Multidimensional subscript operator

Document #: P2128R5
Date: 2021-04-13
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

R5

- Expand motivation, use cases in scientific community, as per EWG request.

POLL: send P2128R4 (revised with motivation suggestions) to electronic polling, targeting CWG.

SF	F	N	A	SA
19	8	0	1	0

- Wording fixes

R4

- Expand motivation
- Expand discussion of alternatives
- Add a feature test macro
- Wording fixes

R3

- Add some discussions about interpreting `t[a][b]` as a syntactic rewrite for a variadic `operator[]` (which 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

- C++ uses `operator[]` for array access.
- C++ libraries use various syntaxes to work around `operator[]` not taking multiple arguments.
- Those syntaxes are inconsistent with single-dimensional C++ arrays. They carry the wrong semantic implications, make compile-time error detection and reporting more difficult, and/or hinder inlining.
- Therefore, we should let `operator[]` take multiple arguments.

What are multidimensional arrays?

Multidimensional arrays map multiple integer indices to a reference to an element of the array. They naturally generalize single-dimensional arrays. Programmers use types that behave like multidimensional arrays to represent objects in many domains, including

- matrices (as in linear algebra) and tensors;
- discretized physical space (e.g., for physics simulations or video games); and
- images (as in graphics).

Two examples of multidimensional arrays are the multidimensional array container `mdarray` (P1684R0 [1]) and the multidimensional array view `mdspan` (P0009R10 [4]).

Other types generalize multidimensional arrays to accept index types other than integers. For example, a type could map from a sequence of string "indices" to an element in a hierarchical data format, like XML (as in XPATH) or INI (as in Windows configuration files).

C++ uses square brackets for one-dimensional array access

C++ preferentially uses `operator[]` for one-dimensional array access. Standard C++ provides several different single-dimensional array types: both "native" arrays `T[]`, and the Standard Library types `array`, `span`, and `vector`. All of these types use `operator[]` as the array access operator, that maps from an integer index to a reference to an array element. While not all users consider `string` and `string_view` to be array types, these types also use `operator[]` as the "array access" operator mapping from an integer offset to a reference to the corresponding string character. Finally, the types `map` and `unordered_map` use `operator[]` for table look-ups returning a reference, as a generalization of array access.

Work-around multidimensional array access syntaxes

It would seem that C++ intends `operator[]` for array access. However, C++ does not currently permit passing multiple arguments to `operator[]`. Thus, C++ multidimensional array types must work around by using a different syntax. Many current libraries take one of the following three options:

- `a(x, y, z)`: the function call operator taking multiple indices, as in the Fortran or Matlab languages;
- `a[x][y][z]`: a chain of single-argument array access operators, as with C array-of-array; or
- `a[{x, y, z}]`: an array access operator taking a `tuple` or tuple-like aggregate index type.

We will go through each in turn, and argue that their disadvantages call out for fixing `operator[]` to accept multiple indices.

Function call operator

One syntax for multidimensional array access is the function call operator taking multiple indices, like this: `a(x, y, z)`. In this example, `a` is a three-dimensional array, and `x`, `y`, and `z` are the three indices. Programming languages such as Fortran, Matlab, and Scala use this syntax, as do many C++ libraries (Armadillo, Boost.uBLAS, Eigen, Kokkos) and C++ Standard Library proposals (`mdspan` (P0009R10 [4]) and `mdarray` (P1684R0 [1]). P0009R10 [4] passed LEWG review in 2018 with this syntax. In addition, the ISO C++ Wiki recommends using multiple-parameter `operator()` over the array-of-arrays syntax `a[x][y][z]`.

Use of `operator()` overloads the same syntax for both array access and calling a function or invocable object. It's true that array access is a special kind of function, mapping indices to a reference. However, using the same syntax for both causes several problems.

