Document Number: DNMSOR0
Date: 2023-02-13
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: EWG-I, EWG
Target: C++23

Non-Member Subscript Operator

ABSTRACT

Currently the subscript operator can only be overloaded as a member function. This paper explores use-cases for a non-member overload and proposes to extend the standard accordingly.

CONTENTS

1 PROBLEM

The subscript operator is currently only overloadable as a member. Most other operators allow non-member overloads. That the subscript operator does not allow non-member overloads seems arbitrary, potentially motivated by a conservative approach only to add features when the use-cases are clear. Therefore, for the rest of this section I will give examples for non-member subscript operators.

In general, non-member subscript is needed / preferable

- if modification of the container type is not possible, or
- if the container type should not depend on the index type (e.g. `#include <vector>` should not implicitly `#include <simd>`), or
- if the implementation of the subscript operation is a detail of the index type rather than the container type.

1.1 VECTOR GATHER/SCATTER ON STD CONTAINERS AND C-ARRAYS

The use-case that got me started was the design of a gather/scatter API for SIMD vector types. Consider a function `f` which modifies an element at a given index:

```
void f(std::vector<int>& data, int i) {
  int x = data[i];
  // do something with x
  data[i] = x;
}
```

Now, let us generalize `i` from one index value to multiple indices, limiting the discussion to `std::simd`[1] for the multi-index type. Obviously, `simd` objects are not convertible to `size_t` and thus the existing subscript operator is not viable. We therefore need a new member `operator[]` overload in `std::vector` with a `simd` subscript argument.

While there's a non-zero chance for the standard library to adopt `simd` integration throughout the library, it is unlikely that non-std containers will do so. Furthermore, the gather and scatter operations on subscripting are really a detail of `simd`, rather than `vector`. The necessary interface to `vector` and all contiguous ranges already exists with `std::ranges::data`. Last but not least, a hidden friend `operator[]` inside `simd` would allow a single generic implementation for all contiguous ranges and all supported `simd` element types. Without non-member overloads the implementation must be copied into every type. (I.e. without non-member `operator[]` generic programming in C++ has a hole.)

1 Assuming `std::experimental::simd` becomes `std::simd` once [**P1928R0**] gains traction again.

The alternative to adding additional member `operator[](simd)` overloads is to rely on a named function instead. This is the most likely outcome if this paper is not adopted. Tony Table 1 shows the significant improvement in generic code if `simd` subscripts are available.

before

```cpp
template<std::ranges::contiguous_range Range,
         class Idx>
void f(Range& data, Idx i) {
  std::conditional_t<std::is_simd_v<Idx>,
                     std::simd<int>, int> x;
  if constexpr (std::is_simd_v<Idx>)
    x = std::simd_gather(std::ranges::data(data), i);
  else
    x = data[i];
  // do something with x
  if constexpr (std::is_simd_v<Idx>)
    std::simd_scatter(std::ranges::data(data), i, x);
  else
    data[i] = x;
}
```

with DNMSOR0

```cpp
template<std::ranges::contiguous_range Range,
         class Idx>
void f(Range& data, Idx i) {




  auto x = std::as_const(data)[i];
  // do something with x



  data[i] = x;
}
```

Tony Table 1: SIMD gather from a `std::vector`

Note that the `simd` subscript expression intuitively[2] asks for `simd<int>::size()` single-element loads from / stores to `data` at `data[i[0]]`, `data[i[1]]`, `data[i[2]]`, ….

1.1.1 Tangent: Reference class decay

The need for `as_const` in Tony Table 1 is unfortunate. The non-const overload of a subscript has to return a reference class type that is able to perform the scatter operation on assignment. Without `as_const` the type of `x` would be the reference class. Since we don't want to have a reference — we wrote `auto` not `auto&` — we have to ensure the `const` overload is used. Note the inconsistency with regard to a function `f` that returns `int&`: the type of `auto x = f()` isn't a reference but a copy of the value (`decay_t<int&>` is `int`). It would be great to get language support to define user-defined reference wrappers with decay semantics. See [**PTODO**].

1.2 Vector load/store subscripts

Analogue to SIMD scatter & gather subscripting, SIMD loads and stores can be implemented as subscripting into a contiguous range. However, instead of a single index a SIMD load/store is an access via multiple contiguous indices. This key difference must be reflected in the index type. A `simd` index type therefore represents a contiguous index range (similar to a `ranges::iota_view<size_t>`). In addition, a `simd` index type could determine the `simd` ABI tag, thus avoiding reference proxy

2 All operators on data-parallel types act element-wise.

types when subscripting `const` ranges. The availability of SIMD loads and stores via `simd` index types improves generality and readability significantly.

