ABSTRACT

This paper describes class templates for portable data-parallel (e.g. SIMD) programming via vector types.

CONTENTS

0 Remarks 1
1 Changelog 1
2 Straw Polls 8
3 Introduction 11
4 Examples 13
5 Naming 13
6 Wording 23
7 Widening Cast 52
8 Split & Concat 54
9 Discussion 56
10 Feature Detection Macros 65
A Acknowledgements 65
B Bibliography 66
0 Remarks

- This document talks about “vector” types/objects. In general this will not refer to the `std::vector` class template. References to the container type will explicitly call out the `std` prefix to avoid confusion.

- In the following, \( W_T \) denotes the number of scalar values (width) in a vector of type \( T \) (sometimes also called the number of SIMD lanes).


- This paper is not supposed to specify a complete API for data-parallel types and operations. It is meant as a useful starting point. Once the foundation is settled on, higher level APIs will be proposed.

1 Changelog

1.1 Changes from r0

Previous revision: [P0214R0].

- Extended the `datapar_abi` tag types with a `fixed_size<N>` tag to handle arbitrarily sized vectors (6.1.1.1).

- Converted `memory_alignment` into a non-member trait (6.1.1.2).

- Extended implicit conversions to handle `datapar_abi::fixed_size<N>` (6.1.2.2).

- Extended binary operators to convert correctly with `datapar_abi::fixed_size<N>` (6.1.3.1).

- Dropped the section on “datapar logical operators”. Added a note that the omission is deliberate (??).

- Added logical and bitwise operators to `mask` (6.1.5.1).

- Modified `mask` compares to work better with implicit conversions (6.1.5.3).

- Modified `where` to support different Abi tags on the `mask` and `datapar` arguments (6.1.5.5).
• Converted the load functions to non-member functions. SG1 asked for guidance from LEWG whether a load-expression or a template parameter to load is more appropriate.

• Converted the store functions to non-member functions to be consistent with the load functions.

• Added a note about masked stores not invoking out-of-bounds accesses for masked-off elements of the vector.

• Converted the return type of `datapar::operator[]` to return a smart reference instead of an lvalue reference.

• Modified the wording of `mask::operator[]` to match the reference type returned from `datapar::operator[]`.

• Added non-trig/pow/exp/log math functions on `datapar`.

• Added discussion on defaulting load/store flags.

• Added sum, product, min, and max reductions for `datapar`.

• Added load constructor.

• Modified the wording of `native_handle()` to make the existence of the functions implementation-defined, instead of only the return type. Added a section in the discussion (cf. Section 9.8).

• Fixed missing flag objects.

1.2 changes from r1

Previous revision: [P0214R1].

• Fixed converting constructor synopsis of `datapar` and `mask` to also allow varying Abi types.

• Modified the wording of `mask::native_handle()` to make the existence of the functions implementation-defined.

• Updated the discussion of member types to reflect the changes in R1.

• Added all previous SG1 straw poll results.

• Fixed `commonabi` to not invent native Abi that makes the operator ill-formed.
• Dropped table of math functions.
• Be more explicit about the implementation-defined Abi types.
• Discussed resolution of the `datapar_abi::fixed_size<N>` design (9.7.4).
• Made the compatible and native ABI aliases depend on `T` (6.1.1.1).
• Added `max_fixed_size` constant (6.1.1.1 p.4).
• Added masked loads.
• Added rationale for return type of `datapar::operator-()` (9.10).

— SG1 guidance:
• Dropped the default load / store flags.
• Renamed the (un)aligned flags to `elementAligned` and `vectorAligned`.
• Added an `overAligned<N>` load / store flag.
• Dropped the ampersand on `nativeHandle` (no strong preference).
• Completed the set of math functions (i.e. add trig, log, and exp).

— LEWG (small group) guidance:
• Dropped `nativeHandle` and add non-normative wording for supporting `static_cast` to implementation-defined SIMD extensions.
• Dropped non-member load and store functions. Instead have `copy_from` and `copy_to` member functions for loads and stores. (6.1.2.3, 6.1.2.4, 6.1.4.3, 6.1.4.4)
  (Did not use the load and store names because of the unfortunate inconsistency with `std::atomic`.)
• Added algorithm overloads for `datapar` reductions. Integrate with `where` to enable masked reductions. (6.1.3.4) This made it necessary to spell out the class `where_expression`. 
1 Changelog

1.3 Changes from r2

Previous revision: [P0214R2].

- Fixed return type of masked reduce (6.1.3.4).
- Added binary min, max, minmax, and clamp (6.1.3.6).
- Moved member min and max to non-member hmin and hmax, which cannot easily be optimized from reduce, since no function object such as std::plus exists (6.1.3.4).
- Fixed neutral element of masked hmin/hmax and drop UB (6.1.3.4).
- Removed remaining reduction member functions in favor of non-member reduce (as requested by LEWG).
- Replaced init parameter of masked reduce with neutral_element (6.1.3.4).
- Extend where_expression to support const datapar objects (6.1.5.5).
- Fixed missing explicit keyword on mask(bool) constructor (6.1.4.2).
- Made binary operators for datapar and mask friend functions of datapar and mask, simplifying the SFINAE requirements considerably (6.1.3.1, 6.1.5.1).
- Restricted broadcasts to only allow non-narrowing conversions (6.1.2.2).
- Restricted datapar to datapar conversions to only allow non-narrowing conversions with fixed_size ABI (6.1.2.2).
- Added generator constructor (as discussed in LEWG in Issaquah) (6.1.2.2).
- Renamed copy_from to memload and copy_to to memstore. (6.1.2.3, 6.1.2.4, 6.1.4.3, 6.1.4.4)
- Documented effect of overaligned_tag<N> as Flags parameter to load/store. (6.1.2.3, 6.1.2.4, 6.1.4.3, 6.1.4.4)
- Clarified cv requirements on T parameter of datapar and mask.
- Allowed all implicit mask conversions with fixed_size ABI and equal size (6.1.4.2).
- Made increment and decrement of where_expression return void.
- Added static_datapar_cast for simple casts (6.1.3.5).
• Clarified default constructor (6.1.2.1, 6.1.2.1).

• Clarified `datapar` and `mask` with invalid template parameters to be complete types with deleted constructors, destructor, and assignment (6.1.2.1, 6.1.2.1).

• Wrote a new subsection for a detailed description of `where_expression` (6.1.1.3).

• Moved masked loads and stores from `datapar` and `mask` to `where_expression` (6.1.1.3). This required two more overloads of `where` to support value objects of type `mask` (6.1.5.5).

• Removed `where_expression::operator!` (6.1.1.3).

• Added aliases `native_datapar`, `native_mask`, `fixed_size_datapar`, `fixed_size_mask` (6.1.1).

• Removed bool overloads of mask reductions awaiting a better solution (6.1.5.4).

• Removed special math functions with f and l suffix and l and ll prefix (6.1.3.7).

• Modified special math functions with mixed types to use `fixed_size` instead of `abi_for_size` (6.1.3.7).

• Added simple ABI cast functions `to_fixed_size`, `to_native`, and `to_compatible` (6.1.3.5).

1.4 changes from r3

Previous revision: [P0214R3].

• Add special math overloads for signed char and short. They are important to avoid widening to multiple SIMD registers and since no integer promotion is applied for `datapar` types.

• Editorial: Prefer using `over` typedef.

• Overload shift operators with `int` argument for the right hand side. This enables more efficient implementations. This signature is present in the Vc library, and was forgotten in the wording.

• Remove empty section about the omission of logical operators.
• Modify mask compares to return a mask instead of bool (6.1.5.3). This resolves an inconsistency with all the other binary operators.

• Editorial: Improve reference member specification (6.1.2.1).

• Require swap(v[0], v[1]) to be valid (6.1.2.1).

• Fixed inconsistency of masked load constructor after move of memload to where_expression (6.1.1.3).

• Editorial: Use Requires clause instead of Remarks to require the memory argument to loads and stores to be large enough (6.1.1.3, 6.1.2.3, 6.1.2.4, 6.1.4.3, 6.1.4.4).

• Add a note to special math functions that precondition violation is UB (6.1.3.7).

• Bugfix: Binary operators involving two datapar::reference objects must work (6.1.2.1).

• Editorial: Replace Note clauses in favor of [ Note: — end note ].

• Editorial: Replace UB Remarks on load/store alignment requirements with Requires clauses.

• Add an example section (4).

— design related:

• Readd bool overloads of mask reductions and ensure that implicit conversions to bool are ill-formed.

• Clarify effects of using an ABI parameter that is not available on the target (6.1.2.1 p.2, 6.1.4.1 p.2, 6.1.1.2 p.6).

• Split where_expression into const and non-const class templates.

• Add section on naming (Section 5).

• Discuss the questions/issues raised on max_fixed_size in Kona (Section 9.11).

• Make max_fixed_size dependent on T.

• Clarify that converting loads and stores only work with arrays of non-bool arithmetic type (6.1.2.3, 6.1.2.4).
• Discuss mask and bitset reduction interface differences (Section 5.5).

• Relax requirements on return type of generator function for the generator constructor (6.1.2.2).

• Remove overly generic datapar_cast function.

• Add proposal for a widening cast function (Section 7).

• Add proposal for split and concat cast functions (Section 8).

• Add noexcept or “Throws: Nothing.” to most functions.

— wording fixes & improvements:

• Remove non-normative noise about ABI tag types (6.1.1.1).

• Remove most of the text about vendor-extensions for ABI tag types, since it’s QoI (6.1.1.1).

• Clarify the differences and intent of compatible<T> vs. native<T> (6.1.1.1).

• Move definition of where_expression out of the synopsis (6.1.1.3).

• Editorial: Improve is_datapar and is_mask wording (6.1.1.2).

• Make ABI tag a consistent term and add is_abi_tag trait (6.1.1.2, 6.1.1.1).

• Clarify that datapar_abi::fixed_size<N> must support all N matching all possible implementation-defined ABI tags (6.1.1.1).

• Clarify abi_for_size wording (6.1.1.2).

• Turn memory_alignment into a trait with a corresponding memory_alignment_v variable template.

• Clarify memory_alignment wording; when it has no value member; and imply its value through a reference to the load and store functions (6.1.1.2).

• Remove exposition-only where_expression constructor and make exposition-only data members private (6.1.1.3).

• Editorial: use “shall not participate in overload resolution unless” consistently.

• Add a note about variability of max_fixed_size (6.1.1.1).
2 Straw Polls

• Editorial: use “target architecture” and “currently targeted system” consistently.

• Add margin notes presenting a wording alternative that avoids “target system” and “target architecture” in normative text.

• Specify result of masked reduce with empty mask (6.1.3.4).

• Editorial: clean up the use of “supported” and resolve contradictions resulting from incorrect use of conventions in the rest of the standard text (6.1.2.1 p.2, 6.1.4.1 p.2, 6.1.1.2).

• Add Section 10 Feature Detection Macros.

2 Straw Polls

2.1 sg1 at chicago 2013

Poll: Pursue SIMD/data parallel programming via types?

\[
\begin{array}{c|c|c|c|c}
SF & F & N & A & SA \\
1 & 8 & 5 & 0 & 0 \\
\end{array}
\]

2.2 sg1 at urbana 2014

• Poll: SF = ABI via namespace, SA = ABI as template parameter

\[
\begin{array}{c|c|c|c|c}
SF & F & N & A & SA \\
0 & 0 & 6 & 11 & 2 \\
\end{array}
\]

• Poll: Apply size promotion to vector operations? SF = \text{shortv} + \text{shortv} = \text{intv}

\[
\begin{array}{c|c|c|c|c}
SF & F & N & A & SA \\
1 & 2 & 0 & 6 & 11 \\
\end{array}
\]

• Poll: Apply “sign promotion” to vector operations? SF = \text{ushortv} + \text{shortv} = \text{ushortv}; SA = no mixed signed/unsigned arithmetic

\[
\begin{array}{c|c|c|c|c}
SF & F & N & A & SA \\
1 & 5 & 5 & 7 & 2 \\
\end{array}
\]
### 2.3 Poll: Make vector types ready for LEWG with arithmetic, compares, write-masking, and math?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### 2.4 Poll: Should subscript operator return an lvalue reference?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>10</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Poll: Should subscript operator return a “smart reference”?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Poll: Specify datapar width using ABI tag, with a special template tag for fixed size.

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Poll: Specify datapar width using <T, N, abi>, where abi is not specified by the user.

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

### 2.5 Poll: Keep native_handle in the wording (dropping the ampersand in the return type)?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Poll: Should the interface provide a way to specify a number for over-alignment?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
• Poll: Should loads and stores have a default load/store flag?

<table>
<thead>
<tr>
<th></th>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

2.6 LEWG AT ISSAQUAH 2016

• Poll: Unary minus on unsigned datapar should be ill formed

<table>
<thead>
<tr>
<th></th>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

• Poll: Reductions only as free functions
  → unanimous consent

• Poll: Jens should work with the author and return with an updated paper
  → unanimous consent

2.7 LEWG AT KONA 2017

• Poll: Want operator<<(signed) to work except where it’s undefined for the underlying integer?
  → unanimous consent

• Poll: Should there be overloads for where and (mask and datapar) reductions with bool/builtin types in place of mask/datapar?

<table>
<thead>
<tr>
<th></th>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

• Poll: Should there be a named “widen” function that widens the element type T from (e.g.) int to long, but rejects int to short? The number of elements is not changed.
  → unanimous consent
  cf. Section 7

• Poll: Should there be a named “concat” functions that concatenates several datapars with the same element type, but potentially different length?
The latter two operations are currently lumped together in datapar_cast. The widen / concat questions, I think, comes down to whether we want the current datapar_cast or not.
  → unanimous consent
  cf. Section 8
• Poll: We still have a few open ends on implicit conversions (https://github.com/mattkretz/wg21-papers/issues/26, https://github.com/mattkretz/wg21-papers/issues/3). The current paper is rather strict, there may be room for more implicit conversions without introducing safety problems. Should we postpone any changes in this area to after TS feedback?
→ unanimous consent

• Poll: Keep or drop datapar_abi::max_fixed_size? (cf. https://github.com/mattkretz/wg21-papers/issues/38)
   Authors will come back with a proposal
   cf. Section 9.11

• Poll: Add a list of possible names for “datapar” and “where” to the paper.
→ unanimous consent
   cf. Section 5

3 INTRODUCTION

3.1 SIMD registers and operations

Since many years the number of SIMD instructions and the size of SIMD registers have been growing. Newer microarchitectures introduce new operations for optimizing certain (common or specialized) operations. Additionally, the size of SIMD registers has increased and may increase further in the future.