First, it's not consistent with use of `operator[]` for one-dimensional array access. Other programming languages are consistent in their choice of array access operator, regardless of the array's dimension. Fortran and Matlab use parentheses for both single-dimensional and multi-dimensional arrays. Python use square brackets, as in `a[x]` or `b[x, y, z]`; Mathematica uses double square brackets, as in `a[[x]]` or `b[[x, y, z]]`. Only C++ switches punctuation depending on the number of dimensions. Novice C++ programmers find the language's inconsistency confusing. One anecdote we gathered, is that it puzzles novices to see what appears to be a function call on the left-hand side of an assignment, as in `a(x,y,z) = 42`. One imagines information flowing out of function calls, not into them.

Second, function calls in C++ carry no particular semantics, while array access generally implies mapping from indices to a reference (or proxy reference).

Third, interfaces that take both invocables and multidimensional arrays need to distinguish between them easily at compile time, in order to catch errors at compile time and produce more user-friendly error messages. A type with a call operator used for indexing might incorrectly satisfy the requirements of the `invocable` concept. The proliferation of asynchronous interfaces that take callbacks makes it even more important to catch errors at compile time. Suppose that `combine(f, A, B, C)` applies the binary function `f` elementwise to the two-dimensional arrays `A` and `B`, assigning the results to the elements of `C`. In our preferred syntax, the "inner loop" would look like this: `C[i,j] = f(A[i,j], B[i,j])`. Using the same syntax for array access and function calls might make `combine(A, f, B, C)` (with the order of `f` and `A` reversed) still compile, resulting in run-time errors, or at least fail with more mysterious build errors.

Fourth, many of the C++ libraries that use `operator()` for array access also provide `operator[]` for one-dimensional arrays. This includes Armadillo, Boost.uBLAS, `boost::multi_array`, Eigen, and Kokkos. Users of these libraries prefer `operator[]` or at least want to be able to write generic code that works for C++'s other one-dimensional array types. In Kokkos' case at least, the library's authors consider `operator()` merely a work-around for `operator[]` not taking multiple arguments.

Chain of single-argument array access operators

Another syntax for multidimensional array access is a chain of single-argument array access operators, like this: `a[x][y][z]`. In this example, `a` is a three-dimensional array, and `x`, `y`, and `z` are the three indices. This has the advantage of being consistent with C's array-of-array(-of-array...) syntax, and with C++ `vector<vector<T>>` and similar types. That makes it easier to write generic code that accepts all these types. However, there are a few problems with this syntax.

First, `a[x][y]` implies that `a[x]` is a valid expression. For any custom array type, this implies that `a[x]` is a proxy reference. This hinders inlining by making the required function call depth for an array access no less than the number of dimensions. Anecdotally, the compiler failing to inline array accesses has a devastating effect on performance. Greater function call depth for frequent operations like array access tends to hinder compiler optimizations like inlining. (It is often difficult to demonstrate this with small code examples. Compilers tend to disable optimizations when faced with larger compilation units.) Furthermore, the proxy reference `a[x]` may be expensive to construct or rarely needed for some array types. For instance, for a sparse matrix in compressed column format, `a[x]` would represent a view of the entire row. It's much more complicated to construct this, than just to get the entry at row `x` and column `y`. Column-oriented access is faster than row-oriented access for this sparse matrix format, so it is more common to get column views than row views.

Second, the notation `a[x][y]` does not tell users about the copy behavior of `a[x]`. If `a` is `vector<vector<int>>`, then `auto a_x = a[x]` makes a deep copy. If `a` is `int**`, then `auto a_x = a[x]` makes a shallow copy. Authors of generic code might be tempted to write `auto& a_x = a[x]`, but if `a[x]` is a proxy reference, then it is unsafe to assign it to a reference. This hinders writing generic code that is both safe and avoids unnecessary copies.

