` aliases will likely be the same. For target architectures with ISA extensions, compiler flags may influence the `native<T>` alias while `compatible<T>` will be the same independent of such flags. — *end note*] [*Example:* Consider a target architecture supporting the extended ABI tags `__simd128` and `__simd256`, where hardware support for `__simd256` only exists for floating-point types. The implementation therefore defines `native<T>` as an alias for

(10.1) — `__simd256` if `T` is a floating-point type, and

(10.2) — `__simd128` otherwise.

— *end example*]

```
template<T, size_t N, class... Abis> struct deduce { using type = see below; };
```

11 The member `type` shall be present if and only if

(11.1) — `T` is a vectorizable type, and

(11.2) — `simd_abi::fixed_size<N>` is supported (see 9.2.1), and

(11.3) — every type in the `Abis` pack is an ABI tag.

12 Where present, the member typedef `type` shall name an ABI tag type that satisfies

(12.1) — `simd_size<T, type> == N`, and

(12.2) — `simd<T, type>` is default constructible (see 9.3.1).

If `N` is `1`, the member typedef `type` is `simd_abi::scalar`. Otherwise, if there are multiple ABI tag types that satisfy the constraints, the member typedef `type` is implementation-defined. [*Note:* It is expected that extended ABI tags can produce better optimizations and thus are preferred over `simd_abi::fixed_size<N>`. Implementations can base the choice on `Abis`, but can also ignore the `Abis` arguments. — *end note*]

13 The behavior of a program that adds specializations for `deduce` is undefined.

9.2.2 simd type traits [parallel.simd.traits]

```
template<class T> struct is_abi_tag { see below };
```

1 The type `is_abi_tag<T>` is a `UnaryTypeTrait` with a base characteristic of `true_type` if `T` is a standard or extended ABI tag, and `false_type` otherwise.

2 The behavior of a program that adds specializations for `is_abi_tag` is undefined.

```
template<class T> struct is_simd { see below };
```

3 The type `is_simd<T>` is a `UnaryTypeTrait` with a base characteristic of `true_type` if `T` is a specialization of the `simd` class template, and `false_type` otherwise.

4 The behavior of a program that adds specializations for `is_simd` is undefined.

```
template<class T> struct is_simd_mask { see below };
```

5 The type `is_simd_mask<T>` is a `UnaryTypeTrait` with a base characteristic of `true_type` if `T` is a specialization of the `simd_mask` class template, and `false_type` otherwise.

6 The behavior of a program that adds specializations for `is_simd_mask` is undefined.

```
template<class T> struct is_simd_flag_type { see below };
```

7 The type `is_simd_flag_type<class T>` is a `UnaryTypeTrait` with a base characteristic of `true_type` if `T` is one of

(7.1) — `element_aligned_tag`, or

(7.2) — `vector_aligned_tag`, or

(7.3) — `overaligned_tag<N>` with `N > 0` and `N` an integral power of two,

and `false_type` otherwise.

8 The behavior of a program that adds specializations for `is_simd_flag_type` is undefined.

```
template<class T, class Abi = simd_abi::compatible<T>> struct simd_size { see below };
```

9 `simd_size<T, Abi>` shall have a member `value` if and only if

(9.1) — `T` is a vectorizable type, and

(9.2) — `is_abi_tag_v<Abi>` is `true`.

[*Note:* The rules are different from those in (9.3.1). — *end note*]

10 If `value` is present, the type `simd_size<T, Abi>` is a `BinaryTypeTrait` with a base characteristic of `integral_constant<size_t, N>` with `N` equal to the number of elements in a `simd<T, Abi>` object. [*Note:* If `simd<T, Abi>` is not supported for the currently targeted system, `simd_size<T, Abi>::value` produces the value `simd<T, Abi>::size()` would return if it were supported. — *end note*]

11 The behavior of a program that adds specializations for `simd_size` is undefined.

```
template<class T, class U = typename T::value_type> struct memory_alignment { see below };
```

12 `memory_alignment<T, U>` shall have a member `value` if and only if

(12.1) — `is_simd_mask_v<T>` is `true` and `U` is `bool`, or

(12.2) — `is_simd_v<T>` is `true` and `U` is a vectorizable type.

13 If `value` is present, the type `memory_alignment<T, U>` is a `BinaryTypeTrait` with a base characteristic of `integral_constant<size_t, N>` for some implementation-defined `N` (see 9.3.5 and 9.5.4). [*Note:* `value` identifies the alignment restrictions on pointers used for (converting) loads and stores for the give type `T` on arrays of type `U`. — *end note*]

14 The behavior of a program that adds specializations for `memory_alignment` is undefined.

```
template<class T, class V> struct rebind_simd { using type = see below; };
```

15 The member `type` is present if and only if

(15.1) — `V` is either `simd<U, Abi0>` or `simd_mask<U, Abi0>`, where `U` and `Abi0` are deduced from `V`, and

(15.2) — `T` is a vectorizable type, and

(15.3) — `simd_abi::deduce<T, simd_size_v<U, Abi0>, Abi0>` has a member type `type`.

16 Let `Abi1` denote the type `deduce_t<T, simd_size_v<U, Abi0>, Abi0>`. Where present, the member typedef `type` names `simd<T, Abi1>` if `V` is `simd<U, Abi0>` or `simd_mask<T, Abi1>` if `V` is `simd_mask<U, Abi0>`.

```
template<int N, class V> struct resize_simd { using type = see below; };
```

17 The member `type` is present if and only if

(17.1) — `V` is either `simd<T, Abi0>` or `simd_mask<T, Abi0>`, where `T` and `Abi0` are deduced from `V`, and

(17.2) — `simd_abi::deduce<T, N, Abi0>` has a member type `type`.

18 Let `Abi1` denote the type `deduce_t<T, N, Abi0>`. Where present, the member typedef `type` names `simd<T, Abi1>` if `V` is `simd<T, Abi0>` or `simd_mask<T, Abi1>` if `V` is `simd_mask<T, Abi0>`.

9.2.3 Where expression class templates [parallel.simd.whereexpr]

```
template<class M, class T> class const_where_expression {
  const M mask;     // exposition only
  T& data;          // exposition only

public:
  const_where_expression(const const_where_expression&) = delete;
  const_where_expression& operator=(const const_where_expression&) = delete;

  T operator-() const && noexcept;
  T operator+() const && noexcept;
  T operator~() const && noexcept;

  template<class U, class Flags> void copy_to(U* mem, Flags f) const &&;
};

template<class M, class T>
class where_expression : public const_where_expression<M, T> {
public:
  template<class U> void operator=(U&& x) && noexcept;
  template<class U> void operator+=(U&& x) && noexcept;
  template<class U> void operator-=(U&& x) && noexcept;
  template<class U> void operator*=(U&& x) && noexcept;
  template<class U> void operator/=(U&& x) && noexcept;
  template<class U> void operator%=(U&& x) && noexcept;
  template<class U> void operator&=(U&& x) && noexcept;
```



```
    template<class U> void operator|=(U&& x) && noexcept;
    template<class U> void operator^=(U&& x) && noexcept;
    template<class U> void operator<<=(U&& x) && noexcept;
    template<class U> void operator>>=(U&& x) && noexcept;

    void operator++() && noexcept;
    void operator++(int) && noexcept;
    void operator--() && noexcept;
    void operator--(int) && noexcept;

    template<class U, class Flags> void copy_from(const U* mem, Flags) &&;
  };
```

1 The class templates `const_where_expression` and `where_expression` abstract the notion of selecting elements of a given object of arithmetic or data-parallel type.