The typical minimal set of SIMD instructions for a given scalar data type comes down to the following:

• Load instructions: load $\mathcal{W}_T$ successive scalar values starting from a given address into a SIMD register.

• Store instructions: store from a SIMD register to $\mathcal{W}_T$ successive scalar values at a given address.

• Arithmetic instructions: apply the arithmetic operation to each pair of scalar values in the two SIMD registers and store the results back to a SIMD register.

• Compare instructions: apply the compare operation to each pair of scalar values in the two SIMD registers and store the results back to a SIMD mask register.

• Bitwise instructions: bitwise operations on SIMD registers.
• Shuffle instructions: permutation and/or blending of scalars in (a) SIMD register(s).

The set of available instructions may differ considerably between different microarchitectures of the same CPU family. Furthermore there are different SIMD register sizes. Future extensions will certainly add more instructions and larger SIMD registers.

3.2 motivation for data-parallel types

SIMD registers and operations are the low-level ingredients to efficient programming for SIMD CPUs. At a more abstract level this is not only about SIMD CPUs, but efficient data-parallel execution (CPUs, GPUs, possibly FPGAs and classical vector supercomputers). Operations on fundamental types in C++ form the abstraction for CPU registers and instructions. Thus, a data-parallel type (SIMD type) can provide the necessary interface for writing software that can utilize data-parallel hardware efficiently. Higher-level abstractions can be built on top of these types. Note that if a low-level access to SIMD is not provided, users of C++ are either constrained to work within the limits of the provided abstraction or resort to non-portable extensions, such as SIMD intrinsics.

In some cases the compiler might generate better code if only the intent is stated instead of an exact sequence of operations. Therefore, higher-level abstractions might seem preferable to low-level SIMD types. In my experience this is a non-issue because programming with SIMD types makes intent very clear and compilers can optimize sequences of SIMD operations just like they can for scalar operations. SIMD types do not lead to an easy and obvious answer for efficient and easily usable data structures, though. But, in contrast to vector loops, SIMD types make unsuitable data structures glaringly obvious and can significantly support the developer in creating more suitable data layouts.

One major benefit from SIMD types is that the programmer can gain an intuition for SIMD. This subsequently influences further design of data structures and algorithms to better suit SIMD architectures.

There are already many users of SIMD intrinsics (and thus a primitive form of SIMD types). Providing a cleaner and portable SIMD API would provide many of them with a better alternative. Thus, SIMD types in C++ would capture and improve on widespread existing practice.

The challenge remains in providing portable SIMD types and operations.
C++ has no means to use SIMD operations directly. There are indirect uses through automatic loop vectorization or optimized algorithms (that use extensions to C/C++ or assembly for their implementation).

All compiler vendors (that I worked with) add intrinsics support to their compiler products to make SIMD operations accessible from C. These intrinsics are inherently not portable and most of the time very directly bound to a specific instruction. (Compilers are able to statically evaluate and optimize SIMD code written via intrinsics, though.)

4 EXAMPLES

4.1 LOOP VECTORIZATION

This shows a low-level approach of manual loop chunking + epilogue for vectorization (“Leave no room for a lower-level language below C++ (except assembler).” [2]). It also shows SIMD loads, operations, write-masking (blending), and stores.

```c
using floatv = native_datapar<float>;
void f() {
    alignas(memory_alignment_v<floatv>) float data[N];
    fill_data(data);
    size_t i = 0;
    for (; i + floatv::size() <= N; i += floatv::size()) {
        floatv v(&data[i], flags::vector_aligned);
        where(v > 100.f, v) = 100.f + (v - 100.f) * 0.1f;
        v.memstore(&data[i], flags::vector_aligned);
    }
    for (; i < N; ++i) {
        float x = data[i];
        if (x > 100.f) {
            x = 100.f + (x - 100.f) * 0.1f;
        }
        data[i] = x;
    }
}
```

5 NAMING

The name datapar was chosen in SG1 after a short discussion, brainstorm session, and straw poll. The following will present naming ideas and a bit of discussion of pros and cons and make recommendations.
The class in question is an array of target-specific size, with elements of type T, and data parallel operation semantics. The actual memory layout and storage size is unspecified. The number of elements is influenced via the second template parameter. If the second template parameter is \texttt{fixed\_size<N>}, an exact number of N elements is used. Operations on objects of the type execute the operation component-wise and concurrently. This allows the user to communicate data parallelism inherent in the problem at hand. An implementation might translate the data parallelism into SIMD instructions, GPU parallelism, serial execution, synchronized multi-core execution, or any mix thereof. The implementation is expected to provide guarantees about the resulting code gen depending on compiler flags and the given ABI parameter (second template parameter), e.g. \texttt{datapar\_<int, datapar\_abi::sse>} uses \texttt{xmm} registers for storage and all ISA extensions enabled via compiler flags.

### Naming Options

- \texttt{vector<T>}
- \texttt{vec<T>}
- \texttt{vecpar<T>}
- \texttt{simd<T>}
- \texttt{datapar<T>}
- \texttt{pack<T>}
- \texttt{simdarray<T> / simdvector<T> / vecarray<T>}
- \texttt{vectorize<T> / simdize<T> / vectize<T> / vectorized<T> / simdized<T> / vectized<T>}

### Discussion

- \texttt{vector<T>}
  
  **PRO** 1. term-of-art in the industry. We talk about “vectorization”, “vector unit”, “vector registers”, ...
  
  2. does work as a mathematical vector, e.g. \texttt{std::reduce\langle std::plus\rangle(x*y)} is the scalar product
1. **[name collision]** `std::vector`: the name is taken. Using a different namespace won’t help: Too much confusion/conflict with `std::vector`, which is not constant-size.

2. ambiguity with mathematically inclined people who may expect operators to behave differently (e.g. I’ve had feedback of users expecting `operator*` to be the dot-product)

- **vec<T>**

  - **pro**
    1. short
    2. pronounceable
    3. usage is somewhat idiomatic: `vec<T>` is a vector-lookalike of T.
    4. term-of-art in the industry. We talk about “vectorization”, “vector unit”, “vector registers”, ...

  - **con**
    1. abbreviation (though rather common)
    2. close to `std::vector`
    3. ambiguity with mathematically inclined people who may expect operators to behave differently (e.g. I’ve had feedback of users expecting `operator*` to be the dot-product)

- **vecpar<T>**

  - **pro**
    1. short
    2. pronounceable
    3. term-of-art in the industry. We talk about “vectorization”, “vector unit”, “vector registers”, ...
    4. resolves ambiguity with math vector

  - **con**
    1. abbreviation (“vector parallel”)
    2. (`par_vec` - it’s `par_unseq` now, so we should be fine)

- **simd<T>**

  - **pro**
    1. short
    2. pronounceable
    3. usage is idiomatic: `simd<T>` is the SIMD thing for T.
    4. Known term in the industry
5. maybe even more to the point than “vector” (note variable-length vector units on traditional vector computers)

**CON**
1. acronym
2. might suggest that the type is not usable for GPUs
3. one `simd<T>` object could drive multiple or partial SIMD registers, multiple partially synchronized threads, one or more non-SIMD registers, a mix of SIMD and non-SIMD registers.

- **datapar<T>**
  
  **PRO**
  1. pronounceable
  2. “data parallel” hints at the intended use: Code expresses inherent data parallelism (intent). Contrast that to “code that uses SIMD registers and operations” (implementation detail).

  **CON**
  1. abbreviation
  2. new term

- **pack<T>**
  
  **PRO**
  1. short
  2. pronounceable
  3. usage is somewhat idiomatic (e.g. `addpd`: “add packed double-precision”)

  **CON**
  1. [name collision] Conflicts with “template parameter pack” usage in variadic templates. These tend to appear in the same context: “You can have a [template parameter] pack of packs [types].” (what?)
  2. no hint about concurrently executing operations in the name

- **simdarray<T>**
  
  **PRO**
  1. matches constant-length `std::array` and math-style of `std::valarray`.
  2. pronounceable
  3. usage is idiomatic: SIMD operations on a fixed-size array

  **CON**
  1. a bit long for daily use
  2. acronym
  3. might suggest that the type is not usable for GPUs

**VARIATIONS**
1. `simdvector<T>`: "vector" suggests `std::vector` behavior - prefer `simdarray<T>`

2. `vecarray<T>`: abbreviation ("vectorized array", not "vector array"); "vector array" misleading

- `vectorize<T>`

  **PRO**
  1. pronounceable
  2. clear meaning: produces a type that is a `vectorized T`
     i.e. action at compile time, so being a verb is fine
  3. clear meaning if proposal is extended to support `std::tuple` for `T` (and structs/classes once we get enough reflection into the language)

- `vectorization<T>`

  **CON**
  1. it is a class, it should be a noun (`vectorization<T>`?)
  2. a bit long for daily use
  3. if data structure vectorization (future extension, cf. [1]) should use a different type/mechanism it would be better to reserve this name for said extension

**VARIATIONS**

1. `simdize<T>`: shorter; downsides of `simd` - see above
2. `vectize<T>`: shorter; abbreviation
3. `vectorized<T>` `simdize<T>`: adjective, still not a noun

**5.1.3 RECOMMENDATION**

I recommend to short-list to:

- `vec<T>`
- `datapar<T>`
- `simd<T>`
- `vecpar<T>`
- `simdarray<T>`
5.2 Mask

The class in question is an array of target-specific size with elements of boolean value. The actual memory layout and storage size is unspecified. This type is the equivalence of `bool` for the `datapar<T>` types. It acts as the return type of `datapar` comparisons and can be used for write-masking, masked loads & stores, and reductions to `bool`.

5.2.1 Naming Options

- `mask<T>`
- `vecmask<T>`
- `boolvec<T>`
- `simdmask<T>`
- `simdbool<T>`
- `parmask<T>`
- `boolpack<T>`

5.2.2 Discussion

Depending on the name chosen for the “`datapar`” class, there are some natural candidates for the `mask` class. In any case, the `mask` name is:

1. a term-of-art,
2. short,
3. pronounceable,
4. idiomatic,
5. noun,
6. no name collision with existing types (as is the case for `vector`).

So I do not see a need for choosing a different (longer) name.
5.3 \textbf{where}

The “where function” wraps a mask object and a reference to a datarar or mask object to implement write-masking, and masked loads & stores. The function acts as special syntax to express that e.g. assignment shall only happen at the element indexes where the mask object is \textit{true}. The where function returns a temporary object (type \texttt{where_expression}) that implements the write-masked operations.

5.3.1 NAMING OPTIONS

\begin{itemize}
  \item \texttt{where}
  \item \texttt{masked}
  \item \texttt{withmask}
  \item \texttt{maskedval}
  \item \texttt{maskedref}
\end{itemize}

5.3.2 DISCUSSION

\begin{itemize}
  \item \texttt{where}
    \begin{itemize}
      \item \texttt{PRO} 1. short
      \item \texttt{PRO} 2. pronounceable
      \item \texttt{PRO} 3. turns code into prose: \texttt{where(x < y, z) += 2;} reads as \texttt{“where x is less then y, modify z by adding 2”}
      \item \texttt{CON} 4. naming reflects relation to \texttt{if} statements
    \end{itemize}
  \item \texttt{masked}
    \begin{itemize}
      \item \texttt{PRO} 1. short
      \item \texttt{PRO} 2. pronounceable
      \item \texttt{CON} 1. too close to \texttt{mask}: ambiguous when spoken
    \end{itemize}
  \item \texttt{withmask}
    \begin{itemize}
      \item \texttt{PRO} 1. pronounceable
    \end{itemize}
\end{itemize}
1. less intuitive to read: \(\text{withmask}(x < y, z) += 2;\)

\[\text{does something with a mask, what?}\]

- **maskedval**
  - **PRO**
    1. pronounceable
    2. communicates: produce a new object that is a *masked value* of the given object
  - **CON**
    1. *value* is not technically correct as it actually holds a reference to the given object
    2. the object returned by `maskedval` may only exists as rvalue; the name suggests otherwise

- **maskedref**
  - **PRO**
    1. pronounceable
    2. communicates: produce a new object that is a *masked reference* to the given object
  - **CON**
    1. the object returned by `maskedref` may only exists as rvalue; the name suggests otherwise

### 5.3.3 Recommendation

My recommendation is to go with `where` for what is in the wording now. If we later want to produce lvalues that act as masked references, I believe we should use a different mechanism/name anyway. Pablo suggested in private communication that `where` could be extended to:

```cpp
where (mask, v1, v2, [auto v1_, auto v2_] { // type of v1_ is a masked reference to v1
    fun(v1_, v2_); // all operations of fun on its parameters are masked
});
```

This suggests that there might not even be a need for allowing `where` or a similar function in the middle of expressions. If we want to follow that path, we might want to revisit masked reductions, which currently use a `const const_where_expression&` parameter.

### 5.4 Memload & Memstore

Loads and stores are the (low-level) conversions between arrays of \(T\) and objects of `datapar<T>`. Converting loads and stores additionally perform widening or narrowing conversions to/from arrays of \(U\), which is convertible to/from \(T\).
std::atomic has member functions called atomic::load and atomic::store: load returns the value of the atomic with a given memory_order; store replaces the value of atomic with the given value using the given memory_order. datapar::load does the reverse of atomic::load: it loads datapar::size() consecutive values starting from the given pointer into the datapar object. datapar::store does the reverse of atomic::store: it stores datapar::size() values from the datapar object to the given pointer.

5.4.1 Naming Options

- load(const U*, Flags), store(U*, Flags)
- memload(const U*, Flags), memstore(U*, Flags)
- load_from(const U*, Flags), store_to(U*, Flags)
- copy_from(const U*, Flags), copy_to(U*, Flags)

5.4.2 Discussion

- load(const U*, Flags), store(U*, Flags)
  
  **PRO**
  1. short
  2. pronounceable
  3. term-of-art

  **CON**
  1. possibly confusing when compared with load and store functions of std::atomic

- memload(const U*, Flags), memstore(U*, Flags)
  
  **PRO**
  1. pronounceable
  2. mem prefix hints at array behind the pointer argument

  **CON**
  1. abbreviation (pretty common, though)

- load_from(const U*, Flags), store_to(U*, Flags)
  
  **PRO**
  1. pronounceable
  2. reads as prose: v.load_from(mem, vector_aligned)

  **CON**

- copy_from(const U*, Flags), copy_to(U*, Flags)
5 Naming

| PRO  | 1. pronounceable                      |
|      | 2. reads as prose: v.copy_from(mem, vector_aligned) |
|      | 3. clarifies that values are copied (user feedback implies that some people expect aliasing) |
| CON  | 1. avoids term-of-art (load/store) |

### 5.4.3 Recommendation

My preference is to go with load and store. The std::atomic class is different enough. I have never received feedback that the copy direction of the load and store functions is confusing.