Tony Table 2 shows a simple loop (i.e. without epilogue) over a given range of `floats` incrementing every value in the range by 1. Without non-member `operator[]` overloads, there is little value

before

```
void f(std::ranges::contiguous_range<float>
       auto& data)
{
  using V = std::simd<float>;
  for (std::size_t i = 0;
         i + V::size() <= data.size();
         i += V::size())
  {
    const float* ptr
      = std::ranges::data(data) + i;
    V v(ptr);
    v += 1;
    v.copy_to(ptr);
  }
}
```

with DNMSOR0

```
void f(std::ranges::contiguous_range<float>
       auto& data)
{

  for (std::simd<float>::index_type i = 0;
         i < data.size(); ++i)

  {
    data[i] += 1;




  }
}
```

Tony Table 2: SIMD loop over a contiguous range

in providing a `simd` index type. Thus, code would have to resort to the low-level load and store operations of the `simd` type (left column in Tony Table 2). However, with suitable `operator[]` overloads, the `simd` index type allows concise, readable code which is also generic wrt. the index type[3] (right column in Tony Table 2).

1.3 VECTOR GATHER/SCATTER OR LOAD/STORE ON PROGRAM-DEFINED RANGES

As noted above, the SIMD gather, scatter, load, and store operations are implementation details of `simd` rather than the ranges where the memory access happens. If subscript operators for SIMD were implemented in some ranges / containers (as members) the support would be inconsistent / patchy. It is much more useful to define the subscript operators to apply to all ranges modelling `contiguous_range`, which *requires a non-member `operator[]`* (preferably as hidden friend of the index type). As a consequence we'd see seamless integration of program-defined ranges integrate with `simd`.

1.4 MULTI-INDEX SUBSCRIPTS

The subscript overloads described in Section 1.1 and Section 1.2 are just as useful for enabling non-SIMD multi-index subscripts. For example, a user could add a `vector<T>::operator[]` overload for a program-defined index type that returns a `span`. Especially for contiguous ranges, which expose

3 except for handling the epilogue in case the range isn't prepared have a matching size

an array of data via `std::ranges::data`, it is a sensible (better) implementation choice to tie the "not `size_t`" `operator[]`s to the index type(s). This requires non-member `operator[]` overloads.

1.5 SUBSCRIPTING WITH SCOPED ENUMERATIONS

The use of scoped enumerations can be cumbersome without the ability to use some of the operators that "just work" for unscoped enumerations. Therefore, it is really useful that operators can be overloaded for scoped enumerations, effectively providing an opt-in for operators. However, `operator[]` stands out. There's no way to allow subscripting with scoped enumerations unless the necessary `operator[]` can be added as a member. That's typically not going to happen, especially for standard library types like `array` and `vector`.

Allowing non-member `operator[]` fills that hole. See Listing 1 for an example.

1.6 COMPATIBLITY LAYER

After P2128 was adopted for C++23, providing multi-dimensional subscript operators for linear algebra (linalg) types (e.g. matrices) will be possible. It is to be expected that some linalg-libaries will provide these subscript operators, but it is unlikely that every type where multi-dimensional indexing is possible will be updated. Therefore, a non-member `operator[]` overload would allow programs to modernize the interface of such types without patching 3rd-party library types (e.g. Listing 2). This is similar to program-defined `make_unique` in C++11 sources.

2 IMPLEMENTATION

Implementing non-member subscript overloads for GCC was as simple as removing two lines of code (two special cases) before P2128 was implemented. After P2128 slightly more work was necessary because operators could only have up to three arguments before P2128 and the new subscript operator code did not look for non-member operators

3 WORDING

The wording for non-member subscript operators can be adopted from the wording for binary operators (§13.5.2 [over.binary]):