2 The first templates argument `M` shall be cv-unqualified `bool` or a cv-unqualified `simd_mask` specialization.

3 If `M` is `bool`, `T` shall be a cv-unqualified arithmetic type. Otherwise, `T` shall either be `M` or `typename M::simd_type`.

4 In this subclause, if `M` is `bool`, `data[0]` is used interchangably for `data`, `mask[0]` is used interchangably for `mask`, and `M::size()` is used interchangably for `1`.

5 The *selected indices* signify the integers $i \in \{j \in \mathbb{N} | j < \texttt{M::size()} \wedge \texttt{mask}[j]\}$. The *selected elements* signify the elements `data[`i`]` for all selected indices i.

6 In this subclause, the type `value_type` is an alias for `T` if `M` is `bool`, or an alias for `typename T::value_type` if `is_simd_mask_v<M>` is `true`.

7 [*Note:* The `where` functions 9.6.5 initialize `mask` with the first argument to `where` and `data` with the second argument to `where`. — *end note*]

```
T operator-() const && noexcept;
T operator+() const && noexcept;
T operator~() const && noexcept;
```

8 *Returns:* A copy of `data` with the indicated unary operator applied to all selected elements.

```
template<class U, class Flags> void copy_to(U* mem, Flags) const &&;
```

9 *Requires:*

(9.1) — If `M` is not `bool`, the largest selected index is less than the number of values pointed to by `mem`.

(9.2) — If the template parameter `Flags` is `vector_aligned_tag`, `mem` shall point to storage aligned by `memory_alignment_v<T, U>`.

(9.3) — If the template parameter `Flags` is `overaligned_tag<N>`, `mem` shall point to storage aligned by `N`.

(9.4) — If the template parameter `Flags` is `element_aligned_tag`, `mem` shall point to storage aligned by `alignof(U)`.

10 *Effects:* Copies the selected elements as if `mem[`i`] = static_cast<U>(data[`i`])` for all selected indices i.

11 *Throws:* Nothing.

12 *Remarks:* This function shall not participate in overload resolution unless

(12.1) — `is_simd_flag_type_v<Flags>` is `true`, and

(12.2) — either

(12.2.1) — `U` is `bool` and `value_type` is `bool`, or

(12.2.2) — `U` is a vectorizable type and `value_type` is not `bool`.

```
template<class U> void operator=(U&& x) && noexcept;
```

13 *Effects:* Replaces `data[`*i*`]` with `static_cast<T>(std::forward<U>(x))[`*i*`]` for all selected indices i.

14 *Remarks:* This operator shall not participate in overload resolution unless `U` is convertible to `T`.

```
template<class U> void operator+=(U&& x) && noexcept;
template<class U> void operator-=(U&& x) && noexcept;
template<class U> void operator*=(U&& x) && noexcept;
template<class U> void operator/=(U&& x) && noexcept;
template<class U> void operator%=(U&& x) && noexcept;
template<class U> void operator&=(U&& x) && noexcept;
template<class U> void operator|=(U&& x) && noexcept;
template<class U> void operator^=(U&& x) && noexcept;
template<class U> void operator<<=(U&& x) && noexcept;
template<class U> void operator>>=(U&& x) && noexcept;
```

15 *Effects:* Replaces `data[`*i*`]` with `static_cast<T>(data @ std::forward<U>(x))[`*i*`]` (where `@` denotes the indicated operator) for all selected indices i.

16 *Remarks:* Each of these operators shall not participate in overload resolution unless the return type of `data @ std::forward<U>(x)` is convertible to `T`. It is unspecified whether the binary operator, implied by the compound assignment operator, is executed on all elements or only on the selected elements.

```
void operator++() && noexcept;
void operator++(int) && noexcept;
void operator--() && noexcept;
void operator--(int) && noexcept;
```

17 *Effects:* Applies the indicated operator to the selected elements.

18 *Remarks:* Each of these operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type `T`.

```
template<class U, class Flags> void copy_from(const U* mem, Flags) &&;
```

19 *Requires:*

(19.1) — If `is_simd_flag_type_v<U>` is `true`, for all selected indices i, i shall be less than the number of values pointed to by `mem`.

(19.2) — If the template parameter `Flags` is `vector_aligned_tag`, `mem` shall point to storage aligned by `memory_alignment_v<T, U>`.

(19.3) — If the template parameter `Flags` is `overaligned_tag<N>`, `mem` shall point to storage aligned by `N`.

(19.4) — If the template parameter `Flags` is `element_aligned_tag`, `mem` shall point to storage aligned by `alignof(U)`.

20 *Effects:* Replaces the selected elements as if `data[`*i*`] = static_cast<value_type>(mem[`*i*`])` for all selected indices i.

21 *Throws:* Nothing.

22 *Remarks:* This function shall not participate in overload resolution unless

(22.1) — `is_simd_flag_type_v<Flags>` is `true`, and

(22.2) — either

(22.2.1) — `U` is `bool` and `value_type` is `bool`, or

(22.2.2) — `U` is a vectorizable type and `value_type` is not `bool`.

9.3 Class template simd [parallel.simd.class]

9.3.1 Class template simd overview [parallel.simd.overview]

```
template<class T, class Abi> class simd {
public:
  using value_type = T;
  using reference = see below;
  using mask_type = simd_mask<T, Abi>;
  using abi_type = Abi;

  static constexpr size_t size() noexcept;

  simd() noexcept = default;

  // 9.3.4, simd constructors
  template<class U> simd(U&& value) noexcept;
  template<class U> simd(const simd<U, simd_abi::fixed_size<size()>>&) noexcept;
  template<class G> explicit simd(G&& gen) noexcept;
  template<class U, class Flags> simd(const U* mem, Flags f);

  // 9.3.5, simd copy functions
  template<class U, class Flags> copy_from(const U* mem, Flags f);
  template<class U, class Flags> copy_to(U* mem, Flags f);

  // 9.3.6, simd subscript operators
  reference operator[](size_t);
  value_type operator[](size_t) const;

  // 9.3.7, simd unary operators
  simd& operator++() noexcept;
  simd operator++(int) noexcept;
  simd& operator--() noexcept;
  simd operator--(int) noexcept;
  mask_type operator!() const noexcept;
  simd operator~() const noexcept;
  simd operator+() const noexcept;
  simd operator-() const noexcept;

  // 9.4.1, simd binary operators
  friend simd operator+(const simd&, const simd&) noexcept;
  friend simd operator-(const simd&, const simd&) noexcept;
  friend simd operator*(const simd&, const simd&) noexcept;
  friend simd operator/(const simd&, const simd&) noexcept;
  friend simd operator%(const simd&, const simd&) noexcept;
  friend simd operator&(const simd&, const simd&) noexcept;
  friend simd operator|(const simd&, const simd&) noexcept;
  friend simd operator^(const simd&, const simd&) noexcept;
  friend simd operator<<(const simd&, const simd&) noexcept;
  friend simd operator>>(const simd&, const simd&) noexcept;
  friend simd operator<<(const simd&, int) noexcept;
  friend simd operator>>(const simd&, int) noexcept;

  // 9.4.2, simd compound assignment
  friend simd& operator+=(simd&, const simd&) noexcept;
  friend simd& operator-=(simd&, const simd&) noexcept;
```



```
    friend simd& operator*=(simd&, const simd&) noexcept;
    friend simd& operator/=(simd&, const simd&) noexcept;
    friend simd& operator%=(simd&, const simd&) noexcept;
    friend simd& operator&=(simd&, const simd&) noexcept;
    friend simd& operator|=(simd&, const simd&) noexcept;
    friend simd& operator^=(simd&, const simd&) noexcept;
    friend simd& operator<<=(simd&, const simd&) noexcept;
    friend simd& operator>>=(simd&, const simd&) noexcept;
    friend simd& operator<<=(simd&, int) noexcept;
    friend simd& operator>>=(simd&, int) noexcept;

    // 9.4.3, simd compare operators
    friend mask_type operator==(const simd&, const simd&) noexcept;
    friend mask_type operator!=(const simd&, const simd&) noexcept;
    friend mask_type operator>=(const simd&, const simd&) noexcept;
    friend mask_type operator<=(const simd&, const simd&) noexcept;
    friend mask_type operator>(const simd&, const simd&) noexcept;
    friend mask_type operator<(const simd&, const simd&) noexcept;
  };
```

1 The class template `simd` is a data-parallel type. The width of a given `simd` specialization is a constant expression, determined by the template parameters.