My second choice is copy_from/to. Avoid the embarrassment of using the terms load and store but having to name them differently just because of std::atomic. I'm certain that if we choose memload/memstore or load_from/store_to the question why we didn't just use load/store will become a FAQ.

### 5.5 Mask Queries

The free functions all_of, any_of, none_of, some_of, popcount, find_first_set, find_last_set could alternatively be member functions of mask. Having them as member functions would be consistent with bitset, which has the member functions all, any, none, and count.

Myers [P0161R0] proposes the bitset member functions low_bit_position and high_bit_position for the same operations as find_first_set(mask) and find_last_set(mask). Fioravante [N3864] proposes similar free functions for integral types: The functions cntt0 and cntt0 do the same operation while avoiding the undefined behavior on all-zero arguments. Maurer [P0553R1] proposes count[lr]_zero and popcount for unsigned integral types (i.e. free functions).

### 5.5.1 Recommendation

I believe the current interface of mask (free functions) is preferable over the interface of bitset. If anything, I believe there should be additional free functions overloaded on bitset if consistency between mask and bitset is desirable.

The operations are generic, in the sense that they are not bound to a single type but to a set of similar types. This makes them perfect candidates for free functions.

The name “low bit position” is easier to read (to me) than “find first set (bit)”. On the other hand,
• a web search of the two function names currently produces more (relevant) results for find_first_set, and

• stating that the function looks for a set bit is clearer than just stating that the function returns a bit position; the latter only implies that it does not consider 0-bits for a “bit position”.

If we rather want to follow the naming convention used by Maurer [P0553R1], then find1_one and findt_one seem to be consistent names. They replace “count” by “find” in the name to indicate the precondition that any_of must return true. Alternatively, the “find” functions could be replaced in favor of “count” functions in this proposal, to streamline with Maurer [P0553R1].

6 Wording

The following is a draft targeting inclusion into the Parallelism TS 2. It defines a basic set of data-parallel types and operations.

6.1 Data-Parallel Types

6.1.1 Header <dpapar> synopsis

```
namespace std {
  namespace experimental {
    namespace datapar_abi {
      struct scalar {};
      template <int N> struct fixed_size {};
      template <typename T> constexpr int max_fixed_size = implementation-defined;
      template <typename T> using compatible = implementation-defined;
      template <typename T> using native = implementation-defined;
    }
    namespace flags {
      struct element_aligned_tag {};
      struct vector_aligned_tag {};
      template <std::align_val_t > struct overaligned_tag {};
      constexpr element_aligned_tag element_aligned();
      constexpr vector_aligned_tag vector_aligned();
      template <std::align_val_t N> constexpr overaligned_tag<int> overaligned = {};
    }
  }
}
```

```
// traits <dpapar>traits

template <class ?> struct is_abi_tag;
template <class ?> constexpr bool is_abi_tag_v = is_abi_tag<T>::value;
```

```
template <class ?> struct is_datapar;
```
template <class ?> constexpr bool is_datapar_v = is_datapar<T>::value;

template <class ?> struct is_mask;

template <class ?> constexpr bool is_mask_v = is_mask<T>::value;

template <class T, size_t N> struct abi_for_size { using type = implementation-defined; };

template <class T, size_t N> using abi_for_size_t = typename abi_for_size<T, N>::type;

template <class T, class Abi = datapar_abi::compatible<T>> struct datapar_size {
    using type = implementation-defined;
};

template <class T, class Abi = datapar_abi::compatible<T>> constexpr size_t datapar_size_v = datapar_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> constexpr size_t memory_alignment_v = memory_alignment<T, U>::value;

// class template datapar [datapar]

template <class T, class Abi = datapar_abi::compatible<T>> class datapar;

template <class T> using native_datapar = datapar<T, datapar_abi::native<T>>;

template <class T, int N> using fixed_size_datapar = datapar<T, datapar_abi::fixed_size<N>>;

// class template mask [mask]

template <class T, class Abi = datapar_abi::compatible<T>> class mask;

template <class T> using native_mask = mask<T, datapar_abi::native<T>>;

template <class T, int N> using fixed_size_mask = mask<T, datapar_abi::fixed_size<N>>;

// casts [datapar.casts]

template <class T, class U, class A> datapar<T, /* see below */ static_datapar_cast(const datapar<U, A>&);

template <class T, class A> datapar<T, datapar_abi::fixed_size<T>> to_fixed_size(const datapar<T, A>&) noexcept;

template <class T, class A> mask<T, datapar_abi::fixed_size<T>> to_fixed_size(const mask<T, A>&) noexcept;

template <class T, size_t N> datapar<T, datapar_abi::native<T>> to_native(const datapar<T, datapar_abi::fixed_size<N>>&) noexcept;

template <class T, size_t N> mask<T, datapar_abi::native<T>> to_native(const mask<T, datapar_abi::fixed_size<N>>&) noexcept;

// reductions [mask.reductions]

template <class T, class Abi> bool all_of(mask<T, Abi>) noexcept;

template <class T, class Abi> bool any_of(mask<T, Abi>) noexcept;

template <class T, class Abi> bool none_of(mask<T, Abi>) noexcept;

template <class T, class Abi> bool some_of(mask<T, Abi>) noexcept;

template <class T, class Abi> int find_first_set(mask<T, Abi>) noexcept;

template <class T, class Abi> int find_last_set(mask<T, Abi>) noexcept;

// casts [datapar.casts]

template <class T, class U, class A> datapar<T, /* see below */ static_datapar_cast(const datapar<U, A>&);

template <class T, class A> datapar<T, datapar_abi::fixed_size<T>> to_fixed_size(const datapar<T, A>&) noexcept;

template <class T, class A> mask<T, datapar_abi::fixed_size<T>> to_fixed_size(const mask<T, A>&) noexcept;

// reductions [mask.reductions]

template <class T, class Abi> bool all_of(mask<T, Abi>) noexcept;

template <class T, class Abi> bool any_of(mask<T, Abi>) noexcept;

template <class T, class Abi> bool none_of(mask<T, Abi>) noexcept;

template <class T, class Abi> bool some_of(mask<T, Abi>) noexcept;

template <class T, class Abi> int find_first_set(mask<T, Abi>) noexcept;

template <class T, class Abi> int find_last_set(mask<T, Abi>) noexcept;
The header `<datapar>` defines the class templates (`datapar`, `mask`, `const_where_expression`, and `where_expression`), several tag types and trait types, and a series of related function templates for concurrent manipulation of the values in `datapar` and `mask` objects.

### 6.1.1.1 datapar ABI tags

```cpp
namespace datapar_abi {
    struct scalar {};
    template <typename T> struct fixed_size {};
    template <typename T> constexpr int max_fixed_size = implementation-defined;
    template <typename T> using compatible = implementation-defined;
    template <typename T> using native = implementation-defined;
}
```
An ABI tag type indicates a choice of target architecture dependent size and binary representation for `datapar` and `mask` objects. 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 `datapar` and `mask`. [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 6.1.2.1 p.2). The ABI tags enable users to safely pass `datapar` and `mask` objects between translation unit boundaries (e.g. function calls or I/O). — end note]

Use of the `scalar` tag type forces `datapar` and `mask` to store a single component (i.e. `datapar<T, datapar_abi::scalar>::size()` returns 1). [Note: `scalar` shall not be an alias for `fixed_size<1>`. — end note]

Use of the `fixed_size<N>` tag type forces `datapar` and `mask` to store and manipulate `N` components (i.e. `datapar<T, datapar_abi::fixed_size<N>::size()` returns `N`). An implementation must support at least any `N ∈ [1 ... 32]`. Additionally, for every supported `datapar<T, A>` (see 6.1.2.1 p.2), where `A` is an implementation-defined ABI tag, `N = datapar<T, A>::size()` must be supported.

[Note: An implementation may choose to forego ABI compatibility between differently compiled translation units for `datapar` and `mask` instantiations using the same `datapar_abi::fixed_size<N>` tag. Otherwise, the efficiency of `datapar<T, A>` is likely to be better than for `datapar<T, fixed_size<datapar_size_v<T, Abi>>` (with `A` not a instance of `datapar_abi::fixed_size`). — end note]

The value of `max_fixed_size<T>` declares that an instance of `datapar<T, fixed_size<N>>` with `N <= max_fixed_size<T>` is supported by the implementation. [Note: It is unspecified whether an implementation supports `datapar<T, fixed_size<N>>` with `N > max_fixed_size<T>`. The value of `max_fixed_size<T>` may depend on compiler flags and may change between different compiler versions. — end note]

An implementation may define additional ABI tag types in the `datapar_abi` namespace, to support other forms of data-parallel computation.

`datapar_abi::compatible<T>` is an alias for the ABI tag with the most efficient data parallel execution for the element type `T` that ensures ABI compatibility on the target architecture. [Example: Consider a target architecture supporting the implementation-defined ABI tags `simd128` and `simd256`, where the `simd256` type requires an optional ISA extension on said target architecture. Also, the target architecture does not support `long double` with either ABI tag. The implementation therefore defines

- `compatible<T>` as an alias for `simd128` for all arithmetic `T`, except `long double`,
- and `compatible<long double>` as an alias for `scalar`. — end example]

`datapar_abi::native<T>` is an alias for the ABI tag with the most efficient data parallel execution for the element type `T` that is supported on the currently targeted system. [Note: 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 currently targeted system supporting the implementation-defined 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
• simd256 if T is a floating-point type,
  • and simd128 otherwise.
— end example ]

6.1.1.2 datar type traits

[datapar.traits]

```cpp
template <class T> struct is_abi_tag;
```

1 The type is_abi_tag<T> is a UnaryTypeTrait with a BaseCharacteristic of true_type if T is the type of a standard or implementation-defined ABI tag, and false_type otherwise.

```cpp
template <class T> struct is_datapar;
```

2 The type is_datapar<T> is a UnaryTypeTrait with a BaseCharacteristic of true_type if T is an instance of the datapar class template, and false_type otherwise.

```cpp
template <class T> struct is_mask;
```

3 The type is_mask<T> is a UnaryTypeTrait with a BaseCharacteristic of true_type if T is an instance of the mask class template, and false_type otherwise.

```
template <class T, size_t N> struct abi_for_size { using type = implementation-defined; }; 
```

4 The member type shall be omitted if
  • T is not a cv-unqualified floating-point or integral type except bool,
  • or if datapar_abi::fixed_size<N> is not supported (see 6.1.1.1 p.3).

5 Otherwise, the member typedef type shall name an ABI tag type that satisfies
  • datapar_size_v<T, type> == N,
  • datapar<T, type> is default constructible (see 6.1.2.1 p.2),
  • datapar_abi::scalar takes precedence over fixed_size [1]. The precedence of implementation-defined ABI tags over datapar_abi::fixed_size<N> is implementation-defined. [ Note: It is expected that implementation-defined ABI tags can produce better optimizations and thus take precedence over datapar_abi::fixed_size<N>. — end note ]

```
template <class T, class Abi = datapar_abi::compatible<T>> struct datapar_size;
```

6 datapar_size<T, Abi> shall have no member value if either
  • T is not a cv-unqualified floating-point or integral type except bool,
  • or is_abi_tag_v<Abi> is false.
  [ Note: The rules are different from 6.1.2.1 p.2 — end note ]

7 Otherwise, the type datapar_size<T, Abi> is a BinaryTypeTrait with a BaseCharacteristic of integral_constant<size_t, N> with N equal to the number of elements in a datapar<T, Abi> object. [ Note: If datapar<T, Abi> is not supported for the currently targeted system, datapar_size<T, Abi>::value produces the value datapar<T, Abi>::size() would return if it were supported. — end note ]
template <class T, class U = typename T::value_type> struct memory_alignment;

memory_alignment<T, U> shall have no member value if either

- T is cv-qualified,
- or U is cv-qualified,
- or !is_datapar_v<T> && !is_mask_v<T>,
- or is_datapar_v<T> and U is not an arithmetic type or U is bool,
- or is_mask_v<T> and U is not bool.

Otherwise, the type memory_alignment<T, U> is a BinaryTypeTrait with a BaseCharacteristic of integral_constant<sizeof_t, N> for some implementation-defined N. [Note: value identifies the alignment restrictions on pointers used for (converting) loads and stores for the given type T on arrays of type U (see 6.1.2.3, 6.1.2.4, 6.1.4.3, 6.1.4.4). — end note ]

6.1.1.3 Class templates const_where_expression and where_expression [datapar.whereexpr]

namespace std {
  namespace experimental {
    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;

      remove_const_t<T> operator-() const &&;

      template <class U, class Flags> [[nodiscard]] V memload(const U* mem, Flags f) const &&;
    }

    template <class U, class T>
    class where_expression : public const_where_expression<M, T> {
      public:
      where_expression(const where_expression&) = delete;
      where_expression& operator=(const where_expression&) = delete;

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

      void operator++();
      void operator++(int);
      void operator--();
      void operator--(int);
    }
The class templates `const_where_expression` and `where_expression<M, T>` combine a predicate and a value object to implement an interface that restricts assignments and/or operations on the value object to the elements selected via the predicate.

The first template argument `M` must be cv-unqualified `bool` or a cv-unqualified mask instantiation.

The second template argument `T` must be a cv-unqualified or `const` qualified type `U`. If `M` is `bool`, `U` must be an arithmetic type. Otherwise, `U` must either be `M` or `M::datapar_type`.

```
const M& mask; // exposition only
T& data; // exposition only
```

[Note: The implementation initializes a `where_expression<M, T>` object with a predicate of type `M` and a reference to a value object of type `T`. The predicate object and a const qualified value object may be copied by the constructor implementation. — end note]

```
remove_const_t<T> operator-() const &&;
```

Returns: If `M` is `bool`, `-data` if `mask` is `true`, `data` otherwise. If `M` is not `bool`, returns an object with the `i`-th element initialized to `-data[i]` if `mask[i]` is `true` and `data[i]` otherwise for all `i ∈ [0, M::size())`.

```
template <class U, class Flags>
[[nodiscard]] remove_const_t<T> memload(const U *mem, Flags) const &&;
```

Remarks: If `remove_const_t<T>` is `bool` or `is_mask_v<remove_const_t<T>>`, the function shall not participate in overload resolution unless `U` is `bool`. Otherwise, the function shall not participate in overload resolution unless `U` is an arithmetic type except `bool`.

