Multidimensional subscript operator

Document #: P2128R4
Date: 2021-03-18
Project: Programming Language C++
Audience: EWG
Reply-to: Mark Hoemmen <mhoemmen@stellarscience.com>
Daisy Hollman <dshollm@sandia.gov>
Corentin Jabot <corentin.jabot@gmail.com>
Isabella Muerte <imuerte@hey.com>
Christian Trott <crtrott@sandia.gov>

Abstract

We propose that user-defined types can define a subscript operator with multiple arguments to better support multi-dimensional containers and views.

Tony tables

Before

```
template<class ElementType, class Extents>
class mdspan {
  template<class... IndexType>
  constexpr reference operator()(IndexType...);
};

int main() {
  int buffer[2*3*4] = { };
  auto s = mdspan<int, extents<2, 3, 4>>(buffer);
  s(1, 1, 1) = 42;
```

After

```
template<class ElementType, class Extents>
class mdspan {
  template<class... IndexType>
  constexpr reference operator[](IndexType...);
};

int main() {
  int buffer[2*3*4] = { };
  auto s = mdspan<int, extents<2, 3, 4>> (buffer);
  s[1, 1, 1] = 42;
}
```

Revisions

R4

- Wording fixes

R3

- Add some discussions about interpreting `t[a][b]` as a syntactic rewrite for a variadic `operator[]` (wich we are not proposing)

R2

- Add explanation about not adapting this proposal to C arrays
- Remove the restriction to require at least one parameter
- Add a paragraph about valarray

Motivation

Types that represent multidimensional views (`mdspan`), containers (`mdarray`), grid, matrixes, images, geometric spaces, etc, need to index multiple dimensions.

In the absence of a more suitable solution, these classes overload the call operator. While this is functionally equivalent to the proposed multidimensional subscript operator, it does not carry the same semantic, making the code harder to read and reason about. It also encourages non-semantical operator overloading.

Proposal

We propose that the `operator[]` should be able to accept 0 or more arguments, including variadic arguments. Both its use and definition would match that of `operator()`.

We make the expressions deprecated in 20 ill-formed while allowing multi-dimensional subscripts expressions in **new** standard types and user types. We do not propose modifications to C arrays as to leave a cycle before giving new meaning to syntax that was still valid in C++20.

What about comma expressions?

In C++20 we deprecated the use of comma expressions in subscript expressions [P1161R3][**?**]. This proposal would make these ill-formed and give a new meaning to commas in subscript expressions. While the timeline is aggressive, we think it is important that this feature be available for the benefit of `mdspan` and `mdarray`. At the time of writing [P1161R3], [**?**] has been implemented by at least GCC, clang, and MSVC. [P1161R3][**?**] further denotes that the cases where comma expressions appear in subscript are vanishingly rare.

However, an implementation could keep supporting the current behavior as an extension, for example, they could fall-back to a comma expression if no overload is found for an expression list, or always assume a comma expression in the presence of a C-array.

Because we should not make C++ more confusing, we think the standard should not continue to support the old meaning of a comma in subscript expressions.

Should we adopt the same syntax for C arrays?

Code that is deprecated in 20, should be ill-formed in 23 rather than a potentially silent change. As such we do not propose the proposed syntax to apply to C arrays. The usefulness of this should be discussed in the C++26 time frame. However C arrays are not widely used by C++, spending time on them might therefore not be useful.

Should we add a multidimentional operator to `valarray`?

Again, we shouldn't change the meaning of existing code in C++23. We should only add multidimensional operators to new in C++23 types such as `mdspan`. If there are users of `valarray` interested in this feature, this can be done in C++26

What about [foo][bar]?

As mentioned in [P1161R3][**?**], an `operator[]` can return an object which has itself an `operator[]`. Therefore chaining multiple `[]` to index a single object isn't a viable proposal.

Should we interpret `t[a][b]` as a syntactic rewrite that looks for a variadic operator[]?

After Richard Smith raised this possibility on the EWG reflector, there was some discussion about this possibility. The rationale behind this suggestion was that current and future generic code that uses multidimensional array semantics (particularly code that may need to work with C-style pointer-to-pointer-like multidimensional arrays) would be syntactically incompatible with types that overload variadic `operator[]`. The authors have ultimately decided to reject this direction for a number of reasons.

First of all, we reject the premise that significant generic code exists or will ever exist that needs to be instantiated with types like `double***`. As experts in scientific computing (expected to be the primary consumer of multidimensional arrays and thus of this proposal), we have seen very little of this sort of generic code. The vast majority of such generic code in production today uses overloads of `operator()`. No suggestion has been made to have `operator()` look for `operator[]` when a sufficient `operator()` overload cannot be found, so such a proposal will not be discussed here (though we find it equally unsatisfying).

Another important reason to shy away from the `t[a][b]` rewrite is related to one of the overall motivations for the variadic subscript operator in general: `t[a]` looks like a valid subexpression, even though it isn't necessarily. This is a confusing and misleading user experience that we'd like to avoid.