Third, `a[x][y]` strongly suggests an array of arrays, perhaps even a C array of arrays (like `int**`). This notation comes with its own semantic expectations, such as:

- the rightmost index is contiguous ("row-wise" storage);
- `a[x]` is a pointer (or an `array` or `vector`) and thus requires allocation (or initialization) for all `x` in range; and
- for `q != r`, `a[q]` and `a[r]` might have different sizes (so that `a` is a "ragged" array).

Fourth, the notations `a(x,y,z)` or `a[x,y,z]` are more friendly to pack expansion than `a[x][y][z]`. It might be possible to extend pack expansion to support a chain of 1-argument `operator[]` (This is explored in [P2355R0], and would require a more substantial change to C++ than what we propose).

The ISO C++ Wiki argues insistently against this syntax. Most C++ libraries that provide multidimensional arrays have also chosen against it.

Array access operator taking a struct of indices

Microsoft AMP's `array` type has an `operator[]` with a single parameter of type `index<Rank>`. This is a `tuple`-like type, where `Rank` is the array's number of dimensions. Users who do not want to construct `index<Rank>` explicitly on each array access can use the syntax `a[{x, y, z}]` (for example), where `a` is a three-dimensional array `a`, and `x`, `y`, and `z` are the three indices. One advantage to this approach is that users can reason about a multidimensional index as a unit. However, this syntax has greater inlining requirements per array expression: the `index<Rank>` object must be constructed, and then unpacked to get the indices. Also, there is no precedent for this `a[{x, y, z}]` syntax in other languages. Furthermore, as with `operator()`, the syntax for multidimensional array access remains inconsistent with that for one-dimensional array access (`a[x,y,z]`) vs. `a[x]`).

General interest and existing practices

The problem this paper solves arises often enough that it is the object of a dedicated question on the ISO C++ Wiki. It is also the object of multiple Stack Overflow questions.

Many languages offer a multidimensional indexing syntax identical to the one we propose, notably C#, D, Julia, Python, R, Raku, Ruby, and Swift. Mathematica uses double square brackets, but permits multiple arguments, like this: `a[[x, y, z]]`.

Summary

The array access operator carries the right meaning for array types in C++, no matter how many dimensions they have. C++ developers see it and immediately know that it maps an index to a reference or reference-like type. Other notations have the wrong semantics. For instance, the function call operator is too generic, while a chain of single-argument array accesses is too specific to arrays-of-arrays. While the notation `a[{x, y, z}]` at least uses the array access operator, the extra level of punctuation is unprecedented, and has potential performance issues. Forcing different punctuation for single-dimensional and multi-dimensional array access is a mistake that other languages do not make, regardless of what symbol they use. In C++, `a[x, y, z]` is the right notation for multidimensional array access.

Proposal

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

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

Frequently asked questions

What about comma expressions?

In C++20 we deprecated the use of comma expressions in subscript expressions P1161R3 [2]. 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 [2] has been implemented by at least GCC, clang, and MSVC. P1161R3 [2] further shows that the cases where comma expressions appear inside the array access operator are vanishingly rare.

Implementations could also continue to support 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.

Should we adopt the same syntax for C arrays?

Code that is deprecated in C++20, should become ill formed in the next version of the C++ Standard (presumably C++23), rather than silently changing meaning. As a result, we do not propose applying the proposed syntax to C arrays. The usefulness of this should be discussed in the C++26 time frame.

Should we add a multidimensional operator to `valarray`?

Again, we do not propose changing the meaning of existing code in C++23. We should only add multidimensional operators to types that will hopefully be new in C++23, 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 [2], an `operator[]` can return an object which itself has an `operator[]`. Therefore chaining multiple `[]` to index a single object isn't a viable proposal on its own. However, in the section below we explore whether a compiler could rewrite `a[x][y][z]` as `a.operator[](x, y, z))` and the challenges with that approach.

Expression rewriting?