2 Every specialization of `simd` shall be a complete type. The specialization `simd<T, Abi>` is supported if `T` is a vectorizable type and

(2.1) — `Abi` is `simd_abi::scalar`, or

(2.2) — `Abi` is `simd_abi::fixed_size<N>`, with `N` constrained as defined in 9.2.1.

If `Abi` is an extended ABI tag, it is implementation-defined whether `simd<T, Abi>` is supported. [*Note:* The intent is for implementations to decide on the basis of the currently targeted system. — *end note*]

If `simd<T, Abi>` is not supported, the specialization shall have a deleted default constructor, deleted destructor, deleted copy constructor, and deleted copy assignment. Otherwise, the following are true:

(2.3) — `is_nothrow_move_constructible_v<simd<T, Abi>>`, and

(2.4) — `is_nothrow_move_assignable_v<simd<T, Abi>>`, and

(2.5) — `is_nothrow_default_constructible_v<simd<T, Abi>>`.

[*Example:* Consider an implementation that defines the extended ABI tags `__simd_x` and `__gpu_y`. When the compiler is invoked to translate to a machine that has support for the `__simd_x` ABI tag for all arithmetic types other than `long double` and no support for the `__gpu_y` ABI tag, then:

(2.6) — `simd<T, simd_abi::__gpu_y>` is not supported for any `T` and has a deleted constructor.

(2.7) — `simd<long double, simd_abi::__simd_x>` is not supported and has a deleted constructor.

(2.8) — `simd<double, simd_abi::__simd_x>` is supported.

(2.9) — `simd<long double, simd_abi::scalar>` is supported.

— *end example*]

3 Default intialization performs no initialization of the elements; value-initialization initializes each element with `T()`. [*Note:* Thus, default initialization leaves the elements in an indeterminate state. — *end note*]

4 Implementations should enable explicit conversion from and to implementation-defined types. This adds one or more of the following declarations to class `simd`:

```
explicit operator implementation-defined() const;
explicit simd(const implementation-defined& init);
```

[*Example:* Consider an implementation that supports the type `__vec4f` and the function `__vec4f _vec4f_addsub(__vec4f, __vec4f)` for the currently targeted system. A user may require the use of `_vec4f_addsub` for maximum performance and thus writes:

```
using V = simd<float, simd_abi::__simd128>;
V addsub(V a, V b) {
  return static_cast<V>(_vec4f_addsub(static_cast<__vec4f>(a), static_cast<__vec4f>(b)));
}
```

— *end example*]

9.3.2 `simd` width [parallel.simd.width]

```
static constexpr size_t size() noexcept;
```

1 *Returns:* The width of `simd<T, Abi>`.

9.3.3 Element references [parallel.simd.reference]

1 A `reference` is an object that refers to an element in a `simd` or `simd_mask` object. `reference::value_type` is the same type as `simd::value_type` or `simd_mask::value_type`, respectively.

2 Class `reference` is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name.

```
class reference // exposition only
{
public:
  reference() = delete;
  reference(const reference&) = delete;

  operator value_type() const noexcept;

  template<class U> reference operator=(U&& x) && noexcept;

  template<class U> reference operator+=(U&& x) && noexcept;
  template<class U> reference operator-=(U&& x) && noexcept;
  template<class U> reference operator*=(U&& x) && noexcept;
  template<class U> reference operator/=(U&& x) && noexcept;
  template<class U> reference operator%=(U&& x) && noexcept;
  template<class U> reference operator|=(U&& x) && noexcept;
  template<class U> reference operator&=(U&& x) && noexcept;
  template<class U> reference operator^=(U&& x) && noexcept;
  template<class U> reference operator<<=(U&& x) && noexcept;
  template<class U> reference operator>>=(U&& x) && noexcept;

  reference operator++() && noexcept;
  value_type operator++(int) && noexcept;
  reference operator--() && noexcept;
```



```
    value_type operator--(int) && noexcept;

    friend void swap(reference&& a, reference&& b) noexcept;
    friend void swap(value_type&& a, reference&& b) noexcept;
    friend void swap(reference&& a, value_type&& b) noexcept;
  };
```

```
operator value_type() const noexcept;
```

3 *Returns:* The value of the element referred to by `*this`.

```
template<class U> reference operator=(U&& x) && noexcept;
```

4 *Effects:* Replaces the referred to element in `simd` or `simd_mask` with `static_cast<value_type>(std::forward<U>(x))`.

5 *Returns:* A copy of `*this`.

6 *Remarks:* This function shall not participate in overload resolution unless `declval<value_type&>() = std::forward>U>(x)` is well-formed.

```
template<class U> reference operator+=(U&& x) && noexcept;
template<class U> reference operator-=(U&& x) && noexcept;
template<class U> reference operator*=(U&& x) && noexcept;
template<class U> reference operator/=(U&& x) && noexcept;
template<class U> reference operator%=(U&& x) && noexcept;
template<class U> reference operator|=(U&& x) && noexcept;
template<class U> reference operator&=(U&& x) && noexcept;
template<class U> reference operator^=(U&& x) && noexcept;
template<class U> reference operator<<=(U&& x) && noexcept;
template<class U> reference operator>>=(U&& x) && noexcept;
```

7 *Effects:* Applies the indicated compound operator to the referred to element in `simd` or `simd_mask` and `std::forward<U>(x)`.

8 *Returns:* A copy of `*this`.

9 *Remarks:* This function shall not participate in overload resolution unless `declval<value_type&>() @= std::forward<U>(x)` (where `@=` denotes the indicated compound assignment operator) is well-formed.

```
reference operator++() && noexcept;
reference operator--() && noexcept;
```

10 *Effects:* Applies the indicated operator to the referred to element in `simd` or `simd_mask`.

11 *Returns:* A copy of `*this`.

12 *Remarks:* This function shall not participate in overload resolution unless the indicated operator can be applied to objects of type `value_type`.

```
value_type operator++(int) && noexcept;
value_type operator--(int) && noexcept;
```

13 *Effects:* Applies the indicated operator to the referred to element in `simd` or `simd_mask`.

14 *Returns:* A copy of the referred to element before applying the indicated operator.

15 *Remarks:* This function shall not participate in overload resolution unless the indicated operator can be applied to objects of type `value_type`.

```
friend void swap(reference&& a, reference&& b) noexcept;
friend void swap(value_type& a, reference&& b) noexcept;
friend void swap(reference&& a, value_type& b) noexcept;
```

16 *Effects:* Exchanges the values a and b refer to.