Finally, we believe the broader issue of generic interoperability with C-style pointer-to-pointer arrays can be addressed with a non-breaking, follow-on proposal in the other direction: if `t[a, b]` fails overload resolution, the compiler could look for `(t[a])[b]`. This would make things like `std::array<std::array<T, 3>, 4>` work out of the box, without any library changes. The authors would welcome such a proposal, but do not believe it should be a part of the initial language change proposed herein. This potential extension also solves the problem that `t[a][b]` is not a syntax that interacts favorably with parameter packs.

Ultimately, our proposal is one that satisfy the needs of library types without adding complexity to overload resolution. It leaves the door to future evolutions if these evolutions prove useful and usable.

Wording

� Expressions [expr]

� Postfix expressions [expr.post]

Postfix expressions group left-to-right.

postfix-expression:
 primary-expression
 ~~*postfix-expression* [*expr-or-braced-init-list*]~~
 postfix-expression [*expression-list*]
 postfix-expression [*braced-init-list*]
 postfix-expression ($_{opt}$*expression-list*)
 simple-type-specifier ($_{opt}$*expression-list*)
 typename-specifier ($_{opt}$*expression-list*)
 simple-type-specifier braced-init-list

� Subscripting [expr.sub]

A postfix expression followed an expression in square brackets is a postfix expression. One of the expressions shall be a glvalue of type “array of `T`” or a prvalue of type “pointer to `T`” and the other shall be a prvalue of unscoped enumeration or integral type. The result is of type “`T`”. The type “`T`” shall be a completely-defined object type.[1] The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`, except that in the case of an array operand, the result is an lvalue if that operand is an lvalue and an xvalue otherwise. The expression `E1` is sequenced before the expression `E2`.

[*Note:* A comma expression appearing as the *expr-or-braced-init-list* of a subscripting expression is deprecated; see [depr.comma.subscript]. — *end note*]

[1]This is true even if the subscript operator is used in the following common idiom: `&x[0]`.

[*Note:* Despite its asymmetric appearance, subscripting is a commutative operation except for sequencing. See [expr.unary] and [expr.add] for details of `*` and `+` and [dcl.array] for details of array types. — *end note*]

~~A *braced-init-list* shall not be used w~~With the built-in subscript operator~~.~~ a *braced-init-list* shall not be used and an *expression-list* shall be a single expression.

� Overloaded operators [over.oper]

� Subscripting [over.sub]

A subscripting operator function is a function named `operator[]` that is a non-static member function ~~with exactly one parameter~~. For an expression of the forms

~~*postfix-expression* `[` *expr-or-braced-init-list* `]`~~

postfix-expression `[` *expression-list* `]`
postfix-expression `[` *brace-init-list* `]`

the operator function is selected by overload resolution ([over.match.oper]). If a member function is selected, the expression is interpreted, respectively, as

~~*postfix-expression* . *operator* `[]` `(` *expr-or-braced-init-list* `)`~~

postfix-expression . *operator* `[]` `(` *expresssion-list* `)`
postfix-expression . *operator* `[]` `(` *braced-init-list* `)`

[*Example:*

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

— *end example*]

� Comma operator [expr.comma]

In contexts where comma is given a special meaning, [*Example:* in lists of arguments to functions ([expr.call]), subscript expressions, and lists of initializers ([decl.init]) — *end example*] the comma operator as described in this subclause can appear only in parentheses. [*Example:*

```
f(a, (t=3, t+2), c);
```

has three arguments, the second of which has the value `5`. — *end example*]

[*Note:* A comma expression appearing as the *expr-or-braced-init-list* of a subscripting expression [expr.sub] is deprecated; see depr.comma.subscript. — *end note*]

� C++ and ISO C++ 2020 [diff.cpp20]

� `[expr.sub]`: declarations [diff.cpp20.expr.sub]

Change: Change the meaning of comma in subscript expressions.
Rationale: Enable repurposing a deprecated syntax to support multidimensional indexing.
Effect on original feature: Valid C++ program that uses a comma expression within a subscript expression may fail to compile.

```
arr[1, 2] //was equivalent to arr[(1, 2)], now equivalent to arr.operator[](1, 2) or ill-formed
```

� Comma operator in subscript expressions[depr.comma.subscript]

A comma expression appearing as the *expr-or-braced-init-list* of a subscripting expression is deprecated. [*Note:* A parenthesized comma expression is not deprecated. — *end note*]
[*Example:*

```
    void f(int *a, int b, int c) {
        a[b,c];                     // deprecated
        a[(b,c)];                   // OK
    }
```

— *end example*]

Implementation

A prototype has been implemented in Clang.

Compiler Explorer Demo.

Github: `https://github.com/cor3ntin/llvm-project/tree/subscript`

Acknowledgments

Thanks to Jens Maurer for his patient help with the wording, and to the many people who provided valuable feedback. Thanks to Matt Godbolt for hosting an experimental compiler with the implementation of this proposal on compiler explorer.

References

[N4861] Richard Smith *Working Draft, Standard for Programming Language C++*
https://wg21.link/N4861