Participants in an e-mail discussion on the EWG reflector suggested different ways that the compiler could rewrite multidimensional array access expressions. All suggestions aimed to simplify writing generic code over many different array types, and to smooth the transition path from old to new code. Three separate suggestions in particular came up:

1. rewrite `a(x)` as `a[x]` if `a` is pointer-to-object, and `a(x,y,z)` as `a[x][y][z]` if `a` is a pointer-to-array;
2. rewrite `a[x][y][z]` as `a.operator[](x,y,z)`, if the latter is variable; or

3. rewrite `a[x,y,z]` as `((a[x])[y])[z]`, only if no multiple-parameter `operator[]` overload is viable.

Suggestion 1: Pointer to object or pointer to array

One e-mail discussion participant suggested having the compiler rewrite `a(x)` as `a[x]` if `a` is pointer-to-object, and `a(x,y,z)` as `a[x][y][z]` if `a` is a pointer-to-array. That is, it would make parentheses a unified syntax for function calls or array access. This option would make C++ more like Scala, which uses parentheses for both array indexing and map lookup.

The main issue with this approach, is that it would make some currently ill-formed code compile, but have incorrect behavior. Here is an example:

```
void call(std::function<void(int)>* fp) {
    fp(4);
}
/* ... */
std::function<void(int)> f = [] () {
    std::cout << "Hi!" << std::endl;
    return;
};
call(&f);
```

We also do not favor using the function call operator (parentheses) as a unified syntax for array access and function calls, as we discussed above.

Rewrite `a[x][y]][z]` as `a.operator[](x,y,z)`?

Richard Smith suggested that the compiler interpret an expression with a chain of single-argument `operator[]` as a call to any viable multiple-parameter `operator[]`. For example, in the expression `a[x][y][z]`, if `a` has a viable three-parameter `operator[]`, then the compiler would rewrite `a[x][y][z]` as `a.operator[](x,y,z)`. Otherwise, the compiler would interpret `a[x][y][z]` as it does currently.

This approach would make a chain of single-parameter `operator[]` a unified syntax for many different array types, including

- `mdspan` (P0009R10 [4]) or `mdarray` (P1684R0 [1]);
- `array<array<T, N>, M>` or `vector<...<vector<T>>...>`;
- `boost::multi_array`; or even
- `T[M][N]`.

(Note that `p[x][y]` already works for `shared_ptr<T[M][N]>`.)

There are a number of issues with this approach. First, this would radically expand the scope of proposed language changes, from a rarely used and deprecated syntax `a[x,y]`, to a commonly used syntax `a[x][y]`. Our proposal would only change the meaning of syntax that is already deprecated.

Second, `a[x]` in the expression `a[x][y]` *looks* like a valid subexpression. In current C++, it *is* a valid subexpression. With this rewrite suggestion, it might not be, depending on the type of `a`. This is a confusing and misleading user experience that we would like to avoid. More generally, `a[x][y]` might not evaluate the same as `(a[x])[y]`. Subsequent discussion on the EWG reflector called this "user-hostile." As we discussed above, one of the motivations for `operator[]` taking multiple parameters is to make clear that array access is a single operation.

Third, this suggestion would prevent or complicate use of a chain of multiple `operator[]`, where each `operator[]` takes multiple parameters. Consider the expression `ini_files["home", username, ".foo.ini"]["section", "key"]`. The arguments of the leftmost `operator[]` would be the components of a `filesystem::path`, and the result of `ini_files["home", username, ".foo.ini"]` would be a reference to a `map`-like object representing the contents of an INI file. The arguments of the rightmost `operator[]` would then form a key to a value in that INI file. If the compiler were to rewrite this as `ini_files["home", username, ".foo.ini", "section", "key"]`, then the file path would be wrong, *and* the expression would have the wrong type. One way to resolve this would be a special opt-out syntax for user-defined `operator[]`. However, this would complicate the language even more. In contrast, our proposal simplifies C++ by making `operator[]` more consistent with `operator()`, and removing the rarely-intended comma operator interpretation of `a[x,y]`.