Returns: If `M` is `bool`, return `mem[0]` if `mask` equals `true` and return `data` otherwise. If `M` is not `bool`, return an object with the `i`-th element initialized to the `i`-th element of `data` if `mask[i]` is `false` and `static_cast<T::value_type>(mem[i])` if `mask[i]` is `true` for all `i ∈ [0, M::size())`.

Requires: If `M` is not `bool`, the largest `i` where `mask[i]` is `true` is less than the number of values pointed to by `mem`.

Requires: If the `Flags` template parameter is of type `flags::vector_aligned_tag`, the pointer value represents an address aligned to `memory_alignment_v<T, U>`. If the `Flags` template parameter is of type `flags::overaligned_tag<N>`, the pointer value represents an address aligned to `N`.

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

Remarks: If `remove_const_t<T>` is `bool` or `is_mask_v<remove_const_t<T>>`, the function shall not participate in overload resolution unless `U` is `bool`. Otherwise, the function shall not participate in overload resolution unless `U` is an arithmetic type except `bool`. 
Effects: If $M$ is bool, assigns data to mem[0] unless mask is false. If $M$ is not bool, copies the elements data[i] where mask[i] is true as if mem[i] = static_cast<U>(data[i]) for all $i \in [0, M::size())$.

Requires: If $M$ is not bool, the largest $i$ where mask[i] is true is less than the number of values pointed to by mem.

Requires: If the Flags template parameter is of type flags::vector_aligned_tag, the pointer value represents an address aligned to memory_alignment_v<remove_const_t<T>, U>. If the Flags template parameter is of type flags::overaligned_tag<N>, the pointer value represents an address aligned to N.

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

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

Effects: If $M$ is bool, applies the indicated operator on data and forward<U>(x) unless mask is false. If $M$ is not bool, applies the indicated operator on data and forward<U>(x) without modifying the elements data[i] where mask[i] is false for all $i \in [0, M::size())$.

Remarks: It is unspecified whether the arithmetic/bitwise operation, which is implied by a compound assignment operator, is executed on all elements or only on the ones written back.

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

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

Effects: If $M$ is bool, applies the indicated operator on data unless mask is false. If $M$ is not bool, applies the indicated operator on data without modifying the elements data[i] where mask[i] is false for all $i \in [0, M::size())$. [Note: It is unspecified whether the inc-/decrement operation is executed on all elements or only on the ones written back. — end note ]

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

Remarks: If T is bool or is_mask_v<T>, the function shall not participate in overload resolution unless U is bool.

Effects: If $M$ is bool, assign mem[0] to data unless mask is false. If $M$ is not bool, replace the elements of data where mask[i] is true such that the $i$-th element is assigned with static_cast<T::value_type>(mem[i]) for all $i \in [0, M::size())$.
Requires: If \( M \) is not \texttt{bool}, the largest \( i \) where \( \text{mask}[i] \) is \texttt{true} is less than the number of values pointed to by \( \text{mem} \).

Requires: If the \texttt{Flags} template parameter is of type \texttt{flags::vector_aligned_tag}, the pointer value represents an address aligned to \texttt{memory_alignment_v<T, U>}. If the \texttt{Flags} template parameter is of type \texttt{flags::overaligned_tag<N>}, the pointer value represents an address aligned to \( N \).

6.1.2 Class template \texttt{datapar}

6.1.2.1 Class template \texttt{datapar} overview

```cpp
namespace std {
namespace experimental {
  template <class T, class Abi> class datapar {
    public:
      using value_type = T;
      using reference = implementation-defined; // see below
      using mask_type = mask<T, Abi>;
      using size_type = size_t;
      using abi_type = Abi;

      static constexpr size_type size() noexcept;
      datapar() = default;
      datapar(const datapar&) = default;
      datapar(datapar&&) = default;

      // implicit broadcast constructor
      template <class U> datapar(U&&);

      // implicit type conversion constructor
      template <class U> datapar(const datapar<U, datapar_abi::fixed_size<size()>>&);

      // generator constructor
      template <class G> datapar(G&& gen);

      // load constructor
      template <class U, class Flags> datapar(const U* mem, Flags);

      // loads [datapar.load]
      template <class U, class Flags> void memload(const U* mem, Flags);

      // stores [datapar.store]
      template <class U, class Flags> void memstore(U* mem, Flags) const;

      // scalar access [datapar.subscr]
      reference operator[](size_type);
      value_type operator[](size_type) const;

      // unary operators [datapar.unary]
      datapar& operator++();
      datapar operator++(int);
  }
}
```
The class template `datapar<T, Abi>` is a one-dimensional smart array. The number of elements in the array is a constant expression, according to the `Abi` template parameter.

The resulting class shall be a complete type with deleted default constructor, deleted destructor, deleted copy constructor, and deleted copy assignment unless all of the following hold:

1. The first template argument `T` is a cv-unqualified integral or floating-point type except `bool` (3.9.1 [basic.fundamental]).
2. The second template argument `Abi` is an ABI tag so that `is_abi_tag_v<Abi>` is true.
3. The `Abi` type is a supported ABI tag. It is supported if

   - `Abi` is `datapar_abi::scalar`, or
— Abi is `std::fixed_size<N>` with `N \leq 32` or implementation-defined additional valid values for `N` (see 6.1.1.1 p.3).

It is implementation-defined whether a given combination of `T` and an implementation-defined ABI tag is supported. [Note: The intent is for implementations to decide on the basis of the currently targeted system. — end note]

[Example: Consider an implementation that defines the implementation-defined 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 except long double and no support for the `gpu_y` ABI tag, then:

• `datapar<T, datapar_abi::gpu_y>` is not supported for any `T` and results in a type with deleted constructor
• `datapar<long double, datapar_abi::simd_x>` is not supported and results in a type with deleted constructor
• `datapar<double, datapar_abi::simd_x>` is supported
• `datapar<long double, datapar_abi::scalar>` is supported

— end example]

3 Default initialization 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 The member type `reference` is an implementation-defined type acting as a reference to an element of type `value_type` with the following properties:

• The type has a deleted default constructor, copy constructor, and copy assignment operator.
• Assignment, compound assignment, increment, and decrement operators shall not participate in overload resolution unless the `reference` object is an rvalue and the corresponding operator of type `value_type` is usable.
• Objects of type `reference` are implicitly convertible to `value_type`.
• If a binary operator is applied to an object of type `reference`, the operator is only applied after converting the `reference` object to `value_type`.
• Calls to `swap(reference &&, value_type &)` and `swap(value_type &, reference &&)` exchange the values referred to by the `reference` object and the `value_type` reference. Calls to `swap(reference &&, reference &&)` exchange the values referred to by the `reference` objects.

```cpp
static constexpr size_type size() noexcept;
```

Returns: the number of elements stored in objects of the given `datapar<T, Abi>` type.

6 [Note: Implementations are encouraged to enable `static_cast` from/to (an) implementation-defined SIMD type(s). This would add one or more of the following declarations to class `datapar`:

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

— end note]

6.1.2.2 `datapar` constructors

[datapar.ctor]
template <class U> datapar(U&&);

Remarks: This constructor shall not participate in overload resolution unless either:

• U is an arithmetic type except bool and every possible value of type U can be represented with type value_type,
• or U is not an arithmetic type and is implicitly convertible to value_type,
• or U is int,
• or U is unsigned int and value_type is an unsigned integral type.

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

Throws: Any exception thrown while converting the argument to value_type.

template <class U> datapar(const datapar<U, datapar_abi::fixed_size<size()>>& x);

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

• abi_type equals datapar_abi::fixed_size<size()>,
• and every possible value of U can be represented with type value_type,
• and, if both U and value_type are integral, the integer conversion rank [N4618, (4.15)] of value_type is greater than the integer conversion rank of U.

Effects: Constructs an object where the i-th element equals static_cast<T>(x[i]) for all i ∈ [0, size()).

template <class G> datapar(G&& gen);

Remarks: This constructor shall not participate in overload resolution unless datapar(gen(integral_constant<size_t, 0>(),)) is well-formed.

Effects: Constructs an object where the i-th element is initialized to gen(integral_constant<size_t, i>()) for all i ∈ [0, size()).

Remarks: The order of calls to gen is unspecified.

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

Remarks: This constructor shall not participate in overload resolution unless U is an arithmetic type except bool.

Effects: Constructs an object where the i-th element is initialized to static_cast<T>(mem[i]) for all i ∈ [0, size()).

Requires: size() is less than or equal to the number of values pointed to by mem.

Requires: If the Flags template parameter is of type flags::vector_aligned_tag, the pointer value represents an address aligned to memory_alignment_v<datapar, U>. If the Flags template parameter is of type flags::overaligned_tag<N>, the pointer value represents an address aligned to N.
template <class U, class Flags> void memload(const U *mem, Flags);

Remarks: This function shall not participate in overload resolution unless U is an arithmetic type except bool.

Effects: Replaces the elements of the datapar object such that the i-th element is assigned with static_cast<T>(mem[i]) for all i ∈ [0, size()).

Requires: size() is less than or equal to the number of values pointed to by mem.

Requires: If the Flags template parameter is of type flags::vector_aligned_tag, the pointer value represents an address aligned to memory_alignment_v<datapar, U>. If the Flags template parameter is of type flags::overaligned_tag<N>, the pointer value represents an address aligned to N.

6.1.2.4 datapar store function

template <class U, class Flags> void memstore(U *mem, Flags);

Remarks: This function shall not participate in overload resolution unless U is an arithmetic type except bool.

Effects: Copies all datapar elements as if mem[i] = static_cast<U>(operator[](i)) for all i ∈ [0, size()).

Requires: size() is less than or equal to the number of values pointed to by mem.

Requires: If the Flags template parameter is of type flags::vector_aligned_tag, the pointer value represents an address aligned to memory_alignment_v<datapar, U>. If the Flags template parameter is of type flags::overaligned_tag<N>, the pointer value represents an address aligned to N.

6.1.2.5 datapar subscript operators

reference operator[](size_type i);

Requires: The value of i is less than size().

Returns: A temporary object of type reference (see 6.1.2.1 p.4) with the following effects:

Effects: The assignment, compound assignment, increment, and decrement operators of reference execute the indicated operation on the i-th element of the datapar object.

Effects: Conversion to value_type returns a copy of the i-th element.

Throws: Nothing.

value_type operator[](size_type i) const;

Requires: The value of i is less than size().

Returns: A copy of the i-th element.

Throws: Nothing.

6.1.2.6 datapar unary operators

[datapar.unary]
**operator++()**

1. **Effects:** Increments every element of *this by one.
2. **Returns:** An lvalue reference to *this after incrementing.
3. **Remarks:** Overflow semantics follow the same semantics as for T.
4. **Throws:** Nothing.

**operator++(int)**

5. **Effects:** Increments every element of *this by one.
6. **Returns:** A copy of *this before incrementing.
7. **Remarks:** Overflow semantics follow the same semantics as for T.
8. **Throws:** Nothing.

**operator--()**

9. **Effects:** Decrement every element of *this by one.
10. **Returns:** An lvalue reference to *this after decrementing.
11. **Remarks:** Underflow semantics follow the same semantics as for T.
12. **Throws:** Nothing.

**operator--(int)**

13. **Effects:** Decrement every element of *this by one.
14. **Returns:** A copy of *this before decrementing.
15. **Remarks:** Underflow semantics follow the same semantics as for T.
16. **Throws:** Nothing.

**operator!() const**

17. **Returns:** A mask object with the i-th element set to !operator[](i) for all i ∈ [0, size()).
18. **Throws:** Nothing.

**operator~() const**

19. **Returns:** A datapar object where each bit is the inverse of the corresponding bit in *this.
20. **Remarks:** datapar::operator~() shall not participate in overload resolution unless T is an integral type.
21. **Throws:** Nothing.

**operator+() const**

22. **Returns:** A copy of *this
23. **Throws:** Nothing.
`datapar operator[](i) const;

Returns: A `datapar` object where the i-th element is initialized to `operator[](i)` for all `i ∈ [0, size())`.

Throws: Nothing.

6.1.3 `datapar` non-member operations

6.1.3.1 `datapar` binary operators

```cpp
friend datapar operator+ (const datapar&, const datapar&);
fraction datapar operator- (const datapar&, const datapar&);
fraction datapar operator* (const datapar&, const datapar&);
fraction datapar operator/ (const datapar&, const datapar&);
fraction datapar operator% (const datapar&, const datapar&);
fraction datapar operator& (const datapar&, const datapar&);
fraction datapar operator| (const datapar&, const datapar&);
fraction datapar operator^ (const datapar&, const datapar&);
friend datapar operator<< (const datapar&, const datapar&);
friend datapar operator>>(const datapar&, const datapar&);
```

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

Returns: A `datapar` object initialized with the results of the component-wise application of the indicated operator.

Throws: Nothing.

```cpp
friend datapar operator<<(const datapar& v, int n);
friend datapar operator>>(const datapar& v, int n);
```

Remarks: Both operators shall not participate in overload resolution unless `value_type` is an unsigned integral type.

Returns: A `datapar` object where the i-th element is initialized to the result of applying the indicated operator to `v[i]` and `n` for all `i ∈ [0, size())`.

Throws: Nothing.

6.1.3.2 `datapar` compound assignment

```cpp
friend datapar& operator+= (datapar&, const datapar&);
friend datapar& operator-= (datapar&, const datapar&);
friend datapar& operator*= (datapar&, const datapar&);
friend datapar& operator/= (datapar&, const datapar&);
friend datapar& operator%= (datapar&, const datapar&);
friend datapar& operator&= (datapar&, const datapar&);
friend datapar& operator|= (datapar&, const datapar&);
friend datapar& operator^= (datapar&, const datapar&);
friend datapar& operator<<=(datapar&, const datapar&);
friend datapar& operator>>=(datapar&, const datapar&);
```
Remarks: Each of these operators shall not participate in overload resolution unless the indicated operator can be applied to objects of type \texttt{value\_type}.

Effects: Each of these operators performs the indicated operator component-wise on each of the corresponding elements of the arguments.

Returns: A reference to the first argument.