9.3.4 simd constructors [parallel.simd.ctor]

```
template<class U> simd(U&&) noexcept;
```

1 *Effects:* Constructs an object with each element initialized to the value of the argument after conversion to value_type.

2 *Remarks:* Let From denote the type remove_cv_t<remove_reference_t<U>>. This constructor shall not participate in overload resolution unless:

(2.1) — From is a vectorizable type and every possibly value of From can be represented with type value_type, or

(2.2) — From is not an arithmetic type and is implicitly convertible to value_type, or

(2.3) — From is int, or

(2.4) — From is unsigned int and value_type is an unsigned integral type.

```
template<class U> simd(const simd<U, simd_abi::fixed_size<size()>>& x) noexcept;
```

3 *Effects:* Constructs an object where the i^{th} element equals static_cast<T>(x[i]) for all i in the range of [0, size()).

4 *Remarks:* This constructor shall not participate in overload resolution unless

(4.1) — abi_type is simd_abi::fixed_size<size()>, and

(4.2) — every possible value of U can be represented with type value_type, and

(4.3) — if both U and value_type are integral, the integer conversion rank (C++17 §7.15) of value_type is greater than the integer conversion rank of U.

```
template<class G> simd(G&& gen) noexcept;
```

5 *Effects:* Constructs an object where the i^{th} element is initialized to gen(integral_constant<size_t, i>()).

6 *Remarks:* This constructor shall not participate in overload resolution unless simd(gen(integral_constant<size_t, i>())) is well-formed for all i in the range of [0, size()).

7 The calls to gen are unsequenced with respect to each other. Vectorization-unsafe standard library functions may not be invoked by gen (C++17 §28.4.3).

```
template<class U, class Flags> simd(const U* mem, Flags);
```

8 *Requires:*

(8.1) — [mem, mem + size()) is a valid range.

(8.2) — If the template parameter Flags is vector_aligned_tag, mem shall point to storage aligned by memory_alignment_v<simd, U>.

(8.3) — If the template parameter Flags is overaligned_tag<N>, mem shall point to storage aligned by N.

(8.4) — If the template parameter Flags is element_aligned_tag, mem shall point to storage aligned by alignof(U).

9 *Effects:* Constructs an object where the i^{th} element is initialized to `static_cast<T>(mem[`i`])` for all i in the range of `[0, size())`.

10 *Remarks:* This constructor shall not participate in overload resolution unless

(10.1) — `is_simd_flag_type_v<Flags>` is `true`, and

(10.2) — `U` is a vectorizable type.

9.3.5 simd copy functions [parallel.simd.copy]

```
template<class U, class Flags> void copy_from(const U* mem, Flags);
```

1 *Requires:*

(1.1) — `[mem, mem + size())` is a valid range.

(1.2) — If the template parameter `Flags` is `vector_aligned_tag`, `mem` shall point to storage aligned by `memory_alignment_v<simd, U>`.

(1.3) — If the template parameter `Flags` is `overaligned_tag<N>`, `mem` shall point to storage aligned by `N`.

(1.4) — If the template parameter `Flags` is `element_aligned_tag`, `mem` shall point to storage aligned by `alignof(U)`.

2 *Effects:* Replaces the elements of the `simd` object such that the i^{th} element is assigned with `static_cast<T>(mem[`i`])` for all i in the range of `[0, size())`.

3 *Remarks:* This function shall not participate in overload resolution unless

(3.1) — `is_simd_flag_type_v<Flags>` is `true`, and

(3.2) — `U` is a vectorizable type.

```
template<class U, class Flags> void copy_to(U* mem, Flags) const;
```

4 *Requires:*

(4.1) — `[mem, mem + size())` is a valid range.

(4.2) — If the template parameter `Flags` is `vector_aligned_tag`, `mem` shall point to storage aligned by `memory_alignment_v<simd, U>`.

(4.3) — If the template parameter `Flags` is `overaligned_tag<N>`, `mem` shall point to storage aligned by `N`.

(4.4) — If the template parameter `Flags` is `element_aligned_tag`, `mem` shall point to storage aligned by `alignof(U)`.

5 *Effects:* Copies all `simd` elements as if `mem[`i`] = static_cast<U>(operator[](`i`))` for all i in the range of `[0, size())`.

6 *Remarks:* This function shall not participate in overload resolution unless

(6.1) — `is_simd_flag_type_v<Flags>` is `true`, and

(6.2) — `U` is a vectorizable type.

9.3.6 simd subscript operators [parallel.simd.subscr]

`reference operator[](size_t i);`

1 *Requires:* `i < size()`.

2 *Returns:* A `reference` (see 9.3.3) referring to the i^{th} element.

3 *Throws:* Nothing.

`value_type operator[](size_t i) const;`

4 *Requires:* `i < size()`.

5 *Returns:* The value of the i^{th} element.

6 *Throws:* Nothing.

9.3.7 simd unary operators [parallel.simd.unary]

1 Effects in this subclause are applied as unary element-wise operations.

`simd& operator++() noexcept;`

2 *Effects:* Increments every element by one.

3 *Returns:* `*this`.

`simd operator++(int) noexcept;`

4 *Effects:* Increments every element by one.

5 *Returns:* A copy of `*this` before incrementing.

`simd& operator--() noexcept;`

6 *Effects:* Decrements every element by one.

7 *Returns:* `*this`.

`simd operator--(int) noexcept;`

8 *Effects:* Decrements every element by one.

9 *Returns:* A copy of `*this` before decrementing.

`mask_type operator!() const noexcept;`

10 *Returns:* A `simd_mask` object with the i^{th} element set to `!operator[]`(i) for all i in the range of `[0, size())`.

`simd operator~() const noexcept;`

11 *Returns:* A `simd` object where each bit is the inverse of the corresponding bit in `*this`.

12 *Remarks:* This operator shall not participate in overload resolution unless `T` is an integral type.

`simd operator+() const noexcept;`

13 *Returns:* `*this`.

`simd operator-() const noexcept;`

14 *Returns:* A `simd` object where the i^{th} element is initialized to `-operator[]`(i) for all i in the range of `[0, size())`.

9.4 simd non-member operations [parallel.simd.nonmembers]

9.4.1 simd binary operators [parallel.simd.binary]

```
friend simd operator+(const simd& lhs, const simd& rhs) noexcept;
friend simd operator-(const simd& lhs, const simd& rhs) noexcept;
friend simd operator*(const simd& lhs, const simd& rhs) noexcept;
friend simd operator/(const simd& lhs, const simd& rhs) noexcept;
friend simd operator%(const simd& lhs, const simd& rhs) noexcept;
friend simd operator&(const simd& lhs, const simd& rhs) noexcept;
friend simd operator|(const simd& lhs, const simd& rhs) noexcept;
friend simd operator^(const simd& lhs, const simd& rhs) noexcept;
friend simd operator<<(const simd& lhs, const simd& rhs) noexcept;
friend simd operator>>(const simd& lhs, const simd& rhs) noexcept;
```

1 *Returns:* A `simd` object initialized with the results of applying the indicated operator to `lhs` and `rhs` as a binary element-wise operation.

2 *Remarks:* Each of these operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type `value_type`.

```
friend simd operator<<(const simd& v, int n) noexcept;
friend simd operator>>(const simd& v, int n) noexcept;
```

3 *Returns:* A `simd` object where the i^{th} element is initialized to the result of applying the indicated operator to `v[`i`]` and `n` for all i in the range of `[0, size())`.