Fourth, it would raise overload resolution and ambiguity questions. Would `operator[](T, T)` requiring user-defined conversions match better than a chain of two `operator[](U)` with exact matches? What if the class has both a single-parameter and a multiple-parameter `operator[]`? Fallback interpretations might introduce even more ambiguity: e.g., would we rewrite `a[x][y][z]` as `a[x,y][z]`?

Finally, we do not think that significant generic code exists or will ever exist that needs to be instantiated with types like `double***`. As experts in scientific computing (a major consumer of multidimensional arrays), we have seen very little of this sort of generic code. The vast majority of such generic code in production today uses multiple-argument `operator()` for multidimensional arrays. Code that uses types like `double***` tends to exploit particular features of pointers, such as

- indirection, where `double***` is a pointer to a `double**` output argument, or
- "ragged" arrays, that is, an array of arrays (of arrays...), where the inner arrays have different sizes.

Code that needs contiguous or strided multidimensional arrays either uses an appropriate class (such as `mdspan`), or represents arrays as a pointer (e.g., `double*`) with attached integer dimensions and stride(s). 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).

Rewrite `a[x,y,z]` as `a[x][y][z]`?

A suggestion in the opposite direction would be for the compiler to rewrite multiple-argument `operator[]` expressions as a chain of single-argument `operator[]` expressions, but only if no viable multiple-parameter `operator[]` exists. For instance, if `a` has a viable three-parameter `operator[]`, then `a[x,y,z]` would just be `a.operator[](x,y,z)`. However, if `a` does not – for instance, if `a` is `vector<vector<vector<int>>>` – then the compiler would rewrite `a[x,y,z]` as `a[x][y][z]`.

We do not propose this, but it has some advantages. First, it would enable writing generic code for C++ array-of array types like `array<array<T, N>, M>` or `vector<...<vector<T>>...>`, C arrays-of-arrays, and multidimensional arrays like `mdspan`. Users would use the array access operator – the most appropriate syntax for multidimensional array access, in our view – for all these types. Second, the syntax avoids conveying the false impression that subexpressions like `a[x]` in `a[x][y][z]` must be valid objects or that they get constructed. Third, it would work with parameter packs, without further language changes.

This approach would have analogous overload resolution and ambiguity questions as the previous rewrite suggestion. However, if those questions could be resolved, then we would welcome it as a nonbreaking, follow-on proposal to ours. We think our proposal satisfies the needs of library types and leaves the door open to future evolution, while making the language easier to understand.

Rewriting schemes aren't needed for a unified array access syntax

One argument for the above rewriting schemes would be to provide a unified array access syntax, that would work for `mdspan`, nested C++ types like `vector<vector<T>>` and `array<array<T>>`, and C arrays of arrays. However, one can get much of this effect just by letting `operator[]` take multiple parameters. The following example subclasses `array<T>`, so that if `T` has an `operator[]`, then it gets any extra arguments passed to the (outer) `operator[]`. This works recursively, for `array<array<T>>`, `array<array<array<T>>>`, etc. We are not proposing this for the C++ Standard, but it illustrates how much one can accomplish with just the small change to the language that this paper proposes.

```
#include <array>
#include <span>

template <class From, class To>
concept convertible_to =
  std::is_convertible_v<From, To> &&
  requires(std::add_rvalue_reference_t<From> (&f)()) {
    static_cast<To>(f());
};

template <typename T, std::size_t S>
struct array;

template <typename T, auto N = 0>
constexpr inline bool _is_array = false;
template <typename T, auto N>
constexpr inline bool _is_array<array<T,N>> = true;
```

```
template <typename T, std::size_t S>
struct array : std::array<T, S>  {
    static constexpr inline std::size_t extent = [] () -> std::size_t {
        if constexpr(_is_array<T>) {
            return 1 + T::extent;
        }
        return 1;
    }();

    constexpr decltype(auto)
    operator[](std::size_t idx) {
        return  *(this->data() + idx);
    }

    constexpr decltype(auto)
    operator[](std::size_t idx, convertible_to<std::size_t> auto&&... args)
    requires (sizeof...(args) < extent) && (sizeof...(args) >= 1) {
        typename std::array<T, S>::reference v = *(this->data() + idx);
        return v.operator[](args...);
    }

    constexpr decltype(auto)
    operator[](std::size_t idx) const {
        return  *(this->data() + idx);
    }

    constexpr decltype(auto)
    operator[](std::size_t idx, convertible_to<std::size_t> auto&&... args) const
    requires (sizeof...(args) < extent) && (sizeof...(args) >= 1) {
        typename std::array<T, S>::reference v = *(this->data() + idx);
        return v.operator[](args...);
    }
};
```