Throws: Nothing.

\begin{verbatim}
friend \texttt{datapar\&} \texttt{operator\textless\textless}(\texttt{datapar\&} \texttt{v}, \texttt{int} \texttt{n});
friend \texttt{datapar\&} \texttt{operator\textgreater\textgreater}(\texttt{datapar\&} \texttt{v}, \texttt{int} \texttt{n});
\end{verbatim}

Remarks: Both operators shall not participate in overload resolution unless \texttt{value\_type} is an unsigned integral type.

Effects: Performs the indicated shift by \texttt{n} operation on the \(i\)-th element of \texttt{v} for all \(i \in \{0, \text{size()}\}\).

Returns: A reference to the first argument.

Throws: Nothing.

6.1.3.3 \texttt{datapar compare operators} \hfill [\texttt{datapar.comparison}] 

\begin{verbatim}
friend \texttt{mask\_type} \texttt{operator\textless\textless}(\texttt{const} \texttt{datapar\&}, \texttt{const} \texttt{datapar\&});
friend \texttt{mask\_type} \texttt{operator\textgreater\textgreater}(\texttt{const} \texttt{datapar\&}, \texttt{const} \texttt{datapar\&});
friend \texttt{mask\_type} \texttt{operator\textasciitilde\textasciitilde}(\texttt{const} \texttt{datapar\&}, \texttt{const} \texttt{datapar\&});
friend \texttt{mask\_type} \texttt{operator\textasciitilde\textasciitilde\textasciitilde}(\texttt{const} \texttt{datapar\&}, \texttt{const} \texttt{datapar\&});
friend \texttt{mask\_type} \texttt{operator\textasciitilde\textasciitilde\textasciitilde\textasciitilde\textasciitilde}(\texttt{const} \texttt{datapar\&}, \texttt{const} \texttt{datapar\&});
friend \texttt{mask\_type} \texttt{operator\textasciitilde\textasciitilde\textasciitilde\textasciitilde\textasciitilde\textasciitilde}(\texttt{const} \texttt{datapar\&}, \texttt{const} \texttt{datapar\&});
\end{verbatim}

Returns: A mask object initialized with the results of the component-wise application of the indicated operator.

Throws: Nothing.

6.1.3.4 \texttt{datapar reductions} \hfill [\texttt{datapar.reductions}]

\begin{verbatim}
template<class BinaryOperation = \texttt{std::plus<>, class T, class Abi}>
T reduce(\texttt{const} \texttt{datapar\&}\texttt{T}, \texttt{Abi}\texttt{1} x, \texttt{BinaryOperation} \texttt{binary\_op} = \texttt{BinaryOperation}());
\end{verbatim}

Returns: \texttt{GENERALIZED\_SUM} (\texttt{binary\_op}, \texttt{x.data}[i], …) for all \(i \in \{0, \text{size()}\}\).

Requires: \texttt{binary\_op} shall be callable with two arguments of type \texttt{T} or two arguments of type \texttt{datapar<T, A1>}, where \texttt{A1} may be different to \texttt{Abi}.

[ Note: This overload of \texttt{reduce} does not require an initial value because \texttt{x} is guaranteed to be non-empty. — end note ]

\begin{verbatim}
template<class BinaryOperation = \texttt{std::plus<>, class M, class V}>
typename V:\texttt{value\_type} reduce(\texttt{const} \texttt{const\_where\_expression\texttt{M}}, \texttt{V}\texttt{\&} x, 
typename V:\texttt{value\_type} \texttt{neutral\_element} = \texttt{default\_neutral\_element},
\texttt{BinaryOperation} \texttt{binary\_op} = \texttt{BinaryOperation}());
\end{verbatim}
Returns: If none_of(x.mask), returns neutral_element. Otherwise, returns GENERALIZED_SUM(binary_op, x.data[i], ...) for all \(i \in \{j \in \mathbb{N}_0 | j < \text{size()} \land x\cdot\text{mask}[j]\}\).

Requires: binary_op shall be callable with two arguments of type \(T\) or two arguments of type datapar<T, A1>, where A1 may be different to Abi.

[Note: This overload of reduce requires a neutral value to enable a parallelized implementation: A temporary datapar object initialized with neutral_element is conditionally assigned from x.data using x.mask. Subsequently, the parallelized reduction (without mask) is applied to the temporary object. — end note]

\[
\text{template } \langle \text{class } T, \text{ class } A \rangle \text{ } T \ h\text{min}\left(\text{const } \text{datapar}\langle T, A\rangle & x\right);
\]

Returns: The value of an element \(x[j]\) for which \(x[j] \leq x[i]\) for all \(i \in [0, \text{size()}]\).

Throws: Nothing.

\[
\text{template } \langle \text{class } M, \text{ class } V \rangle \text{ } T \ h\text{min}\left(\text{const } \text{const}\_\text{where}\_\text{expression}\langle M, V\rangle & x\right);
\]

Returns: If none_of(x.mask), the return value is numeric_limits<V::value_type>::max(). Otherwise, returns the value of an element \(x\cdot\text{data}[j]\) for which \(x\cdot\text{mask}[j] = \text{true}\) and \(x\cdot\text{data}[j] \leq x\cdot\text{data}[i]\) for all \(i \in [0, \text{size()}]\).

Throws: Nothing.

\[
\text{template } \langle \text{class } T, \text{ class } A \rangle \text{ } T \ h\text{max}\left(\text{const } \text{datapar}\langle T, A\rangle & x\right);
\]

Returns: The value of an element \(x[j]\) for which \(x[j] \geq x[i]\) for all \(i \in [0, \text{size()}]\).

Throws: Nothing.

\[
\text{template } \langle \text{class } M, \text{ class } V \rangle \text{ } T \ h\text{max}\left(\text{const } \text{const}\_\text{where}\_\text{expression}\langle M, V\rangle & x\right);
\]

Returns: If none_of(x.mask), the return value is numeric_limits<V::value_type>::min(). Otherwise, returns the value of an element \(x\cdot\text{data}[j]\) for which \(x\cdot\text{mask}[j] = \text{true}\) and \(x\cdot\text{data}[j] \geq x\cdot\text{data}[i]\) for all \(i \in [0, \text{size()}]\).

Throws: Nothing.

6.1.3.5 datapar casts

\[
\text{template } \langle \text{class } T, \text{ class } U, \text{ class } A \rangle \text{ } \text{datapar}\langle T, /*\text{see below}*/ \rangle \ \text{static}\_\text{datapar}\_\text{cast}\left(\text{const } \text{datapar}\langle U, A\rangle & x\right);
\]

Remarks: The return type is datapar<T, A> if either U and T are equal or U and T are integral types that only differ in signedness. Otherwise, the return type is datapar<T, datapar_abi::fixed_size<datapar <U, A>::size()>).

Returns: A datapar object with the \(i\)-th element initialized to static_cast<T>{x[i]}.

Throws: Nothing.
template <class T, class A>
    datapar<T, datapar_abi::fixed_size<datapar_size_v<T, A>>> to_fixed_size(const datapar<T, A>& x) noexcept;

template <class T, class A>
    mask<T, datapar_abi::fixed_size<datapar_size_v<T, A>>> to_fixed_size(const mask<T, A>& x) noexcept;

Returns: An object with the i-th element initialized to x[i].

template <class T, size_t N>
    datapar<T, datapar_abi::native<T>> to_native(const datapar<T, datapar_abi::fixed_size<N>>& x) noexcept;

template <class T, size_t N>
    mask<T, datapar_abi::native<T>> to_native(const mask<T, datapar_abi::fixed_size<N>>& x) noexcept;

Remarks: These functions shall not participate in overload resolution unless datapar_size_v<T, datapar_abi::native<T>> is equal to N.

Returns: An object with the i-th element initialized to x[i].

6.1.3.6 datapar algorithms [datapar.alg]

template <class T, class A>
    datapar<T, A> min(const datapar<T, A>& a, const datapar<T, A>& b) noexcept;

Returns: An object with the i-th element initialized with the smaller value of a[i] and b[i] for all i ∈ [0, size()).

template <class T, class A>
    datapar<T, A> max(const datapar<T, A>& a, const datapar<T, A>& b) noexcept;

Returns: An object with the i-th element initialized with the larger value of a[i] and b[i] for all i ∈ [0, size()).

template <class T, class A>

Returns: An object with the i-th element in the first pair member initialized with the smaller value of a[i] and b[i] for all i ∈ [0, size()). The i-th element in the second pair member is initialized with the larger value of a[i] and b[i] for all i ∈ [0, size()).

template <class T, class A>
    datapar<T, A> clamp(const datapar<T, A>& v, const datapar<T, A>& lo, const datapar<T, A>& hi);

40
4 \emph{Requires:} No element in $lo$ shall be greater than the corresponding element in $hi$.

5 \emph{Returns:} An object with the $i$-th element initialized with $lo[i]$ if $v[i]$ is smaller than $lo[i], hi[i]$ if $v[i]$ is larger than $hi[i]$, otherwise $v[i]$ for all $i \in \{0, \ldots, size()\}$.

6.1.3.7 \texttt{datapar} math library

```
namespace std {
    namespace experimental {
        template <class Abi> using scharv = datapar<signed char, Abi>; // exposition only
        template <class Abi> using shortv = datapar<short, Abi>; // exposition only
        template <class Abi> using intv = datapar<int, Abi>; // exposition only
        template <class Abi> using llongv = datapar<long long int, Abi>; // exposition only
        template <class T, class V> using samesize = fixed_size_datapar<T, V::size ()>; // exposition only
    }
    namespace math {
        template <class Abi> floatv<Abi> acos(floatv<Abi> x);
        template <class Abi> doublev<Abi> acos(doublev<Abi> x);
        template <class Abi> ldoublev<Abi> acos(ldoublev<Abi> x);
        template <class Abi> floatv<Abi> asin(floatv<Abi> x);
        template <class Abi> doublev<Abi> asin(doublev<Abi> x);
        template <class Abi> ldoublev<Abi> asin(ldoublev<Abi> x);
        template <class Abi> floatv<Abi> atan(floatv<Abi> x);
        template <class Abi> doublev<Abi> atan(doublev<Abi> x);
        template <class Abi> ldoublev<Abi> atan(ldoublev<Abi> x);
        template <class Abi> floatv<Abi> atan2(floatv<Abi> y, floatv<Abi> x);
        template <class Abi> doublev<Abi> atan2(doublev<Abi> y, doublev<Abi> x);
        template <class Abi> ldoublev<Abi> atan2(ldoublev<Abi> y, ldoublev<Abi> x);
        template <class Abi> floatv<Abi> cos(floatv<Abi> x);
        template <class Abi> doublev<Abi> cos(doublev<Abi> x);
        template <class Abi> ldoublev<Abi> cos(ldoublev<Abi> x);
        template <class Abi> floatv<Abi> sin(floatv<Abi> x);
        template <class Abi> doublev<Abi> sin(doublev<Abi> x);
        template <class Abi> ldoublev<Abi> sin(ldoublev<Abi> x);
        template <class Abi> floatv<Abi> tan(floatv<Abi> x);
        template <class Abi> doublev<Abi> tan(doublev<Abi> x);
        template <class Abi> ldoublev<Abi> tan(ldoublev<Abi> x);
        template <class Abi> floatv<Abi> acosh(floatv<Abi> x);
        template <class Abi> doublev<Abi> acosh(doublev<Abi> x);
        template <class Abi> ldoublev<Abi> acosh(ldoublev<Abi> x);
        template <class Abi> floatv<Abi> asinh(floatv<Abi> x);
        template <class Abi> doublev<Abi> asinh(doublev<Abi> x);
        template <class Abi> ldoublev<Abi> asinh(ldoublev<Abi> x);
        template <class Abi> floatv<Abi> atanh(floatv<Abi> x);
    }
}
```
template <class Abi> doublev<Abi> atanh(doublev<Abi> x);
template <class Abi> ldoublev<Abi> atanh(ldoublev<Abi> x);

template <class Abi> floatv<Abi> cosh(floatv<Abi> x);
template <class Abi> doublev<Abi> cosh(doublev<Abi> x);
template <class Abi> ldoublev<Abi> cosh(ldoublev<Abi> x);

template <class Abi> floatv<Abi> sinh(floatv<Abi> x);
template <class Abi> doublev<Abi> sinh(doublev<Abi> x);
template <class Abi> ldoublev<Abi> sinh(ldoublev<Abi> x);

template <class Abi> floatv<Abi> tanh(floatv<Abi> x);
template <class Abi> doublev<Abi> tanh(doublev<Abi> x);
template <class Abi> ldoublev<Abi> tanh(ldoublev<Abi> x);

template <class Abi> floatv<Abi> exp(floatv<Abi> x);
template <class Abi> doublev<Abi> exp(doublev<Abi> x);
template <class Abi> ldoublev<Abi> exp(ldoublev<Abi> x);

template <class Abi> floatv<Abi> exp2(floatv<Abi> x);
template <class Abi> doublev<Abi> exp2(doublev<Abi> x);
template <class Abi> ldoublev<Abi> exp2(ldoublev<Abi> x);

template <class Abi> floatv<Abi> expm1(floatv<Abi> x);
template <class Abi> doublev<Abi> expm1(doublev<Abi> x);
template <class Abi> ldoublev<Abi> expm1(ldoublev<Abi> x);

template <class Abi> floatv<Abi> ilogb(floatv<Abi> x);
template <class Abi> doublev<Abi> ilogb(doublev<Abi> x);
template <class Abi> ldoublev<Abi> ilogb(ldoublev<Abi> x);

template <class Abi> floatv<Abi> ldexp(floatv<Abi> x, samesize<int, floatv<Abi>> exp);
template <class Abi> doublev<Abi> ldexp(doublev<Abi> x, samesize<int, doublev<Abi>> exp);
template <class Abi> ldoublev<Abi> ldexp(ldoublev<Abi> x, samesize<int, ldoublev<Abi>> exp);

template <class Abi> floatv<Abi> log(floatv<Abi> x);
template <class Abi> doublev<Abi> log(doublev<Abi> x);
template <class Abi> ldoublev<Abi> log(ldoublev<Abi> x);

template <class Abi> floatv<Abi> log10(floatv<Abi> x);
template <class Abi> doublev<Abi> log10(doublev<Abi> x);
template <class Abi> ldoublev<Abi> log10(ldoublev<Abi> x);