4 *Remarks:* These operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type `value_type`.

9.4.2 simd compound assignment [parallel.simd.cassign]

```
friend simd& operator+=(simd& lhs, const simd& rhs) noexcept;
friend simd& operator-=(simd& lhs, const simd& rhs) noexcept;
friend simd& operator*=(simd& lhs, const simd& rhs) noexcept;
friend simd& operator/=(simd& lhs, const simd& rhs) noexcept;
friend simd& operator%=(simd& lhs, const simd& rhs) noexcept;
friend simd& operator&=(simd& lhs, const simd& rhs) noexcept;
friend simd& operator|=(simd& lhs, const simd& rhs) noexcept;
friend simd& operator^=(simd& lhs, const simd& rhs) noexcept;
friend simd& operator<<=(simd& lhs, const simd& rhs) noexcept;
friend simd& operator>>=(simd& lhs, const simd& rhs) noexcept;
```

1 *Effects:* These operators apply the indicated operator to `lhs` and `rhs` as an element-wise operation.

2 *Returns:* `lhs`.

3 *Remarks:* These operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type `value_type`.

```
friend simd& operator<<=(simd& lhs, int n) noexcept;
friend simd& operator>>=(simd& lhs, int n) noexcept;
```

4 *Effects:* Equivalent to: `return operator@=(lhs, simd(n));`

5 *Remarks:* These operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type `value_type`.

9.4.3 simd compare operators [parallel.simd.comparison]

```
friend mask_type operator==(const simd& lhs, const simd& rhs) noexcept;
friend mask_type operator!=(const simd& lhs, const simd& rhs) noexcept;
friend mask_type operator>=(const simd& lhs, const simd& rhs) noexcept;
friend mask_type operator<=(const simd& lhs, const simd& rhs) noexcept;
friend mask_type operator>(const simd& lhs, const simd& rhs) noexcept;
friend mask_type operator<(const simd& lhs, const simd& rhs) noexcept;
```

1 *Returns:* A `simd_mask` object initialized with the results of applying the indicated operator to `lhs` and `rhs` as a binary element-wise operation.

9.4.4 Reductions [parallel.simd.reductions]

1 In this subclause, `BinaryOperation` shall be a binary element-wise operation.

```
template<class T, class Abi, class BinaryOperation = plus<>>
  T reduce(const simd<T, Abi>& x, BinaryOperation binary_op = {});
```

2 *Requires:* `binary_op` shall be callable with two arguments of type `T` returning `T`, or callable with two arguments of type `simd<T, A1>` returning `simd<T, A1>` for every `A1` that is an ABI tag type.

3 *Returns:* *`GENERALIZED_SUM`*`(binary_op, x.data[`*i*`], ...)` for all *i* in the range of `[0, size())`(C++17 §29.2).

4 *Throws:* Any exception thrown from `binary_op`.

```
template<class M, class V, class BinaryOperation>
  typename V::value_type reduce(const const_where_expression<M, V>& x,
                                typename V::value_type identity_element,
                                BinaryOperation binary_op = {});
```

5 *Requires:* `binary_op` shall be callable with two arguments of type `T` returning `T`, or callable with two arguments of type `simd<T, A1>` returning `simd<T, A1>` for every `A1` that is an ABI tag type. The results of `binary_op(identity_element, x)` and `binary_op(x, identity_element)` shall be equal to `x` for all finite values `x` representable by `V::value_type`.

6 *Returns:* If `none_of(x.mask)`, returns `identity_element`. Otherwise, returns *`GENERALIZED_SUM`*`(binary_op, x.data[`*i*`], ...)` for all selected indices *i*.

7 *Throws:* Any exception thrown from `binary_op`.

```
template<class M, class V>
  typename V::value_type reduce(const const_where_expression<M, V>& x, plus<> binary_op) noexcept;
```

8 *Returns:* If `none_of(x.mask)`, returns `0`. Otherwise, returns *`GENERALIZED_SUM`*`(binary_op, x.data[`*i*`], ...)` for all selected indices *i*.

```
template<class M, class V>
  typename V::value_type reduce(const const_where_expression<M, V>& x, multiplies<> binary_op) noexcept;
```

9 *Returns:* If `none_of(x.mask)`, returns `1`. Otherwise, returns *`GENERALIZED_SUM`*`(binary_op, x.data[`*i*`], ...)` for all selected indices *i*.

```
template<class M, class V>
  typename V::value_type reduce(const const_where_expression<M, V>& x, bit_and<> binary_op) noexcept;
```

10 *Requires:* `is_integral_v<V::value_type>` is `true`.

11 *Returns:* If `none_of(x.mask)`, returns `~V::value_type()`. Otherwise, returns *`GENERALIZED_SUM`*`(binary_op, x.data[`*i*`], ...)` for all selected indices *i*.


```
template<class M, class V>
  typename V::value_type reduce(const const_where_expression<M, V>& x, bit_or<> binary_op) noexcept;
template<class M, class V>
  typename V::value_type reduce(const const_where_expression<M, V>& x, bit_xor<> binary_op) noexcept;
```

12 *Requires:* `is_integral_v<V::value_type>` is `true`.

13 *Returns:* If `none_of(x.mask)`, returns `0`. Otherwise, returns *GENERALIZED_SUM*`(binary_op, x.data[i], ...)` for all selected indices i.

```
template<class T, class Abi> T hmin(const simd<T, Abi>& x) noexcept;
```

14 *Returns:* The value of an element `x[j]` for which `x[j] <= x[i]` for all i in the range of `[0, size())`.

```
template<class M, class V> typename V::value_type hmin(const const_where_expression<M, V>& x) noexcept;
```

15 *Returns:* If `none_of(x.mask)`, the return value is `numeric_limits<V::value_type>::max()`. Otherwise, returns the value of an element `x.data[j]` for which `x.mask[j] == true` and `x.data[j] <= x.data[i]` for all selected indices i.

```
template<class T, class Abi> T hmax(const simd<T, Abi>& x) noexcept;
```

16 *Returns:* The value of an element `x[j]` for which `x[j] >= x[i]` for all i in the range of `[0, size())`.

```
template<class M, class V> typename V::value_type hmax(const const_where_expression<M, V>& x) noexcept;
```

17 *Returns:* If `none_of(x.mask)`, the return value is `numeric_limits<V::value_type>::lowest()`. Otherwise, returns the value of an element `x.data[j]` for which `x.mask[j] == true` and `x.data[j] >= x.data[i]` for all selected indices i.

9.4.5 Casts [parallel.simd.casts]

```
template<class T, class U, class Abi> see below simd_cast(const simd<U, Abi>& x) noexcept;
```

1 Let `To` denote `T::value_type` if `is_simd_v<T>` is `true`, or `T` otherwise.

2 *Returns:* A `simd` object with the i^{th} element initialized to `static_cast<To>(x[i])` for all i in the range of `[0, size())`.

3 *Remarks:* The function shall not participate in overload resolution unless

(3.1) — every possible value of type `U` can be represented with type `To`, and

(3.2) — either

(3.2.1) — `is_simd_v<T>` is `false`, or

(3.2.2) — `T::size() == simd<U, Abi>::size()` is `true`.

4 The return type is

(4.1) — `T` if `is_simd_v<T>` is `true`;

(4.2) — otherwise, `simd<T, Abi>` if `U` is the same type as `T`;

(4.3) — otherwise, `simd<T, simd_abi::fixed_size<simd<U, Abi>::size()>>`

```
template<class T, class U, class Abi> see below static_simd_cast(const simd<U, Abi>& x) noexcept;
```

5 Let `To` denote `T::value_type` if `is_simd_v<T>` is `true` or `T` otherwise.

6 *Returns:* A `simd` object with the i^{th} element initialized to `static_cast<To>(x[i])` for all i in the range of `[0, size())`.