This `array` subclass enables code like the following:

```
// 2 x 3 array of arrays
array aa {array{1, 2, 3}, array{4, 5, 6}};
static_assert(decltype(aa)::extent == 2);

assert( (aa[0,1] == 2) ); // extra parens for macro reasons
assert( (aa[1,2] == 6) );

array bb {array{7, 8, 9}, array{10, 11, 12}};
array cc {array{13, 14, 15}, array{16, 17, 18}};
array dd {array{19, 20, 21}, array{22, 23, 24}};

// 4 x 2 x 3 array of arrays of arrays
array aaa{ aa, bb, cc, dd };
static_assert(decltype(aaa)::extent == 3);
    assert( (aaa[1, 1, 1] == 11) );
```

Non-member `operator[]`?

After our presentation of P2128R3 to EWG, further e-mail list discussion brought up non-member `operator[]`. Permitting this would let users use our proposed `a[x,y,z]` syntax with existing types that do not use multiple-parameter `operator[]`, without modifying those types. This would enable a migration strategy towards adoption of our proposed syntax. Others suggested that it could be useful for SIMD indexed array access operations.

We think nonmember `operator[]` is out of scope of this paper, though we would not object to it being considered by a separate follow-on paper. As a migration strategy, it has the issue that a nonmember `operator[]` could break in the future, if the class' author later adds a member `operator[]`. This would create an ambiguous overload set, and therefore, an ill-formed program. Nevertheless, there might be motivating use cases.

`static operator[]`

Our proposal does not support static `operator[]` declaration, but we would not oppose such a proposal

We do recommend it should be consistent with the call operator, as is explored in P1169R0 [3].

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*]

[*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*$_{opt}$`]`
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*$_{opt}$`)`
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*]

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

� 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*]

Feature test macros

Insert an entry in the table into [tab:cpp.predefined.ft]

```
__cpp_multidimensional_subscript | <DATE OF ADOPTION>
```

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

[1] D. S. Hollman, Christian Trott, Mark Hoemmen, and Daniel Sundernland. P1684R0: mdarray: An owning multidimensional array analog of mdspan. `https://wg21.link/p1684r0`, 6 2019.

[2] Corentin Jabot. P1161R3: Deprecate uses of the comma operator in subscripting expressions. `https://wg21.link/p1161r3`, 2 2019.

[3] Barry Revzin and Casey Carter. P1169R0: static operator(). `https://wg21.link/p1169r0`, 10 2018.

[4] Christian Trott, Bryce Adelstein Lelbach, Daniel Sunderland, D. S. Hollman, H. Carter Edwards, Mauro Bianco, Ben Sander, Athanasios Iliopoulos, John Michopoulos, and Mark Hoemmen. P0009R10: mdspan. `https://wg21.link/p0009r10`, 2 2020.

[N4885] Thomas Köppe *Working Draft, Standard for Programming Language C++*
`https://wg21.link/N4885`

[P2355R0] Davis Herring *Postfix fold expressions*
`https://wg21.link/P2355R0`