template <class Abi> floatv<Abi> log1p(floatv<Abi> x);
template <class Abi> doublev<Abi> log1p(doublev<Abi> x);
template <class Abi> ldoublev<Abi> log1p(ldoublev<Abi> x);

template <class Abi> floatv<Abi> log2(floatv<Abi> x);
template <class Abi> doublev<Abi> log2(doublev<Abi> x);
template <class Abi> ldoublev<Abi> log2(ldoublev<Abi> x);

template <class Abi> floatv<Abi> logb(floatv<Abi> x);
template <class Abi> doublev<Abi> logb(doublev<Abi> x);
template <class Abi> ldoublev<Abi> logb(ldoublev<Abi> x);
template<
class Abi>
floatv<Abi> modf(floatv<Abi> value, floatv<Abi>* iptr);
template<
class Abi>
doublev<Abi> modf(doublev<Abi> value, doublev<Abi>* iptr);
template<
class Abi>
ldoublev<Abi> modf(ldoublev<Abi> value, ldoublev<Abi>* iptr);

template<
class Abi>
floatv<Abi> scalbn(floatv<Abi> x, samesize<int, floatv<Abi>> n);
template<
class Abi>
doublev<Abi> scalbn(doublev<Abi> x, samesize<int, doublev<Abi>> n);
template<
class Abi>
ldoublev<Abi> scalbn(ldoublev<Abi> x, samesize<int, ldoublev<Abi>> n);

template<
class Abi>
floatv<Abi> scalbln(floatv<Abi> x, samesize<long int, floatv<Abi>> n);
template<
class Abi>
doublev<Abi> scalbln(doublev<Abi> x, samesize<long int, doublev<Abi>> n);
template<
class Abi>
ldoublev<Abi> scalbln(ldoublev<Abi> x, samesize<long int, ldoublev<Abi>> n);

template<
class Abi>
fcharv<Abi> abs(scharv<Abi> j);
template<
class Abi>
shortv<Abi> abs(shortv<Abi> j);
template<
class Abi>
intv<Abi> abs(intv<Abi> j);
template<
class Abi>
longv<Abi> abs(longv<Abi> j);
template<
class Abi>
longv<Abi> abs(longv<Abi> j);
template<
class Abi>
floatv<Abi> abs(floatv<Abi> j);
template<
class Abi>
doublev<Abi> abs(doublev<Abi> j);
template<
class Abi>
ldoublev<Abi> abs(ldoublev<Abi> j);

template<
class Abi>
floatv<Abi> hypot(floatv<Abi> x, floatv<Abi> y);
template<
class Abi>
doublev<Abi> hypot(doublev<Abi> x, doublev<Abi> y);
template<
class Abi>
ldoublev<Abi> hypot(ldoublev<Abi> x, ldoublev<Abi> y);

template<
class Abi>
floatv<Abi> hypot(floatv<Abi> x, floatv<Abi> y, floatv<Abi> z);
template<
class Abi>
doublev<Abi> hypot(doublev<Abi> x, doublev<Abi> y, doublev<Abi> z);
template<
class Abi>
ldoublev<Abi> hypot(ldoublev<Abi> x, ldoublev<Abi> y, ldoublev<Abi> z);

template<
class Abi>
floatv<Abi> pow(floatv<Abi> x, floatv<Abi> y);
template<
class Abi>
doublev<Abi> pow(doublev<Abi> x, doublev<Abi> y);
template<
class Abi>
ldoublev<Abi> pow(ldoublev<Abi> x, ldoublev<Abi> y);

template<
class Abi>
floatv<Abi> sqrt(floatv<Abi> x);
template<
class Abi>
doublev<Abi> sqrt(doublev<Abi> x);
template<
class Abi>
ldoublev<Abi> sqrt(ldoublev<Abi> x);

template<
class Abi>
floatv<Abi> erf(floatv<Abi> x);
template<
class Abi>
doublev<Abi> erf(doublev<Abi> x);
template<
class Abi>
ldoublev<Abi> erf(ldoublev<Abi> x);

template<
class Abi>
floatv<Abi> erfc(floatv<Abi> x);
template<
class Abi>
doublev<Abi> erfc(doublev<Abi> x);
template<
class Abi>
ldoublev<Abi> erfc(ldoublev<Abi> x);

template<
class Abi>
floatv<Abi> lgamma(floatv<Abi> x);
template<
class Abi>
doublev<Abi> lgamma(doublev<Abi> x);
template<
class Abi>
ldoublev<Abi> lgamma(ldoublev<Abi> x);

template<
class Abi>
floatv<Abi> tgamma(floatv<Abi> x);
template<
class Abi>
doublev<Abi> tgamma(doublev<Abi> x);
template<
class Abi>
ldoublev<Abi> tgamma(ldoublev<Abi> x);
template <class Abi> floatv<Abi> ceil(floatv<Abi> x);
template <class Abi> doublev<Abi> ceil(doublev<Abi> x);

template <class Abi> floatv<Abi> floor(floatv<Abi> x);
template <class Abi> doublev<Abi> floor(doublev<Abi> x);

template <class Abi> floatv<Abi> nearbyint(floatv<Abi> x);
template <class Abi> doublev<Abi> nearbyint(doublev<Abi> x);

template <class Abi> floatv<Abi> rint(floatv<Abi> x);
template <class Abi> doublev<Abi> rint(doublev<Abi> x);

template <class Abi> samesize<long int, floatv<Abi>> lrint(floatv<Abi> x);
template <class Abi> samesize<long int, doublev<Abi>> lrint(doublev<Abi> x);

template <class Abi> floatv<Abi> round(floatv<Abi> x);
template <class Abi> doublev<Abi> round(doublev<Abi> x);

template <class Abi> samesize<long int, floatv<Abi>> lround(floatv<Abi> x);
template <class Abi> samesize<long int, doublev<Abi>> lround(doublev<Abi> x);

template <class Abi> floatv<Abi> trunc(floatv<Abi> x);
template <class Abi> doublev<Abi> trunc(doublev<Abi> x);

template <class Abi> floatv<Abi> fmod(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> fmod(doublev<Abi> x, doublev<Abi> y);

template <class Abi> floatv<Abi> remainder(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> remainder(doublev<Abi> x, doublev<Abi> y);

template <class Abi> floatv<Abi> remquo(floatv<Abi> x, floatv<Abi> y, samesize<int, floatv<Abi>>* quo);
template <class Abi> doublev<Abi> remquo(doublev<Abi> x, doublev<Abi> y, samesize<int, doublev<Abi>>* quo);

template <class Abi> floatv<Abi> copysign(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> copysign(doublev<Abi> x, doublev<Abi> y);

template <class Abi> doublev<Abi> nan(const char* tagp);
template <class Abi> floatv<Abi> nanf(const char * tagp);
template <class Abi> ldoublev<Abi> nanl(const char * tagp);

template <class Abi> floatv<Abi> nextafter(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> nextafter(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> nextafter(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> nexttoward(floatv<Abi> x, ldoublev<Abi> y);
template <class Abi> doublev<Abi> nexttoward(doublev<Abi> x, ldoublev<Abi> y);
template <class Abi> ldoublev<Abi> nexttoward(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> fdim(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> fdim(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> fdim(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> fmax(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> fmax(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> fmax(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> fmin(floatv<Abi> x, floatv<Abi> y);
template <class Abi> doublev<Abi> fmin(doublev<Abi> x, doublev<Abi> y);
template <class Abi> ldoublev<Abi> fmin(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> floatv<Abi> fma(floatv<Abi> x, floatv<Abi> y, floatv<Abi> z);
template <class Abi> doublev<Abi> fma(doublev<Abi> x, doublev<Abi> y, doublev<Abi> z);
template <class Abi> ldoublev<Abi> fma(ldoublev<Abi> x, ldoublev<Abi> y, ldoublev<Abi> z);

template <class Abi> samesize<int, floatv<Abi>> fpclassify(floatv<Abi> x);
template <class Abi> samesize<int, doublev<Abi>> fpclassify(doublev<Abi> x);
template <class Abi> samesize<int, ldoublev<Abi>> fpclassify(ldoublev<Abi> x);

template <class Abi> mask<float, Abi> isfinite(floatv<Abi> x);
template <class Abi> mask<double, Abi> isfinite(doublev<Abi> x);
template <class Abi> mask<long double, Abi> isfinite(ldoublev<Abi> x);

template <class Abi> mask<float, Abi> isinf(floatv<Abi> x);
template <class Abi> mask<double, Abi> isinf(doublev<Abi> x);
template <class Abi> mask<long double, Abi> isinf(ldoublev<Abi> x);

template <class Abi> mask<float, Abi> isnan(floatv<Abi> x);
template <class Abi> mask<double, Abi> isnan(doublev<Abi> x);
template <class Abi> mask<long double, Abi> isnan(ldoublev<Abi> x);

template <class Abi> mask<float, Abi> isnormal(floatv<Abi> x);
template <class Abi> mask<double, Abi> isnormal(doublev<Abi> x);
template <class Abi> mask<long double, Abi> isnormal(ldoublev<Abi> x);

template <class Abi> mask<float, Abi> signbit(floatv<Abi> x);
template <class Abi> mask<double, Abi> signbit(doublev<Abi> x);
template <class Abi> mask<long double, Abi> signbit(ldoublev<Abi> x);

template <class Abi> mask<float, Abi> isgreater(floatv<Abi> x, floatv<Abi> y);
template <class Abi> mask<double, Abi> isgreater(doublev<Abi> x, doublev<Abi> y);
template <class Abi> mask<long double, Abi> isgreater(ldoublev<Abi> x, ldoublev<Abi> y);

template <class Abi> mask<float, Abi> isgreaterequal(floatv<Abi> x, floatv<Abi> y);
template <class Abi> mask<double, Abi> isgreaterequal(doublev<Abi> x, doublev<Abi> y);
template <class Abi> mask<long double, Abi> isgreaterequal(ldoublev<Abi> x, ldoublev<Abi> y);
Each listed function concurrently applies the indicated mathematical function component-wise. The results per component are not required to be binary equal to the application of the function which is overloaded for the element type. [Note: If a precondition of the indicated mathematical function is violated, the behavior is undefined. —end note] If abs() is called with an argument of type datapar<X, Abi> for which is_unsigned<X>::value is true, the program is ill-formed.

6.1.4 Class template mask

6.1.4.1 Class template mask overview

namespace std {
namespace experimental {

template <class T, class Abi> class mask {
public:
    using value_type = bool;
    using reference = implementation-defined; // see datapar::reference
    using datapar_type = datapar<T, Abi>;
    using size_type = size_t;
    using abi_type = Abi;

    static constexpr size_type size() noexcept;

    mask() = default;

    mask(const mask&) = default;
    mask(mask&&) = default;
    mask& operator=(const mask&) = default;
    mask& operator=(mask&&) = default;

    template <class Abi> mask<float, Abi> isless(floatv<Abi> x, floatv<Abi> y);
    template <class Abi> mask<double, Abi> isless(doublev<Abi> x, doublev<Abi> y);
    template <class Abi> mask<long double, Abi> isless(ldoublev<Abi> x, ldoublev<Abi> y);
    template <class Abi> mask<float, Abi> islessequal(floatv<Abi> x, floatv<Abi> y);
    template <class Abi> mask<double, Abi> islessequal(doublev<Abi> x, doublev<Abi> y);
    template <class Abi> mask<long double, Abi> islessequal(ldoublev<Abi> x, ldoublev<Abi> y);
    template <class Abi> mask<float, Abi> islessgreater(floatv<Abi> x, floatv<Abi> y);
    template <class Abi> mask<double, Abi> islessgreater(doublev<Abi> x, doublev<Abi> y);
    template <class Abi> mask<long double, Abi> islessgreater(ldoublev<Abi> x, ldoublev<Abi> y);
    template <class Abi> mask<float, Abi> isunordered(floatv<Abi> x, floatv<Abi> y);
    template <class Abi> mask<double, Abi> isunordered(doublev<Abi> x, doublev<Abi> y);
    template <class Abi> mask<long double, Abi> isunordered(ldoublev<Abi> x, ldoublev<Abi> y);
}
}
}
The class template \texttt{mask\langle T, Abi \rangle} is a one-dimensional smart array of booleans. The number of elements in the array is a constant expression, equal to the number of elements in \texttt{datapar\langle T, Abi \rangle}.

The resulting class shall be a complete type with deleted default constructor, deleted destructor, deleted copy constructor, and deleted copy assignment unless all of the following hold:

- The first template argument \( T \) is a cv-unqualified integral or floating-point type except \texttt{bool} (3.9.1 [basic.fundamental]).
- The second template argument \( Abi \) is an ABI tag so that \texttt{is\_abi\_tag\_v\langle Abi \rangle} is true.
- The \( Abi \) type is a supported ABI tag. It is supported if
  - \( Abi \) is \texttt{datapar\_abi::scalar}, or
  - \( Abi \) is \texttt{datapar\_abi::fixed\_size\langle N \rangle} with \( N \leq 32 \) or implementation-defined additional valid values for \( N \) (see 6.1.1.1 p.3).
It is implementation-defined whether a given combination of `T` and an implementation-defined ABI tag is supported. [Note: The intent is for implementations to decide on the basis of the currently targeted system. — end note]

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

```cpp
static constexpr size_type size() noexcept;
```

**Returns:** the number of boolean elements stored in objects of the given `mask<T, Abi>` type.

[Note: Implementations are encouraged to enable `static_cast` from/to (an) implementation-defined SIMD mask type(s). This would add one or more of the following declarations to class `mask`:

```cpp
explicit operator implementation-defined() const;
explicit datapar(const implementation-defined init);
```

— end note]

### 6.1.4.2 mask constructors [mask ctor]

```cpp
explicit mask(value_type) noexcept;
```

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

```cpp
template <class U> mask(const mask<U, datapar_abi::fixed_size<size()>>& x) noexcept;
```

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

**Effects:** Constructs an object of type `mask` where the `i`-th element equals `x[i]` for all `i ∈ [0, size())`.

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

**Effects:** Constructs an object where the `i`-th element is initialized to `mem[i]` for all `i ∈ [0, size())`.

**Requires:** `size()` is less than or equal to the number of values pointed to by `mem`.

**Requires:** If the `Flags` template parameter is of type `flags::vector_aligned_tag`, the pointer value represents an address aligned to `memory_alignment_v<mask>`. If the `Flags` template parameter is of type `flags::overaligned_tag<N>`, the pointer value represents an address aligned to `N`.