7 *Remarks:* The function shall not participate in overload resolution unless either

(7.1) — `is_simd_v<T>` is `false`, or

(7.2) — `T::size() == simd<U, Abi>::size()` is `true`.

8 The return type is

(8.1) — `T` if `is_simd_v<T>` is `true`;

(8.2) — otherwise, `simd<T, Abi>` if either `U` is the same type as `T` or `make_signed_t<U>` is the same type as `make_signed_t<T>`;

(8.3) — otherwise, `simd<T, simd_abi::fixed_size<simd<U, Abi>::size()>>`.

```
template<class T, class Abi>
  fixed_size_simd<T, simd_size_v<T, Abi>> to_fixed_size(const simd<T, Abi>& x) noexcept;
template<class T, class Abi>
  fixed_size_simd_mask<T, simd_size_v<T, Abi>> to_fixed_size(const simd_mask<T, Abi>& x) noexcept;
```

9 *Returns:* A data-parallel object with the i^{th} element initialized to `x[`i`]` for all i in the range of `[0, size())`.

```
template<class T, int N> native_simd<T> to_native(const fixed_size_simd<T, N>& x) noexcept;
template<class T, int N> native_simd_mask<T> to_native(const fixed_size_simd_mask<T, N>& x) noexcept;
```

10 *Returns:* A data-parallel object with the i^{th} element initialized to `x[`i`]` for all i in the range of `[0, size())`.

11 *Remarks:* These functions shall not participate in overload resolution unless `simd_size_v<T, simd_abi::native<T>> == N` is `true`.

```
template<class T, int N> simd<T> to_compatible(const fixed_size_simd<T, N>& x) noexcept;
template<class T, int N> simd_mask<T> to_compatible(const fixed_size_simd_mask<T, N>& x) noexcept;
```

12 *Returns:* A data-parallel object with the i^{th} element initialized to `x[`i`]` for all i in the range of `[0, size())`.

13 *Remarks:* These functions shall not participate in overload resolution unless `simd_size_v<T, simd_abi::compatible<T>> == N` is `true`.

```
template<size_t... Sizes, class T, class Abi>
  tuple<simd<T, simd_abi::deduce_t<T, Sizes>>...>
    split(const simd<T, Abi>& x) noexcept;
template<size_t... Sizes, class T, class Abi>
  tuple<simd_mask<T, simd_abi::deduce_t<T, Sizes>>...>
    split(const simd_mask<T, Abi>& x) noexcept;
```

14 *Returns:* A `tuple` of data-parallel objects with the i^{th} `simd`/`simd_mask` element of the j^{th} `tuple` element initialized to the value of the element `x` with index i + sum of the first j values in the `Sizes` pack.

15 *Remarks:* These functions shall not participate in overload resolution unless the sum of all values in the `Sizes` pack is equal to `simd_size_v<T, Abi>`.

```
template<class V, class Abi>
  array<V, simd_size_v<typename V::value_type, Abi> / V::size()>
    split(const simd<typename V::value_type, Abi>& x) noexcept;
template<class V, class Abi>
  array<V, simd_size_v<typename V::simd_type::value_type, Abi> / V::size()>
    split(const simd_mask<typename V::simd_type::value_type, Abi>& x) noexcept;
```


16 *Returns:* An `array` of data-parallel objects with the i^{th} `simd`/`simd_mask` element of the j^{th} `array` element initialized to the value of the element in `x` with index `i + j * V::size()`.

17 *Remarks:* These functions shall not participate in overload resolution unless either:

(17.1) — `is_simd_v<V>` is `true` and `simd_size_v<typename V::value_type, Abi>` is an integral multiple of `V::size()`, or

(17.2) — `is_simd_mask_v<V>` is `true` and `simd_size_v<typename V::simd_type::value_type, Abi>` is an integral multiple of `V::size()`.

```
template<size_t N, class T, class A>
  array<resize_simd<simd_size_v<T, A> / N, simd<T, A>>, N>
    split_by(const simd<T, A>& x) noexcept;
template<size_t N, class T, class A>
  array<resize_simd<simd_size_v<T, A> / N, simd_mask<T, A>>, N>
    split_by(const simd_mask<T, A>& x) noexcept;
```

18 *Returns:* An array `arr`, where `arr[i][j]` is initialized by `x[i * (simd_size_v<T, A> / N) + j]`.

19 *Remarks:* The functions shall not participate in overload resolution unless `simd_size_v<T, A>` is an integral multiple of `N`.

```
template<class T, class... Abis>
  simd<T, simd_abi::deduce_t<T, (simd_size_v<T, Abis> + ...)>> concat(
    const simd<T, Abis>&... xs) noexcept;
template<class T, class... Abis>
  simd_mask<T, simd_abi::deduce_t<T, (simd_size_v<T, Abis> + ...)>> concat(
    const simd_mask<T, Abis>&... xs) noexcept;
```

20 *Returns:* A data-parallel object initialized with the concatenated values in the `xs` pack of data-parallel objects: The i^{th} `simd`/`simd_mask` element of the j^{th} parameter in the `xs` pack is copied to the return value's element with index i + the sum of the width of the first j parameters in the `xs` pack.

```
  template<class T, class Abi, size_t N>
    resize_simd<simd_size_v<T, Abi> * N, simd<T, Abi>>
      concat(const array<simd<T, Abi>, N>& arr) noexcept;
  template<class T, class Abi, size_t N>
    resize_simd<simd_size_v<T, Abi> * N, simd_mask<T, Abi>>
      concat(const array<simd_mask<T, Abi>, N>& arr) noexcept;
```

21 *Returns:* A data-parallel object, the i^{th} element of which is initialized by `arr[i / simd_size_v<T, Abi>][i % simd_size_v<T, Abi>]`.

9.4.6 Algorithms [parallel.simd.alg]

```
template<class T, class Abi> simd<T, Abi> min(const simd<T, Abi>& a, const simd<T, Abi>& b) noexcept;
```

1 *Returns:* The result of the element-wise application of `std::min(a[i], b[i])` for all i in the range of `[0, size())`.

```
template<class T, class Abi> simd<T, Abi> max(const simd<T, Abi>& a, const simd<T, Abi>& b) noexcept;
```

2 *Returns:* The result of the element-wise application of `std::max(a[i], b[i])` for all i in the range of `[0, size())`.

```
template<class T, class Abi>
  pair<simd<T, Abi>, simd<T, Abi>> minmax(const simd<T, Abi>& a, const simd<T, Abi>& b) noexcept;
```


3 *Returns:* A pair initialized with

(3.1) — the result of element-wise application of std::min(a[*i*], b[*i*]) for all i in the range of [0, size()) in the first member, and

(3.2) — the result of element-wise application of std::max(a[*i*], b[*i*]) for all i in the range of [0, size()) in the second member.

```
template<class T, class Abi> simd<T, Abi>
  clamp(const simd<T, Abi>& v, const simd<T, Abi>& lo, const simd<T, Abi>& hi);
```

4 *Requires:* No element in lo shall be greater than the corresponding element in hi.

5 *Returns:* The result of element-wise application of std::clamp(v[*i*], lo[*i*], hi[*i*]) for all i in the range of [0, size()).