§13.5.5 [over.sub] operator[] shall be implemented either by a non-static member function with ~~exactly~~ one parameter or by a non-member function with two parameters. It implements the subscripting syntax

postfix-expression [*expression*]

or

postfix-expression [*braced-init-list*]

```cpp
enum class Color : char {
  Red,
  Green,
  Blue
};

struct Hsv {
  short hue, sat, val;
};

using ArgbArray = std::array<std::uint32_t, 3>;
using HsvArray = std::array<Hsv, 3>;

const ArgbArray ArgbValue = {{
  0xFFFF0000,
  0xFF00FF00,
  0xFF0000FF,
}};

const HsvArray HsvValue = {{
  {0, 1000, 1000},
  {120, 1000, 1000},
  {240, 1000, 1000}
}};

auto rgb = ArgbValue[Color::Red];  // ERROR
auto hsv = HsvValue[Color::Green]; // ERROR

constexpr inline std::uint32_t operator[](const ArgbArray &values, Color c)
{ return values[static_cast<std::size_t>(c)]; }

constexpr inline Hsv operator[](const HsvArray &values, Color c)
{ return values[static_cast<std::size_t>(c)]; }

auto rgb = ArgbValue[Color::Red];  // now it works
auto hsv = HsvValue[Color::Green]; // now it works
```

Listing 1: Scoped enumerations sometimes should be able to opt-in to being used as subscript index

```
// linalg.h - external code, cannot be modified
namespace linalg {
  struct matrix {
    float & operator()(size_t row, size_t col);
  };
}

// utils.h - local utility header
namespace linalg {
  inline float& operator[](matrix& self, size_t row, size_t col) {
    return self(row, col);
  }
}

linalg::matrix mat = ...;
mat[3, 3] = 1;
```

Listing 2: Extending a type with missing multi-dimensional subscript operator

Thus, a subscripting expression `x[y]` is interpreted as either `x.operator[](y)` ~~for a class object `x` of type `T` if `T::operator[](T1)` exists~~or `operator[](x, y)` ~~and if the operator is selected as the best match function by the overload resolution mechanism (13.3.3)~~. If both forms of the operator function have been declared, the rules in 13.3.1.2 determine which, if any, interpretation is used. [*Example:*

```
struct X {
  Z operator[](std::initializer_list<int>);
};
X x;
x[{1,2,3}] = 7;          // OK: meaning x.operator[]({1,2,3})
int a[10];
a[{1,2,3}] = 7;          // error: built-in subscript operator
```

— *end example*]

In addition, Table 11 in §13.3.1.2 [over.match.oper] mentions the subscript operator and needs to be adjusted:

§13.3.1.2 [over.match.oper] Table 11 — Relationship between operator and function call notation

Subclause	**Expression**	**As member function**	**As non-member function**
13.5.1	@a	(a).operator@()	operator@(a)
13.5.2	a@b	(a).operator@(b)	operator@(a, b)
13.5.3	a=b	(a).operator=(b)	
13.5.5	a[b]	(a).operator[](b)	operator[](a, b)
13.5.6	a->	(a).operator->()	
13.5.7	a@	(a).operator@(0)	operator@(a, 0)

The following paragraph in [over.match.oper] needs to discuss the subscript operator in addition to unary and binary operators:

---

§13.3.1.2 [over.match.oper] For a unary operator `@` with an operand of a type whose cv-unqualified version is `T1`, for a binary operator `@` with a left operand of a type whose cv-unqualified version is `T1` and a right operand of a type whose cv-unqualified version is `T2`, and for a subscript operator (where in the following `@` identifies `[]`) with a left operand of a type whose cv-unqualified version is `T1` and subscript operand of a type whose cv-unqualified version is `T2`, three sets of candidate functions, designated *member candidates*, *non-member candidates* and *built-in candidates*, are constructed as follows:

- If `T1` is a complete class type or a class currently being defined, the set of member candidates is the result of the qualified lookup of `T1::operator@` (13.3.1.1.1); otherwise, the set of member candidates is empty.
- The set of non-member candidates is the result of the unqualified lookup of `operator@` in the context of the expression according to the usual rules for name lookup in unqualified function calls (3.4.2) except that all member functions are ignored. However, if no operand has a class type, only those non-member functions in the lookup set that have a first parameter of type `T1` or "reference to (possibly cv-qualified) `T1`", when `T1` is an enumeration type, or (if there is a right operand) a second parameter of type `T2` or "reference to (possibly cv-qualified) `T2`", when `T2` is an enumeration type, are candidate functions.
- For the operator `,`, the unary operator `&`, or the operator `->`, the built-in candidates set is empty. For all other operators, the built-in candidates include all of the candidate operator functions defined in 13.6 that, compared to the given operator,
  - have the same operator name, and
  - accept the same number of operands, and
  - accept operand types to which the given operand or operands can be converted according to 13.3.3.1, and
  - do not have the same parameter-type-list as any non-member candidate that is not a function template specialization.

---

4 EVALUATION

Since, currently the declaration of non-member subscript operators is ill-formed, there are no backwards compatibility issues to expect.