### 6.1.4.3 mask load function [mask.load]

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

**Effects:** Replaces the elements of the `mask` object such that the `i`-th element is assigned with `mem[i]` for all `i ∈ [0, size())`.

**Requires:** `size()` is less than or equal to the number of values pointed to by `mem`.

**Requires:** If the `Flags` template parameter is of type `flags::vector_aligned_tag`, the pointer value represents an address aligned to `memory_alignment_v<mask>`. If the `Flags` template parameter is of type `flags::overaligned_tag<N>`, the pointer value represents an address aligned to `N`. 
6.1.4.4 mask store function

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

1. **Effects:** Copies all `mask` elements as if `mem[i] = operator[](i)` for all `i ∈ [0, size())`.
2. **Requires:** `size()` is less than or equal to the number of values pointed to by `mem`.
3. **Requires:** If the `Flags` template parameter is of type `flags::vector_aligned_tag`, the pointer value represents an address aligned to `memory_alignment_v<mask>`. If the `Flags` template parameter is of type `flags::overaligned_tag<N>`, the pointer value represents an address aligned to `N`.

6.1.4.5 mask subscript operators

```cpp
reference operator[](size_type i);
```

1. **Requires:** The value of `i` is less than `size()`.
2. **Returns:** A temporary object of type `reference` (see 6.1.2.1 p.4) with the following effects:
3. **Effects:** The assignment, compound assignment, increment, and decrement operators of `reference` execute the indicated operation on the `i`-th element of the `mask` object.
4. **Effects:** Conversion to `value_type` returns a copy of the `i`-th element.
5. **Throws:** Nothing.

```cpp
value_type operator[](size_type i) const;
```

1. **Requires:** The value of `i` is less than `size()`.
2. **Returns:** A copy of the `i`-th element.
3. **Throws:** Nothing.

6.1.4.6 mask unary operators

```cpp
mask operator!() const noexcept;
```

1. **Returns:** A mask object with the `i`-th element set to the logical negation for all `i ∈ [0, size())`.

6.1.5 mask non-member operations

6.1.5.1 mask binary operators

```cpp
friend mask operator&&(const mask&, const mask&) noexcept;
friend mask operator||(const mask&, const mask&) noexcept;
friend mask operator&(const mask&, const mask&) noexcept;
friend mask operator|(const mask&, const mask&) noexcept;
friend mask operator^(const mask&, const mask&) noexcept;
```

49
Returns: A mask object initialized with the results of the component-wise application of the indicated operator.

6.1.5.2 mask compound assignment

friend mask& operator=(mask&, const mask&) noexcept;
friend mask& operator|=(mask&, const mask&) noexcept;
friend mask& operator^=(mask&, const mask&) noexcept;

Effects: Each of these operators performs the indicated operator component-wise on each of the corresponding elements of the arguments.

Returns: A reference to the first argument.

6.1.5.3 mask compares

friend mask operator==(const mask&, const mask&) noexcept;
friend mask operator!=(const mask&, const mask&) noexcept;

Returns: A mask object initialized with the results of the component-wise application of the indicated operator.

6.1.5.4 mask reductions

template<class T, class Abi> bool all_of(mask<T, Abi>) noexcept;

Returns: true if all boolean elements in the function argument equal true, false otherwise.

template<class T, class Abi> bool any_of(mask<T, Abi>) noexcept;

Returns: true if at least one boolean element in the function argument equals true, false otherwise.

template<class T, class Abi> bool none_of(mask<T, Abi>) noexcept;

Returns: true if none of the boolean element in the function argument equals true, false otherwise.

template<class T, class Abi> bool some_of(mask<T, Abi>) noexcept;

Returns: true if at least one of the boolean elements in the function argument equals true and at least one of the boolean elements in the function argument equals false, false otherwise.

template<class T, class Abi> int popcount(mask<T, Abi>) noexcept;

Returns: The number of boolean elements that are true.

template<class T, class Abi> int find_first_set(mask<T, Abi> m);
6 Wording

Requires: \text{any\_of}(m) \text{ returns } \text{true}

Returns: The lowest element index \text{i} where \text{m}[i] == \text{true}.

\textbf{template <class T, class Abi> int find_last_set(mask<T, Abi> m)};

Requires: \text{any\_of}(m) \text{ returns } \text{true}

Returns: The highest element index \text{i} where \text{m}[i] == \text{true}.

\textbf{template <class T, class Abi> bool all_of (implementation-defined) noexcept ;}
\textbf{template <class T, class Abi> bool any_of (implementation-defined) noexcept ;}
\textbf{template <class T, class Abi> bool none_of (implementation-defined) noexcept ;}
\textbf{template <class T, class Abi> bool some_of (implementation-defined) noexcept ;}
\textbf{template <class T, class Abi> int popcount (implementation-defined) noexcept ;}
\textbf{template <class T, class Abi> int find_first_set (implementation-defined) noexcept ;}
\textbf{template <class T, class Abi> int find_last_set (implementation-defined) noexcept ;}

Remarks: The functions shall not participate in overload resolution unless the argument is of type \text{bool}.

Returns: \text{all\_of} and \text{any\_of} return their arguments; \text{none\_of} returns the negation of its argument; \text{some\_of} returns \text{false}; \text{popcount} returns the integral representation of its argument; \text{find\_first\_set} and \text{find\_last\_set} return 0.

6.1.5.5 Masked assignment  

\textbf{template <class T, class A> where_expression <mask<T, A>, datapar<T, A>> where (}
\textbf{\text{const typename datapar<T, A>::mask_type & k, datapar<T, A>& v) noexcept ;}
\textbf{template <class T, class A> const where_expression <mask<T, A>, const datapar<T, A>> where (}
\textbf{\text{const typename datapar<T, A>::mask_type & k, const datapar<T, A>& v) noexcept ;}
\textbf{template <class T, class A> where_expression <mask<T, A>, mask<T, A>> where (}
\textbf{\text{const remove_const_t <mask<T, A>> & k, mask<T, A>& v) noexcept ;}
\textbf{template <class T, class A> const where_expression <mask<T, A>, const mask<T, A>> where (}
\textbf{\text{const remove_const_t <mask<T, A>> & k, const mask<T, A>& v) noexcept ;}

Returns: An object of type \text{where\_expression} (see 6.1.1.3) initialized with the predicate \text{k} and the value reference \text{v}.

\textbf{template <class T> where_expression<bool, T> where(implementation-defined k, T & v) noexcept ;}

Remarks: The function shall not participate in overload resolution unless

- \text{T} is neither a \text{datapar} nor a \text{mask} instantiation, and
- the first argument is of type \text{bool}.

Returns: An object of type \text{where\_expression} (see 6.1.1.3) initialized with the predicate \text{k} and the value reference \text{v}.
7 WIDENING CAST

The following presents an option for extending the above wording with a cast function that only allows “lossless” conversions of the element type. I present three options: The first option requires a `datapar` type as cast type argument. This choice provides control over the returned ABI tag. The second option requires an element type as cast type argument. This choice uses the typically much simpler cast argument. The third option works with either.

Note that `static_datapar_cast`, which is defined in 6.1.3.5, uses an element type as cast type argument. Thus, “Option 2” is equivalent to the `static_datapar_cast` function, differing only in the requirement that the conversion must be “lossless”.

Examples:

```cpp
using floatv = native_datapar<float>;
floatv x = ...;

// Option 1:
auto a = datapar_cast<fixed_size_datapar<double, floatv::size>>() (x);

// Option 2:
auto b = datapar_cast<double>(x);

// Option 3:
auto c = datapar_cast<fixed_size_datapar<double, floatv::size>>() (x);
auto d = datapar_cast<double>(x);
```

7.1 OPTION 1: DATAPAR TEMPLATE ARGUMENT

Add to the synopsis in 6.1.1:

```cpp
template <class V, class T, class A> V datapar_cast(const datapar<T, A>&);
```

Append to 6.1.3.5:

```cpp
template <class V, class T, class A> V datapar_cast(const datapar<T, A>& x);
```

Remarks: The function shall not participate in overload resolution unless
- `is_datapar_v<V>`,
- `V::size()` is equal to `datapar<T, A>::size()`,
- and every possible value of type `T` can be represented with type `datapar::value_type`. 52
7.2 option 2: element type template argument

Add to the synopsis in 6.1.1:

```cpp
template <class T, class U, class A>
datapar<T, /* see below */ datapar<const datapar<U, A>>>()
datapar_cast(const datapar<U, A>&);
```

Append to 6.1.3.5:

```cpp
template <class T, class U, class A>
datapar<T, /* see below */ datapar<const datapar<U, A>>>()
datapar_cast(const datapar<U, A>&);
```

Remarks: The function shall not participate in overload resolution unless every possible value of type U can be represented with type T.

Remarks: The return type is `datapar<T, A>` if either U and T are equal or U and T are integral types that only differ in signedness. Otherwise, the return type is `datapar<T, datapar_abi::fixed_size<datapar<U, A>::size>>()`. Returns: A `datapar` object with the i-th element initialized to `static_cast<T>(x[i])`. Throws: Nothing.

7.3 option 3: both

Add to the synopsis in 6.1.1:

```cpp
template <class T, class U, class A>
/* see below */ datapar_cast<const datapar<U, A>>();
```

Append to 6.1.3.5:

```cpp
template <class T, class U, class A>
/* see below */ datapar_cast<const datapar<U, A>>();
```
8 Split & Concat

The following presents an option for extending the above wording with two cast functions. The split function allows to turn one `datapar` or `mask` object into two or more `datapar`/`mask` objects with smaller element counts. The concat function allows to combine multiple `datapar` or `mask` objects into a single `datapar`/`mask` object consisting of all the input elements.

Here is a simple example for `split` and `concat`:

```cpp
fixed_size_datapar<float, 12> x = ...;
auto [a, b] = split<8, 4>(x);
// e.g. on x86 you'd get: decltype(a) == datapar<float, avx>
// and: decltype(b) == datapar<float, sse>
x = concat(a + 1, b + 2);
```

The `abi_for_size_t` choice below could also be changed to use the `fixed_size` ABI tag unconditionally. Since a `fixed_size` `datapar` is implicitly convertible to a `non-fixed_size` `datapar` type with equal `size()`, this may be the more generic solution. I have a slight preference for `abi_for_size_t`, since it more naturally supports the pattern of splitting a `fixed_size` object into several native `datapar` objects. That pattern is not fully covered by the second `split` variant (e.g. consider the example above).

The same discussion of `abi_for_size_t` vs. `fixed_size` is valid for the return type of `concat`.

Add to the synopsis in 6.1.1:

```cpp
template <size_t... Sizes, class T, class A>
tuple<datapar<T, abi_for_size_t<Sizes>>...> split(const datapar<T, A>&);
```

```cpp
template <size_t... Sizes, class T, class A>
tuple<mask<T, abi_for_size_t<Sizes>>...> split(const mask<T, A>&);
```

```cpp
template <class V, class T, class A>
```
Append to 6.1.3.5:

1. Remarks: These functions shall not participate in overload resolution unless the sum of all values in the Sizes pack is equal to datapar_size_v<T, A>.

2. Returns: A tuple of datapar/mask objects with the i-th datapar/mask element of the j-th tuple element initialized to the value of the element in x with index i + partial sum of the first j values in the Sizes pack.

3. Remarks: These functions shall not participate in overload resolution unless
   - is_datapar_v<V> for the first signature / is_mask_v<V> for the second signature,
   - and datapar_size_v<T, A> is an integral multiple of V::size().

4. Returns: An array of datapar/mask objects with the i-th datapar/mask element of the j-th array element initialized to the value of the element in x with index i + j * V::size().

5. Returns: A datapar/mask object initialized with the concatenated values in the xs pack of datapar/mask objects. The i-th datapar/mask element of the j-th parameter in the xs pack is copied to the return value’s element with index i + partial sum of the size() of the first j parameters in the xs pack.
9 Discussion

9.1 Member Types

The member types may not seem obvious. Rationales:

value_type
In the spirit of the value_type member of STL containers, this type denotes the logical type of the values in the vector.

reference
Used as the return type of the non-const scalar subscript operator.

mask_type
The natural mask type for this datapar instantiation. This type is used as return type of compares and write-mask on assignments.

datapar_type
The natural datapar type for this mask instantiation.

size_type
Standard member type used for size() and operator[].

abi_type
The Abi template parameter to datapar.

9.2 Conversions

The datapar conversion constructor only allows implicit conversion from datapar template instantiations with the same Abi type and compatible value_type. Discussion in SG1 showed clear preference for only allowing implicit conversion between integral types that only differ in signedness. All other conversions could be implemented via an explicit conversion constructor. The alternative (preferred) is to use datapar_cast consistently for all other conversions.

After more discussion on the LEWG reflector, in Issaquah, and between me and Jens, we modified conversions to be even more conservative. No implicit conversion will ever allow a narrowing conversion of the element type (and signed - unsigned is narrowing in both directions).
9.3 broadcast constructor

The `datapar` broadcast constructor is not declared explicit to ease the use of scalar prvalues in expressions involving data-parallel operations. The operations where such a conversion should not be implicit consequently need to use SFINAE / concepts to inhibit the conversion.

Experience from Vc shows that the situation is different for `mask`, where an implicit conversion from `bool` typically hides an error. (Since there is little use for broadcasting `true` or `false`.)

9.4 aliasing of subscript operators

The subscript operators return an rvalue. The const overload returns a copy of the element. The non-const overload returns a smart reference. This reference behaves mostly like an lvalue reference, but without the requirement to implement assignment via type punning. At this point the specification of the smart reference is very conservative / restrictive: The reference type is neither copyable nor movable. The intention is to avoid users to program like the operator returned an lvalue reference. The return type is significantly larger than an lvalue reference and harder to optimize when passed around. The restriction thus forces users to do element modification directly on the `datapar` / `mask` objects.

Guidance from SG1 at JAX 2016:

Poll: Should subscript operator return an lvalue reference?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>10</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Poll: Should subscript operator return a “smart reference”?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

9.5 compound assignment

The semantics of compound assignment would allow less strict implicit conversion rules. Consider `datapar<int>() *= double();` the corresponding binary multiplication operator would not compile because the implicit conversion to `datapar<double>` is non-portable. Compound assignment, on the other hand, implies an implicit conversion back to the type of the expression on the left of the assignment operator. Thus, it is possible to define compound operators that execute the operation correctly on the promoted type without sacrificing portability. There are two arguments for not relaxing the rules for compound assignment, though:
1. Consistency: The conversion of an expression with compound assignment to a binary operator might make it ill-formed.