9.4.7 Math library [parallel.simd.math]

1 For each set of overloaded functions within <cmath>, there shall be additional overloads sufficient to ensure that if any argument corresponding to a double parameter has type simd<T, Abi>, where is_floating_point_v<T> is true, then:

(1.1) — All arguments corresponding to double parameters shall be convertible to simd<T, Abi>.

(1.2) — All arguments corresponding to double* parameters shall be of type simd<T, Abi>*.

(1.3) — All arguments corresponding to parameters of integral type U shall be convertible to fixed_size_simd<U, simd_size_v<T, Abi>>.

(1.4) — All arguments corresponding to U*, where U is integral, shall be of type fixed_size_simd<U, simd_size_v<T, Abi>>*.

(1.5) — If the corresponding return type is double, the return type of the additional overloads is simd<T, Abi>. Otherwise, if the corresponding return type is bool, the return type of the additional overload is simd_mask<T, Abi>. Otherwise, the return type is fixed_size_simd<R, simd_size_v<T, Abi>>, with R denoting the corresponding return type.

It is unspecified whether a call to these overloads with arguments that are all convertible to simd<T, Abi> but are not of type simd<T, Abi> is well-formed.

2 Each function overload produced by the above rules applies the indicated <cmath> function element-wise. For the mathematical functions, the results per element only need to be approximately equal to the application of the function which is overloaded for the element type.

3 The behavior is undefined if a domain, pole, or range error occurs when the input argument(s) are applied to the indicated <cmath> function.

4 If abs is called with an argument of type simd<X, Abi> for which is_unsigned_v<X> is true, the program is ill-formed.

9.5 Class template simd_mask [parallel.simd.mask.class]

9.5.1 Class template simd_mask overview [parallel.simd.mask.overview]

```
template<class T, class Abi> class simd_mask {
public:
  using value_type = bool;
  using reference = see below;
  using simd_type = simd<T, Abi>;
```



```
    using abi_type = Abi;

    static constexpr size_t size() noexcept;

    simd_mask() noexcept = default;

    // 9.5.3, Constructors
    explicit simd_mask(value_type) noexcept;
    template<class U>
      simd_mask(const simd_mask<U, simd_abi::fixed_size<size()>>&) noexcept;
    template<class Flags> simd_mask(const value_type* mem, Flags);

    // 9.5.4, Copy functions
    template<class Flags> void copy_from(const value_type* mem, Flags);
    template<class Flags> void copy_to(value_type* mem, Flags);

    // 9.5.5, Subscript operators
    reference operator[](size_t);
    value_type operator[](size_t) const;

    // 9.5.6, Unary operators
    simd_mask operator!() const noexcept;

    // 9.6.1, Binary operators
    friend simd_mask operator&&(const simd_mask&, const simd_mask&) noexcept;
    friend simd_mask operator||(const simd_mask&, const simd_mask&) noexcept;
    friend simd_mask operator&(const simd_mask&, const simd_mask&) noexcept;
    friend simd_mask operator|(const simd_mask&, const simd_mask&) noexcept;
    friend simd_mask operator^(const simd_mask&, const simd_mask&) noexcept;

    // 9.6.2, Compound assignment
    friend simd_mask& operator&=(simd_mask&, const simd_mask&) noexcept;
    friend simd_mask& operator|=(simd_mask&, const simd_mask&) noexcept;
    friend simd_mask& operator^=(simd_mask&, const simd_mask&) noexcept;

    // 9.6.3, Comparisons
    friend simd_mask operator==(const simd_mask&, const simd_mask&) noexcept;
    friend simd_mask operator!=(const simd_mask&, const simd_mask&) noexcept;
  };
```

1 The class template `simd_mask` is a data-parallel type with the element type `bool`. The width of a given `simd_mask` specialization is a constant expression, determined by the template parameters. Specifically, `simd_mask<T, Abi>::size() == simd<T, Abi>::size()`.

2 Every specialization of `simd_mask` shall be a complete type. The specialization `simd_mask<T, Abi>` is supported if `T` is a vectorizable type and

(2.1) — `Abi` is `simd_abi::scalar`, or

(2.2) — `Abi` is `simd_abi::fixed_size<N>`, with `N` constrained as defined in (9.2.1).

If `Abi` is an extended ABI tag, it is implementation-defined whether `simd_mask<T, Abi>` is supported. [*Note:* The intent is for implementations to decide on the basis of the currently targeted system. — *end note*]

If `simd_mask<T, Abi>` is not supported, the specialization shall have a deleted default constructor, deleted destructor, deleted copy constructor, and deleted copy assignment. Otherwise, the following are true:

(2.3) — `is_nothrow_move_constructible_v<simd_mask<T, Abi>>`, and

(2.4) — `is_nothrow_move_assignable_v<simd_mask<T, Abi>>`, and

(2.5) — `is_nothrow_default_constructible_v<simd_mask<T, Abi>>`.

3 Default initialization performs no intialization of the elements; value-initialization initializes each element with `false`. [*Note:* Thus, default initialization leaves the elements in an indeterminate state. — *end note*]

4 Implementations should enable explicit conversion from and to implementation-defined types. This adds one or more of the following declarations to class `simd_mask`:

```
explicit operator implementation-defined() const;
explicit simd_mask(const implementation-defined& init) const;
```

5 The member type `reference` has the same interface as `simd<T, Abi>::reference`, except its `value_type` is `bool`. (9.3.3)

9.5.2 `simd_mask` width [parallel.simd.mask.width]

```
static constexpr size_t size() noexcept;
```

1 *Returns:* The width of `simd<T, Abi>`.

9.5.3 Constructors [parallel.simd.mask.ctor]

```
explicit simd_mask(value_type x) noexcept;
```

1 *Effects:* Constructs an object with each element initialized to `x`.

```
template<class U> simd_mask(const simd_mask<U, simd_abi::fixed_size<size()>>& x) noexcept;
```

2 *Effects:* Constructs an object of type `simd_mask` where the i^{th} element equals `x[`i`]` for all i in the range of `[0, size())`.

3 *Remarks:* This constructor shall not participate in overload resolution unless `abi_type` is `simd_abi::fixed_size<size()>`.

```
template<class Flags> simd_mask(const value_type* mem, Flags);
```

4 *Requires:*

(4.1) — `[mem, mem + size())` is a valid range.

(4.2) — If the template parameter `Flags` is `vector_aligned_tag`, `mem` shall point to storage aligned by `memory_alignment_v<simd_mask>`.

(4.3) — If the template parameter `Flags` is `overaligned_tag<N>`, `mem` shall point to storage aligned by `N`.

(4.4) — If the template parameter `Flags` is `element_aligned_tag`, `mem` shall point to storage aligned by `alignof(value_type)`.

5 *Effects:* Constructs an object where the i^{th} element is initialized to `mem[`i`]` for all i in the range of `[0, size())`.

6 *Throws:* Nothing.

7 *Remarks:* This constructor shall not participate in overload resolution unless `is_simd_flag_type_v<Flags>` is `true`.

9.5.4 Copy functions [parallel.simd.mask.copy]

```
template<class Flags> void copy_from(const value_type* mem, Flags);
```

1 *Requires:*

(1.1) — `[mem, mem + size())` is a valid range.

(1.2) — If the template parameter `Flags` is `vector_aligned_tag`, `mem` shall point to storage aligned by `memory_alignment_v<simd_mask>`.

(1.3) — If the template parameter `Flags` is `overaligned_tag<N>`, `mem` shall point to storage aligned by `N`.

(1.4) — If the template parameter `Flags` is `element_aligned_tag`, `mem` shall point to storage aligned by `alignof(value_type)`.

2 *Effects:* Replaces the elements of the `simd_mask` object such that the i^{th} element is replaced with `mem[i]` for all i in the range of `[0, size())`.

3 *Throws:* Nothing.

4 *Remarks:* This function shall not participate in overload resolution unless `is_simd_flag_type_v<Flags>` is `true`.

```
template<class Flags> void copy_to(value_type* mem, Flags);
```

5 *Requires:*

(5.1) — `[mem, mem + size())` is a valid range.

(5.2) — If the template parameter `Flags` is `vector_aligned_tag`, `mem` shall point to storage aligned by `memory_alignment_v<simd_mask>`.

(5.3) — If the template parameter `Flags` is `overaligned_tag<N>`, `mem` shall point to storage aligned by `N`.

(5.4) — If the template parameter `Flags` is `element_aligned_tag`, `mem` shall point to storage aligned by `alignof(value_type)`.

6 *Effects:* Copies all `simd_mask` elements as if `mem[i] = operator[](i)` for all i in the range of `[0, size())`.

7 *Throws:* Nothing.

8 *Remarks:* This function shall not participate in overload resolution unless `is_simd_flag_type_v<Flags>` is `true`.

9.5.5 Subscript operators [parallel.simd.mask.subscr]

```
reference operator[](size_t i);
```

1 *Requires:* `i < size()`.

2 *Returns:* A `reference` (see 9.3.3) referring to the i^{th} element.

3 *Throws:* Nothing.

```
value_type operator[](size_t i) const;
```

4 *Requires:* `i < size()`.

5 *Returns:* The value of the i^{th} element.

6 *Throws:* Nothing.

9.5.6 Unary operators [parallel.simd.mask.unary]

```
simd_mask operator!() const noexcept;
```

1 *Returns:* The result of the element-wise appliation of `operator!`.

9.6 Non-member operations [parallel.simd.mask.nonmembers]

9.6.1 Binary operators [parallel.simd.mask.binary]

```
friend simd_mask operator&&(const simd_mask& lhs, const simd_mask& rhs) noexcept;
friend simd_mask operator||(const simd_mask& lhs, const simd_mask& rhs) noexcept;
friend simd_mask operator& (const simd_mask& lhs, const simd_mask& rhs) noexcept;
friend simd_mask operator| (const simd_mask& lhs, const simd_mask& rhs) noexcept;
friend simd_mask operator^ (const simd_mask& lhs, const simd_mask& rhs) noexcept;
```

1 *Returns:* A `simd_mask` object initialized with the results of applying the indicated operator to `lhs` and `rhs` as a binary element-wise operation.

9.6.2 Compound assignment [parallel.simd.mask.cassign]

```
friend simd_mask& operator&=(simd_mask& lhs, const simd_mask& rhs) noexcept;
friend simd_mask& operator|=(simd_mask& lhs, const simd_mask& rhs) noexcept;
friend simd_mask& operator^=(simd_mask& lhs, const simd_mask& rhs) noexcept;
```

1 *Effects:* These operators apply the indicated operator to `lhs` and `rhs` as a binary element-wise operation.

2 *Returns:* `lhs`.

9.6.3 Comparisons [parallel.simd.mask.comparison]

```
friend simd_mask operator==(const simd_mask&, const simd_mask&) noexcept;
friend simd_mask operator!=(const simd_mask&, const simd_mask&) noexcept;
```

1 *Returns:* A `simd_mask` object initialized with the results of applying the indicated operator to `lhs` and `rhs` as a binary element-wise operation.

9.6.4 Reductions [parallel.simd.mask.reductions]

```
template<class T, class Abi> bool all_of(const simd_mask<T, Abi>& k) noexcept;
```

1 *Returns:* `true` if all boolean elements in `k` are `true`, `false` otherwise.

```
template<class T, class Abi> bool any_of(const simd_mask<T, Abi>& k) noexcept;
```

2 *Returns:* `true` if at least one boolean element in `k` is `true`, `false` otherwise.

```
template<class T, class Abi> bool none_of(const simd_mask<T, Abi>& k) noexcept;
```

3 *Returns:* `true` if none of the one boolean elements in `k` is `true`, `false` otherwise.

```
template<class T, class Abi> bool some_of(const simd_mask<T, Abi>& k) noexcept;
```

4 *Returns:* `true` if at least one of the one boolean elements in `k` is `true` and at least one of the boolean elements in `k` is `false`, `false` otherwise.

```
template<class T, class Abi> int popcount(const simd_mask<T, Abi>& k) noexcept;
```

5 *Returns:* The number of boolean elements in `k` that are `true`.

```
template<class T, class Abi> int find_first_set(const simd_mask<T, Abi>& k);
```

6 *Requires:* `any_of(k)` returns `true`.

7 *Returns:* The lowest element index i where `k[`i`]` is `true`.

8 *Throws:* Nothing.

```
template<class T, class Abi> int find_last_set(const simd_mask<T, Abi>& k);
```

9 *Requires:* `any_of(k)` returns `true`.

10 *Returns:* The greatest element index i where `k[`i`]` is `true`.

11 *Throws:* Nothing.

```
bool all_of(T) noexcept;
bool any_of(T) noexcept;
bool none_of(T) noexcept;
bool some_of(T) noexcept;
int popcount(T) noexcept;
```

12 *Returns:* `all_of` and `any_of` return their arguments; `none_of` returns the negation of its argument; `some_of` returns `false`; `popcount` returns the integral representation of its argument.

13 *Remarks:* The parameter type `T` is an unspecified type that is only constructible via implicit conversion from `bool`.

```
int find_first_set(T);
int find_last_set(T);
```

14 *Requires:* The value of the argument is `true`.

15 *Returns:* `0`.

16 *Throws:* Nothing.

17 *Remarks:* The parameter type `T` is an unspecified type that is only constructible via implicit conversion from `bool`.

9.6.5 `where` functions [parallel.simd.mask.where]

```
template<class T, class Abi>
  where_expression<simd_mask<T, Abi>, simd<T, Abi>>
    where(const typename simd<T, Abi>::mask_type& k, simd<T, Abi>& v) noexcept;
template<class T, class Abi>
  const_where_expression<simd_mask<T, Abi>, simd<T, Abi>>
    where(const typename simd<T, Abi>::mask_type& k, const simd<T, Abi>& v) noexcept;
template<class T, class Abi>
  where_expression<simd_mask<T, Abi>, simd_mask<T, Abi>>
    where(const type_identity_t<simd_mask<T, Abi>>& k, simd_mask<T, Abi>& v) noexcept;
template<class T, class Abi>
  const_where_expression<simd_mask<T, Abi>, simd_mask<T, Abi>>
    where(const type_identity_t<simd_mask<T, Abi>>& k, const simd_mask<T, Abi>& v) noexcept;
```

1 *Returns:* An object (9.2.3) with `mask` and `data` initialized with `k` and `v` respectively.

```
template<class T>
  where_expression<bool T>
    where(see below k, T& v) noexcept;
template<class T>
  const_where_expression<bool, T>
    where(see below k, const T& v) noexcept;
```


2 *Remarks:* The functions shall not participate in overload resolution unless

(2.1) — `T` is neither a `simd` nor a `simd_mask` specialization, and

(2.2) — the first argument is of type `bool`.

3 *Returns:* An object (9.2.3) with `mask` and `data` initialized with `k` and `v` respectively.