2. The implicit conversion in the `int * double` case could be expensive and unintended. This is already a problem for builtin types, where many developers multiply `float` variables with `double` prvalues, though.

### 9.6 Return Type of Masked Assignment Operators

The assignment operators of the type returned by `where(mask, datapar)` could return one of:

- A reference to the `datapar` object that was modified.
- A temporary `datapar` object that only contains the elements where the `mask` is `true`.
- A reference to the `where_expression` object.
- Nothing (i.e. `void`).

My first choice was a reference to the modified `datapar` object. However, then the statement `(where(x < 0, x) *= -1) += 2` may be surprising: it adds 2 to all vector entries, independent of the mask. Likewise, `y += (where(x < 0, x) *= -1)` has a possibly confusing interpretation because of the `mask` in the middle of the expression.

Consider that write-masked assignment is used as a replacement for `if`-statements. Using `void` as return type therefore is a more fitting choice because `if`-statements have no return value. By declaring the return type as `void` the above expressions become ill-formed, which seems to be the best solution for guiding users to write maintainable code and express intent clearly.

### 9.7 Fundamental SIMD Type or Not?

#### 9.7.1 The Issue

There was substantial discussion on the reflectors and SG1 meetings over the question whether C++ should define a fundamental, native SIMD type (let us call it `fundamental<T>`) and additionally a generic data-parallel type which supports an arbitrary number of elements (call it `arbitrary<T, N>`). The alternative to defining both types is to only define `arbitrary<T, N = default_size<T>>`, since it encompasses the `fundamental<T>` type.

With regard to this proposal this second approach would add a third template parameter to `datapar` and `mask` as shown in Listing 1.
9 Discussion

1 template <class T, size_t N = datapar_size_v<T, datapar_abi::compatible<T>>,
2 class Abi = datapar_abi::compatible<T>>
3 class datapar;

Listing 1: Possible declaration of the class template parameters of a datapar class with arbitrary width.

9.7.2 Standpoints

The controversy is about how the flexibility of a type with arbitrary \( N \) is presented to the users. Is there a (clear) distinction between a "fundamental" type with target-dependent (i.e. fixed) \( N \) and a higher-level abstraction with arbitrary \( N \) which can potentially compile to inefficient machine code? Or should the C++ standard only define arbitrary and set it to a default \( N \) value that corresponds to the target-dependent \( N \). Thus, the default \( N \), of arbitrary would correspond to fundamental.

It is interesting to note that \( \text{arbitrary}<T, 1> \) is the class variant of \( T \). Consequently, if we say there is no need for a fundamental type then we could argue for the deprecation of the built-in arithmetic types, in favor of \( \text{arbitrary}<T, 1> \). [Note: This is an academic discussion, of course. — end note]

The author has implemented a library where a clear distinction is made between fundamental\( <T, \text{Abi}> \) and arbitrary\( <T, N> \). The documentation and all teaching material says that the user should program with fundamental. The arbitrary type should be used in special circumstances, or wherever fundamental works with the arbitrary type in its interfaces (e.g. for gather & scatter or the ldexp & frexp functions).

9.7.3 Issues

The definition of two separate class templates can alleviate some source compatibility issues resulting from different \( N \) on different target systems. Consider the simplest example of a multiplication of an int vector with a float vector:

\[
\text{arbitrary}<\text{float}>() \ast \text{arbitrary}<\text{int}>(); \quad /\!/ \text{compiles for some targets, fails for others}
\text{fundamental}<\text{float}>() \ast \text{fundamental}<\text{int}>(); \quad /\!/ \text{never compiles, requires explicit cast}
\]

The datapar\( <T> \) operators are specified in such a way that source compatibility is ensured. For a type with user definable \( N \), the binary operators should work slightly different with regard to implicit conversions. Most importantly, arbitrary\( <T, N> \) solves the issue of portable code containing mixed integral and floating-point values. A user would typically create aliases such as:

\[
\text{using floatvec = datapar<float>};
\text{using intvec = arbitrary<int, floatvec::size()>};
\]
using doublevec = arbitrary<int, floatvec::size>();

Objects of types floatvec, intvec, and doublevec will work together, independent of the target system.

Obviously, these type aliases are basically the same if the \( N \) parameter of arbitrary has a default value:

using floatvec = arbitrary<float>;
using intvec = arbitrary<int, floatvec::size()>;
using doublevec = arbitrary<int, floatvec::size()>>;

The ability to create these aliases is not the issue. Seeing the need for using such a pattern is the issue. Typically, a developer will think no more of it if his code compiles on his machine. If `arbitrary<float>() * arbitrary<int>()` just happens to compile (which is likely), then this is the code that will get checked in to the repository. Note that with the existence of the fundamental class template, the \( N \) parameter of the arbitrary class would not have a default value and thus force the user to think a second longer about portability.

9.7.4 progress

SG1 Guidance at JAX 2016:
Poll: Specify datapar width using ABI tag, with a special template tag for fixed size.

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Poll: Specify datapar width using \(<T, N, abi>\), where abi is not specified by the user.

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

At the Jacksonville meeting, SG1 decided to continue with the `datapar<T, Abi>` class template, with the addition of a new Abi type that denotes a user-requested number of elements in the vector (`datapar_abi::fixed_size<N>`). This has the following implications:

- There is only one class template with a common interface for fundamental and arbitrary (fixed_size) vector types.

- There are slight differences in the conversion semantics for datapar types with the fixed_size Abi type. This may look like the vector<bool> mistake all over again. I’ll argue below why I believe this is not the case.

- The fundamental class instances could be implemented in such a way that they do not guarantee ABI compatibility on a given architecture where translation
units are compiled with different compiler flags (for micro-architectural differences).

- The fixed_size class instances, on the other hand, could be implemented to be the ABI stable types (if an implementation thinks this is an important feature). In implementation terms this means that fundamental types are allowed to be passed via registers on function calls. fixed_size types can be implemented in such a way that they are only passed via the stack, and thus an implementation only needs to ensure equal alignment and memory representation across TU borders for a given T, N.

The conversion differences between the fundamental and fixed_size class template instances are the main motivation for having a distinction (cf. discussion above). The differences are chosen such that, in general, fundamental types are more restrictive and do not turn into fixed_size types on any operation that involves no fixed_size types. Operations of fixed_size types allow easier use of mixed precision code as long as no elements need to be dropped / generated (i.e. the number of elements of all involved datapar objects is equal or a builtin arithmetic type is broadcast).

Examples:

1. Mixed int–float operations

```cpp
using floatv = datapar<float>; // native ABI
using float_sized_abi = datapar ABI::fixed_size<floatv::size>();
using intv = datapar<int, float_sized_abi>;
auto x = floatv() + intv();
```

Line 5 is well-formed: It states that \( N = \text{floatv::size()} \) additions shall be executed concurrently. The type of x is datapar<float>, because it stores \( N \) elements and both types intv and floatv are implicitly convertible to datapar<float>. Line 6 is also well-formed because implicit conversion from datapar<T, Abi> to datapar<U, datapar_abi::fixed_size<N>> is allowed whenever \( N == \text{datapar<T, Abi>::size()} \).

2. Native int vectors

```cpp
using intv = datapar<int>; // native ABI
using int_sized_abi = datapar ABI::fixed_size<intv::size>();
using floatv = datapar<float, int_sized_abi>;
auto x = floatv() + intv();
```
int v y = floatv() + intv();

Line 5 is well-formed: It states that $N (= \text{intv::size()}$) additions shall be executed concurrently. The type of $x$ is `datapar<float_v, int_sized_abi>` (i.e. `floatv`) and never `datapar<float>`, because...

... the Abi types of `intv` and `floatv` are not equal.

... either `datapar<float>::size() != N` or `intv` is not implicitly convertible to `datapar<float>`.

... the last rule for `commonabi(V0, V1, T)` sets the Abi type to `int_sized_abi`.

Line 6 is also well-formed because implicit conversion from `datapar<T, datapar_abi::fixed_size<N>>` to `datapar<U, Abi>` is allowed whenever $N == \text{datapar<U, Abi>::size()}$.

9.8 NATIVE HANDLE

The presence of a `native_handle` function for accessing an internal data member such as e.g. a vector builtin or SIMD intrinsic type is seen as an important feature for adoption in the target communities. Without such a handle the user is constrained to work within the (limited) API defined by the standard. Many SIMD instruction sets have domain-specific instructions that will not easily be usable (if at all) via the standardized interface. A user considering whether to use `datapar` or a SIMD extension such as vector builtins or SIMD intrinsics might decide against `datapar` just for fear of not being able to access all functionality.¹

I would be happy to settle on an alternative to exposing an lvalue reference to a data member. Consider implementation-defined support casting (static_cast?) between `datapar` and non-standard SIMD extension types. My understanding is that there could not be any normative wording about such a feature. However, I think it could be useful to add a non-normative note about making static_cast(?) able to convert between such non-standard extensions and `datapar`.

Guidance from SG1 at Oulu 2016:

Poll: Keep `native_handle` in the wording?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

¹ Whether that’s a reasonable fear is a different discussion.
SIMD loads and stores require at least an alignment option. This is in contrast to implicit loads and stores present in C++, where alignment is always assumed. Many SIMD instruction sets allow more options, though:

- Streaming, non-temporal loads and stores
- Software prefetching

In the Vc library I have added these as options in the load store flag parameter of the `load` and `store` functions. However, non-temporal loads & stores and prefetching are also useful for the existing builtin types. I would like guidance on this question: should the general direction be to stick to only alignment options for datapar loads and stores?

The other question is on the default of the load and store flags. Some argue for setting the default to aligned, as that's what the user should always aim for and is most efficient. Others argue for unaligned since this is safe per default. The Vc library before version 1.0 used aligned loads and stores per default. After the guidance from SG1 I changed the default to unaligned loads and stores with the Vc 1.0 release. Changing the default is probably the worst that could be done, though. For Vc 2.0 I will drop the default.

For datapar I prefer no default:

- This makes it obvious that the API has the alignment option. Users should not just take the default and think no more of it.
- If we decide to keep the load constructor, the alignment parameter (without default) nicely disambiguate the load from the broadcast.
- The right default would be application/domain/experience specific.
- Users can write their own load/store wrapper functions that implement their chosen default.

Guidance from SG1 at Oulu 2016:
Poll: Should the interface provide a way to specify a number for over-alignment?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Poll: Should loads and stores have a default load/store flag?

<table>
<thead>
<tr>
<th>SF</th>
<th>F</th>
<th>N</th>
<th>A</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

The discussion made it clear that we only want to support alignment flags in the load and store operations. The other functionality is orthogonal.

9.10 *Unary Minus Return Type*

The return type of `datapar<T, Abi>::operator-()` is `datapar<T, Abi>`. This is slightly different to the behavior of the underlying element type `T`, if `T` is an integral type of lower integer conversion rank than `int`. In this case integral promotion promotes the type to `int` before applying unary minus. Thus, the expression `-T()` is of type `int` for all `T` with lower integer conversion rank than `int`. This is widening of the element size is likely unintended for SIMD vector types.

Fundamental types with integer conversion rank greater than `int` are not promoted and thus a unary minus expression has unchanged type. This behavior is copied to element types of lower integer conversion rank for `datapar`.

There may be one interesting alternative to pursue here: We can make it ill-formed to apply unary minus to unsigned integral types. Anyone who wants to have the modulo behavior of a unary minus could still write `0u - x`.

9.11 *Max Fixed Size*

In Kona, LEWG asked why `max_fixed_size` is not dependent on `T`. After some consideration I am convinced that the correct solution is to make `max_fixed_size` a variable template, dependent on `T`.

The reason to restrict the number of elements `N` in a fixed-size `datapar` type at all, is to inhibit misuse of the feature. The intended use of the fixed-size ABI, is to work with a number of elements that is somewhere in the region of the number of elements that can be processed efficiently concurrently in hardware. Implementations may want to use recursion to implement the fixed-size ABI. While such an implementation can, in theory, scale to any `N`, experience shows that compiler memory usage and compile times grow significantly for “too large” `N`. The optimizer also has a hard time to optimize register / stack allocation optimally if `N` becomes “too large”. Unsuspecting users might not think of such issues and try to map their complete problem to a single `datapar` object. Allowing implementations to restrict `N` to a value that they can and want to support thus is useful for users and implementations. The value itself should not be prescribed by the standard as it is really just a QoI issue.
However, why should the user be able to query the maximum $N$ supported by the implementation?

- In principle, a user can always determine the number using SFINAE to find the maximum $N$ that he can still instantiate without substitution failure. Not providing the number thus provides no “safety” against “bad usage”.

- A developer may want to use the value to document assumptions / requirements about the implementation, e.g. with a static assertion.

- A developer may want to use the value to make code portable between implementations that use a different `max_fixed_size`.

Making the `max_fixed_size` dependent on $T$ makes sense because most hardware can process a different number of elements in parallel depending on $T$. Thus, if an implementation wants to restrict $N$ to some sensible multiple of the hardware capabilities, the number must be dependent on $T$.

In Kona, LEWG also asked whether there should be a provision in the standard to ensure that a native `datapar` of 8-bit element type is convertible to a fixed-size `datapar` of 64-bit element type. It was already there (6.1.1 p.3: “for every supported `datapar<T, A>` (see 6.1.2.1 p.2), where `A` is an implementation-defined ABI tag, $N = \text{datapar<T, A>::size()}$ must be supported”). Note that this does not place a lower bound on `max_fixed_size`. The wording allows implementations to support values for fixed-size `datapar` that are larger than `max_fixed_size`. I.e. $N \leq \text{max_fixed_size}$ works; whether $N > \text{max_fixed_size}$ works is unspecified.

10 FEATURE DETECTION MACROS

For the purposes of SD-6, feature detection initially will be provided through the shipping vehicle (TS) itself. For a standalone feature detection macro, I recommend `__cpp_lib_datapar`. If LEWG decides to rename the `datapar` class template, the feature detection macro needs to be renamed accordingly.

ACKNOWLEDGEMENTS

- This work was supported by GSI Helmholtzzentrum für Schwerionenforschung and the Hessian LOEWE initiative through the Helmholtz International Center for FAIR (HIC for FAIR).
• Jens Maurer contributed important feedback and suggestions on the API. Thanks also for presenting the paper in Kona 2017 and Toronto 2017.

• Thanks to Hartmut Kaiser for presenting in Issaquah 2016.

• Geoffrey Romer did a very helpful review of the wording.

Bibliography


